

Neuron Data Elements Environment Elements Application Services

Version 4.1

C++ Programmer's Guide

© Copyright 1986–1997, Neuron Data, Inc. All Rights Reserved.

This software and documentation is subject to and made available only pursuant to the terms of the Neuron Data License Agreement and may be used or copied only in accordance with the terms of that agreement. It is against the law to copy the software except as specifically allowed in the agreement. This document may not, in whole or in part, be copied, photocopied, reproduced, translated, or reduced to any electronic medium or machine-readable form without prior consent, in writing, from Neuron Data, Inc.

Use, duplication, or disclosure by the U.S. Government is subject to restrictions set forth in the Neuron Data License Agreement and in subparagraph (c)(1) of the Commercial Computer Software-Restricted Rights Clause at FAR 52.227-19; subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013; subparagraph (d) of the Commercial Computer Software—Licensing clause at NASA FAR supplement 16-52.227-86; or their equivalent.

Information in this document is subject to change without notice and does not represent a commitment on the part of Neuron Data. THE SOFTWARE AND DOCUMENTATION ARE PROVIDED “AS IS” WITHOUT WARRANTY OF ANY KIND INCLUDING WITHOUT LIMITATION, ANY WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. FURTHER, NEURON DATA DOES NOT WARRANT, GUARANTEE, OR MAKE ANY REPRESENTATIONS REGARDING THE USE, OR THE RESULTS OF THE USE, OF THE SOFTWARE OR WRITTEN MATERIAL IN TERMS OF CORRECTNESS, ACCURACY, RELIABILITY, OR OTHERWISE.

Open Interface Element™, Data Access Element™, Intelligent Rules Element™, and Web Element™ are trademarks of, and are developed and licensed by Neuron Data, Inc., Mountain View, California. NEXPERT OBJECT® and NEXPERT® are registered trademarks of, and are developed and licensed by, Neuron Data, Inc., Mountain View, California.

Other brand or product names are the trademarks or registered trademarks of their respective holders.

Contents

Preface

Purpose of this Manual	xv
Audience	xv
How to Use This Manual	xv
Related Manuals	xvi

1. Introducing EE Application Services

Introduction	1
Building Block Mechanisms	1
Data Source/View Mechanism	2
Application Services Classes	2

2. Using Data Source/View

Using Data Source/View in an Application	5
Propagating Events	5
Controlled Access to Data Sources	6
Locking Data in Table Datasources.	7
Locking Data in List Datasources.	7
Data Source/View Examples	8
OI Example	8
DA Example	9
IR Example	9
Data Source Internals.....	9
Internals for OI Core Data Sources.....	9
Internals for DA Data Sources.....	10
RecordSetDataSource Implementation	10
Properties from VariantTable	10
Methods from VariantTable	11
Internals for IRE Data Sources	12
Example	13
Input Table (LBox)	14
Selection table (ListBox)	15
IRE Text Edit	16

3. Tree Datasource: Managing Hierarchical Data

Concepts	17
Tree Datasource	18
Node	18
Tree	20
Node Accessor	21
Cursor	23
Edit Object	24
Options for the NDTView and NDBrows Views.....	28
cursor	28

initexpandlevel	29
autosize	30
Building a Tree Datasource	30
Creating and Destroying a Tree Datasource	31
Creating and Destroying a Node Accessor	31
Creating and Destroying an Edit Object	32
Adding Nodes	33
Managing Memory	42
Editing a Tree Datasource.....	44
Datasource-Level Editing	47
Node-Level Editing	51
Advanced Topics.....	52
Node-Count Functions	52
Managing the Cursor	55
Acting on Multiple Nodes	55
Persistent Data Storage and Relational Tables	57
4. Graph Datasource: Managing Graph Data	
Concepts	59
Graph Datasource	59
Node	60
Edge	63
Graph	66
Accessor	70
Cursor	73
Edit Object	75
Options for the NDDGram View	80
autosize	81
cursor	82
readonly	83
Diagrammer	83
Custom Node and Link Options	98
Building a Graph Datasource.....	112
Creating a Graph Datasource	113
Creating and Destroying an Edit Object	113
Creating Accessors	114
Creating Nodes	117
Creating Edges	124
5. Args Class	
Overview	127
API Overview	127
Scanning the List of Command Arguments.....	128
6. ArNum Class	
Overview	131
API Principle.....	131
Macros.....	131
Constructors and Destructor	132

Clone, Copy, Reset	132
Changing the Length of the Array	133
Global Queries	133
Accessing Elements.....	133
Finding Elements	134
Adding Elements.....	135
Removing Elements	135
Sorting.....	136
Removing Duplicates	136
7. ArObj Class	
Overview	139
API Principle.....	139
Constructors and Destructor	142
Clone, Copy, Reset.....	142
Changing the Length.....	142
Global Queries.....	143
Accessing Elements.....	143
Finding Elements	143
Adding Elements.....	144
Removing Elements	145
Sorting.....	146
Removing Duplicates	146
8. ArPtr Class	
Technical Overview	147
API Principles	147
Macros	147
Constructors and Destructor	148
Constructors	148
Destructor	148
Clone, Copy, Reset.....	148
Changing the length of the array.....	149
Global Queries.....	149
Accessing Elements.....	149
Finding Elements	150
Adding Elements.....	151
Removing elements	152
Sorting.....	152
Removing Duplicates	153
9. ARRay Class	
Overview	155
10. ARRec Class	
Overview	157
API Principle	157
Macros	157
Constructors and Destructor	158
Constructors	158
Destructor	158
Clone, Copy, Reset.....	158
Changing the length	158

Global Queries	159
Accessing Elements	159
Finding Elements	159
Adding elements	160
Removing Elements	161
Sorting	162
Removing Duplicates	162
11. Avl Class	
Overview	163
Data Structures	163
AvlTree and AvlNode Classes	164
AvlNode Class	164
Constructors and Destructor	164
Accessing the AvlNode Key	164
Scanning AvlNodes	165
AvlTree Class	165
Constructors and Destructor	165
Queries	166
Propagating an Action	166
Current Node API	167
12. Base Class	
Technical Summary	169
Basic Data Types	169
BoolEnum	173
CpyEnum	173
CmpEnum	174
PerfEnum	174
VertEnum and HorzEnum	175
Version Enum	175
Debugging Macros	176
Exit Status	179
Miscellaneous Basic Macros	180
13. BBuf Class	
Overview	183
Examples:	183
Examples:	183
BBuf Class	185
Specialization Flags	185
Data Structures	185
Constructors and Destructor	186
Constructors	186
Destructor	187
Read and Write Operations	187
Seek Operations	188
Accessing Private Fields	189
Installing Custom Paging Methods	191
14. Cell Class	
Technical Summary	193
Data Structures	193

Cell Range Operations.....	194
15. Char Class	
Technical Summary	195
Environment Variables.....	197
Data Structures	197
Character Length.....	199
Character Code.....	199
Basic Character Classification	201
Basic Character Conversion.....	202
Conversions between ASCII and EBCDIC.....	205
16. Cs Class	
Overview	207
Code Sets	207
Creating and Destroying.....	211
Constructors	211
Destructor	211
Convenience Macros.....	211
Predefined Code Sets.....	213
Local Macros	213
ISO LATIN1 Character Information Definition.....	213
ASCII Character Information Definition	213
JIS0208 Character Information Definition	214
JIS0201 Character Information Definition	214
17. Ct Class	
Technical Summary	215
Data Types	216
Enumerated Types.....	216
Creating and Disposing	219
Member Functions	219
18. Ds Module	
Design Overview.....	223
Classes.....	223
View Interface.....	223
Edition Interface	224
Update Interface.....	225
Contained/Container Data Source Interface	225
Creating and Disposing	226
Class	226
Edition Operation	226
Modifications Implementation	226
Data Source	226
19. Err Class	
Overview	227
Disciplined Exceptions	227
Error Handling And Reporting	228
Entry/Exit Macros	229
Error Recovery	229

Retry	229
Signalling A Failure	230
Fatal Errors	231
Error Contexts	232
Error Tracing	233
Global Variables And Initialization	233
Advanced Error Reporting	235
Summary Of Error Handling And Reporting	235
Reporting Errors for Calls to Third Party APIs	236
Data Structures	237
ErrFrame API for Error Reporting and Discrimination	237
ErrFrame Class	238
Macros.....	239
Context Messages and Tracing	239
Misc Macros For Error Reporting.....	240
ERR_LIB, ERR_EXTERN.....	240
Initialization Macros.....	241
Fatal Errors.....	241
Signaling Failures.....	241
Generating Warnings	242
Querying the Error State	242
Assertions.....	242
Error Reporting	243
Error Conditions Signaled by Error Module.....	243
Exiting from the Application.....	243
UNIX Exception Handling.....	243
W16 Exceptions Handling	244
Mac Exceptions Handling.....	245

20. File Class

Technical Summary	247
Data Structures	251
Enumerated Types.....	252
Accessing File Attributes	255
Checking Existence and Access Rights of a File	258
Opening and Closing a File	259
Querying and Changing Position in a File	262
Reading and Writing	264
Miscellaneous Functions.....	269
Default Search Path.....	270
Direct access to native File I/O	271
Errors	272

21. FMgr Class

Technical Summary	273
Data Types	273
Enumerated Types.....	276
Querying and Changing File/Directory Attributes.....	281
Finding File Type by Mac Type or by File Extension	283
Creating	286
Copying	287
Moving.....	288

Deleting	289
Performing an Action	291
22. FName Class	
Technical Summary	295
Data Types	299
Enumerated Types	301
File Name Syntax	304
Find Path Name Syntax	305
Checking Path Name Validity	305
Evaluating Variable Expressions	306
Conversion between Syntaxes	307
Conversion Status	308
Extracting File Components	309
Directories Specified as Paths or as Files	310
Top Directory	312
Current Volume / Current Directory	313
Parent Directory	314
Home Directory	315
Absolute / Relative Parts	316
Comparing File Names	316
Generating Temporary and Backup File Names	317
23. Hash Class	
Overview	319
Data Structures	319
NDHashInfo	319
Constructors and Destructor	320
Convenience Functions	320
Resetting a Hash Table	320
Creating and Disposing Hash Tables	320
Defining a Hash Table	320
Querying the Hash Table Information	321
Using Hash Tables	322
Perform An Action On All The Entries	322
Default Methods	323
Default Hashing	323
Default Comparison	323
Default String Cloning	323
Hash Table Entries	323
Statistics	324
24. Heap Class	
Overview	325
Heap Class	325
Constructor and Destructor	325
Heap Size	325
Heap Manipulation	326
25. ISet Class	
Overview	327
Data Structures	327
Constructors and Destructor Interval Sets	327

Special Intervals	328
Adding and Removing Intervals	328
Comparing and Combining Two Sets.....	329
26. MCH Class	
Technical Summary	331
Compiler Information	336
27. Nfier Class	
Overview	339
Creating and Disposing	340
Broadcasting a Notification	340
Notifier Client Creation and Destruction	340
Associating Client Data with the Notifier Client Pointer.....	341
Notifier Client Registration and Unregistration.....	341
Convenience: Unregistration, destruction and deallocation	341
28. Pack Class	
Overview	343
Short Description of the Compression Algorithms:	343
Choice of a Compression Algorithm:	343
Constructors and Destructor	344
API Usage.....	344
Compression	344
Decompression	345
Worst Case Performances	346
RLE (Run Length Encoding)	346
PackBits.....	346
CCITT Fax Compression.....	347
Overview	347
Examples:	347
General Case	348
29. Plfd Class	
Overview	349
Scope of Documented API	349
Permanent Field Data Types	349
Field Categories	350
Data Structures	350
WARNING:	350
30. Point Class	
Overview	351
Constructors / Destructor	351
Sets and Queries.....	352
31. Pool Class	
Overview	355
Pool oriented memory management	355
Pool Definition.....	356
Constructors and Destructor	356
Constructors	356

Destructor	357
Setting/Querying the Information on a Memory Pool	357
Allocating and Deallocating	357
Statistics	358
32. Ptr Class	
Technical Overview	359
Data Types	359
Enumerated Types	360
Alignment	361
Alloc, Free, and Realloc	361
Functions for Memory Copy, Move, Set.....	363
Statistics	365
Low-level Byte Copies.....	365
Machine-Independent Memory Representations for Integers	366
Memory Representations for Strings	367
Errors Signalled by Ptr Class.....	368
33. RClas Class	
Persistent Data.....	371
Allocation/Deallocation	372
Member Functions	373
Querying Database of Resource Classes.....	373
Testing Inheritance	374
34. Rect Class	
Technical Summary	377
Point Functions.....	378
Rect Functions	379
Rectangles Defined by Origin and Extent	381
Rectangles Defined by Beginning and End.....	382
35. Res Class	
Technical Summary	383
Creating and Disposing	388
Saving To a Resource Database	389
Output to a Text Resource File.....	389
Resource Library Initialization.....	389
Loading and Finding Resources	390
Accessing the Name of a Resource	394
Accessing Client Data of a Resource	395
Accessing Children of a Resource	395
Accessing the Class of A Resource	396
Resource States	396
Resource Notifications	396
Sending Notifications	399
Sending versus Posting	399
Sending A Notification With Data	399
Responding to a Notification	402
Control Data.....	402
Command Management	403
Command Routing	403
General Purpose	403

Command Sources	403
Handling Command Notifications	404
Resource Scripting	404
Error Handling Utilities	404
36. Region Class	
Technical Summary	405
Enumerated Types	405
Empty Region	405
Region Rectangular Bounds	406
Region Translation	406
Comparisons with other Regions	406
Operations between Two Regions	407
Operations between a Region and a Rectangle	408
Regions Specified by a Polygon	409
Performing an Action on Each Rectangle Component of a Region	410
37. RLib Class	
Technical Summary	411
Accessing Libraries	411
Loading, Unloading, and Closing	412
38. SBuf Class	
Technical Summary	415
Simple Queries	415
Iteration	416
Miscellaneous Queries	416
Changing Contents	417
Case Conversion	418
Matching	419
39. Sclpt Class	
Technical Summary	421
Widget Scripts	421
Variables	422
Script Data Types	422
Statements	422
SELF	423
Using the Scripting Environment	423
Extending the Script Language	424
Registering Constants	424
Registering Events	424
Registering Verbs	425
Running a Script in Standalone Applications	428
Bare Scripts	429
40. Set Class	
Overview	431
Constructors and Destructor	431
Special Shared Sets	431
Adding, Removing, Accessing Elements	431
Comparing and Combining Two Sets	433

41. Str Class	
Technical Summary	435
Data Types	437
Cloning and Disposing.....	439
Set and Append.....	440
String Length	442
Iterating through Strings.....	442
Writing into String Buffers	444
Basic String Comparisons	448
Testing Matches.....	450
Searching	451
Scanning of Numeric Values.....	454
Formating the Numeric Values.....	458
Basic Conversions	459
Loading from Resources	461
Conversions Between Code Types	462
42. StrL Class	
Technical Summary	465
Class	465
Accessing the Strings.....	465
43. StrR Class	
Technical Summary	469
Class	469
Loading a String Resource.....	469
Accessing Text	470
Accessing the Id.....	470
44. Var Class	
Type System.....	471
“Variant” Management.....	473
Class	473
Conversion Methods	474
Information Methods.....	477
45. VarDs Class	
Variant Data Source Value.....	479
Notifications.....	479
Variant Data Source	480
46. VarGr	
Design Overview.....	481
Graph Properties	481
Graph Title	481
Node and Edge Accessors	482
Node Accessor	482
Edge Accessors	482
“All” Edge Accessor	482
“In” edge accessor	483
“Out” Edge Accessor	483
Undirected Edge Accessor	483
Node Accessors Navigation	483

Edge-Accessor Navigation.....	485
“All” Edge Accessors	485
“In” Edge Accessors	486
“Out” Edge Accessors	487
Undirected Edge Accessors	487
Adding and Removing Nodes	488
Adding and Removing Edges.....	488
Graph-Node Properties.....	490
Accessor Validity	490
Node Counts	490
Node ID	491
Node Value	491
Node XOrigin	492
Node YOrigin	492
Node Height	493
Node Width	493
Additional Node Properties	494
Graph-Edge Properties.....	495
Accessor Validity	495
Edge Count	495
Edge ID	495
Edge Value	496
Directed Edge	496
Additional Edge Properties	497
Node-Relationship Discovery	498
Getting and Setting the Cursors.....	498
Convenience Methods.....	499
Advanced Objects and Methods.....	500
Node and Edge Objects	500
Edit Objects	507
Modification Descriptions	507
Class Operations	507
47. VarLs Class	
Design Overview.....	509
Class.....	509
Reading and Writing in the List.....	509
List Title	509
Row Titles	510
Row Values	510
Modifying the List.....	511
Reading and Setting the Cursor Row.....	511
Edition Objects.....	512
Modification Descriptions	513
Notifications.....	513
48. VarTb Class	
Technical Overview	515
Class	515
Table Interaction.....	515

Read Support	515
Row Title	516
TableTitle	516
Reading and Setting the Cursor Row and Column	517
Edition Support	517
Edition Objects.....	519
Modifications Queries	520
Row Interaction	520
Column Interaction.....	520
Cell Interaction	521
Virtual Interface Implementation	521
Variant List Implementation	521
Variant List Row Implementation	521
Variant List Row Implementation	521
Variant List Cell Implementation	521
49. VarTr	
Design Overview.....	523
Tree-Datasource Properties	523
Tree Title	523
Node-Accessor Navigation.....	524
Convenient Navigation	525
Adding and Removing Nodes	525
Class Operations	526
Tree-Node Properties	526
Tree-Node Discovery and Navigation	526
Reading and Setting the Cursor.....	526
Modifying the Tree Datasource	527
Tree-Node Values	527
Tree-Node IDs	528
Modifying the Tree-Node Datasource	528
Modification Descriptions	529
50. VStr Class	
Technical Summary	531
Changing Contents	531
Queries.....	532
Concatenation, Insertion, Deletion.....	533
Comparisons	534
Loading Resources.....	535
Arrays Of Strings	535



Preface

Purpose of this Manual

This manual describes Open Interface Element™, the C++ language application programming interface (API) for developing applications with graphical user interfaces for any standard windowing system. The Open Interface API is a highly modular ANSI C++ library. The modules group calls by categories that closely follow standard interface functionality.

In this document “Open Interface Element™” and “Open Interface” will be used interchangeably.

Audience

This manual is designed for people who understand programming concepts, the C++ language, and Open Interface. If you are not familiar with programming concepts, you may need to review an introductory programming book before you use the API. If you are not familiar with Open Interface, you may need to review the Programming Guide. For a complete list of available documents, see Related Manuals below.

How to Use This Manual

To communicate the API’s functionality to you in this manual, we have chosen to adhere to the organization of the software itself. This approach was adopted in favor of the traditional chapter-oriented reference manual for several reasons. Most importantly it permits you to transfer your experience using the Open Interface API directly into using this manual to locate call descriptions. Therefore the body of this reference documents each call in alphabetical order, by software module.

Overall, we believe this yields a significant usability improvement. Because the API is already highly modular and the calls themselves follow a standard naming convention that places the module name in front of the call, you will always be able to find the call by deciding which module it belongs to. To aid in this task, the Reference Manual includes a standard table of contents and running page heads.

Document table of contents

A standard listing of all the calls contained in this volumen of the reference following this Preface. Because all calls are organized by module name they appear in alphabetical order. Although, you may notice that each module’s data structures and enumerated types always begin a new module.

Page heads

If you have acquainted yourself with the organization of the API, you are ready to use the reference to locate calls. To help you find the desired module and call, each page shows the module name printed across the top of the page. Simply flipping the top corner of the page will reveal this information.

Related Manuals

This manual is a member of the Open Interface document set. Each document addresses a different aspect of the product. To avoid duplicating information between manuals, references to related topics in other manuals are given when needed. It is therefore recommended that you familiarize yourself with the complete set of Open Interface documents as follows.

Programming Guide

Reference Manual: Volume 1 and Volume 2

Reference Manual Supplement

User's Guide

Introducing EE Application Services

Introduction

The Elements Environment Application Services (EAS) provide the support layer for built-in memory and print management, graphic primitives, error handling, file I/O, asynchronous event management, and string manipulation services to reduce the development time spent coding these low-level, platform specific functions. These services enable the portability of the graphical presentation layers of an application as well as the integration layers.

EAS provides is the underlying support for Internationalization to allow quick ports of applications to any of a dozen single-byte or double-byte foreign languages including Japanese.

Internationalization features include character sets, porting support and rendering, edit-in-place, standard or native in-text widgets, and string manipulation services.

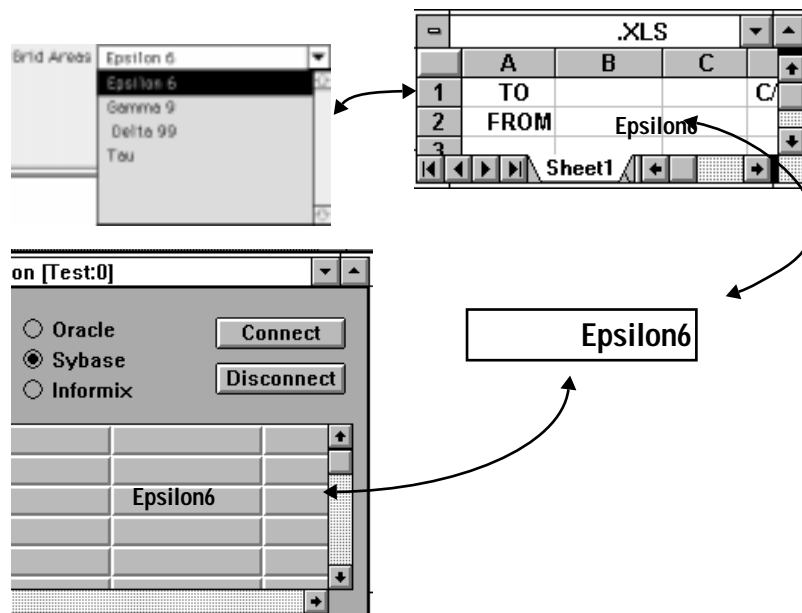
Building Block Mechanisms

In addition to these low-level services, EE Application Services feature higher level application development services that provide more complex building blocks which you can use to assemble your application.

These building block mechanisms, such as the **Data Source/View** mechanism, free the application developer from performing repetitive coding tasks related to the manipulation of data, data sources, and the display of data for complex widgets such as tables and list boxes.

Data Source/View Mechanism

Data Source/View is a mechanism designed to provide underlying bi-directional linkage protocols between views and data sources for applications written in C, C++, and scripts.



The Data Source View mechanism allows you to present and access the same data (such as information from a database) in multiple views, such as a spreadsheet-like table, a choice box, or an input field.

This manual describes the architecture of the data source/view mechanism and includes information for using it in developing applications. The OOScript language class definitions that support the Data Source/View mechanism are described in Chapter 2, "Core Reference", of the *OOScript Language Reference* manual.

Application Services Classes

The EE Application Services or (Core Services) are provided through C/C++ or OOScript language classes. These classes include:

- VStr
- Str
- SBuf
- Base
- APP
- CT
- File
- FMgr
- FName
- MCH
- Ptr

- Variant...
- Data Source...
- Resource (Rlib, RClass,...)

The C/C++ classes are described in the OIE API Reference. The OOScript Language Reference describes the equivalent classes for scripting.

Using Data Source/View

The Data Source/View (DS/V) mechanism provides a data-centric as opposed to widget-centric approach to programming. The following sections describe the high-level tasks required for using the DS/V mechanism to create applications. DS/V examples and internals for the Open Interface Element (OIE), Intelligent Rules Element (IRE), and the Data Access Element are also included.

Using Data Source/View in an Application

The typical procedure for using Data Source/Views is as follows:

1. Create and initialize a data source from a server (Core, DA, or IR) or use the Resource Manager method, `LoadInit()`, to load and initialize a data source of a given type:
 - `VariantDataSource`
 - `VariantListDataSource`
 - `VariantTableDataSource`
 - `RecordSetDataSource`
 - `NxDataSource`
 - `NxTableDataSource`
2. Create one or more view or views.
3. Register a view (ListBox, ChoiceBox, TextEdit or CheckBox) to a data source.

Note: Some data sources support only certain views. For more information about datasource views, see the *OOScript Programmer's Guide*.

4. Set the view option using the property you want from the `ViewOptions` metaclass.

Note: For a complete list of view options, see the *OOScript Programmer's Guide*.

5. Populate the data source with data. Populating the data source can also be done prior to registering the view.

Propagating Events

When a view (widget) is registered to a data source, the view's default notification handler is reset to the DSV handler as appropriate for the type of widget. It is important to allow DSV to still process these notifications. The following events are handled by DSV for each type of view:

CBOX_NFY...

- MOUSECLICK
- KEYCHAR
- ELTSELECTED
- ELTDRAW
- GETELTSTRING
- END

CBUT_NFY....

- HIT
- PROPOSE
- END

LBOX_NFY...

- CELLDRAW
- CELLSTRING
- END
- ENDEDIT
- SELOPERATION
- STARTEDIT
- VALIDATE

TED_NFY....

- END
- KEYCHAR
- QUERYVALIDATE
- VALIDATE
- HIT

Note: If you are using callbacks, you must use the default procedure for these events or you will disable the data source update mechanism.

Controlled Access to Data Sources

To prevent conflicts in accessing the same data, the Data Source/View mechanism provides synchronized and controlled access by allowing only one view to modify the data at any one time.

Simple data manipulations (typically cell-type operations) do not require an explicit edit to be initiated on the data source. Data updates are handled, transparently, by the view registration default methods. When data is changed through the views, an "atomic" edition is performed on the data source that begins an edition, updates the data and ends.

Programmatic control over data source updating can be done when an application updates data sources by explicitly beginning an edition on the data source, performing the updates, and ending or aborting the edition.

Complex operations require building an edition. A complex operation might be performing multiple operations (locally or globally, in the case of a list or table) on a data source.

The following procedure describes the steps you must follow to implement synchronization and controlled access:

1. Before modification of data takes place, you must make an Edition authorization request. This request locks the data source or part of the data source.
2. Request to start a edition on a data source (or a subset of a data source such as a cell, row or column in the case of a table).
 - If the data source has an open edition (i.e., is locked) the request is denied.
 - If the there is an open edition, the views will access the data from the data source in a read-only mode.
3. Once the data source is locked, you can make any changes to the data source or the part you locked. You can make your changes to the data source through the edition.
4. End or abort the edition.

When all updates to the data are complete, you must do one of the following:

- “end” the data source edition (all changes are made).
- “abort” the edition (all changes are not made).

Note: In this release, Data Source Views only supports “End” or “Abort” i.e., all changes are made or none are made.

5. If the owner of the data source (in the case of IR or DA data sources) has updated the data during your edition, your attempt to end the edition and update the data source may be rejected.
6. If your request has been granted, you obtain a lock on the data source (or subset). No one, other than you or the owner of the data source, can abort the edition.

Note: For IR data sources, the owner is the Rules Processor. For DA data sources, the owner is DA itself.

Locking Data in Table Datasources.

In table data sources, Data Source/Views can lock data at any one of the following levels:

- Cell
- Row
- Column
- Entire Table

The locking is exclusive. If you lock a cell and try to also lock the same row or lock the entire table, the lock request is rejected.

Locking Data in List Datasources.

In list data sources, Data Source/Views can lock data at any of the following levels:

- Cell
- Entire list

Controlled Access Example

Here is an example of an edit operation upon initializing a table datasource:

```
edit := internal_ds.StartEdit();
if(!isnull(edit));

edit.RowCount(2,7);

edit.ColumnTitle(0) = "Company";
edit.ColumnTitle(1) = "Contact";
edit.ColumnTitle(2) = "Address";
edit.ColumnTitle(3) = "City";
edit.ColumnTitle(4) = "State";
edit.ColumnTitle(5) = "Zip";
edit.ColumnTitle(6) = "YTD Purchases";

edit.CellValue(0,0) = "XYZ Corporation";
edit.CellValue(0,1) = "Jane Doe";
edit.CellValue(0,2) = "123 Main Street";
edit.CellValue(0,3) = "Anytown";
edit.CellValue(0,4) = "CA";
edit.CellValue(0,5) = "10001-0000";
edit.CellValue(0,6) = 12500.00;

edit.CellValue(1,0) = "Sony";
edit.CellValue(1,1) = "Doris Doubleday";
edit.CellValue(1,2) = "268 River Oaks Parkway";
edit.CellValue(1,3) = "San Jose";
edit.CellValue(1,4) = "CA";
edit.CellValue(1,5) = "94041-1230";
edit.CellValue(1,6) = 80000.00;

edit.End();
```

Data Source/View Examples

The following examples illustrate the use of the Data Source/Views mechanism using Neuron Data's OOScript language. The coding is similar in C/C++.

OI Example

The following example links a TextEdit to a VariantDataSource with a simple atomic edition performed automatically by setting the value of the VariantDataSource.

Linking a TextEdit to a Variant Data Source

```
// "ted"      a TextEdit object reference
// "coreserver" a reference to the Core server

// Create a datasource.
ds := coreserver.VariantDataSources.Create();
ds.Init(); // initialize

ds = "hello"; // "atomic" edition performed here
ds.RegisterView(ted);

// the TextEdit then displays the data in the ds data source
```

DA Example

The following example links a DA DatabaseViewDataSource (created from a DatabaseView) to a ListBox.

Linking a DA DatabaseViewData Source to a List Box

```
// "databaseview" a DatabaseView object reference
// "lbox"      a ListBox object reference
// "coreserver" a reference to the Core server
// "daeserver" a reference to the DA Core server

// "databaseview" has already been populated with data from
// a database somewhere...
databaseview :=

daeserver.DatabaseViewDataSources.CreateFromDatabaseView;

DatabaseView.RegisterView(lbox);

// the ListBox then displays the data in the data source
```

IR Example

The following example links a Text Edit with a IR slot with automatic and implicit controlled edition.

Linking a TextEdit with an NXP slot

```
// Assume that a Text Edit object reference is in the ted
// variable
// and nxsvr contains the Nx serve
ds := nxsvr.NxTableDataSources.Create();
ds.Atom = nxsvr.Objects.Car.Color; // assuming that Car.Color
is a slot in NXP
ds.Strategy = nxsvr.Engine.VSTRAT_VFWRD;
ds.RegisterView(ted);
// the rest (local update, forwarding the data,...) is handled
// automatically by the IRE data source
```

Data Source Internals

Data Source Internals defines the relationships and inheritance of the data sources for the OIE, DAE, and IRE.

Internals for OI Core Data Sources

The OI Core data sources `VariantDataSource`, `VariantListDataSource` and `VariantTableDataSources` can all be created dynamically or stored as persistent resources.

The data that they contain must (in the present release) be assigned at runtime. Data in these data sources cannot be persistently stored.

Internals for DA Data Sources

The DA class `RecordSetDataSource` is a subclass of the `VariantTableDataSource`, and inherits all of the methods and properties from that data source.

The `RecordSetDataSource` maintains a “contains a” relationship with the `RecordSet` that it was created from. This means that there is only one copy of the data. The views registered to a `RecordSetDataSource` are viewing the data that is in the `RecordSet` itself.

Note: A `RecordSet` is created and saved in an RC file. Since the data source needs to rely on a mechanism to derive its data, the `RecordSet` needs to be loaded manually and initialized in the database view or the Resource (RC file?) must be explicitly loaded.

RecordSetDataSource Implementation

The `RecordSetDataSource` inherits the `VariantTableDataSource` interfaces, but certain operations possible through this interface are not suitable for a `RecordSet`, such as setting row titles (there are no row titles in the `RecordSet`).

The tables below describe the properties and methods from the `VariantTable` as applied to the `RecordSetDataSource`. Properties or methods not listed below are not implemented.

Properties from VariantTable

The `VariantTable` class provides some standard operations for handling modifications to the `RecordSet` through its properties. If you change the property of a `datasource`, depending upon the options you set, you will change the data contained or represented in the `RecordSet` and views. The final data storage mechanism (database, flat file) is not changed until it is explicitly updated.

The following `VariantTableDataSource` properties can be used to perform operations on the `RecordSetDataSource`.

Use this property...	To...
<code>RowCount</code>	Return the number of records in the <code>RecordSet</code> .
<code>ColumnCount</code>	Return the number of columns in the <code>RecordSet</code>
<code>ColumnTitle</code>	Return the name of the column in the <code>RecordSet</code>
<code>CursorRow</code>	Perform either a Get or Set <code>CursorRow</code> operation: Use a <code>CursorRow Set</code> to set the current record position in the <code>RecordSet</code> . Use a <code>CursorRow Get</code> to return the current record position in the <code>RecordSet</code> .
<code>Cells</code>	Perform a Get or a Set:

Use this property...	To...
	Use a Cell Get to retrieve the value from the RecordSet for the specified row (record) and column.
	Use a Cell Set to set the value into the RecordSet for the specified row (record) and column.

The following VariantTableDataSource properties can be used to perform operations on the DatabaseViewDataSource.

Use this property...	To...
RowCount	Return the number of records in the DatabaseView.
ColumnCount	Return the number of columns in the DatabaseView.
ColumnName	Return the name of the column in the DatabaseView
CursorRow	Perform either a Get or Set CursorRow operation: Use a CursorRow Set to set the current record position in the DatabaseView. Use a CursorRow Get to return the current record position in the DatabaseView.
Cells	Perform a Get or a Set: Use a Cell Get to retrieve the value from the DatabaseView for the specified row (record) and column. Use a Cell Set to set the value into the DatabaseView for the specified row (record) and column.

For more information about the VariantTable class, refer to Chapter 2, “Core Reference,” of the *OOScript Language Reference* manual.

Methods from VariantTable

The VariantTable class provides some standard operations for handling modifications to the RecordSet. The following VariantTable methods can be used to perform operations on the RecordSetDataSource.

Use this Method...	To...
AddColumn	Not implemented.
AddRow	Add a record to the RecordSet at the specified index.
RowColumnCount	Not implemented.
RemoveRow	Not implemented.
RemoveColumn	Not implemented.

Using the RecordSet contained in a RecordSetDataSource

The RecordSet that is contained in a RecordSetDataSource needs to be used to update the database when necessary. You can invoke the following methods on a RecordSetDataSource to update a RecordSet.

Use this Method...	To...
AddRecord	Add a row to the end of the RecordSetDataSource.
RemoveNthRecord	Delete a row from the RecordSetDataSource.

When to Use RecordSet Data

In general, once you have created a `RecordSetDataSource` from a `RecordSet`, you should avoid updating the data in the `RecordSet` using its own interface. Only the basic operations of adding and removing rows on the `RecordSet` will be reflected in the `RecordSetDataSource`. Full control over updating data and positioning the current record in a `RecordSet` is provided through the `RecordSetDataSource` interface.

You should use the data in the `RecordSet`, when you need to update the backend database with the values that have been updated in the `RecordSet`. It is then more convenient to extract the data from the `RecordSet` using its own interface for operations like parameter binding.

Internals for IRE Data Sources

In the same way that you create data sources from Core or DA data, you can create data sources from IR data. The update is done automatically (deferred is not currently supported).

Note: User input is forwarded to the inference engine only after a `Continue/Start`. In the case of the table: slot value changes or objects added or deleted from a class are reflected automatically in the data source.

IRE supports two types of data sources: atomic data sources which are instances of the class `NxDataSource`, and table data sources which are instances of the class `NxTableDataSource`. The `NxDataSources` are used to display values of IRE slots into Text Edits or into button states, while `NxTableDataSources` are used to display the slot values of objects of a specific class or sub-objects of a specific object.

You should use the IRE data sources whenever you want to either present multiple views of slots values or whenever you want to have the changes of the slot values dynamically updated on the screen while the IRE rule processor is running. Whether the user changes the values or the IRE rule processor, the slots are updated immediately (no deferred) and all views synchronized if the correct options have been set on the views (see the view option `CURSOR`). You just need to create a data source, set its IOE properties, associate it with one or more views and eventually set the view options.

Creating IRE data sources are a little bit different from the Data Access data sources. IRE Data sources have additional properties to be set at creation:

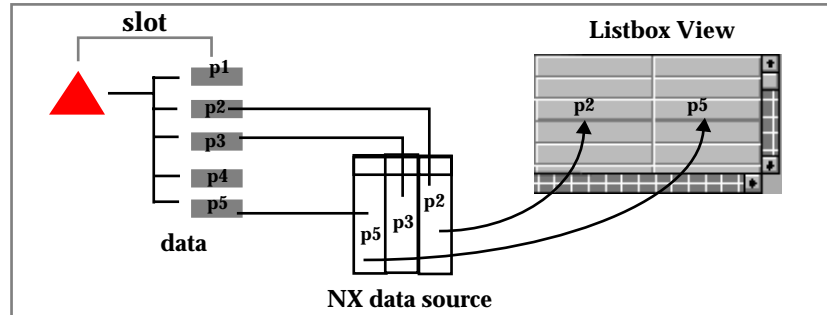
- **Atom**

In the case of an atomic data source, the `Atom` property should be set to the IRE slot whose value will be displayed or edited. In the case of a table data source, the `Atom` property should be set to either the IRE class whose direct children objects (and indirect objects through subclasses) will be displayed, or the IRE object whose direct children objects will be displayed.

- **ColumnProperty(Index)**– only for `NxTableDataSources`

This property defines the mapping between the data sources columns and the IRE properties of the `Atom`. The mapping isn't always on a one-to-one basis. The column of data (derived from a data source) shown in

a view might have less IRE slot properties than the original object property actually contains. The ViewOption property of the Data Source controls what data is displayed. For each column (Index) of the data source, you should associate the IOE object reference of a IRE property of the IRE atom.



You can derive a small subset of original data at the data source layer and use an even smaller subset of that for your view.

Figure 2-1 Mapping of data, data source, and view

Note: You should first define the dimensions of your data source tables prior setting the `ColumnProperty()` on a data source. This can be done by using the method `RowColCount(x,y)` where `x` is the number of rows (0 for infinite), and `y` the number of columns (0 for infinite).

■ Strategy

This property defines which strategy will be applied when volunteering the value back to the IRE slots after the user updated the value from the views. If the Strategy property of the data source is not set, the IRE data source will update the IRE slots using the strategy defined in the `DefaultVolunteerStrategy` property of the Engine meta-class. Refer to the IRE IOE server Reference for the list of potential values for this property.

Currently the IRE data sources are supporting only the following views: Text Edits and buttons for single data sources, ListBoxes for table data sources.

Note: There is no special behavior for choice boxes. You need to get the value of slots and stored them in choice items.

Example

The following example links an IRE class Tanks with three IRE properties Name, Level, HasProblem:

Linking IRE class with IRE properties

```
ds := nxsvr.NxTableDataSources.Create();
ds.Atom = nxsvr.Classes.Tanks;
ds.RowColCount (0,3);
ds.ColumnProperty(0)= nxsvr.Properties.Name;
ds.ColumnProperty(1)= nxsvr.Properties.Level;
```

```
ds.ColumnProperty(2)= nxsvr.Properties.Hasproblem;
ds.RegisterView(myLBox);
```

The following VariantTable properties and methods, when applied to the NxTableDataSource, perform the specified changes. Properties or methods not listed below are not supported.

Use this method or property...	To...
RowCount	Return the number of records in the NxTableDataSource.
ColumnCount	Return the number of columns in the NxTableDataSource.
ColumnName	Return the name of the column in the NxTableDataSource.
CursorRow	Perform a Get or Set operation: Get: Returns current record position in the NxTableDataSource. Set: Sets the current record position in the NxTableDataSource.
Cells	Perform a Get or Set operation: Get: Retrieves the value from the NxTableDataSource for the specified row (IRE object) and column. Set: Sets the value into the NxTableDataSource for the specified row (IRE object) and column.

Note: RowColumnCount, AddRow, AddColumn, RemoveColumn and RemoveRow are not allowed operations on NxTableDataSources. You need to directly use the methods Delete/CreateObject on the IRE class/object.

If the IRE Rule processor adds or removes objects from the class or object the data source is based on the view will be updated accordingly.

As you design an application usually you have two types of tables:

- An input table where the user can edit the values.
- A selection table where the user can select a current row.

Input Table (LBox)

In the case of an input table, the data source transparently handles the update of the back-end data (IRE slots) and the updates of the other views registered to this data source. The IRE data sources uses the strategy set on the data source or the Default Volunteer Strategy set on the Engine Object to volunteer back the value to the IRE slot, when the cell edition is done.

Listbox views are by default input table if a Text Edit has been attached for edition (Refer to the LBox editor section of the Open Interface User's Guide). Non-editable columns can be defined through the view option "noeditcolumn" and specifying the range of non-editable columns. If you do set the headers on the listbox the data source will display automatically the name of the IRE object for each listbox row, and the name of the property for each listbox column. The title of these headers can be changed by setting the Title property of the data source columns and/or rows.

To start the edition, you should use the following keys:

- double click (cell edition)
- CTR+e (cell edition)

- CTR+m (continued edition)
- ESC (abort edition)

Note: You still need to attach a Text Edit to the listbox to set the View in an edit mode

The following example shows an input table with one column that is non-editable:

Input table with non-editable column

```
on event WGTSSINITIALIZED
    tanks := rulesvr.Classes.tanks;
    rProps := rulesvr.Properties;
    // Use a table data source to link the listbox to the
    // class Tanks
    ds := rulesvr.NxTableDataSources.Create();
    ds.RowCount(0,3); // set the size of the data
                       // source

    ds.Atom = tanks;
    ds.RegisterView( SELF);
    // set the column mapping with field and column labels
    ds.ColumnProperty(0) = rProps.Name;
    ds.Columns(0).Title = "Tanks";
    ds.ColumnProperty(1) = rProps.level;
    ds.Columns(1).Title = "Level";
    ds.ColumnProperty(2) = rProps.problem;
    ds.Columns(2).Title = "Has Problem";
    // set the view non editable for columns 0 and 2
    SELF.ViewOptions.UneditableColumns = "[0...0][2...2]";
end event
```

Selection table (ListBox)

In the case of a selection table, the data source transparently handles the selection but you should trap the CELLSELECTED event of the view or set a callback for CellSelectedProc, to process the selection update. If the cursor property of the data source has been set to “controls”, the data source just sets the current row to the row selected by the user. And you can get the current row index by looking up the property CursorRow of the data source, while the property CursorColumn indicates the current column of the selection.

The current cell contents (text) can be access through the property cells as follows:

```
currentCellContents = string
(SELF.Data.Cells(SELF.Data.CursorRow,
dsEmp.CursorColumn))
```

Note: From the view you can access the data source by using the property Data of the Listbox object.

Listbox views are selection table if you set the Listbox selection flag (in particular, Single vs. multiple selections) on the Listbox. (Refer to the LBox editor section of the Open Interface User’s Guide). If you do set the headers on the listbox the data source will display automatically the name of the IRE

object for each listbox row, and the name of the property for each listbox column. The title of these headers can be changed by setting the RowTitle and ColumnTitle properties of the data source.

The following example registers a listbox to a data source at its initialization. This is a selection table which sets the contents of another data source based on the information contained in the current cell (1 column listbox).

Registering a listbox to a data source

```

on event WGT_INITIALIZED
    dsEmp := nxsvr.NxTableDataSources.Create();
    dsEmp.Atom = nxsvr.Classes.employees;
    dsEmp.RowColumnCount(0,1);
    dsEmp.ColumnProperty(0) = nxsvr.Properties.name;
    dsEmp.RegisterView(SELF);
    SELF.ViewOptions.CursorOption = "controls";
end event

on event LBOX_CELLSELECTED
    dsEmp := SELF.Data;
    theEmployee = string(dsEmp.Cells(dsEmp.CursorRow, 0));
    if (theEmployee == "Unknown") // verify whether the
        // Employee is a valid IRE object
        return;
    if (isnull(dsEmp2)) // verify whether the other data
        // source has been created
        return;
    dsEmp2.Atom := nxsvr.Objects.$theEmployee;
    dsEmp2.RowColumnCount(0,2);
    dsEmp2.ColumnProperty(0) = nxsvr.Properties.nature;
    dsEmp2.ColumnProperty(1) = nxsvr.Properties.amount;
end event

```

In this particular example, the "\$" is used to force the evaluation of the variable theEmployee prior the resolution of the object expression. DsEmp2 is in fact set to the IRE object whose name is value of theEmployee.

IRE Text Edit

The following example links a Text Edit with a IRE slot with automatic and implicit controlled edition.

Linking a Text Edit with an IRE slot

```

// Assume that a Text Edit object reference is in the ted
// variable nxr contains the Nx server
ds := nxsvr.NxTableDataSource.Create();
ds.Atom := nxsvr.Object.Car.Color; // assuming that Car.Color
// ia a valid slot in IRE
ds.Strategy = nxsvr.Engine.VSTRAT_VFWRD;
ds.RegisterView(ted);
// the rest (local update, forwarding the data,...) is handled
// automatically by the NEXPERT data source

```

Tree Datasource: Managing Hierarchical Data

A *tree datasource* is a container of hierarchically organized nodes. The tree datasource is similar to the other datasources—for example, list (sequential) and table (tabular) datasources—in that it is based on a specific data model. In this case, the data model is a **hierarchy**.

You can display the contents of the tree datasource in the **NDTView** and **NDBrows** views, which are supplied by the Open Interface Element. The Elements Environment *datasource/views* mechanism supports the interface between the datasource and the **NDTView** and **NDBrows** views.

This chapter discusses these topics:

- Concepts
- Options for the NDTView and NDBrows Views
- Building a Tree Datasource
- Editing a Tree Datasource
- Advanced Topics

Note: Data stored in the tree datasource is **not** persistent. However, you can write a routine to traverse the datasource and write its contents to a persistent data-storage medium, such as a local hard disk or database.

If you haven't already done so, read the chapters on the **NDTView** and **NDBrows** widgets in the *Open Interface Element C++ Programmer's Guide*. See Chapter 3 of this book for information about registering a view with a datasource.

Concepts

Storing information based on a hierarchical data model, the tree datasource is founded on these basic concepts:

- Tree Datasource
- Node
- Tree
- Node Accessor
- Cursor
- Edit Object

This section discusses the preceding concepts, which are then used in “Building a Tree Datasource” on page 30 to tell you how to program a tree datasource.

Tree Datasource

The *tree datasource*—an object of the **NDVarTr** class—is a container class that stores and manages hierarchically arranged nodes. When you dispose the tree datasource, any contained objects are also disposed.

Using the APIs supplied with the tree datasource (**NDVarTr** object), you can:

- Program the creation and destruction of nodes contained by the datasource object
- Enumerate the nodes in the datasource by index and traverse them using the methods in the **NDVarTr** API

Node

A *node* is the elementary component of a tree. Each node has these properties:

- ID and Value
- Navigational References

As Figure 3-1 shows, each node stores references to:

- Its parent node
- The next sibling or root node
- The previous sibling or root node
- Its first child node

If any of these references accesses a memory location where no node exists, then the reference indicates that the current node is the last valid node. For example, if the Parent reference accesses an empty node location, the node is a *root node*, which has no parent.

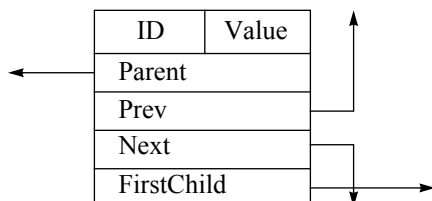


Figure 3-1 The Structure of a Node

ID and Value

Each node in the tree datasource has an **ID** property and a **Value** property. Both the **ID** and **Value** properties:

- Store variant data
- Can contain any variant-supported type

For example, **ID** may be a variant containing a string, while **Value** may be an object reference.

ID

You can assign any variant data to a node **ID** property. Node **IDs** need not be unique, but they may be more useful if they are. You can set the **ID**:

- When you create a node
- During a separate editing session

A unique node **ID** can be very helpful. This is especially true if you need to associate it with the primary key of a relational-database table. For example, if a node represents an employee in an organization, you may want to:

- Set **ID** to an employee number
- Set **Value** to the employee name
- Then associate the node with a row from a table datasource that shares a common employee number

Value

Like the node **ID** property, you can set the node **Value** to any variant value. The **Value** property represents the “data” part of the node contents. You can use the node **Value** any way you want. For example, you may simply set it to an employee name in an organizational hierarchy, or you may set it to an employee number that acts as a key to display employee data stored in a row of a table datasource.

Navigational References

Each node supports API tree traversal through these mechanisms:

- Parent Reference
- Next and Previous Sibling References
- First Child Reference

These references provide access to the corresponding nodes relative to the currently accessed nodes. For more information about accessing nodes, see “Node Accessor” on page 21.

Parent Reference

The *parent* reference, which Figure 3–1 shows as “Parent,” provides access to the parent node. If the current node is a root node, the parent reference is meaningless.

Next and Previous Sibling References

The *sibling* references, which Figure 3–1 shows as “Next” and “Prev,” provides access to the next and previous sibling nodes, respectively. If the current node is a root node, these references provide access to the next and previous root nodes, respectively.

First Child Reference

The *first child* reference, which Figure 3–1 shows as “FirstChild,” provides access to the first child node. After accessing the first child node, you can use the first child reference again to descend deeper into the hierarchy. Alternately, you can use the sibling references to access the siblings of the first child node.

Tree

A *tree* is a hierarchical node network that emanates from a single root node. A tree datasource may store one or more trees. Each tree has exactly one root node. Therefore, the tree datasource can contain only as many trees as it does root nodes.

The notion of a *subtree* is also supported to a limited extent. A subtree may be based on any node in the datasource. While you can remove an entire subtree, there is no API support for “relocating” a subtree to a new position in the datasource. In other words, you cannot use the API to assign a subtree to a new parent node.

These concepts are instrumental in the description of trees and tree navigation:

- Root Node
- Parent-Child Node Relationship
- Sibling Node Relationship

Root Node

A *root node* is a node that has no parent node, but can have child nodes. This is the topmost node in a tree hierarchy. It is always the first node created after the tree datasource is created.

Root nodes each have one unique feature that differentiates them from non-root nodes: they have no parent node. As Figure 3–2 shows, the Parent reference of a root node accesses an empty node location:

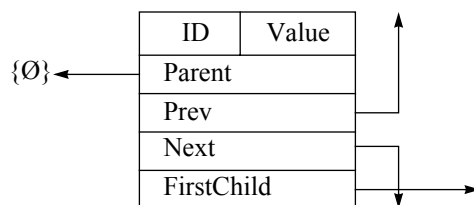


Figure 3–2 Unique Characteristics of a Root Node

Relative to a root node, you can position a *node accessor*. With a *node accessor*, you can add child nodes and other root nodes (from which you can build other trees) to the tree datasource. Like the maximum number of child nodes, the number of root nodes is limited by the size of an **Int16** datatype on each platform.

Parent-Child Node Relationship

The parent-child relationship is a convenient way to explain the relationships in the tree datasource. Figure 3–3 shows how node references establish the relationships between *parent nodes* and their *child nodes*:

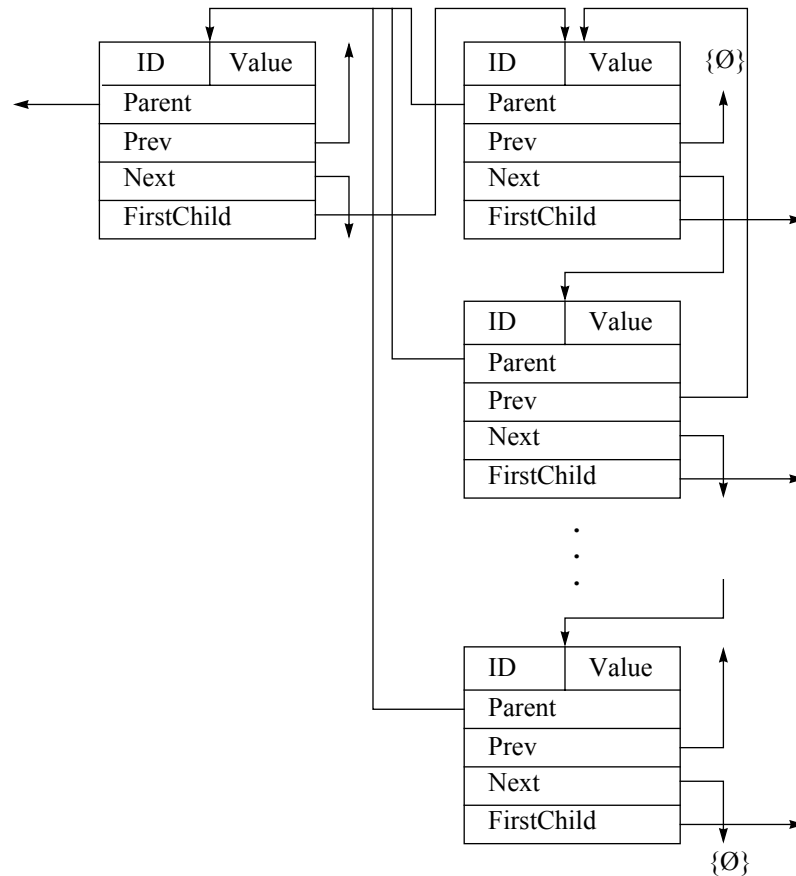


Figure 3–3 Parent-Child Node Relationship

Any node can have child nodes. Any tree in the tree datasource can expand to the full extent of the memory available in the executing system. For any parent node, the number of child nodes that can be indexed by the tree datasource is limited to the size of an **Int32** datatype. For example, if you are using 16-bit integers, a parent node can have no more than 2^{16} child nodes.

Sibling Node Relationship

In addition to child nodes, each node can have *sibling* nodes. In Figure 3–3, the Prev reference of the first child accesses an empty node location; there is no “previous sibling.” Likewise, the Next reference of the last sibling node accesses an empty node location; there is no “next sibling.”

Node Accessor

A *node accessor* is a node indexing mechanism that references and traverses the nodes in the tree datasource. You cannot access the nodes directly,

therefore you **must** use a node accessor to access them. You must also use accessors to identify the node in a node-level edit operation.

You need at least one node accessor to traverse—using the **NDVarTrNodeAccessor** API—the nodes in a tree datasource. After moving the node accessor to the appropriate node in the hierarchy, your application can modify either the datasource structure or the node properties.

This code fragment shows how to create and destroy a node accessor:

```
// Declare a tree-datasource pointer variable, and assign a
// tree-datasource object to it.
VarTrPtr mTreeDs = new NDVarTr;

// Declare a node-accessor pointer variable, and assign a
// node-accessor object to it.
VarTrNodeAccessorPtr mNodeAccessor = new NDVarTrNodeAccessor;
...
// Destroy the node accessor.
delete mNodeAccessor;
...
// Destroy the tree datasource.
delete mTreeDs;
```

Using the APIs, you can create edit objects to support either node-level or datasource-level modifications using the node accessors. For more information about node accessors, see “Adding Nodes” on page 33 and “Destroying a Node-Accessor Object” on page 43.

With a node accessor, you can traverse the node hierarchy using functions in the **NDVarTrNodeAccessor** API. With these functions, you can move the accessor relative to its current node location. You can also move it directly to a specific location using the indexing scheme shown in Figure 3-4:

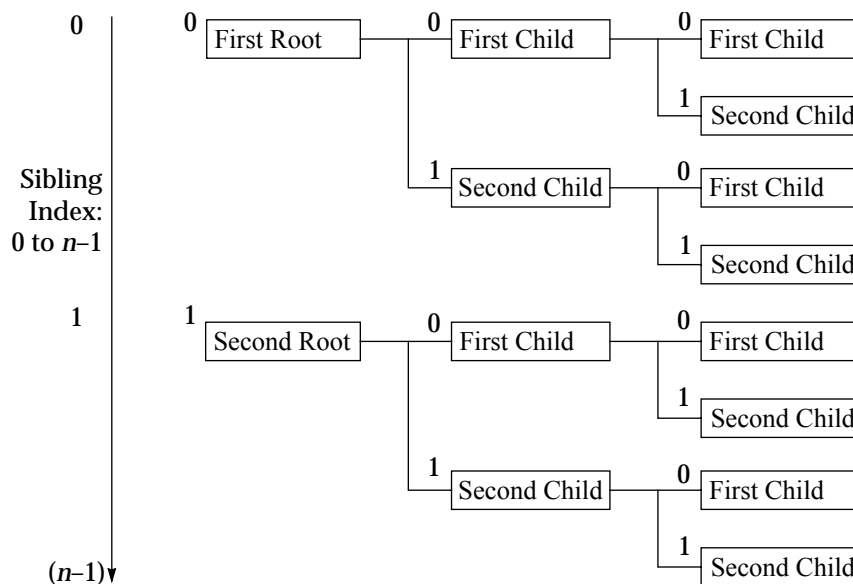


Figure 3-4 Node Indexing in the Tree Datasource

The sibling index in Figure 3-4 ranges from 0 to $n-1$, where n is a one-based counter that represents the number of sibling nodes at a particular level of the hierarchy. The origin of a tree is the root node. The sibling index of the

first root node is 0. The index of the last root node is the number of root nodes minus one ($n-1$).

The zero-based sibling index is useful when moving the node accessor directly to the n th root, child, or sibling node (see “Adding Nodes” on page 33 for examples of how to use the API). The following functions return one-based counters:

- **GetNumRoots()**
- **GetNumChildren()**
- **GetNumSiblings()**

These functions work well with the **GoNthRoot()**, **GoNthChild()**, and **GoNthSibling()** functions to position the node accessor on the next empty node location. These are further described in “Node-Count Functions” on page 52.

The sibling index applies to root nodes, too, even though they do not share a common parent node. Each tier of the hierarchy uses the same index scheme. Using the sibling index, combined with the depth of the node in the hierarchy, a composite index of this form uniquely identifies each node in the datasource:

```
(<root index>, <child index>, ..., <tier <n> index>)
```

where the number of indices in the composite index equals the tier number, n , of the node being represented. The order of the sibling indices in the composite index is from most significant to least significant, or from the root level downward.

For example, using the preceding notation in Figure 3–4, the node, “First Root”->“Second Child”->“First Child,” has a composite index of (0,1,0).

Cursor

The tree datasource supports a *node cursor*, which is a property of the tree datasource. Like the **Title** property, you can set and get the cursor. You can:

- Set the cursor by associating it with a node accessor using the **NDVarTr::SetCursor(*accessor*)** function
- Then access the node at the current cursor location using the **NDVarTr::GetCursor()** function

When a **NDTView** or **NDBrows** view is registered with a tree datasource, you can set a view option to either control the datasource cursor through the view or simply reflect the current location of the datasource cursor as it traverses the internal hierarchy.

This code fragment shows how to set and get a cursor:

```
// Declare a tree-datasource pointer variable, and assign a
// tree-datasource object to it.
VarTrPtr mTreeDs = new NDVarTr;

// Declare a browser pointer variable, and assign a browser
// object to it.
BrowsPtr mBrowsWgt = new NDBrows;

// Register the browser with the tree datasource, and set the
// "cursor" view option.
mTreeDs->RegisterView(mBrowsWgt);
mTreeDs->SetViewOption(mBrowsWgt, "cursor", "CONTROLS");
```

```

// Declare a node-accessor pointer variable, and assign a
// node-accessor object to it.
VarTrNodeAccessorPtr mNodeAccessor = new NDVarTrNodeAccessor;

// Declare two variant pointer variables, and assign variant
// objects to them.
VarPtr mVarID = new NDVar;
VarPtr mVarValue = new NDVar;

// Set a cursor at the location of the node accessor.
mTreeDs->SetCursor(mNodeAccessor);

// Position the node accessor.
...
// Use "convenience" API functions to edit the ID and Value
// properties of the node at the current cursor location.
mVarID->SetStr("0000");
mVarValue->SetStr("Node");
mTreeDs->SetNodeID(mTreeDs->GetCursor(), mVarID);
mTreeDs->SetNodeValue(mTreeDs->GetCursor(), mVarValue);
...
// Destroy the variant objects.
delete mVarID;
delete mVarValue;

// Destroy the node accessor.
delete mNodeAccessor;

// Destroy the tree datasource.
delete mTreeDs;

```

“Destroying a Node-Accessor Object” on page 43 shows an alternative use of the `GetCursor()` function. For information about setting the cursor behavior, see “Options for the NDTVView and NDBrows Views” on page 28.

Edit Object

To perform edit operations on the tree datasource or the nodes it contains, your application must use an *edit object*. The tree datasource uses edit objects to:

- Create working copies of the data
- Protect the datasource from corruption resulting from simultaneous editing sessions sharing a common datasource

The tree datasource supports these editing levels:

- Datasource Editing
- Node Editing

If the data to be modified is locked by another view, no edit object can be created. This locks your application out of the data. To prevent your application from hanging when it encounters a data lock, you can create your edit object within a conditional construct that checks for the availability of the data and supplies an alternative if the data is locked.

Editing the datasource includes the following four steps:

1. Create an edit object
2. Execute the edit operations
3. Commit the edit operations
4. Destroy the edit object

In addition to the direct approach to managing edit objects, a set of “convenience” APIs supplies functions that manage the edit objects

automatically for single edit operations. For more information about the “convenience” APIs, see “Convenience API Functions” on page 26.

Datasource Editing

When you want to modify the structure of the datasource—for example, to create new nodes—your application needs a *datasource edit object*. When a datasource edit object is created to support an editing operation for one view, no other view can create an edit object for that datasource. This includes node edit objects for editing node data, because the node you may want to edit may also be edited during the datasource-level editing session.

The datasource edit object is created, locking the datasource, when an object of the **NDVarTr** class executes the **StartEdit()** function. This is a public function inherited from the **NDDs** class. The tree datasource is unlocked when the **NDDsEdit::End()** function executes, as shown in this example:

```
// Declare pointer variables and assign objects to them.
VarTrPtr mTreeDs = new NDVarTr;
VarTrNodeAccessorPtr mNodeAccessor = new NDVarTrNodeAccessor;

// Declare a datasource edit pointer and assign an edit object
// to it.
VarTrEditPtr mEditTreeDs = mTreeDs->StartEdit();

// Position the node accessor and edit the tree.
mNodeAccessor->GoNthRoot(mTreeDs->GetNumRootNodes());
mEditTreeDs->AddNode(mNodeAccessor);
...
mNodeAccessor->GoFirstRoot();
mEditTreeDs->RemoveNode(mNodeAccessor);

// Commit the edit operations and destroy the edit object.
mEditTreeDs->End();

// Destroy other objects.
delete mNodeAccessor;
delete mTreeDs;
```

When the **DsEdit::End()** function executes, all tree modifications are committed, and the datasource-level lock is released.

Node Editing

When you want to edit the data properties of a node in a tree datasource—for example, to change the node **Value**—your application needs only a *node edit object*, not a datasource edit object. Instead of locking the entire datasource from access by other views, you only need to lock the node you want to modify.

A node edit object is created, locking the accessed node, when an object of the **NDVarTr** class executes the **StartNodeEdit()** function. This is a public function inherited from the **NDDs** class. All edit operations are committed, the edit object is destroyed, and the accessed node is unlocked when the **DsEdit::End()** function executes, as this example shows:

```
// Declare pointer variables and assign objects to them.
VarTrPtr mTreeDs = new NDVarTr;
VarTrNodeAccessorPtr mNodeAccessor = new NDVarTrNodeAccessor;
VarPtr mVarID = new NDVar;
VarPtr mVarValue = new NDVar;

// Position the node accessor and edit nodes.
mNodeAccessor->GoFirstRoot();
while (mVarTr->IsValid(mNodeAccessor)) {
```

```

        mNodeAccessor->GoNext();
    }

    // Datasource-level edit object created, edits committed, and
    // edit object destroyed by the AddNode() function. See
    // "Convenience API Functions" for more information.
    mTreeDs->AddNode(mNodeAccessor);

    // Declare a node-level edit pointer and create an edit object.
    VarTrNodeEditPtr mEditNode =
        mTreeDs->StartNodeEdit(mNodeAccessor);

    // StartNodeEdit returns NULL if the node accessor is not on a
    // valid node. The following conditional ensures that the edit
    // operations are not attempted if the edit object was not
    // created.
    if (mEditNode != NULL) {
        mVarID->SetStr("0000");
        mVarValue->SetStr("New Node");
        mEditNode->SetID(mVarID);
        mEditNode->SetValue(mVarValue);
        ...
    }
    // Commit the edit operations and dispose of the node-level edit
    // object.
    mEditNode->End();
} // End if.
...
// Dispose of other objects.
delete mVarID;
delete mVarValue;
delete mNodeAccessor;
delete mTreeDs;

```

The code in the preceding example creates a node edit object after the `NDVarTr::AddNode()` function, because the “convenience” API functions create their own edit objects. The `AddNode()` function would fail to execute if a node edit object was created before it. As with datasource edit objects, all node modifications are committed and the node-level lock is released when the `NDDsEdit::End()` function executes.

Convenience API Functions

When editing a tree datasource, you can use either the standard APIs or the convenience APIs to complete the edit operations. When using the standard APIs, you must:

1. Create an edit object to start the edit operation
2. Perform any necessary editions to the datasource
3. Commit the edit operation
4. Destroy the edit object

When using the “convenience” APIs, steps 1, 3, and 4 from the preceding list are completed automatically. You can perform both:

- Node Editing with the “Convenience” APIs
- Datasource Editing with the “Convenience” APIs

In other words, your application can use the “convenience” API to edit the datasource or its contents without formally creating an edit object. For example, when the `NDVarTr::AddNode(accessor)` function executes:

- An edit object is automatically created
- The new node is added at the location specified by the node accessor
- The edit operations are committed

■ The edit object is destroyed

The “convenience” API functions are useful for performing single edit operations. However, these functions can inhibit performance when used to perform batch edit operations.

Datasource Editing with the “Convenience” APIs

If you want to change the ID and Value properties of a specific node in the datasource, the “convenience” API functions are most useful. To directly manage the required edit object would add some unnecessary complexity to your application logic.

In the next example, a datasource edit object is create by the “convenience” function, **NDVarTr::AddNode()**. This creates a datasource edit object, adds a node, commits the node addition to the datasource, and destroys the edit object.

```
// Declare pointer variables and assign objects to them.
VarTrPtr mTreeDs = new NDVarTr;
VarTrNodeAccessorPtr mNodeAccessor = new NDVarTrNodeAccessor;

// Move the node accessor to the next empty root-node location.
mNodeAccessor->GoFirstRoot();
while (mVarTr->IsNodeValid(mNodeAccessor)) {
    mNodeAccessor->GoNext();
}

// Add a node using the "convenience" API. A datasource edit
// object is created, edit operations are committed, and the
// edit object is destroyed by the mTreeDs->AddNode() function.
mTreeDs->AddNode(mNodeAccessor);
...
// Dispose of other objects.
delete mVarID;
delete mVarValue;
delete mNodeAccessor;
delete mTreeDs;
```

If the preceding code fragment was intended to build a hierarchy of nodes—for example, a datasource with 10 root nodes, each with 10 children, and so on—the “convenience” API functions would not be appropriate. For such operations, use batched edit operations as described in “Datasource Editing” on page 25.

Node Editing with the “Convenience” APIs

If you want to change the ID and Value properties of a specific node in the datasource, the “convenience” API functions are most useful. To directly manage the required edit object would add some unnecessary complexity to your application logic.

In the next example, a node edit object is create by each of the convenience API functions, **NDVarTr::SetNodeID()** and **NDVarTr::SetNodeValue()**. Each of these functions creates a node edit object, commits its edit operation, and destroys the edit object.

```
// Declare pointer variables and assign objects to them.
VarTrPtr mTreeDs = new NDVarTr;
VarTrNodeAccessorPtr mNodeAccessor = new NDVarTrNodeAccessor;
VarPtr mVarID = new NDVar;
VarPtr mVarValue = new NDVar;

// Move the node accessor to the next empty root-node location.
mNodeAccessor->GoFirstRoot();
```

```

while (mVarTr->IsValid(mNodeAccessor)) {
    mNodeAccessor->GoNext();
}

// Add a node using the "convenience" API. A datasource edit
// object is created, edit operations are committed, and the
// edit object is destroyed by the mTreeDs->AddNode() function.
mTreeDs->AddNode(mNodeAccessor);

// Set the variant objects to some initializing values.
mVarID->SetStr("0000");
mVarValue->SetStr("New Node");

// Set the node ID and Value properties using the "convenience"
// APIs. A node edit object is created, edit operations are
// committed, and the edit objects are destroyed by each of the
// following two functions.
mTreeDs->SetNodeID(mNodeAccessor, mVarID);
mTreeDs->SetNodeValue(mNodeAccessor, mVarValue);
...
// Dispose of other objects.
delete mVarID;
delete mVarValue;
delete mNodeAccessor;
delete mTreeDs;

```

If the preceding code fragment was intended to traverse and initialize each node in the hierarchy, the “convenience” API functions would not be appropriate. For such operations, use batched edit operations as described in “Node Editing” on page 25.

Options for the NDTView and NDBrows Views

The tree datasource supports these view options for the **NDTView** and **NDBrows** views:

- **cursor**
- **initexpandlevel**
- **autosize** (**NDBrows** view, only)

To set view options, use the second and third parameters of the **NDDs::SetViewOption()** function, as shown here:

```

<datasource>->SetViewOption((ResPtr)<view>,
    [{"cursor", "{CONTROLS|REFLECTS}"}]
    [{"initexpansionlevel", "{0..<n>}"}]
    [{"autosize", "{FALSE|TRUE}"}]
    );

```

cursor

The **cursor** view option determines whether the view cursor controls or reflects the position of the datasource cursor. The **cursor** view option has two possible settings: **CONTROLS** (the default) and **REFLECTS**. Here is the format for the setting the “cursor” option:

```

<datasource>->SetViewOption((ResPtr)<view>,
    {"cursor", "{CONTROLS|REFLECTS}"});

```

With **cursor** set to **CONTROLS**, the cursor position (active node) in the view determines the position of the datasource cursor. This ensures that the datasource cursor and view cursor are synchronized.

With **cursor** set to **REFLECTS**, the view cursor reflects the current location of the datasource cursor. This setting ensures that the view is continually updated when the node accessor is moved programmatically.

When multiple views are registered with a common datasource, each registered view can manipulate the position of the datasource cursor if **cursor** is set to **CONTROLS**. For example, if two views control the position of the datasource cursor, moving one view cursor changes the position of the datasource cursor, which the other registered view reflects.

In this example, the mTreeViewWgt cursor reflects the current location of the datasource cursor, while mBrowsWgt controls the position of the datasource cursor:

```
VarTrPtr mTreeDs;
BrowsPtr mBrowsWgt;
TViewPtr mTreeViewWgt;
...
mTreeDs->RegisterView((ResPtr)mBrowsWgt)
mTreeDs->RegisterView((ResPtr)mTreeViewWgt)
...
mTreeDs->SetViewOption((ResPtr)mTreeViewWgt,
                      "cursor", "REFLECTS");
mTreeDs->SetViewOption((ResPtr)mBrowsWgt,
                      "cursor", "CONTROLS");
```

If both view cursors control the position of the datasource cursor, any change in the cursor position of one view is automatically reflected in the other view.

initexpandlevel

The **initexpandlevel** option sets the number of levels to which the root node expands in the display when the view is registered with a tree datasource. Here is the syntax for using the **initexpandlevel** option:

```
<datasource>->SetViewOption((ResPtr)<view>,
                            {"initexpandlevel", "{0..<n>}");
```

where *n* is the number of levels of expansion from the root node in the mTreeDs hierarchy. The default expansion level depends on the type of view to which it applies. By default, **NDBrows** views are fully expanded (*n* expansion levels displayed), while **NDTView** views are collapsed to root nodes only (zero expansion levels displayed).

Warning: If two views sharing a common tree datasource have initial expansion levels that differ, the displayed views may also differ, depending on the setting of the **cursor** option.

In the following code fragment, assume the tree datasource, mTreeDs, has six expansion levels. Widgets mTreeViewWgt and mBrowsWgt share mTreeDs as a common datasource with mBrowsWgt controlling the datasource cursor and mTreeViewWgt reflecting it. However, if the mBrowsWgt cursor is placed on a level 4 node, the mTreeViewWgt cursor is unable to reflect its position in the display, because it is initially expanded only to two expansion levels.

```
VarTrPtr mTreeDs;
BrowsPtr mBrowsWgt;
TViewPtr mTreeViewWgt;
...
mTreeDs->RegisterView((ResPtr)mBrowsWgt)
mTreeDs->RegisterView((ResPtr)mTreeViewWgt)
...
```

```

mTreeDs->SetViewOption((ResPtr)mTViewWgt,
    "cursor", "REFLECTS");
mTreeDs->SetViewOption((ResPtr)mBrowsWgt,
    "cursor", "CONTROLS");
mTreeDs->SetViewOption((ResPtr)mTViewWgt,
    "initexpandlevel", "6");
mTreeDs->SetViewOption((ResPtr)mBrowsWgt,
    "initexpandlevel", "2");

```

In the next example, however, the `mTViewWgt` cursor can correctly reflect the position of the datasource cursor, because it is expanded to the same level as the `mBrowsWgt` widget, which controls the datasource cursor:

```

VarTrPtr mTreeDs;
BrowsPtr mBrowsWgt;
TViewPtr mTViewWgt;
...
mTreeDs->RegisterView((ResPtr)mBrowsWgt)
mTreeDs->RegisterView((ResPtr)mTViewWgt)
...
mTreeDs->SetViewOption((ResPtr)mTViewWgt,
    "cursor", "REFLECTS");
mTreeDs->SetViewOption((ResPtr)mBrowsWgt,
    "cursor", "CONTROLS");
mTreeDs->SetViewOption((ResPtr)mTViewWgt,
    "initexpandlevel", "6");
mTreeDs->SetViewOption((ResPtr)mBrowsWgt,
    "initexpandlevel", "6");

```

autosize

For **NDBrows** views only, you can set the **autosize** option to **TRUE** to create automatically sized nodes. With **autosize** enabled, all bounding boxes for the sibling nodes at a given expansion level have the maximum width for nodes at that level. Here is the syntax for using the **autosize** option:

```

<datasource>->SetViewOption((ResPtr)<view>,
    {"autosize", "{FALSE|TRUE}");

```

The default value for the **autosize** option is **FALSE**. With the default setting, the bounding-box widths are based on the string lengths of the node **Value** properties.

This code fragment shows how to enable the **autosize** option:

```

VarTrPtr mTreeDs;
BrowsPtr mBrows1;
mTreeDs->RegisterView((ResPtr)mBrows1)
...
mTreeDs->SetViewOption((ResPtr)mBrows1, "autosize", "TRUE");

```

Building a Tree Datasource

A *tree datasource* is a container of hierarchically arranged nodes. It can consist of one or more trees. Each node has variant **ID** and **Value** properties. These may be supplied when the node is created or during a separate editing session.

Building a tree datasource involves these tasks:

- Creating and Destroying a Tree Datasource
- Creating and Destroying a Node Accessor
- Creating and Destroying an Edit Object
- Adding Nodes

■ Managing Memory

The preceding list is somewhat simplified, but does explain the basic process, parts of which you may need to reiterate.

If you are constructing your datasource hierarchy interactively using the **NDBrows** and **NDTView** views—probably a more realistic approach—see “Editing a Tree Datasource” on page 44 for examples that show the programmatic aspect of the datasource/views relationship. For more information about options for the supported view widgets, see “Options for the NDTView and NDBrows Views” on page 28.

Creating and Destroying a Tree Datasource

Before you can begin creating trees in the tree datasource, your application must first create a tree datasource. This code fragment creates `mTreeDs` as a **VarTrPtr** variable and assigns an object of the **NDVarTr** object class to it:

```
// Create a tree-datasource object.
VarTrPtr mTreeDs = new NDVarTr;
...
// Destroy the tree-datasource object.
delete mTreeDs;
```

The preceding code fragment creates and destroys a tree datasource with the structure shown in Figure 3-5. The simple box in Figure 3-5 represents the memory location of the datasource object, `mTreeDs`.



Figure 3-5 Untitled Tree Datasource

The examples in the following sections build on this simple representation to construct a tree structure hierarchically from left (parents) to right (children). For more information about tree datasources, see “Tree Datasource” on page 18.

After the datasource object has served its purpose, use the **delete** operator to destroy it. For more information about memory management, see “Destroying the Tree-Datasource Object” on page 43.

Creating and Destroying a Node Accessor

A *node accessor* is an indexing mechanism that references the nodes in the hierarchy. With a node accessor, you can use the tree-datasource APIs to traverse the hierarchy. You need a node accessor in two instances:

- When you are simply updating information about a particular node
- When you are making structural changes—such as adding or removing nodes—to the tree datasource

This code fragment shows how to create and destroy a node accessor:

```
// Declare a tree-datasource pointer variable, and assign a
// tree-datasource object to it.
VarTrPtr mTreeDs = new NDVarTr;

// Declare a node-accessor pointer variable, and assign a
// node-accessor object to it.
VarTrNodeAccessorPtr mNodeAccessor = new NDVarTrNodeAccessor;
...
```

```
// Destroy the node accessor.
delete mNodeAccessor;
...
```

This creates *mNodeAccessor* as a **VarTrNodeAccessorPtr** variable and assigns an object of the **NDVarTrNodeAccessor** object class to it. When your application is finished with the node accessor, use the **delete** operator to free the memory allocated for it. For more information about memory management, see “Destroying a Node-Accessor Object” on page 43.

For more information about tree-datasource edit objects, see “Node Accessor” on page 21.

Creating and Destroying an Edit Object

To build a tree datasource, you have to modify its hierarchical structure. To do so, you need a datasource edit object. This code fragment creates and destroys a datasource edit object, *mEditTreeDs*:

```
VarTrPtr mTreeDs = new NDVarTr;

// Declare a datasource-edit pointer variable, and assign an
// edit object to it.
VarTrEditPtr mEditTreeDs = mTreeDs->StartEdit();

// Edit operations defined.
...
// Commit edit operations to the datasource and destroy the edit
// object.
mEditTreeDs->End();
...
```

In this example, *mEditTreeDs* is a **VarTrEditPtr** variable and is assigned a datasource edit object—the value returned by the **NDVarTr::StartEdit()** function. When the **NDDsEdit::End()** function executes, all editing operations are committed to the datasource, and the edit object is destroyed. For more information about tree-datasource edit objects, see “Datasource Editing” on page 25.

As a first use of the datasource edit object, assign a title to the datasource, as shown here:

```
VarTrPtr mTreeDs = new NDVarTr;
VarTrEditPtr mEditTreeDs;
...
// Create a datasource edit object.
mEditTreeDs = mTreeDs->StartEdit();

// Set the title of the tree datasource.
mEditTreeDs->SetTitle("Tree Datasource");

// Commit editing operations to the datasource and destroy the
// edit object.
mEditTreeDs->End();

// Destroy the datasource.
delete
```

Note the use of the string literal in quotation marks. Unlike the node **ID** and **Value** properties, the **NDVarTrEdit::SetTitle()** function accepts a string as the datasource title. Building on the example from Figure 3–5, executing the

preceding `NDVarTrEdit::SetTitle()` function adds a title to the tree datasource, as Figure 3-6 shows:

Figure 3-6 Titled Tree Datasource

Adding Nodes

After creating a node accessor, you use the `NDVarTrNodeAccessor` API to position the node accessor where you want to create a node. You must perform these tasks when:

- Creating the First Root Node
- Creating Child Nodes and Siblings
- Creating Additional Trees

Creating the First Root Node

To create the first root node, you need:

- A tree datasource
- A datasource edit object
- A node accessor

Building on the structure in Figure 3-6, the next task in creating your tree datasource is to add the first root node to it, which Figure 3-7 shows:

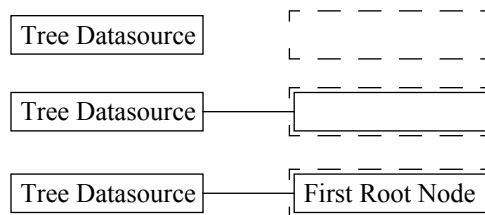


Figure 3-7 Creating the First Root Node in a Tree Datasource

These are the typical tasks in creating the first root node:

1. Moving the Node Accessor to the First Unoccupied Root-Node Location
2. Adding a Node
3. Setting the Node ID and Value Properties

This code fragment shows how to implement these steps to create the first root node:

```
VarTrPtr mTreeDs = new NDVarTr;
VarTrNodeAccessorPtr mNodeAccessor = new NDVarTrNodeAccessor;
VarPtr mVarID = new NDVar;
VarPtr mVarValue = new NDVar;
VarTrEditPtr mEditTreeDs = mTreeDs->StartEdit();

// Set the title of the tree datasource.
mEditTreeDs->SetTitle("Tree Datasource");

// Move the node accessor to the first unoccupied root-node
// location.
```

```

mNodeAccessor->GoNthRoot(mTreeDs->GetNumRoots());

// Add a node at the first empty root-node location.
mEditTreeDs->AddNode(mNodeAccessor);

// Set the node ID and Value properties.
mVarID->SetStr("New");
mVarValue->SetStr("First Root Node");
mEditTreeDs->SetNodeID(mNodeAccessor, mVarID);
mEditTreeDs->SetNodeValue(mNodeAccessor, mVarValue);
mEditTreeDs->End();
...
delete mVarID;
delete mVarValue;
delete mNodeAccessor;
delete mTreeDs;

```

You can also create the first root node using a “convenience” API. This creates and disposes of the edit object for you. This code fragment shows how to use the “convenience” functions to create the first root node in the datasource:

```

VarTrPtr mTreeDs = new NDVarTr;
VarTrNodeAccessorPtr mNodeAccessor = new NDVarTrNodeAccessor;
VarPtr mVarID = new NDVar;
VarPtr mVarValue = new NDVar;

// Set the title of the tree datasource using the "convenience"
// API.
mTreeDs->SetTitle("Tree Datasource");

// Move the node accessor to the first unoccupied root-node
// location.
mNodeAccessor->GoFirstRoot();
while (mTreeDs->IsNodeValid(mNodeAccessor)) {
    mNodeAccessor->GoNext();
}

// Add a node.
mTreeDs->AddNode(mNodeAccessor);

// Set the node ID and Value properties using "convenience" API
// functions.
mVarID->SetStr("New");
mVarValue->SetStr("First Root Node");
mTreeDs->SetNodeID(mNodeAccessor, mVarID);
mTreeDs->SetNodeValue(mNodeAccessor, mVarValue);
...
delete mVarID;
delete mVarValue;
delete mNodeAccessor;
delete mTreeDs;

```

The “convenience” API functions:

1. Create an edit object.
2. Perform the specified operation.
3. Dispose of the edit object when the operation is complete.

Tip: Because these “convenience” functions create and dispose of an edit object for each operation, they are not very efficient for performing batches of editing operations.

Moving the Node Accessor to the First Unoccupied Root-Node Location

When you are creating the first root node in a tree datasource, traversing the datasource to find the first unoccupied root node is very simple. All of the functions in the **NDVarTrNodeAccessor** API move the accessor to the same

node. The `NDVarTrNodeAccessor::GoFirstRoot()` function is used for simplicity and clarity in the preceding code fragment.

The following traversal approach is a more universal. That is because it moves the node accessor to the first unoccupied root-node location, regardless of the number of root nodes in the datasource.

```
// Declarations.
VarTrPtr mTreeDs = new NDVarTr;
VarTrNodeAccessorPtr mNodeAccessor = new NDVarTrNodeAccessor;
...
// Move the node accessor to the first unoccupied root-node
// location.
mNodeAccessor->GoNthRoot(mTreeDs->GetNumRoots());
...
```

In this case, the number of existing root nodes is used as an argument for the `NDVarTrNodeAccessor::GoNthRoot()` function. The value returned by the `NDVarTr::GetNumRoots()` function is a one-based counter, while the `NDVarTrNodeAccessor::GoNthRoot()` function expects a zero-based index. This ensures that the node accessor points to the next unoccupied root node.

Warning: The `NDVarTr::GetNumRoots()` function returns the number of root nodes in the datasource at the time the edit object is created. Do *not* use this return value as a control-loop counter, unless the edit object is created and destroyed within the loop, as the “convenience” API functions do.

You may want to use this nested construct to check the validity of the node before adding a new node:

```
// Declarations.
VarTrPtr mTreeDs = new NDVarTr;
VarTrNodeAccessorPtr mNodeAccessor = new NDVarTrNodeAccessor;
...
// Move the node accessor to the first unoccupied root-node
// location.
mNodeAccessor->GoFirstRoot();
while (mTreeDs->IsValid(mNodeAccessor))
    mNodeAccessor->GoNext();
}
...
```

Adding a Node

After positioning the node accessor at the first unoccupied node location, execute the `VarTrEdit::AddNode()` function to add a node:

```
// Declarations.
VarTrPtr mTreeDs = new NDVarTr;
VarTrNodeAccessorPtr mNodeAccessor = new NDVarTrNodeAccessor;
...
// Move the node accessor to the first unoccupied root-node
// location.
mNodeAccessor->GoFirstRoot();
while (mTreeDs->IsValid(mNodeAccessor))
    mNodeAccessor->GoNext();
}
mTreeDs->AddNode(mNodeAccessor);
...
```

Note: You can also “insert” nodes in the hierarchy. For more information, see “Inserting Nodes versus Adding Nodes” on page 47.

Setting the Node **ID** and **Value** Properties

Although setting the node **ID** and **Value** properties are optional tasks when you are building the node hierarchy, you may want to add a routine to your code to ensure the uniqueness of the node **IDs**. The **ID** field, and possibly the **Value** field, must be unique when associating a node with row data in a table datasource.

To add data to the nodes as you create them, you can use the “convenience” API functions—**NDVarTr::SetNodeID()** and **NDVarTr::SetNodeValue()**—to set the **ID** and **Value** properties. Alternately, the application can end the datasource-level editing session and start a node-level editing session. For more information, see “Node-Level Editing” on page 51.

Creating Child Nodes and Siblings

After creating the first root node, you can methodically create multiple generations of child and sibling nodes. Creating child and sibling nodes is similar to creating the first root node. In each case, you:

1. Move the node accessor to the appropriate node location.
2. Add a node.
3. Optionally set the node **ID** and **Value** properties.

Creating the First Child Node

Continuing with the structure in Figure 3–7, the next task in building your tree datasource is to add the first child node, as Figure 3–8 shows:

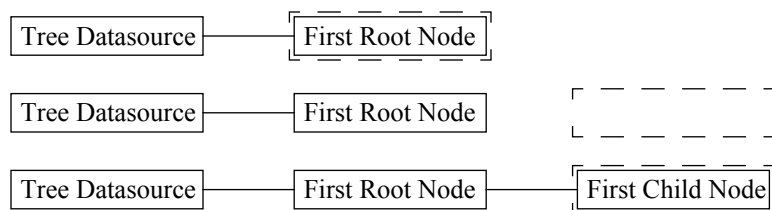


Figure 3–8 Creating the First Child Node

To create the first child node, use the **NDVarTrNodeAccessor** API to traverse the node hierarchy, relative to the first root node, to the first unoccupied child node. This code fragment shows how to create the first child node, using the **NDVarTrNodeAccessor::GoFirstChild()** function to position the node accessor:

```
VarTrPtr mTreeDs = new NDVarTr;
VarTrNodeAccessorPtr mNodeAccessor = new NDVarTrNodeAccessor;
VarPtr mVarID = new NDVar;
VarPtr mVarValue = new NDVar;
VarTrEditPtr mEditTreeDs = mTreeDs->StartEdit();

// Move the node accessor to the first unoccupied root-node
// location.
mNodeAccessor->GoNthRoot(mTreeDs->GetNumRoots());

// Add a node.
mEditTreeDs->AddNode(mNodeAccessor);

// Set the node ID and Value properties.
```

```

mVarID->SetStr("0");
mVarValue->SetStr("First Root Node");
mEditTreeDs->SetNodeID(mNodeAccessor, mVarID);
mEditTreeDs->SetNodeValue(mNodeAccessor, mVarValue);

// Relative to the root node, move the node accessor to the
// first unoccupied child-node location.
mNodeAccessor->GoFirstChild();

// Add a node at the first child-node location.
mEditTreeDs->AddNode(mNodeAccessor);

// Set the node ID and Value properties.
mVarID->SetStr("0,0");
mVarValue->SetStr("First Child Node");
mEditTreeDs->SetNodeID(mNodeAccessor, mVarID);
mEditTreeDs->SetNodeValue(mNodeAccessor, mVarValue);
mEditTreeDs->End();

delete mVarID;
delete mVarValue;
delete mNodeAccessor;
delete mTreeDs;

```

The following traversal approach is more universal. This is because it moves the node accessor to the first unoccupied child-node location, regardless of the number of child nodes.

```

// Declarations.
VarTrPtr mTreeDs = new NDVarTr;
VarTrNodeAccessorPtr mNodeAccessor = new NDVarTrNodeAccessor;
...
// Relative to the first root node, move the node accessor to
// the first unoccupied child-node location.
mNodeAccessor->GoFirstRoot();
if (mTreeDs->IsValid(mNodeAccessor)) {
    mNodeAccessor->GoFirstChild();
    while (mTreeDs->IsValid(mNodeAccessor))
        mNodeAccessor->GoNext();
} // End while.
} // End if.
...

```

In this case, the number of existing sibling nodes is used as an argument for the `NDVarTrNodeAccessor::GoNthSibling()` function. The value returned by the `NDVarTr::GetNumSiblings()` function is a one-based counter, while the `NDVarTrNodeAccessor::GoNthSibling()` function expects a zero-based index. This ensures that the node accessor points to the next unoccupied sibling node.

Warning: Like the `NDVarTr::GetNumRoots()` function, the `NDVarTr::GetNumSiblings()` function returns the number of sibling nodes relative to the node accessor when the edit object is created. Do **not** use this return value as a control-loop counter, unless the edit object is created and destroyed within the loop.

This approach first checks the validity of the root node. If the root node is valid, the node accessor moves to the next empty child location, as in the preceding example.

```

// Declarations.
VarTrPtr mTreeDs = new NDVarTr;
VarTrNodeAccessorPtr mNodeAccessor = new NDVarTrNodeAccessor;
...
// Relative to the first root node, move the node accessor to
// the first unoccupied child-node location.
mNodeAccessor->GoFirstRoot();
if (mTreeDs->IsValid(mNodeAccessor)) {

```

```

        mNodeAccessor->GoNthChild(mTreeDs->
                                GetNumChildren(mNodeAccessor));
    } // End if.
    ...

```

Creating Sibling Nodes

Building on the structure in Figure 3-8, the next task in creating your tree datasource is to add the second child, or next sibling, node, as Figure 3-9 shows:

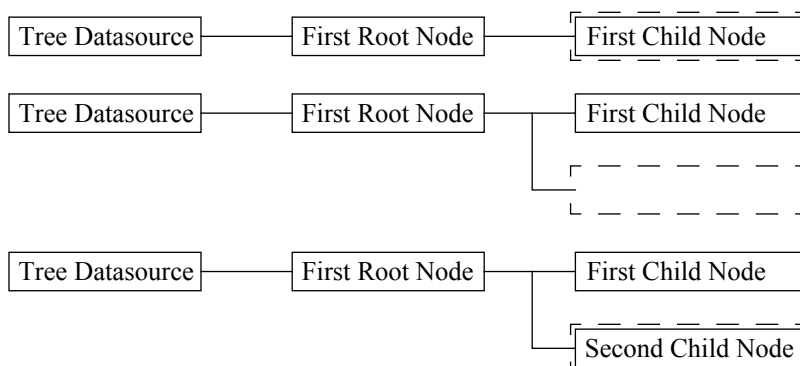


Figure 3-9 Creating the Second Child Node

To create the second child node, move the node accessor, relative to the first child node, to the first unoccupied sibling node. This code fragment shows how to create the second child node, which Figure 3-10 shows, using the **NDVarTrNodeAccessor::GoNext()** function to position the node accessor:

```

VarTrPtr mTreeDs = new NDVarTr;
VarTrNodeAccessorPtr mNodeAccessor = new NDVarTrNodeAccessor;
VarPtr mVarID = new NDVar;
VarPtr mVarValue = new NDVar;
VarTrEditPtr mEditTreeDs = mTreeDs->StartEdit();

// Move the node accessor to the first unoccupied root-node
// location.
mNodeAccessor->GoNthRoot(mTreeDs->GetNumRoots());

// Add a node.
mEditTreeDs->AddNode(mNodeAccessor);

// Set the node ID and Value properties.
mVarID->SetStr("0");
mVarValue->SetStr("First Root Node");
mEditTreeDs->SetNodeID(mNodeAccessor, mVarID);
mEditTreeDs->SetNodeValue(mNodeAccessor, mVarValue);

// Relative to the first root node, move the node accessor to
// the first child-node location.
mNodeAccessor->GoFirstChild();

// Add a node at the first child-node location.
mEditTreeDs->AddNode(mNodeAccessor);

// Set the node ID and Value properties.
mVarID->SetStr("0,0");
mVarValue->SetStr("First Child Node");
mEditTreeDs->SetNodeID(mNodeAccessor, mVarID);
mEditTreeDs->SetNodeValue(mNodeAccessor, mVarValue);

// Relative to the first child node, move the node accessor to
// the next unoccupied sibling-node location.

```



```

mNodeAccessor->GoNext();

// Add a node at the next sibling-node location.
mEditTreeDs->AddNode(mNodeAccessor);

// Set the node ID and Value properties.
mVarID->SetStr("0,1");
mVarValue->SetStr("Second Child Node");
mEditTreeDs->SetNodeID(mNodeAccessor, mVarID);
mEditTreeDs->SetNodeValue(mNodeAccessor, mVarValue);
mEditTreeDs->End();

delete mVarID;
delete mVarValue;
delete mNodeAccessor;
delete mTreeDs;

```

The following traversal method is more universal. This is because it moves the node accessor to the first unoccupied child-node location, regardless of the number of children.

The next code fragment, the internal **while** loop moves the node accessor to the first unoccupied child-node location, regardless of the number of children. After each pass through the **while** loop:

- A node is added.
- The node accessor is returned to the parent-node.

In this example, the “convenience” API functions are used to clarify the application logic. An **Int16** constant, `maxNodes`, is set to 10 to limit the number of children that are created:

```

// Declarations.
VarTrPtr mTreeDs = new NDVarTr;
VarTrNodeAccessorPtr mNodeAccessor = new NDVarTrNodeAccessor;
...
// Relative to the first root node, create nodes at the first
// 10 child-node locations.
mNodeAccessor->GoFirstRoot();
for (Int16 i = mTreeDs->GetNumChildren(mNodeAccessor),
     Int16 maxNodes = 10; i < maxNodes; i++) {
    while (mTreeDs->IsValidNode(mNodeAccessor)) {
        mNodeAccessor->GoNthChild(mTreeDs->
                                GetNumChildren(mNodeAccessor));
    }
    mTreeDs->AddNode(mNodeAccessor);
    mNodeAccessor->GoParent();
}
...

```

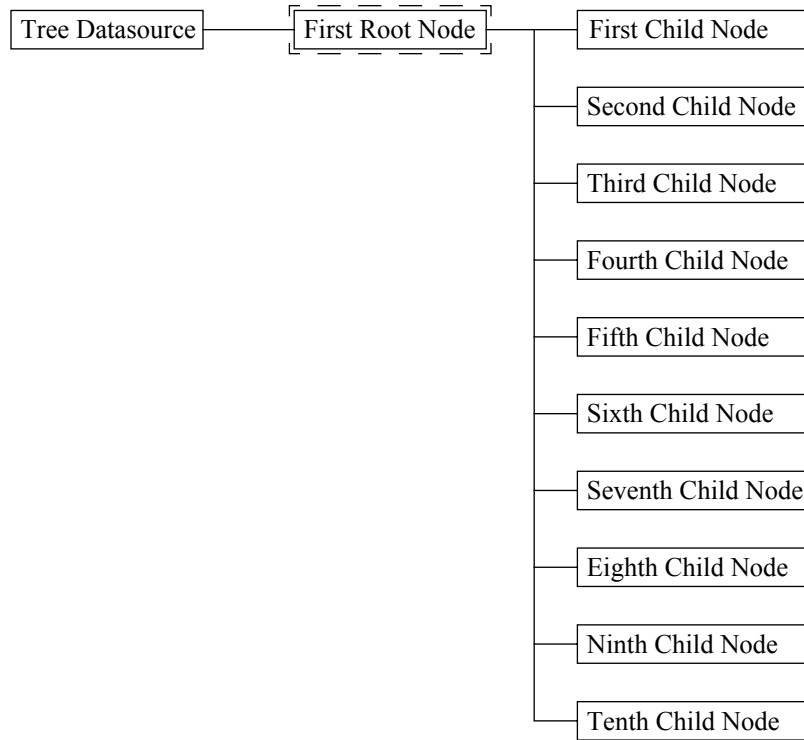


Figure 3-10 Creating the Next Sibling Node

Creating Additional Trees

To create additional trees, you need additional root nodes. Using each root node as a starting point, you can build trees by employing the programming techniques discussed in “Creating Child Nodes and Siblings” on page 36. Figure 3-11 shows the creation of a second root node, from which you can methodically build another tree:

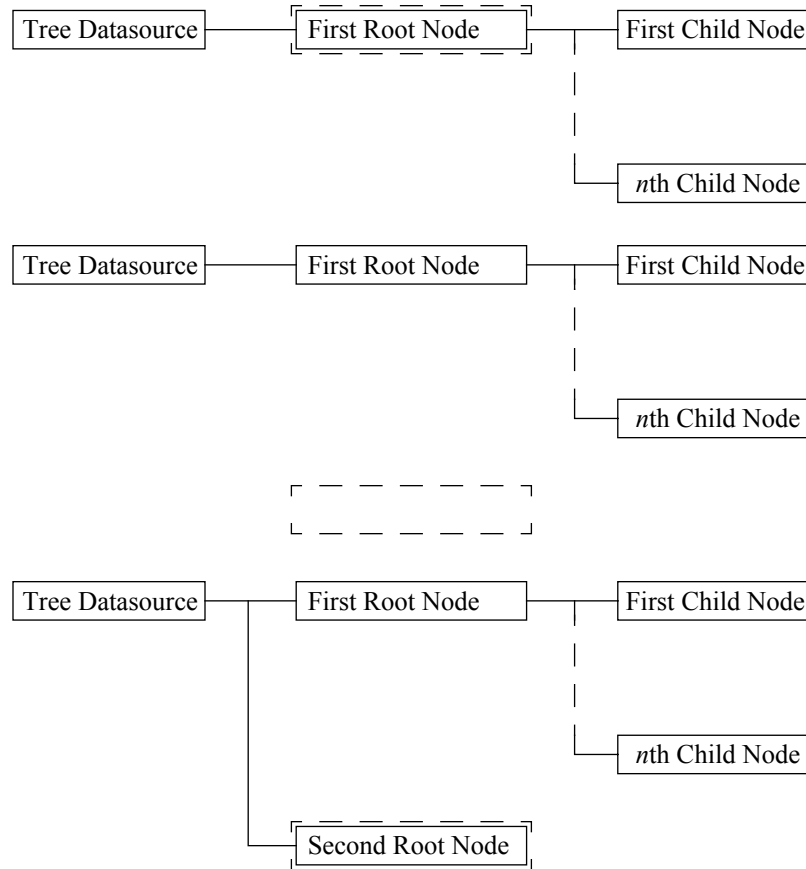


Figure 3-11 Creating Additional Trees

To create additional root nodes, move the node accessor to the first root node, then use the `NDVarTrNodeAccessor::GoNext()` function, as needed, to move the accessor to the first unoccupied root node. This code fragment shows how to create a second root node:

```

VarTrPtr mTreeDs = new NDVarTr;
VarTrNodeAccessorPtr mNodeAccessor = new NDVarTrNodeAccessor;
VarPtr mVarID = new NDVar;
VarPtr mVarValue = new NDVar;
VarTrEditPtr mEditTreeDs = mTreeDs->StartEdit();

// Move the node accessor to the first unoccupied root-node
// location.
mNodeAccessor->GoFirstRoot();
while (mNodeAccessor->IsNodeValid()) {
    mNodeAccessor->GoNext();
}
// Add a node at the first empty root-node location.
mEditTreeDs->AddNode(mNodeAccessor);

// Set the node ID and Value properties.
mVarID->SetStr("1");
mVarValue->SetStr("Second Root Node");
mEditTreeDs->SetNodeID(mNodeAccessor, mVarID);
mEditTreeDs->SetNodeValue(mNodeAccessor, mVarValue);
mEditTreeDs->End();
...
delete mVarID;
delete mVarValue;

```

```
delete mNodeAccessor;
delete mTreeDs;
```

You can also use this traversal approach to move the node accessor directly to the first unoccupied root-node location, regardless of the number of root nodes:

```
// Declarations.
VarTrPtr mTreeDs = new NDVarTr;
VarTrNodeAccessorPtr mNodeAccessor = new NDVarTrNodeAccessor;
...
// Move the node accessor to the first unoccupied root-node
// location.
mNodeAccessor->GoNthRoot(mTreeDs->GetNumRoots());
...
```

In this case, the number of existing root nodes is used as an argument for the **NDVarTrNodeAccessor::GoNthRoot()** function. The value returned by the **NDVarTr::GetNumRoots()** function is a one-based counter, while the **NDVarTrNodeAccessor::GoNthRoot()** function expects a zero-based index. This ensures that the node accessor points to the next unoccupied root node.

Managing Memory

Memory usage accumulates as you create objects for your datasource, regardless of the specific object type. To adequately manage your memory usage, you should destroy an object after it has served its purpose.

The code fragments that show you how to work with the tree datasource consistently illustrate the destruction of edit objects. However, you should also destroy both the tree-datasource and node-accessor objects.

This code fragment shows a general framework for creating and destroying objects; the pairs of creation and destruction calls are indented for clarity:

```
// Declare a tree-datasource pointer variable, and assign a
// tree-datasource object to it.
VarTrPtr mTreeDs = new NDVarTr;

// Declare an edit pointer variable, and assign an edit
object
// to it.
VarTrEditPtr mEditTreeDs = mTreeDs->StartEdit();

// Declare a node-accessor pointer variable, and assign a
// a node-accessor object to it.
VarTrNodeAccessorPtr mNodeAccessor =
    new NDVarTrNodeAccessor;

// Position the node accessor and edit the tree.
...

// Destroy the node-accessor object.
delete mNodeAccessor;

...

// Commit the changes and destroy the edit object.
mEditTreeDs->End();

...

// Destroy the tree-datasource object.
delete mTreeDs;
```

The formal creation and destruction of the datasource-level edit object, illustrated in the preceding code fragment, is most useful when performing editing operations in batches. For single operations, the “convenience” API

is effective. This is because, using the cursor as a node reference, it automatically creates an edit object, completes the editing operation, and disposes of the edit object—all in one step.

Destroying the Tree-Datasource Object

The tree-datasource object may be the object least often destroyed in your application. However, good object construction and destruction habits can minimize application errors created by memory leaks.

For example, an application that sequentially loads and unloads several different organizational hierarchies continues to increase its memory usage if the datasource objects are not destroyed and created again when needed. Simply disassociating a view from a datasource—**NDDs::UnregisterView()** function—is not enough to avoid memory leaks in your application.

This code fragment shows how to create and destroy a tree datasource:

```
// Declare a tree-datasource pointer variable, and assign a
// tree-datasource object to it.
VarTrPtr mTreeDs = new NDVarTr;
...
// Destroy the tree-datasource object.
delete mTreeDs;
```

Before destroying your tree datasource, you will undoubtedly need to compose a routine to traverse the hierarchy and write the data to a persistent storage medium, such as a flat-file format on a local hard disk. You might want to associate each node in the tree datasource with a row in a table datasource, using the query mechanism supplied by the Data Access Element.

Destroying a Node-Accessor Object

Depending on the logic of your application, you may want to create multiple node accessors or aliases to a single node accessor. If you create multiple node-accessor objects and assign them to pointer variables, you must destroy each of them separately, as shown here:

```
// Declare a tree-datasource pointer variable, and assign a
// tree-datasource object to it.
VarTrPtr mTreeDs = new NDVarTr;

// Declare three node-accessor pointer variables, and assign
// node-accessor objects to them.
VarTrNodeAccessorPtr mNodeAccessor_1 =
    new NDVarTrNodeAccessor;
VarTrNodeAccessorPtr mNodeAccessor_2 =
    new NDVarTrNodeAccessor;
VarTrNodeAccessorPtr mNodeAccessor_3 =
    new NDVarTrNodeAccessor;

// Move the node accessors and edit the tree datasource.
...
// Destroy each of the node accessors separately.
delete mNodeAccessor_1;
delete mNodeAccessor_2;
delete mNodeAccessor_3;
...
// Destroy the tree-datasource object.
delete mTreeDs;
```

You can also set aliases to a node accessor by assigning the node-accessor pointer returned by the **NDVarTr::GetCursor()** function to a node-accessor

pointer variable. This code fragment shows how you might use such aliases to your node accessor:

```
// Declare a tree-datasource pointer variable, and assign a
// tree-datasource object to it.
VarTrPtr mTreeDs = new NDVarTr;

// Declare a node-accessor pointer variable, and assign a
// node-accessor object to it.
VarTrNodeAccessorPtr mNodeAccessor = new NDVarTrNodeAccessor;

// Set a cursor at the location of the node accessor.
mTreeDs->SetCursor(mNodeAccessor);

// Declare two additional node-accessor pointer variables, and
// assign the node-accessor pointers returned by the
// GetCursor() function to them.
VarTrNodeAccessorPtr mManager = mTreeDs->GetCursor();
VarTrNodeAccessorPtr mEmployee = mTreeDs->GetCursor();
...
// Move the node accessor and aliases, and edit the tree
// datasource.
...
// Destroy the node-accessor object.
delete mNodeAccessor;

// Destroy the tree-datasource object.
delete mTreeDs;
```

In the preceding example, only `mNodeAccessor` must be destroyed, because `mManager` and `mEmployee` are only aliases to the node-accessor object, `mNodeAccessor`. Upon destroying the node accessor object, the two aliases become meaningless.

Editing a Tree Datasource

The concept of “editing” the tree datasource described in this section is based on the assumption that you are using one or more views to control the movement of the cursor in the datasource. You should keep this in mind when evaluating the code examples in this section.

When editing a tree datasource, these editing levels apply:

- Datasource-Level Editing
- Node-Level Editing

The notion of “subtree” locking is not yet supported, so editing operations must lock either the entire tree or a single node. Editing operations involve these steps:

1. Move the node accessor to a specific location.
2. Create either a datasource or a node edit object.
3. Perform editing operations.
4. End the editing operation and commit the modifications.

All editing operations require a node-accessor pointer with a node-accessor object assigned to it. For information about node-accessor declarations, see “Creating and Destroying a Node Accessor” on page 31.

In addition to the node accessor, you need either a datasource-level or node-level edit object. Specifically, you need:

- A datasource edit object when modifying a tree structure
- A node edit object to limit modifications to the node **ID** and **Value** properties

The *edit object* is a working copy of the tree or node you are editing. All modifications are committed to the datasource when the `NDDsEdit::End()` function executes.

Note: See “Datasource Editing” on page 25 and “Node Editing” on page 25 for more information about datasource and node edit objects, respectively.

After declaring the node accessor and the required edit objects, you can position the node accessor on any node location in the tree datasource. You can traverse the tree datasource to complete any datasource-level or node-level editing operations using the node-accessor API listed in Table 3-1:

Table 3-1 Basic Functions for Traversing the Tree Datasource

Function	Description
<code>accessor->GoFirstRoot()</code>	Move <i>accessor</i> to the first root node.
<code>accessor->GoFirstChild()</code>	Move <i>accessor</i> to the first child node relative to the current <i>accessor</i> location.
<code>accessor->GoFirstSibling()</code>	Move <i>accessor</i> to the first sibling node of the current <i>accessor</i> location.
<code>accessor->GoParent()</code>	Move <i>accessor</i> to the parent node of the current <i>accessor</i> location.
<code>accessor->GoPrev()</code>	Move <i>accessor</i> to the previous sibling node of the current <i>accessor</i> location.
<code>accessor->GoNext()</code>	Move <i>accessor</i> to the next sibling node of the current <i>accessor</i> location.

The functions in Table 3-2 require the indexing system described in “Tree Datasource” on page 18. Note that nodes are indexed from 0.

Table 3-2 “Convenience” Functions for Traversing the Tree Datasource

Function	Description
<code>accessor->GoNthRoot(index)</code>	Move <i>accessor</i> to the <i>n</i> th root node, specified by <i>index</i> (zero-based), of the tree datasource.
<code>accessor->GoNthChild(index)</code>	Move <i>accessor</i> to the <i>n</i> th child node, specified by <i>index</i> (zero-based), relative to the current <i>accessor</i> location.
<code>accessor->GoNthSibling(index)</code>	Move <i>accessor</i> to the <i>n</i> th sibling node, specified by <i>index</i> (zero-based), relative to the current <i>accessor</i> location.
<code>accessor->GoID(id)</code>	Move <i>accessor</i> to the node with the specified <i>id</i> .

The functions in Table 3–3 comprise a useful API with which to retrieve information from the tree datasource and the nodes it contains:

Table 3–3 Functions for Getting Miscellaneous Information

Function	Description
<code>mTreeDs->GetTitle()</code>	Get the title of the tree datasource, <i>mTreeDs</i> .
<code>mTreeDs->GetNumRoots()</code>	Get the number (one-based) of root nodes in the tree datasource, <i>mTreeDs</i> .
<code>mTreeDs->GetNumChildren(accessor)</code>	Get the number (one-based) of child nodes relative to the current <i>accessor</i> location.
<code>mTreeDs->GetNumSiblings(accessor)</code>	Get the number (one-based) of sibling nodes relative to the current <i>accessor</i> location.
<code>mTreeDs->QueryNodeID(accessor, idPtr)</code>	Copy the data setting of the node ID property at the current <i>accessor</i> location to the address to which <i>idPtr</i> points.
<code>mTreeDs->QueryNodeValue(accessor, valuePtr)</code>	Copy the data setting of the node Value property at the current <i>accessor</i> location to the address to which <i>valuePtr</i> points.
<code>mTreeDs->GetNodeID(accessor)</code>	Get a pointer to the variant object that stores the data of the node ID property at the current <i>accessor</i> location.
<code>mTreeDs->GetNodeValue(accessor)</code>	Get a pointer to the variant object that stores the data of the node Value property at the current <i>accessor</i> location.
<code>mTreeDs->IsNodeValid(accessor)</code>	Returns a boolean value indicating whether a node exists at the current <i>accessor</i> location.

This code shows how to create and dispose of a datasource edit object:

```
VarTrPtr mTreeDs = new NDVarTr;
VarTrNodeAccessorPtr mNodeAccessor = new NDVarTrNodeAccessor;

// Declare a datasource edit object and a pointer to it.
VarTrEditPtr mEditTreeDs = mTreeDs->StartEdit();

// Edit operations.
...

// Commit the edit operations, and destroy the edit object.
mEditTreeDs->End();
```

You can use the datasource edit object to edit at the node level, too. If you want to restrict modifications to the node level only, this code shows how to create and dispose of a node edit object:

```
VarTrPtr mTreeDs = new NDVarTr;
VarTrNodeAccessorPtr mNodeAccessor = new NDVarTrNodeAccessor;

// Declare a node edit object and a pointer to it.
VarTrNodeEditPtr mEditNode = mTreeDs->StartNodeEdit();
...
mEditNode->End();
```

Modifications are committed when you call the **NDDsEdit::End()** function. This also releases the data lock that was created when the **NDVarTr::StartEdit()** function executed. If you call another

NDVarTr::StartEdit() before the **NDDsEdit::End()** function executes, the **StartEdit()** function will fail, because the data is still locked.

Datasource-Level Editing

When you make structural changes to a tree datasource, you need a datasource edit object. This locks the entire datasource to prevent simultaneous editing of the datasource from another view.

These are datasource-level editing operations:

- Setting the Title of the Tree Datasource
- Inserting Nodes versus Adding Nodes
- Modifying Node Data Using the “Convenience” API
- Removing a Node
- Removing a Tree

The functions in Table 3–4 support the structural modifications to the tree datasource and are supplied by the **NDVarTrEdit** API.

Table 3–4 Functions for Structural Modifications to the Tree Datasource

Function	Description
<code>mTreeDs->SetTitle()</code>	Sets the Title property of the tree datasource, <i>mTreeDs</i> .
<code>mEditTreeDs->AddNode()</code>	Add a node at the current <i>accessor</i> location.
<code>mEditTreeDs->RemoveNode()</code>	Remove the node at the current <i>accessor</i> location. Child nodes of the node to be removed become the children of the parent node of the removed node.
<code>mEditTreeDs->RemoveTree()</code>	Remove the hierarchy beneath the current <i>accessor</i> location.

Setting the Title of the Tree Datasource

The tree datasource has a **Title** property to which you can assign a string value using the **NDVarTrEdit::SetTitle(string)** function. Likewise, you can retrieve the current title assigned to the tree datasource using the **NDVarTr::GetTitle()** function.

Inserting Nodes versus Adding Nodes

You can add a node at any valid node-accessor location. If a node is already present, the new node is *inserted* before the existing node. If there is no node at the current accessor location, you can *add* a node.

Figure 3-12 shows the insertion of a “third” child node between the “first” and “second” child nodes. The inserted node becomes the sibling of the first two nodes.

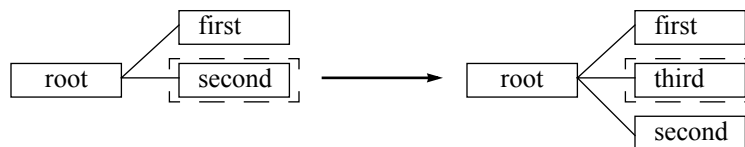


Figure 3-12 Inserting a Child Node

This code fragment shows how to programmatically *insert* a child node, “third,” with the node-accessor position at the “second” node location:

```

VarTrPtr mTreeDs = new NDVarTr;
VarTrNodeAccessorPtr mNodeAccessor = new NDVarTrNodeAccessor;
VarPtr mVarValue = new NDVar;

// Empty accessor location.
mNodeAccessor->GoNthRoot(mTreeDs->GetNumRoots());
mTreeDs->AddNode(mNodeAccessor); // Add first root node.
mVarValue->SetStr("root");
mTreeDs->SetNodeValue(mNodeAccessor, mVarValue);

// Empty accessor location.
mNodeAccessor->GoNthChild(mTreeDs->
    GetNumChildren(mNodeAccessor));
mTreeDs->AddNode(mNodeAccessor); // Add first child node.
mVarValue->SetStr("first");
mTreeDs->SetNodeValue(mNodeAccessor, mVarValue);

// Empty accessor location.
mNodeAccessor->GoNext();
mTreeDs->AddNode(mNodeAccessor); // Add second child node.
mVarValue->SetStr("second");
mTreeDs->SetNodeValue(mNodeAccessor, mVarValue);

// Add the third child node at the "second" node location.
mTreeDs->AddNode(mNodeAccessor); // Add third child node.
mVarValue->SetStr("third");
mTreeDs->SetNodeValue(mNodeAccessor, mVarValue);
...
delete mVarID;
delete mVarValue;
delete mNodeAccessor;
delete mTreeDs;
  
```

If you add a node at an unoccupied accessor location, a new node is created with a Next reference that accesses an empty node location. In other words, the new node is the last node in the sibling or root list.

```

VarTrPtr mTreeDs = new NDVarTr;
VarTrNodeAccessorPtr mNodeAccessor = new NDVarTrNodeAccessor;
VarPtr mVarValue = new NDVar;

// Empty root-level accessor location.
mNodeAccessor->GoNthRoot(mTreeDs->GetNumRoots());

// Add first root node.
mTreeDs->AddNode(mNodeAccessor);
mVarValue->SetStr("1st root");
mTreeDs->SetNodeValue(mNodeAccessor, mVarValue);

// Empty root-level accessor location.
mNodeAccessor->GoNext(accessor);

// Add second root node.
  
```

```

mTreeDs->AddNode(mNodeAccessor);
mVarValue->SetStr("2nd root");
mTreeDs->SetNodeValue(mNodeAccessor, mVarValue);
...
// Empty root-level accessor location. */
mNodeAccessor->GoNext();

// Add nth root node.
mTreeDs->AddNode(mNodeAccessor);
mVarValue->SetStr("nth root");
mTreeDs->SetNodeValue(mNodeAccessor, mVarValue);
...
delete mVarID;
delete mVarValue;
delete mNodeAccessor;
delete mTreeDs;

```

Modifying Node Data Using the “Convenience” API

Using the “convenience” functions supplied with the `NDVarTrEdit` class, you can perform these node-level editing operations using a datasource edit object:

- Setting the Node **ID**
- Setting the Node **Value**

You can set the node **ID** using the `NDVarTrEdit::SetNodeID()` function. Likewise, you can set the value using the `NDVarTrEdit::SetNodeValue()` function.

With the “convenience” functions in Table 3–5, you can easily modify node data during a datasource editing session. You do not need a node edit object. However, you do need a datasource edit object to prevent editing of the datasource from other views.

Table 3–5 “Convenience” Functions for Modifying Node Data

Function	Description
<code>mTreeDs->SetNodeID(accessor, id)</code>	Set the ID property of the node at the current <i>accessor</i> location to <i>id</i> .
<code>mTreeDs->SetNodeValue(accessor, value)</code>	Set the Value property of the node at the current <i>accessor</i> location to <i>value</i> .

This code fragment uses the “convenience” API to set the node **ID** of the first child of the first root to “0,0” and to set the node **Value** to “first child”:

```

VarTrPtr mTreeDs = new NDVarTr;
VarTrNodeAccessorPtr mNodeAccessor = new NDVarTrNodeAccessor;
VarPtr mVarID = new NDVar;
VarPtr mVarValue = new NDVar;

// Go to the first child of the first root node.
mNodeAccessor->GoFirstRoot();
mNodeAccessor->GoFirstChild();

// Set the node ID and Value properties.
mVarID->SetStr("0,0");
mVarValue->SetStr("first child");
mTreeDs->SetNodeID(mNodeAccessor, mVarID);
mTreeDs->SetNodeValue(mNodeAccessor, mVarValue);
...
delete mVarID;
delete mVarValue;

```

```
delete mNodeAccessor;
delete mTreeDs;
```

Removing a Node

When you remove a node from the tree hierarchy, child nodes become the children of the parent of the node being removed. If the node being removed is a root node, its child nodes become root nodes.

Figure 3-13 shows the nodes with labels that indicate their positions in the hierarchy. Note that the immediate child nodes of the root node become root nodes.

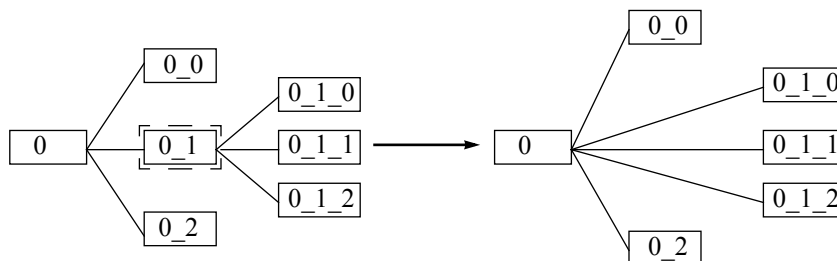


Figure 3-13 Removing a Node

This code fragment shows how to edit the tree structure on the left side of Figure 3-13 to produce the structure on the right:

```
VarTrPtr mTreeDs = new NDVarTr;
VarTrNodeAccessorPtr mNodeAccessor = new NDVarTrNodeAccessor;
VarTrEditPtr mEditTreeDs = mTreeDs->StartEdit();

// Create the tree structure on the left side of Figure 3-13.
...
// Go to the second child node of the first root node.
mNodeAccessor->GoFirstRoot();
mNodeAccessor->GoNthChild(1);

// Remove a node.
mEditTreeDs->RemoveNode(mNodeAccessor);
mEditTreeDs->End();
...
delete mNodeAccessor;
delete mTreeDs;
```

Removing a Tree

When you remove a tree or subtree from the tree hierarchy, the child nodes of the current node are removed recursively. If the accessed node is a root node, **NDVarTrEdit::RemoveTree()** removes the entire tree. Figure 3-14 shows the nodes with labels that indicate their positions in the hierarchy:

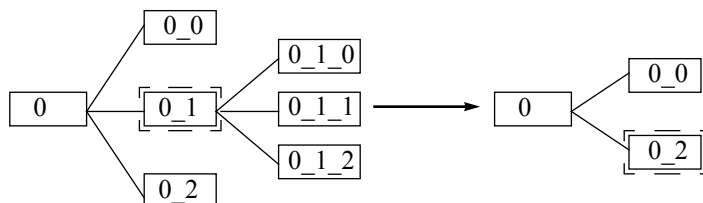


Figure 3-14 Removing a Tree

This code fragment shows how to edit the tree structure on the left side of Figure 3–14 to produce the structure on the right:

```

VarTrPtr mTreeDs = new NDVarTr;
VarTrNodeAccessorPtr mNodeAccessor = new NDVarTrNodeAccessor;
VarTrEditPtr mEditTreeDs = mTreeDs->StartEdit(mNodeAccessor);

// Create the tree structure on the left side of Figure 3–14.
...
// Go to the second child of the first root node.
mNodeAccessor->GoFirstRoot();
mNodeAccessor->GoNthChild(1);

// Remove a tree.
mEditTreeDs->RemoveTree(mNodeAccessor);
mEditTreeDs->End();
...
delete mNodeAccessor;
delete mTreeDs;

```

Node-Level Editing

There are only two operations you can perform at the node level:

- Setting the Node ID
- Setting the Node Value

You can use the functions in Table 3–6 to get and set node data. While all of these are useful, only `NDVarTrNodeEdit::SetID()` and `NDVarTrNodeEdit::SetValue()` require you to declare a node edit object.

Table 3–6 Functions for Getting and Setting Node Data

Function	Description
<code>mTreeDs->QueryNodeID(accessor, idPtr)</code>	Copy the ID property of the node at the current <i>accessor</i> location to the address stored by <i>idPtr</i> .
<code>mTreeDs->QueryNodeValue(accessor, valuePtr)</code>	Copy the Value property of the node at the current <i>accessor</i> location to the address stored by <i>valuePtr</i> .
<code>mTreeDs->GetNodeID(accessor)</code>	Get a pointer to the variant object that stores the data of the ID property at the current <i>accessor</i> location.
<code>mTreeDs->GetNodeValue(accessor)</code>	Get a pointer to the variant object that stores the data of the Value property of the node at the current <i>accessor</i> location.
<code>mEditNode->SetID(id)</code>	Set the ID property of the node at the current <i>accessor</i> location to <i>id</i> .
<code>mEditNode->SetValue(value)</code>	Set the Value property of the node at the current <i>accessor</i> location to <i>value</i> .
<code>mEditTreeDs->SetNodeID(accessor, id)</code>	Set the ID property of the node at the current <i>accessor</i> location to <i>id</i> .
<code>mEditTreeDs->SetNodeValue(accessor, value)</code>	Set the Value property of the node at the current <i>accessor</i> location to <i>value</i> .
<code>mTreeDs->IsNodeValid(accessor)</code>	Get a boolean value indicating whether a node exists in the tree datasource at the current <i>accessor</i> location.

Setting the Node ID

You can:

- Get a copy of the node **ID** using `NDVarTr::QueryNodeID()`
- Get a pointer to the **ID** using `NDVarTr::GetNodeID()`
- Set the **ID** using `NDVarTrNodeEdit::SetID()`

This code sets the node **ID** of the first child of the first root to “0,0” and sets the node value to “first child”:

```
VarTrPtr mTreeDs = new NDVarTr;
VarTrNodeAccessorPtr mNodeAccessor = new NDVarTrNodeAccessor;
VarPtr mVarID = new NDVar;
VarPtr mVarValue = new NDVar;

// Go to the first child of the first root node.
mNodeAccessor->GoFirstRoot();
mTreeDs->AddNode(mNodeAccessor);

mNodeAccessor->GoFirstChild();
mTreeDs->AddNode(mNodeAccessor);

// Set the node ID and Value properties.
VarTrNodeEditPtr mEditNode =
    mTreeDs->StartNodeEdit(mNodeAccessor);
mVarID->SetStr("0,0");
mVarValue->SetStr("first child");
mEditNode->SetID(mNodeAccessor, mVarID);
mEditNode->SetValue(mNodeAccessor, mVarValue);
mEditNode->End();
...
delete mVarID;
delete mVarValue;
delete mNodeAccessor;
delete mTreeDs;
```

Setting the Node Value

You can:

- Get a copy of the **Value** using `NDVarTr::QueryNodeValue()`
- Get a pointer to the **Value** using `NDVarTr::GetNodeValue()`
- Set the **Value** using `NDVarTrNodeEdit::SetValue()`

Advanced Topics

The following issues are more advanced topics that provide very useful information when creating a datasource application:

- Node-Count Functions
- Managing the Cursor
- Acting on Multiple Nodes
- Persistent Data Storage and Relational Tables

Node-Count Functions

In the examples supplied in the following sections, these node-count functions in the `NDVarTr` API are used to help traverse the tree datasource:

- `GetNumRoots()`
- `GetNumChildren()`
- `GetNumSiblings()`

Before planning the logic of your application, you should familiarize yourself with the basic behaviors of these node-count functions. These functions return the number of applicable nodes that are present in the datasource at the time you create the edit object. The current number of applicable nodes is not reflected until the editing operations are committed to the datasource.

The following code fragment does **not** work. In this case, the **while** loop never terminates, because the edit operations are not committed to the datasource before each evaluation of the control criterion. Specifically, the **NDVarTr::GetNumRoots()** function returns the same value on each cycle of the loop, so the control criterion never evaluates **FALSE**.

```
VarTrPtr mTreeDs = new NDVarTr;
VarTrNodeAccessorPtr mNodeAccessor = new NDVarTrNodeAccessor;
VarTrEditPtr mEditTreeDs = mTreeDs->StartEdit();

// Add nodes as long as the number of root nodes is less than
// maxRoots.
Int16 maxRoots = 10;
while (mTreeDs->GetNumRoots() < maxRoots) {
    mNodeAccessor->GoNthRoot(mTreeDs->GetNumRoots());
    mEditTreeDs->AddNode(mNodeAccessor);
}
...
mEditTreeDs->End();
```

However, if you create and destroy the edit object within the **while** loop—effectively, the same as using the “convenience” API function—you can use the return value from the **NDVarTr::GetNumRoots()** function as a counter. In this example, the **NDVarTr::StartEdit()** and **NDDsEdit::End()** functions are located, respectively, as the first and last executable lines of code within the **while** loop:

```
VarTrPtr mTreeDs = new NDVarTr;
VarTrEditPtr mEditTreeDs; // No edit object created.
VarTrNodeAccessorPtr mNodeAccessor = new NDVarTrNodeAccessor;
Int16 maxRoots = 10;
while (mTreeDs->GetNumRoots() < maxRoots) {
    // Create edit object.
    mEditTreeDs = mTreeDs->StartEdit();
    mNodeAccessor->GoNthRoot(mTreeDs->GetNumRoots());
    mEditTreeDs->AddNode(mNodeAccessor);
    // Commit the edit operation and destroy the edit object.
    mEditTreeDs->End();
}
...

```

The preceding example is the same as using the “convenience” API as shown here:

```
VarTrPtr mTreeDs = new NDVarTr;
VarTrEditPtr mEditTreeDs; // No edit object created.
VarTrNodeAccessorPtr mNodeAccessor = new NDVarTrNodeAccessor;
Int16 maxRoots = 10;
while (mTreeDs->GetNumRoots() < maxRoots) {
    mNodeAccessor->GoNthRoot(mTreeDs->GetNumRoots());
    // Create edit object, add a node, commit the edit operation,
    // and destroy the edit object.
    mEditTreeDs->AddNode(mNodeAccessor);
}
...

```

The next code fragment does **not** work. Upon initial examination, this appears to be a simpler way to perform the intended tasks of the two preceding examples:

```

VarTrPtr mTreeDs = new NDVarTr;
VarTrEditPtr mEditTreeDs = mTreeDs->StartEdit();
VarTrNodeAccessorPtr mNodeAccessor = new NDVarTrNodeAccessor;
Int16 maxRoots = 10;
Int16 i = 0;
mNodeAccessor->GoFirstRoot();
while (mTreeDs->IsNodeValid(mNodeAccessor) && (i < maxRoots))
    mEditTreeDs->AddNode(mNodeAccessor);
    mNodeAccessor->GoNext();
    i++;
}
...
mEditTreeDs->End();

```

In the preceding code fragment, the **IsNodeValid()** control-loop criterion evaluates **FALSE** after only the first pass through the loop, because the node created in the first pass is not yet committed to the datasource.

In this code fragment, the **Int16** declaration and **while** loop create the same nodes as the preceding code fragment, except that the nodes are created as children of the parent node instead of as siblings of the first child:

```

VarTrPtr mTreeDs = new NDVarTr;
VarTrNodeAccessorPtr mNodeAccessor = new NDVarTrNodeAccessor;
NDVarPtr mVarID = new NDVar;
NDVarPtr mVarValue = new NDVar;
VarTrEditPtr mEditTreeDs; // No edit object created.

// Using the "convenience" API, add a node at the next empty
// root-node location.
mNodeAccessor->GoNthRoot(mTreeDs->GetNumRoots());
mTreeDs->AddNode(mNodeAccessor);

// Set variants and use them to set the node ID and Value
// properties.
mVarID->SetStr("r0000");
mVarValue->SetStr("First Root Node");
mTreeDs->SetNodeID(mNodeAccessor, mVarID);
mTreeDs->SetNodeValue(mNodeAccessor, mVarValue);

// Add nodes as long as the number of root nodes is less than
// maxNodes.
Int16 maxNodes = 10;
while (mTreeDs->GetNumChildren(mNodeAccessor) < maxNodes) {

// Create a datasource edit object, and assign it to the edit
// pointer variable.
mEditTreeDs = mTreeDs->StartEdit();

// Descend from the parent to the next "empty" child-node
// location.
mNodeAccessor->GoNthChild(
    mTreeDs->GetNumChildren(mNodeAccessor));

// Add a node and set the node ID and Value properties.
mEditTreeDs->AddNode(mNodeAccessor);
mVarID->SetStr("c0000");
mVarValue->SetStr("Child Node");
mEditTreeDs->SetNodeID(mNodeAccessor, mVarID);
mEditTreeDs->SetNodeValue(mNodeAccessor, mVarValue);

// Commit the changes to the datasource, and destroy the edit
// object.
mEditTreeDs->End();

// Return the node accessor to the parent node.
mNodeAccessor->GoParent();
}

// Dispose of the accessor and datasource.

```



```
delete mNodeAccessor;
delete mTreeDs;
```

In this case, the number of existing child nodes is used as an argument for the `NdVarTrNodeAccessor::GoNthChild()` function. The value returned by the `NdVarTr::GetNumChildren()` function is a one-based counter, while the `NdVarTrNodeAccessor::GoNthChild()` function expects a zero-based index. This ensures that the node accessor points to the next unoccupied sibling node.

Managing the Cursor

When building or editing the tree datasource interactively through a view widget, you can set a cursor in the datasource to follow the movement of the cursor in the view. The functions in Table 3-7 supply cursor management for the tree datasource:

Table 3-7 Cursor Functions

Function	Description
<code>mTreeDs->SetCursor(accessor)</code>	Set <i>accessor</i> as the datasource cursor.
<code>mTreeDs->GetCursor()</code>	Get the datasource cursor.

Acting on Multiple Nodes

Some datasource editing operations may require more than one node accessor. One example of the use of multiple node accessors is for moving or copying a node, or a subtree represented by the node, to a new location in the hierarchy. This is the same as assigning the node, subtree, or copy a new parent.

Figure 3-15 shows the child nodes of the first root being assigned the second root node as a new parent node:

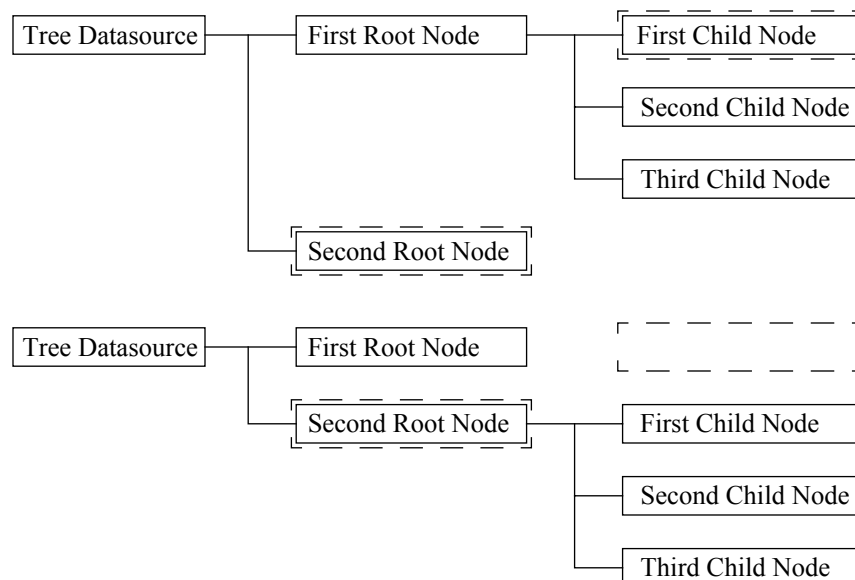


Figure 3-15 Assigning Child Nodes a New Parent Node

This code fragment uses the “convenience” API functions to perform this operation in the tree datasource using two node accessors, `mNodeAccessorFrom` and `mNodeAccessorTo`:

```

VarTrPtr mTreeDs = new NDVarTr;
VarTrNodeAccessorPtr mNodeAccessorFrom =
    new NDVarTrNodeAccessor;
VarTrNodeAccessorPtr mNodeAccessorTo =
    new NDVarTrNodeAccessor;
NDVarPtr mVarID = new NDVar;
NDVarPtr mVarValue = new NDVar;

// Using the "convenience" APIs, create the initial hierarchy:
// one root root node with three children and a second root node
// with no children.
mNodeAccessorFrom->GoFirstRoot();
mTreeDs->AddNode(mNodeAccessorFrom);
mVarID->SetStr("0");
mVarValue->SetStr("First Root Node");
mTreeDs->SetNodeID(mNodeAccessorFrom, mVarID);
mTreeDs->SetNodeValue(mNodeAccessorFrom, mVarValue);

mNodeAccessorFrom->GoFirstChild();
mTreeDs->AddNode(mNodeAccessorFrom);
mVarID->SetStr("0,0");
mVarValue->SetStr("First Child Node");
mTreeDs->SetNodeID(mNodeAccessorFrom, mVarID);
mTreeDs->SetNodeValue(mNodeAccessorFrom, mVarValue);

mNodeAccessorFrom->GoNext();
mTreeDs->AddNode(mNodeAccessorFrom);
mVarID->SetStr("0,1");
mVarValue->SetStr("Second Child Node");
mTreeDs->SetNodeID(mNodeAccessorFrom, mVarID);
mTreeDs->SetNodeValue(mNodeAccessorFrom, mVarValue);

mNodeAccessorFrom->GoNext();
mTreeDs->AddNode(mNodeAccessorFrom);
mVarID->SetStr("0,2");
mVarValue->SetStr("Third Child Node");
mTreeDs->SetNodeID(mNodeAccessorFrom, mVarID);
mTreeDs->SetNodeValue(mNodeAccessorFrom, mVarValue);

mNodeAccessorFrom->GoNthRoot(mTreeDs->GetNumRoots());
mTreeDs->AddNode(mNodeAccessorFrom);
mVarID->SetStr("1");
mVarValue->SetStr("Second Root Node");
mTreeDs->SetNodeID(mNodeAccessorFrom, mVarID);
mTreeDs->SetNodeValue(mNodeAccessorFrom, mVarValue);

// Move the "From" accessor to the first root node.
mNodeAccessorFrom->GoFirstRoot();

// Move the "To" accessor to the second root-node.
mNodeAccessorTo->GoNthRoot(1);

// Execute the while loop until all children of the first root
// node are relocated.
while (mTreeDs->IsValid(mNodeAccessorFrom)) {

    // Move the "From" accessor to the FIRST child of the first
    // root node.
    mNodeAccessorFrom->GoFirstChild();

    // Move the "To" accessor to the LAST EMPTY child location of
    // the second root node.
    mNodeAccessorTo->GoNthChild(
        mTreeDs->GetNumChildren(mNodeAccessorTo));

    mTreeDs->AddNode(mNodeAccessorTo);

```

```

mTreeDs->SetNodeID(mNodeAccessorTo,
                  mTreeDs->GetNodeID(mNodeAccessorFrom));
mTreeDs->SetNodeValue(mNodeAccessorTo,
                     mTreeDs->GetNodeValue(mNodeAccessorFrom));
// Remove the child node from the first root.
mTreeDs->RemoveNode(mNodeAccessorFrom);

// Move the "From" and "To" node accessors to the first and
// second root nodes, respectively.
mNodeAccessorFrom->GoParent();
mNodeAccessorTo->GoParent();
}
delete mVarID;
delete mVarValue;
delete mNodeAccessorFrom;
delete mNodeAccessorTo;
delete mTreeDs;

```

Persistent Data Storage and Relational Tables

The tree datasource has no mechanism for persistent data storage. However, you can design your own scheme for writing data to, and reading data from, a persistent storage medium. You may want to store the hierarchy in a flat-file format. Alternately, you could create a database schema or spreadsheet file to store the pertinent information for the hierarchy.

Relying on the uniqueness of the node **ID**, you can also use the inherent hierarchical design of the tree datasource to store node information as part of the corresponding row data in a table datasource. You would construct the table datasource indirectly by a query to a relational table. You could then design an algorithm to reconstruct the tree datasource from the row data—one row for each node—in the table datasource each time the application executes.

To associate nodes with row data from a relational table, use a table datasource to store extended data pertaining to the node. Allow the tree datasource to store the relationship between the rows. In this case, each node **ID** in the tree datasource is unique and corresponds to a column value in exactly one row in the relational table.

Graph Datasource: Managing Graph Data

A *graph datasource* is a container of freely arranged nodes and edges. The graph datasource is similar to the other datasources—for example, list (sequential), table (tabular), and tree (hierarchical) datasources—because it is based on a specific data model. In this case, the data model is a **graph**, which combines hierarchical and neighbor relationships.

You can display the contents of the graph datasource in an **NDDGram** view, which is supplied by the Open Interface Element. The Elements Environment *datasource/views* mechanism supports the interface between the datasource and the **NDDGram** view.

This chapter discusses these topics:

- Concepts
- Options for the NDDGram View
- Building a Graph Datasource

Note: Data stored in the graph datasource is **not** persistent. However, you can write a routine to traverse the datasource and write its contents to a persistent data-storage medium, such as a local hard disk or database.

If you haven't already done so, read the chapter on the **NDDGram** widget in the *Open Interface Element C++ User's Guide*. See Chapter 3 of this book for information about registering a view with a datasource.

Concepts

The graph datasource stores information for a graph data model. These concepts describe its use:

- Graph Datasource
- Node
- Edge
- Graph
- Accessor
- Cursor
- Edit Object

This section discusses the preceding concepts, which are then used in “Building a Graph Datasource” on page 112 to tell you how to program a graph datasource.

Graph Datasource

The *graph datasource*—an object of the **NDVarGr** class—is a container class that stores and manages nodes and the edges that define the relationships

between them. When your application disposes of a graph datasource, any contained objects are also disposed.

Using the APIs supplied with the graph datasource (**NDVarGr** object), your application can add and remove nodes and edges contained by the datasource object. You can also use the methods in the **NDVarGr** API to enumerate the nodes and edges in the datasource and to traverse them with an index.

Node

A *node* is one of the two basic components of a graph (the other being *edge*; see “Edge” on page 63). Each node has these properties:

- ID and Value
- XOrigin and YOrigin
- Height and Width
- Custom Node Properties
- Edge References

Each node also has references to its:

- “In” edges
- “Out” edges
- Undirected edges

The API uses these references to traverse the graph datasource. In the conceptual figures that follow, beginning with Figure 4–1, these references are named, respectively:

- InEdge
- OutEdge
- UndirEdge

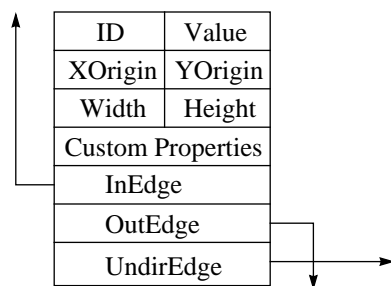


Figure 4–1 Structure of a Node

As Figure 4–1 shows, each node has references to its:

- Parents
- Children
- Neighbors

If any of these references accesses a memory location where no edge exists, then the reference indicates that the current edge is the last valid edge. If the InEdge reference has access to no valid edge location, then the edge is a *root node*, which has no parent. Likewise, if the OutEdge reference has access to no valid edge location, then the node has no children.

You can also think of the edge references in Figure 4-1 as the *edge-reference* mechanisms shown in Figure 4-2:

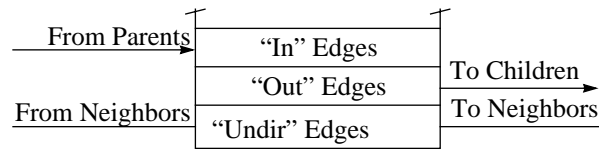


Figure 4-2 Edge-Reference Mechanisms

The arrows in the edge-reference mechanism in Figure 4-2 represent the edges that connect the related nodes. The arrows in Figure 4-1, on the other hand, refer to the edges that define the node relationships.

You can use the functions in the **NDVarGraphNodeAccessor** API to:

- Get the number of related parent, child, and neighbor nodes
- Access any of them by index

For more information about node accessors, see “Node Accessor” on page 71.

ID and Value

Each node in the graph datasource has an **ID** property and a **Value** property. Both the **ID** and **Value** properties store *variant* (**NDVar**) data and can contain any variant-supported type. For example, the **ID** property may be expressed as a variant containing a string, while the **Value** property may be an object reference.

ID

You can assign any variant data to a node **ID** property. Node IDs need not be unique, but they may be more useful if they are. You can assign data to the **ID** property when you create a node or during separate edit sessions.

A unique node ID can be very helpful, especially if you need to associate it with the primary key of a relational-database table. For example, if a node represents a device on a computer network, you may want to set the node **ID** to its asset number, set the **Value** to the device name, and associate the node with a row in a table datasource that shares the same asset number.

Value

Like the node **ID**, you can set the node **Value** property to any variant type. The **Value** property represents the “data” part of the node contents. Your use of the **Value** may range from an employee name in an organizational hierarchy to an employee number acting as a key to display employee data stored in a row of a table datasource.

XOrigin and YOrigin

You can use the **XOrigin** and **YOrigin** properties to store the *x* and *y* coordinates, respectively, of the upper left corner of the bounding box for the node when it is represented in the **NDDGram** view. Specifically, **XOrigin** and **YOrigin** store the number of pixels from the left side and top of the **NDDGram** view.

Note: **XOrigin** and **YOrigin** can actually store any variant data.

Height and Width

You can use the **Height** and **Width** properties to store the lengths, in pixels, of the sides of the bounding box for a node in the **NDDGram** view.

If you do not set these properties, the size of the node is determined by the **NDDGram** view options or the default sizes for the view widget. If there are values, however, they override any defaults that the view supplies.

Note: As with the **XOrigin** and **YOrigin** properties, you can set the **Height** and **Width** properties to any variant value.

For more information about **NDDGram** view options, see “Options for the **NDDGram** View” on page 80.

Custom Node Properties

Custom node properties are additional properties that you can define, which qualify individual nodes or collections of nodes. You can create a property and set or get its value using the **NDVarGr** and **NDVarGrEdit** APIs. For more information, see “Custom Node Properties” on page 99.

Edge References

Each node has three *edge references* to support navigation to its parent, child, and neighbor nodes. Edge references are used by the APIs to traverse and edit the graph datasource.

Edges are either *directed*—that is, “in” or “out” edges—or *undirected*. A node may have any number of parents, children, and neighbors. Any two nodes can have multiple edges relating them. If a pair of nodes is related through multiple edges, all of the edges **must** be either directed or undirected. Two nodes **cannot** be related through both directed and undirected edges.

Directed-Edge References

A reference to a directed edge defines a *parent-child* (hierarchical or antecedent) node relationship. In Figure 4-11:

- The **OutEdge** reference of the parent node defines the link from the parent or “source” node to the directed edge.
- The **ToNode** reference of the directed edge defines the link to the child or “target” node.

Also in Figure 4-11:

- The **InEdge** reference of the child node defines the link from the child node to the directed edge.
- The **FromNode** reference of the directed edge defines the link back to the parent node.

Undirected-Edge References

A reference to an undirected edge defines a *neighbor* node relationship. In Figure 4-12:

- The UndirEdge reference of one neighbor defines the link to the undirected edge.
- The ToNode reference of the undirected edge defines the link to the neighboring node.

Also in Figure 4-12:

- The UndirEdge reference of the neighboring node defines the link to the undirected edge.
- The FromNode of the undirected edge defines the link back to the initial neighbor node.

Edge

An *edge* is one of the two basic components of a graph (the other being *node*; see “Node” on page 60). It indicates a relationship between any two nodes in a graph datasource. Edges **cannot** exist apart from nodes. As a result, an edge becomes undefined if the node at either end of it is removed.

Each edge has these properties:

- ID and Value
- Directed
- Custom Properties

In addition to these properties, each edge also references the nodes at either end of it. Figure 4-3 shows these as “FromNode” and “ToNode.” These references respectively access either:

- The parent and child nodes
- The *n*th and *n*+1 neighbor nodes

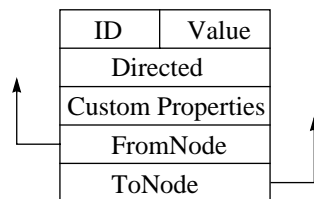


Figure 4-3 Edge Structure

Alternately, you can think of the references in Figure 4-3 as the *node references* shown in Figure 4-4 and Figure 4-5. In these edge-centric figures, the node-reference arrows represent the OutEdge and InEdge references of the respective parent and child nodes.

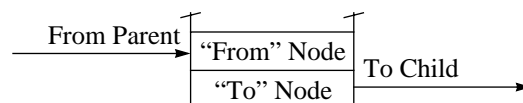


Figure 4-4 Node-Reference Mechanisms for Parent-Child Nodes

In Figure 4–5, the node-reference mechanism uses connecting lines without arrows to represent neighbor relationships. Undirected edges still use `ToNode` and `FromNode` references to access the nodes at either end, but they are significant only when traversing the nodes to find a particular node using an index.

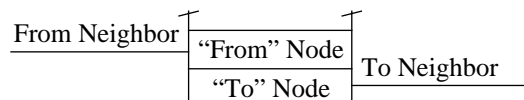


Figure 4–5 Node-Reference Mechanisms for Neighbor Nodes

ID and Value

Each edge in the graph datasource has an **ID** property and a **Value** property. Both the **ID** and **Value** properties store variant data. For example, the **ID** may be set to a variant containing a string, while the **Value** may be an object reference.

ID

You can assign any variant data to an edge **ID** property. Edge IDs need not be unique, but they may be more useful if they are. You can set the IDs when you create the edge or during a separate edit session.

Like the node **ID**, a unique edge **ID** can be very helpful, especially if you need to associate it with the primary key of a relational-database table. If a node represents a device on a network, an edge may represent the cable that connects the networked devices, which also needs a unique asset number. You can set its asset number to the edge **ID**, set the cable name to the **Value**, and associate the node with a row in a table datasource that shares the asset number.

Following this paradigm, you may consider the nodes “active” network components, while the edges are “passive” components. All components in the network are material resources, each requiring an asset number. For cable that is ordered in bulk, an asset number may be used for an entire roll. Edges that represent segments of bulk-ordered cable in the graph datasource may all share a common edge **ID**. Other passive components that are separately ordered, such as cables for peripheral devices and software keys, may have different asset numbers and unique edge IDs.

Value

Edge **Value** properties, like edge IDs, are variant types. They represent the “data” part of the edge contents. You may use the **Value** property to simply elaborate on the relationship between the two nodes at either end of the edge, or you may adopt numerous other uses for it.

Directed

Each edge has a **Directed** property that indicates whether the edge is directed or undirected:

- If the **Directed** property is **TRUE**, the edge connects a parent node to a child node.

- If the **Directed** property is **FALSE**, the edge connects two neighbor nodes.

Any two nodes can have multiple edges relating them. If multiple edges relate a pair of nodes, all of the edges **must** be either directed or undirected. Two nodes **cannot** be related through both directed and undirected edges.

While you can define multiple directed or multiple undirected edges between nodes, you cannot define both directed and undirected edges between a pair of nodes. The node relationship in Figure 4–6 is invalid, because both directed and undirected edges relate the two nodes.

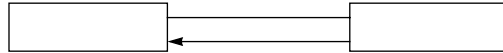


Figure 4–6 Not Supported: Combining Directed and Undirected Edges

Directed Edges

A *directed* edge indicates a *parent-child*, or antecedent, relationship between two nodes. Each directed edge can access, through the `FromNode` reference, to the node whose `OutEdge` reference accesses it. Each directed edge also can access, through the `ToNode` reference, the node whose `InEdge` reference accesses it. For an illustration of this relationship, see “Parent-Child Node Relationship” on page 69.

As shown in Figure 4–7, the parent-child relationship loosely describes a hierarchical node relationship. Because any two nodes may have multiple directed edges defining their relationship, one node may relate to another node as both a parent and a child. However, they may not be related by a combination of directed and undirected edges.

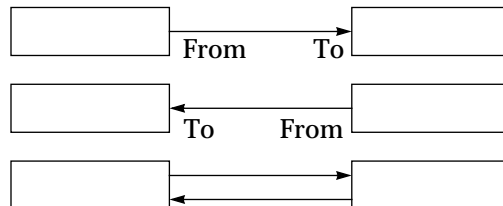


Figure 4–7 Single and Multiple Directed Edges

Although you can change a directed edge to an undirected edge by setting **Directed** to **FALSE**, you *cannot* change the direction of a directed edge. `FromNode` and `ToNode` (Figure 4–3) are determined by the order in which nodes are specified when the edge is added, which determines its direction.

If you need to change the direction of an edge, remove the existing edge using the **RemoveEdgeBetween**(*source*, *target*) function, and add another directed edge using the **AddDirEdge**(*source*, *target*) function from either the **NDVarGr** or **NDVarGrEdit** API. Using the parent-child paradigm, the parent node would be the *source* node.

Undirected Edges

An *undirected edge* indicates a *neighbor*, or *nonhierarchical*, relationship between two nodes. You can create an undirected edge, using the **AddUndirEdge(*node1*, *node2*)** function from either the **NDVarGr** or the **NDVarGrEdit** API, to connect any two nodes that are not already related hierarchically.

Each undirected edge, through the *FromNode* and *ToNode* references, respectively, accesses *node1* and *node2* specified by the **AddUndirEdge()** function. For an illustration of this relationship, see “Neighbor Node Relationship” on page 70.

Figure 4–8 shows the nonhierarchical relationship between neighbor nodes. You can create multiple undirected edges to relate any two nodes just as you can with directed edges. However, with undirected edges, there is no real significance to the *FromNode* and *ToNode* references, except that they determine the order in which the nodes in the datasource are traversed.

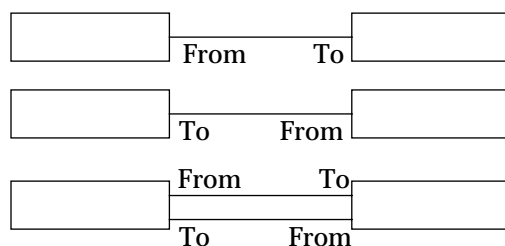


Figure 4–8 Single and Multiple Undirected Edges

For example, when traversing neighboring nodes, the **GoNext()** and **GoPrev()** functions from the **NDVarGrNodeAccessor** API use the *FromNode* and *ToNode* references to move the accessor to the *n*th neighboring node.

If you need to change the traversal order, remove the existing edge using the **RemoveEdgeBetween(*node1*, *node2*)** function, and add another undirected edge using the **AddUndirEdge(*node1*, *node2*)** function from the **NDVarGr** API. If you need to change several edges, use the functions from the **NDVarGrEdit** API to complete the edit operations more efficiently.

Custom Properties

Custom properties are additional properties that you can define. You can create a property and set and get its value using the **NDVarGr** and **NDVarGrEdit** APIs. For more information, see “Custom Link Properties” on page 106.

Graph

A *graph* is a general mathematical abstraction that can represent many data models. Each graph consists of:

- Data represented by node objects
- Edge objects representing the relationships between the nodes

Thus, a graph is a collection of nodes with various relationships between them

A *graph datasource* may contain one or more graphs. It may also be considered a graph, itself. A graph may be either connected or disconnected. In a *connected* graph, a node accessor can traverse all nodes in the graph through a common edge using the `NDVarGrNodeAccessor` API. In a *disconnected* graph, one or more of the nodes is not related to the other nodes through an edge.

These concepts are instrumental in describing graphs:

- Root Node
- Parent-Child Node Relationship
- Neighbor Node Relationship

Nodes may have parents, children, and neighbors. These associations are established using directed and undirected edges. A directed edge indicates a parent-child relationship. An undirected edge indicates a neighbor relationship.

In Figure 4–9, nodes A, C, D, and G are root nodes, because they have no parent nodes. Node A is the parent of node B. Node B has two neighbors, nodes C and D, which are also neighbors to each other. Node D has two children, nodes E and F. In addition to being a root node, node G is also disconnected from the remainder of the graph, because it has no edges.

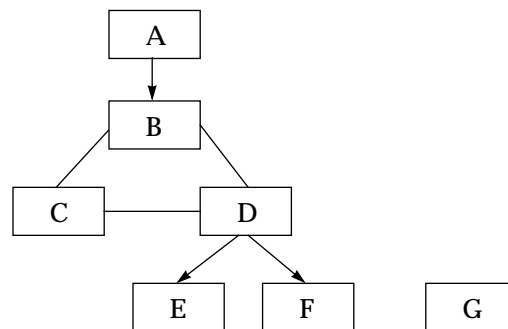


Figure 4–9 Node Relationships in a Graph

For more information about nodes, see “Node” on page 60. For more information about edges, see “Edge” on page 63.

Root Node

A *root node* is a node that has no parent node, but may have child and neighbor nodes. This characteristic differentiates them from all other nodes.

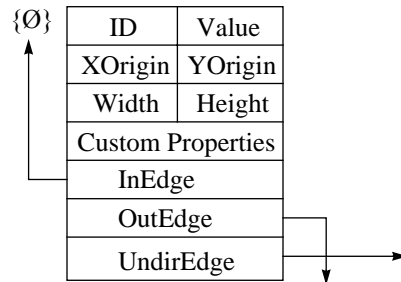


Figure 4-10 Unique Characteristics of a Root Node

Relative to a root node, you can position a *node accessor* to add other root nodes. Root nodes can be related through undirected edges, or they may simply be *disconnected*—inaccessible through a common edge—from the other root nodes.

Note: If you add a directed edge between two root nodes, one of the related nodes becomes a child of the other and is no longer a root node.

Parent-Child Node Relationship

A *parent-child* node relationship is hierarchical or antecedent. You establish such a relationship in a graph datasource through a directed edge. Figure 4-11 shows how the node references in the node-edge-node structure establish the parent-child relationship.

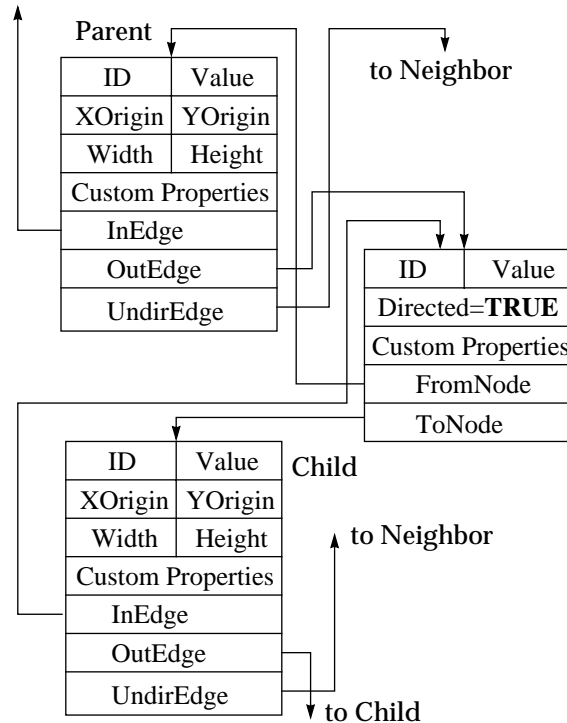


Figure 4-11 Parent-Child Node Relationship

You connect the parent and child nodes through an edge with its **Directed** property set to **TRUE**. For directed edges, **FromNode** identifies the parent node, and **ToNode** identifies the child node.

Using the **NDVarGrNodeAccessor** API, you can instruct a node accessor to move from a parent node to its first child node with the **GoFirstChild()** function. You can then use the **GoNext()** and **GoPrev()** functions to traverse the child nodes of the parent node.

The **GoNext()** function is meaningless unless one of the “GoFirst” or “GoNth” functions executes first. These functions include:

- **GoFirstRoot()** and **GoNthRoot()**
- **GoFirstNeighbor()** and **GoNthNeighbor()**
- **GoFirstParent()** and **GoNthParent()**
- **GoFirstChild()** and **GoNthChild()**

To add a child node after the last child node:

1. Move the accessor to the last valid child node.
2. Execute the **GoNext()** function.

- Using either the **NDVarGr** or **NDVarGrEdit** API, execute the **AddNode()** function.

Because a node can have multiple parents, you can use the **GoNext()** and **GoPrev()** functions to traverse the parent nodes of a child node. You can also add new parents.

Neighbor Node Relationship

Two nodes that share an undirected edge are *neighbors*. Neighbor nodes may or may not share a common parent. Figure 4-12 shows the node-edge-node relationship of two neighbors.

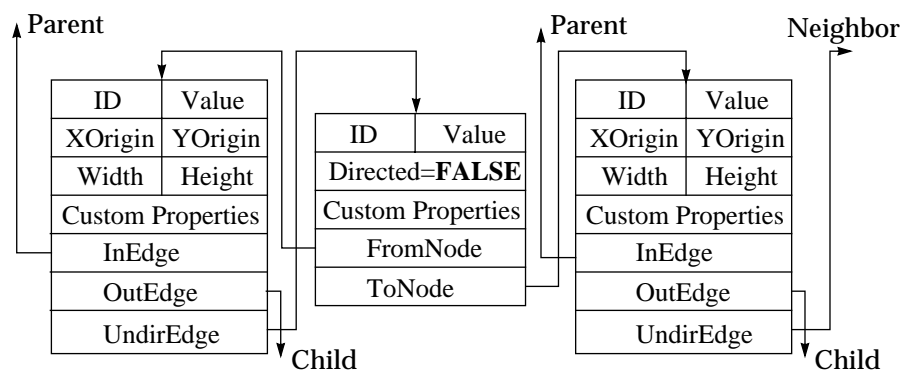


Figure 4-12 Neighbor Node Relationship

The two nodes in Figure 4-12 are connected through an edge with its **Directed** property set to **FALSE**. For undirected edges, **FromNode** and **ToNode** define the order in which nodes are traversed, but do not imply a direction.

Using the **NDVarGrNodeAccessor** API, you can instruct a node accessor to move from a node to its first neighbor with the **GoFirstNeighbor()** function. You can then use the **GoNext()** and **GoPrev()** functions to traverse the neighbor nodes.

The **GoNext()** function is meaningless unless one of the “GoFirst” or “GoNth” functions executes first. These functions include:

- **GoFirstRoot()** and **GoNthRoot()**
- **GoFirstNeighbor()** and **GoNthNeighbor()**
- **GoFirstParent()** and **GoNthParent()**
- **GoFirstChild()** and **GoNthChild()**

To add a neighbor node:

- Position the accessor on a valid node.
- Execute the **GoNext()** function.
- Using the **NDVarGr** or **NDVarGrEdit** API, execute the **AddNode()** function.

Accessor

An *accessor* is an index mechanism by which you can traverse the nodes and edges in the graph datasource. You cannot access nodes or edges directly,

therefore you **must** use accessors to access them. You must also use accessors to identify any node or edge to be modified by an edit operation.

The graph datasource supplies two basic accessor types:

- Node Accessor
- Edge Accessor

There are four types of edge accessors, which give you added flexibility and provide optimal navigational performance for your application.

The graph datasource also support a node cursor and an edge cursor. For more information about cursors, see “Cursor” on page 73.

Node Accessor

A *node accessor* is a node index mechanism that references and traverses the nodes in the graph datasource. You cannot access the nodes directly, therefore you **must** use a node accessor to access them. You must also use accessors to identify the node in a node-level edit operation.

You need at least one node accessor to traverse—using the **NDVarGrNodeAccessor** API—the nodes in a graph datasource. After moving the node accessor to the appropriate node in the graph, your application can modify either the datasource structure or the properties of the nodes it contains.

In many cases, you need two node accessors to identify the endpoints of an edge relating a pair of nodes. This code fragment shows how to create and destroy two node accessors:

```
VarGrPtr mGraphDs = new NDVarGr;

// Declare node-accessor pointers and assign node-accessor
// objects to each of them.
VarGrNodeAccessorPtr mNodeAccessorFrom =
    new NDVarGrNodeAccessor;
VarGrNodeAccessorPtr mNodeAccessorTo =
    new NDVarGrNodeAccessor;
...
// Destroy the node accessors.
delete mNodeAccessorFrom;
delete mNodeAccessorTo;
...
// Destroy the graph datasource.
delete mGraphDs;
```

The preceding code fragment declares two accessors—`mNodeAccessorFrom` and `mNodeAccessorTo`—because two nodes are required to define an edge between them.

Edge Accessor

An *edge accessor* is an edge index mechanism that references and traverses the edges in the graph datasource. You cannot access edges directly, therefore you **must** use an edge accessor to access them. There are three types of edges:

- “In” edges
- “Out” edges
- Undirected edges

These three edge-accessor APIs support the preceding respective edge types:

- **NDVarGrInEdgeAccessor**
- **NDVarGrOutEdgeAccessor**
- **NDVarGrUndirEdgeAccessor**

Use the API functions supplied with these accessor objects to traverse and edit the edges in the datasource. APIs for “in,” “out,” and undirected edge accessors pertain only to edges of the node referenced by the node accessor that defined the edge accessor.

In addition to the type-specific accessor APIs, your application can traverse all of the edges in the graph datasource using the functions supplied with the universal **NDVarGrAllEdgeAccessor** API. “All” edge accessors do *not* require a node accessor when they are created; these pertain to the entire graph datasource.

This code fragment shows how to create an undirected edge between two nodes and how to set the edge **Value** property:

```
VarGrPtr mGraphDs = new NDVarGr;
VarPtr mVarValue = new NDVar;

// Declare a datasource edit pointer and assign an edit object
// to it.
VarGrEditPtr mEditGraphDs = mGraphDs->StartEdit();

// Declare two node accessors for the two nodes at either end of
// of the edge to be added. Assign node-accessor objects to
// each of them.
VarGrNodeAccessorPtr mNodeAccessorFrom =
    new NDVarGrNodeAccessor;
VarGrNodeAccessorPtr mNodeAccessorTo =
    new NDVarGrNodeAccessor;

// Declare an undirected edge-accessor pointer and assign an
// undirected edge accessor to it based on the node accessed by
// mNodeAccessorFrom.
VarGrUndirEdgeAccessorPtr mUndirEdgeAccessor =
    new NDVarGrUndirEdgeAccessor(mNodeAccessorFrom);

// Create several root nodes.
...
// Move mNodeAccessorFrom to the first root node, and move
// mNodeAccessorTo to the second root node.
mNodeAccessorFrom->GoFirstRoot();
mNodeAccessorTo->GoFirstRoot();
mNodeAccessorTo->GoNext();

// Add an undirected edge between the nodes accessed by
// mNodeAccessorFrom and mNodeAccessorTo.
mEditGraphDs->AddUndirEdge(mNodeAccessorFrom,
                           mNodeAccessorTo);

// Use mUndirEdgeAccessor to set the Value property of the new
// edge to "Neighbor."
mVarValue->SetStr("Neighbor");
mEditGraphDs->SetEdgeValue(mUndirEdgeAccessor,
                           mVarValue);
mEditGraphDs->End();
```

In the preceding example:

1. The creation of several root nodes is implied.
2. Two node accessors are created to access the first two root nodes.
3. An undirected edge accessor is declared, using **mNodeAccessorFrom** as an argument.

4. An undirected edge is defined using the two node accessors.
5. The variant, "Neighbor," is assigned to the **Value** property referenced by the undirected edge accessor.

Cursor

The graph datasource supports two types of *cursor*:

- *Node Cursor*
- *Edge Cursor*

The node cursor and edge cursor are properties of the graph datasource. Like the **Title** property, you can set and get the node and edge cursors.

When a **NDDGram** view is registered with the graph datasource, you can set an option to cause the view either control the datasource cursor or simply reflect the current location of the datasource cursor as it traverses the internal hierarchy.

Node Cursor

The node cursor is a property of the graph datasource. You can:

- Set the node cursor by associating it with a node accessor using the **NDVarGr::SetNodeCursor(*nodeaccessor*)** function.
- Access the node at the current cursor location using the **NDVarGr::GetNodeCursor()** function.

This code fragment shows how to set and get a node cursor:

```
// Declare a graph-datasource pointer variable, and assign a
// graph-datasource object to it.
VarGrPtr mGraphDs = new NDVarGr;

// Declare a browser pointer variable, and assign a browser
// object to it.
DGramPtr mDGRAMWgt = (NDDGramPtr)GetNamedWgt("DGRAM");

// Register the diagrammer with the graph datasource, and set
// the "cursor" view option.
mGraphDs->RegisterView(mDGRAMWgt);
mGraphDs->SetViewOption(mDGRAMWgt, "cursor", "CONTROLS");

// Declare a node-accessor pointer variable, and assign a
// node-accessor object to it.
VarGrNodeAccessorPtr mNodeAccessor = new NDVarGrNodeAccessor;

// Declare two variant pointer variables, and assign variant
// objects to them.
VarPtr mVarID = new NDVar;
VarPtr mVarValue = new NDVar;

// Set a cursor at the location of the node accessor.
mGraphDs->SetNodeCursor(mNodeAccessor);

// Position the node accessor.
...
// Use "convenience" API functions to edit the ID and Value
// properties of the node at the current cursor location.
mVarID->SetStr("0000");
mVarValue->SetStr("Node");
mGraphDs->SetNodeID(mGraphDs->GetNodeCursor(), mVarID);
mGraphDs->SetNodeValue(mGraphDs->GetNodeCursor(), mVarValue);
...
// Destroy the variant objects.
delete mVarID;
```

```

delete mVarValue;
...
// Destroy the node accessor.
delete mNodeAccessor;
...
// Destroy the graph datasource.
delete mGraphDs;

```

For information about setting the cursor behavior, see “Options for the NDDGram View” on page 80.

Edge Cursor

The edge cursor is a property of the graph datasource. You can:

- Set the node cursor by associating it with a node accessor using the **NDVarGr::SetEdgeCursor(*edgeaccessor*)** function.
- Access the node at the current cursor location using the **NDVarGr::GetEdgeCursor()** function.

This code fragment shows how to set and get an edge cursor:

```

// Declare a graph-datasource pointer variable, and assign a
// graph-datasource object to it.
VarGrPtr mGraphDs = new NDVarGr;

// Declare a browser pointer variable, and assign a browser
// object to it.
DGramPtr mDGRAMWgt = (NDDGramPtr)GetNamedWgt("DGRAM");

// Register the diagrammer with the graph datasource, and set
// the "cursor" view option.
mGraphDs->RegisterView(mDGRAMWgt);
mGraphDs->SetViewOption(mDGRAMWgt, "cursor", "CONTROLS");

// Declare an edge-accessor pointer variable, and assign an
// edge-accessor object to it.
VarGrAllEdgeAccessorPtr mAllEdgeAccessor =
    new NDVarGrAllEdgeAccessor;

// Declare two variant pointer variables, and assign variant
// objects to them.
VarPtr mVarID = new NDVar;
VarPtr mVarValue = new NDVar;

// Set a cursor at the location of the node accessor.
mGraphDs->SetEdgeCursor(mAllEdgeAccessor);

// Position the edge accessor.
...
// Use "convenience" API functions to edit the ID and Value
// properties of the edge at the current cursor location.
mVarID->SetStr("0000");
mVarValue->SetStr("Edge");
mGraphDs->SetEdgeID(mGraphDs->GetEdgeCursor(), mVarID);
mGraphDs->SetEdgeValue(mGraphDs->GetEdgeCursor(), mVarValue);
...
// Destroy the variant objects.
delete mVarID;
delete mVarValue;
...
// Destroy the node accessor.
delete mNodeAccessor;
...
// Destroy the graph datasource.
delete mGraphDs;

```

For information about setting the cursor behavior, see “Options for the NDDGram View” on page 80.

Edit Object

To perform edit operations on the graph datasource or the nodes it contains, your application must use an *edit object*. The graph datasource uses edit objects to:

- Create working copies of the data
- Protect the datasource from corruption resulting from simultaneous editing sessions sharing a common datasource

The graph datasource supports these editing levels:

- Datasource Editing
- Node Editing
- Edge Editing

If the data to be modified is locked by another view, no edit object can be created. This locks your application out of the data. To prevent your application from hanging when it encounters a data lock, you can create your edit object within a conditional construct that checks for the availability of the data and supplies an alternative if the data is locked.

Editing the datasource includes the following four steps:

1. Create an edit object
2. Execute the edit operations
3. Commit the edit operations
4. Destroy the edit object

In addition to the direct approach to managing edit objects, a set of “convenience” APIs supplies functions that manage the edit objects automatically for single edit operations. For more information about the “convenience” APIs, see “Convenience API Functions” on page 77.

Datasource Editing

When you want to modify the structure of the datasource—for example, to create new nodes and edges—you need a *datasource edit object*. When you create a datasource edit object for a particular view, no other view can create an edit object for that datasource. This includes edit objects for editing node and edge data, because the node or edge you may want to edit may also be edited during the datasource-level edit session.

The datasource edit object is created, locking the datasource, when an **NDVarGr** object executes the **StartEdit()** function. This is a public function inherited from the **NDDs** class. The graph datasource is unlocked when the **NDDsEdit::End()** function executes, as shown in this example:

```
VarGrPtr mGraphDs = new NDVarGr;

// Declare and assign node and edge accessors.
VarGrNodeAccessorPtr mNodeAccessorFrom =
    new NDVarGrNodeAccessor;
VarGrNodeAccessorPtr mNodeAccessorTo =
    new NDVarGrNodeAccessor;
VarGrAllEdgeAccessorPtr mAllEdgeAccessor =
    new NDVarGrAllEdgeAccessor;

// Create the datasource-level edit object.
VarGrEditPtr mEditGraphDs = mGraphDs->StartEdit();

// Position the node accessor and edit the graph.
```

```
mGraphDs->SetNodeCursor(mNodeAccessorFrom);
mGraphDs->CreateOutEdge(GetNodeCursor(), mNodeAccessorTo);
...
mEditGraphDs->End();
```

When the **DsEdit::End()** function executes, all graph modifications are committed, and the datasource-level lock is released.

Node Editing

To set the data properties of a node in a graph datasource—for example, to update its x and y coordinates—you do *not* need to lock the entire datasource from access by other views with a datasource edit object. Instead, you only need to lock the node that you want to modify.

A *node edit object* is created, locking the node referenced by the node accessor, when an object of the **NDVarGr** class executes the **StartNodeEdit(accessor)** function. This is a public function inherited from the **NDDs** class. The accessed node is unlocked when the **DsEdit::End()** function executes, as shown in this example:

```
VarGrPtr mGraphDs = new NDVarGr;
VarGrNodeAccessorPtr mNodeAccessor = new NDVarGrNodeAccessor;

// Execute the StartNodeEdit() function.
VarGrNodeEditPtr mEditNode =
    mGraphDs->StartNodeEdit(mNodeAccessor);

// Position the node accessor and edit node data.
...
mEditNode->End();
```

When the **DsEdit::End()** function executes, all node modifications are committed, and the node-level lock is released.

Edge Editing

To set the data properties of an edge in a graph datasource—for example, to change its label—you do *not* need to lock the entire datasource from access by other views with a datasource edit object. Instead, you only need to lock the edge that you want to modify.

An *edge edit object* is created, locking the edge referenced by the edge accessor, when an object of the **NDVarGr** class executes the **StartEdgeEdit(accessor)** function. This is a public function inherited from the **NDDs** class. The accessed node is unlocked when the **DsEdit::End()** function executes, as shown in this example:

```
VarGrPtr mGraphDs = new NDVarGr;
VarGrAllEdgeAccessorPtr mAllEdgeAccessor =
    new NDVarGrNodeAccessor;

// Execute the StartEdgeEdit() function.
VarGrEdgeEditPtr mEditEdge =
    mGraphDs->StartEdgeEdit(mEdgeAccessor));

// Position the edge accessor and edit edge data.
...
mEditEdge->End();
```

When the **DsEdit::End()** function executes, all edge modifications are committed, and the edge-level lock is released.

Convenience API Functions

When editing a graph datasource, you can use either the standard APIs or the convenience APIs to complete the edit operations. When using the standard APIs, you must:

1. Create an edit object to start the edit operation.
2. Perform any necessary editions to the datasource.
3. Commit the edit operations.
4. Destroy the edit object.

When using the “convenience” APIs, steps 1, 3, and 4 from the preceding list are completed automatically. You can perform:

- Datasource Editing with the “Convenience” APIs
- Node Editing with the “Convenience” APIs
- Edge Editing with the “Convenience” APIs

In other words, your application can use the “convenience” API to edit the datasource or its contents without formally creating an edit object. For example, when the `NDVarGr::AddNode(accessor)` function executes:

- An edit object is automatically created
- The new node is added at the location specified by the node accessor
- The edit operations are committed
- The edit object is destroyed

The “convenience” API functions are useful for performing single edit operations. However, these functions can inhibit performance when used to perform batch edit operations.

Datasource Editing with the “Convenience” APIs

If you want to change the ID and Value properties of a specific node in the datasource, the “convenience” API functions are most useful. To directly manage the required edit object would add some unnecessary complexity to your application logic.

In the next example, a datasource edit object is create by the “convenience” function, `NDVarGr::AddNode()`. This creates a datasource edit object, adds a node, commits the node addition to the datasource, and destroys the edit object.

```
// Declare pointer variables and assign objects to them.
VarGrPtr mGraphDs = new NDVarGr;
VarGrNodeAccessorPtr mNodeAccessor = new NDVarGrNodeAccessor;

// Move the node accessor to the next empty root-node location.
mNodeAccessor->GoFirstRoot();
while (mVarGr->IsValid(mNodeAccessor)) {
    mNodeAccessor->GoNext();
}

// Add a node using the "convenience" API. A datasource edit
// object is created, edit operations are committed, and the
// edit object is destroyed by the mGraphDs->AddNode()
// function.
mGraphDs->AddNode(mNodeAccessor);
...
// Dispose of other objects.
delete mVarID;
delete mVarValue;
```

```
delete mNodeAccessor;
delete mGraphDs;
```

If the preceding code fragment was intended to build a complete node network, the “convenience” API functions would not be appropriate. For such operations, use batched edit operations as described in “Datasource Editing” on page 75.

Node Editing with the “Convenience” APIs

If you want to change the ID and Value properties of a specific node in the datasource, the “convenience” API functions are most useful. To directly manage the required edit object would add some unnecessary complexity to your application logic.

In the next example, a node edit object is create by each of the convenience API functions, `NDVarGr::SetNodeID()` and `NDVarGr::SetNodeValue()`. Each of these functions creates a node edit object, commits its edit operation, and destroys the edit object.

```
// Declare pointer variables and assign objects to them.
VarGrPtr mGraphDs = new NDVarGr;
VarGrNodeAccessorPtr mNodeAccessor = new NDVarGrNodeAccessor;
VarPtr mVarID = new NDVar;
VarPtr mVarValue = new NDVar;

// Move the node accessor to the next empty root-node location.
mNodeAccessor->GoFirstRoot();
while (mVarGr->IsNodeValid(mNodeAccessor)) {
    mNodeAccessor->GoNext();
}

// Add a node using the "convenience" API. A datasource edit
// object is created, edit operations are committed, and the
// edit object is destroyed by the mGraphDs->AddNode()
// function.
mGraphDs->AddNode(mNodeAccessor);

// Set the variant objects to some initializing values.
mVarID->SetStr("0000");
mVarValue->SetStr("New Node");

// Set the node ID and Value properties using the "convenience"
// APIs. A node edit object is created, edit operations are
// committed, and the edit objects are destroyed by each of the
// following two functions.
mGraphDs->SetNodeID(mNodeAccessor, mVarID);
mGraphDs->SetNodeValue(mNodeAccessor, mVarValue);
...
// Dispose of other objects.
delete mVarID;
delete mVarValue;
delete mNodeAccessor;
delete mGraphDs;
```

If the preceding code fragment was intended to traverse and initialize each node in the hierarchy, the “convenience” API functions would not be appropriate. For such operations, use batched edit operations as described in “Node Editing” on page 76.

Edge Editing with the “Convenience” APIs

If you want to change the ID and Value properties of a specific edge in the datasource, the “convenience” API functions are most useful. To directly manage the required edit object would add some unnecessary complexity to your application logic.

In the next example, an edge edit object is created by each of the convenience API functions, `NDVarGr::SetEdgeID()` and `NDVarGr::SetEdgeValue()`. Each of these functions creates an edge edit object, commits its edit operation, and destroys the edit object.

```
// Declare pointer variables and assign objects to them.
VarGrPtr mGraphDs = new NDVarGr;
VarGrNodeAccessorPtr mNodeAccessorFrom =
    new NDVarGrNodeAccessor;
VarGrNodeAccessorPtr mNodeAccessorTo =
    new NDVarGrNodeAccessor;
VarGrAllEdgeAccessorPtr mAllEdgeAccessor =
    new NDVarGrAllEdgeAccessor();
VarPtr mVarID = new NDVar;
VarPtr mVarValue = new NDVar;

// Move the "source" node accessor to the next empty root-edge
// location.
mNodeAccessorFrom->GoFirstRoot();
while (mVarGr->IsValid(mNodeAccessorFrom)) {
    mNodeAccessorFrom->GoNext();
}

// Add a node using the "convenience" API at the "source"
// accessor location. A datasource edit object is created, edit
// operations are committed, and the edit object is destroyed
// by the mGraphDs->AddNode() function.
mGraphDs->AddNode(mNodeAccessorFrom);

// Set the variant objects to some initializing values.
mVarID->SetStr("n0000");
mVarValue->SetStr("New Node");

// Set the node ID and Value properties using the "convenience"
// APIs. A node edit object is created, edit operations are
// committed, and the edit objects are destroyed by each of the
// following two functions.
mGraphDs->SetNodeID(mNodeAccessorFrom, mVarID);
mGraphDs->SetNodeValue(mNodeAccessorFrom, mVarValue);

// Move the "target" node accessor to the next empty root-edge
// location.
mNodeAccessorTo->GoNthRoot(mGraphDs->GetNumRoots());
while (mVarGr->IsValid(mNodeAccessorTo)) {
    mNodeAccessorTo->GoNext();
}

// Add a node using the "convenience" API at the "target"
// accessor location. A datasource edit object is created, edit
// operations are committed, and the edit object is destroyed
// by the mGraphDs->AddNode() function.
mGraphDs->AddNode(mNodeAccessorTo);

// Set the node ID and Value properties using the "convenience"
// APIs. A node edit object is created, edit operations are
// committed, and the edit objects are destroyed by each of the
// following two functions.
mGraphDs->SetNodeID(mNodeAccessorTo, mVarID);
mGraphDs->SetNodeValue(mNodeAccessorTo, mVarValue);

// Add a directed edge between the two root nodes using the
// "convenience" API. A datasource edit object is created, edit
// operations are committed, and the edit object is destroyed
// by the mGraphDs->AddEdge() function.
mGraphDs->AddDirEdge(mNodeAccessorFrom, mNodeAccessorTo);

// Set the variant objects to some initializing values.
mVarID->SetStr("e0000");
mVarValue->SetStr("New Edge");

// Set the edge ID and Value properties using the "convenience"
```

```

// APIs. An edge edit object is created, edit operations are
// committed, and the edit objects are destroyed by each of the
// following two functions.
mAllEdgeAccessor->GoBetween(mNodeAccessorFrom,
                           mNodeAccessorTo);
mGraphDs->SetEdgeID(mAllEdgeAccessor, mVarID);
mGraphDs->SetEdgeValue(mAllEdgeAccessor, mVarValue);
...
// Dispose of other objects.
delete mVarID;
delete mVarValue;
delete mNodeAccessorFrom;
delete mNodeAccessorTo;
delete mAllEdgeAccessor;
delete mGraphDs;

```

If the preceding code fragment was intended to traverse and initialize each edge in the datasource, the “convenience” API functions would not be appropriate. For such operations, use batched edit operations as described in “Edge Editing” on page 76.

Options for the NDDGram View

The graph datasource supports the general-purpose options for **NDDGram** views:

- **autosize**
- **cursor**
- **readonly**
- **Diagrammer**
- **Custom Node and Link Options**

For example, the **Diagrammer** option accepts a parameter string that applies to the entire **NDDGram** view. You can also define node and link properties to which you can assign values.

To set view options, use the second and third arguments of the **NDDs::SetViewOption()** argument list, as shown here:

```

<datasource>->SetViewOption(<view>,
                           [{"autosize", "{FALSE|TRUE}"}|
                            [{"cursor", "{CONTROLS|REFLECTS}"}|
                             {"readonly", "{FALSE|TRUE}"}|
                             {"Diagrammer", "<parameter_list>"}|
                             [{"<node_property>:<value>", "<parameter_list>"}|
                              [{"<link_property>:<value>", "<parameter_list>"}]
                            ]
                           );

```

In the preceding syntax, *parameter_list* is an expression that represents a series of pertinent parameter settings. A parameter list is enclosed in quotation marks, can have any number of parameter settings within it, and follows this format:

```

"<parameter_1>=<value_1>;
 <parameter_2>=<value_2>;
 ...;
 <parameter_n>=<value_n>"

```

You can only use parameter lists when setting node and link parameters with:

- The **Diagrammer** option

- Any options you may define using the *node_property:value* and *link_property:value* formats

Table 1-1 lists the types and possible values for each of these parameters.

Table 4-1 Parameter Values for **Diagrammer** and Custom Options

Type	Values	Default Value
Boolean	0 FALSE NO OTHERS 1 TRUE YES	See Option or Parameter
NodeShape	0 DEFAULT RECT RECTANGLE 1 ROUNDRECT ROUNDRECTANGLE 2 ELLIPSE 3 DIAMOND 4 HEXAGON 5 TRIANGLE	RECTANGLE
LinkShape	DEFAULT DIAGONAL RIGHTANGLE	DIAGONAL
Pen	Pen.<pen_name> <module_name>.<pen_name>	NDDGram-specific
Font	Font.<font_name> <module_name>.<font_name>	NDDGram-specific
Color	Color.<color_name> <module_name>.<color_name>	NDDGram-specific

This `SetViewOption()` function illustrates the parameter values listed in Table 1-1:

```
mGraphDs->SetViewOption(mDGRAMWgt,
    "Diagrammer",
    "NodeW = 72; NodeH = 18;
    XGrid = 36; YGrid = 36;
    GridAlignment = TRUE;
    BgColor = Color.Black;

    NodeStandardDDData.Shape = ROUNDRECTANGLE;
    NodeStandardDDData.BgColor = Color.Navy;
    NodeFocusDDData.BgColor = Color.Blue;
    NodeStandardDDData.LabelColor = Color.White;
    NodeStandardDDData.FramePen = Pen.Solid;
    NodeStandardDDData.LabelFont = Font.Arias;

    Orientation = VERTICAL;
    LinkStandardDDData.Shape = RIGHTANGLE;
    LinkStandardDDData.LinkDirColor = Color.Blue;
    LinkFocusDDData.LinkDirColor = Color.Red;
    LinkStandardDDData.LinkUndirColor = Color.Navy;
    LinkFocusDDData.LinkUndirColor = Color.Maroon;
    LinkStandardDDData.LinkPen = Pen.Solid;
    LinkStandardDDData.LinkLabel = Font.Arias;
    "
);
```

autosize

You can set the **autosize** option to **TRUE** to create automatically sized nodes. With **autosize** enabled, bounding boxes for all nodes in a specified expansion level are the maximum width for nodes at that level. Here is the syntax for using the **autosize** option:

```
<datasource>->SetViewOption(<view>,
                             "autosize", "{FALSE|TRUE}");
```

The **autosize** default is **FALSE**. With **autosize** set to **TRUE**, the bounding-box widths are based on the string lengths of the node **Value** properties. This code fragment shows how to enable the **autosize** option:

```
VarGrPtr mGraphDs;
DGraphPtr mDGraphWgt = (NDDGraphPtr)GetNamedWgt("DGraph");
...
mGraphDs->RegisterView(mDGraphWgt)
mGraphDs->SetViewOption(mDGraphWgt, "autosize", "TRUE");
```

CURSOR

The **cursor** option determines whether the view cursor controls or reflects the position of the datasource cursor. The **cursor** option has two possible settings:

- **CONTROLS** (the default)
- **REFLECTS**

Here is the format for the setting the **cursor** option:

```
<datasource>->SetViewOption(<view>,
                             "cursor", "{CONTROLS|REFLECTS}");
```

With **cursor** set to **CONTROLS**, the cursor position or active node in the view determines the position of the datasource cursor. This ensures that the datasource cursor and view cursor are synchronized.

With **cursor** set to **REFLECTS**, the view cursor reflects the current location of the datasource cursor. This setting ensures that the view is continually updated when the datasource cursor is moved programmatically. When you change the view cursor, the datasource cursor is not updated.

When multiple views are registered with a common datasource and with **cursor** set to **CONTROLS**, each registered view can manipulate the position of the datasource cursor. For example, if two views control the datasource cursor, movement of one view cursor changes the position of the datasource cursor, which is reflected in the other registered view.

This example has two **NDDGraph** views, **mDGraph1** and **mDGraph2**, registered to a common graph datasource, **mGraphDs**. The **mDGraph1** cursor **reflects** the current location of the datasource cursor. The **mDGraph2** cursor **controls** the position of the datasource cursor:

```
VarGrPtr mGraphDs;
DGraphPtr mDGraph1;
DGraphPtr mDGraph2;
mGraphDs->RegisterView(mDGraph1)
mGraphDs->RegisterView(mDGraph2)
...
mGraphDs->SetViewOption(mDGraph2,
                        "cursor", "REFLECTS");
mGraphDs->SetViewOption(mDGraph2,
                        "cursor", "CONTROLS");
```

In the next example, both view cursors control the position of the datasource cursor. Any change in the cursor position of one view is automatically reflected in the other view.

```
VarGrPtr mGraphDs;
DGraphPtr mDGraph1;
DGraphPtr mDGraph2;
mGraphDs->RegisterView(mDGraph1)
mGraphDs->RegisterView(mDGraph2)
```

```

...
mGraphDs->SetViewOption(mDGRAM2,
                        "cursor", "CONTROLS");
mGraphDs->SetViewOption(mDGRAM2,
                        "cursor", "CONTROLS");

```

readonly

When **readonly** is **FALSE** (the default), you can right-click on a node or link to display a popup menu. This lets you access the Node Edition and Link Edition dialogs. In these, you can change the **ID**, **Value**, **Width**, or **Height** of the node or the **Value** property or **Directed** property of a link.

```

<datasource>->SetViewOption(<view>,
                            "readonly", "{FALSE|TRUE}");

```

With **readonly** set to **TRUE**, the application user *cannot* right-click to display the popup menu and, therefore, cannot access the Node and Link Edition dialogs.

Diagrammer

You can use the **Diagrammer** option to set many different parameters:

- Basic Diagrammer Parameters
- “Standard” View Settings for Nodes and Links
- “Focus” View Settings for Nodes and Links

These parameters have varied scopes. As a result, only a general syntax for the **Diagrammer** option is shown here:

```

<datasource>->SetViewOption(<view>,
                            "Diagrammer",
                            "[<basic_parameter_list>] |
                             [<standard_node_and_link_parameter_list>] |
                             [<focus_node_and_link_parameter_list>]"
                            );

```

A complete parameter list for each scope is supplied in the sections that follow.

Basic Diagrammer Parameters

With the basic Diagrammer parameters, you can set these diagram characteristics:

- Dimensions of the node bounding box
- Grid and snap
- Magnification
- Overview configuration
- Link orientation
- Background

Here is the syntax supporting these characteristics:

```

<datasource>->SetViewOption(<view>,
                            "Diagrammer",
                            "[NodeW = <integer>;] |
                             [NodeH = <integer>;] |
                             [XGrid = <integer>;] |
                             [YGrid = <integer>;] |
                             [GridAlignment = {FALSE|TRUE};] |
                             [ScaleFactor = <real>;] |
                             [Overview = {0|DEFAULT|NO|HIDE|NOOVERVIEW} |
                             1|TOP]";

```

```

                2|LEFT
                };] |
[Orientation =
                {0|DEFAULT|HORZ|HORIZONTAL},
                1|VERT|VERTICAL
                };] |
[Cycles = {FALSE|TRUE};] |
[BitmapFile, <filename>;] |
[BgColor = <color_resource>;]"
);

```

NodeW

This integer value specifies the width of a node in pixels. If the **Autosize** option is enabled, this option and the **NodeH** parameter are ignored.

This syntax shows how to specify the node width:

```

<datasource>->SetViewOption(<view>,
    "Diagrammer",
    "...";
    NodeW = <integer>;
    "...")
);

```

This code fragment sets the node width to 72 pixels—that is, 1 inch on a 72-pixel-per-inch display:

```

mGraphDs->SetViewOption(mDGRAMWGT,
    "Diagrammer", "NodeW = 72");

```

Type: **Int32**

Default: **NDDGram**-specific

Synonyms: **Width**

NodeH

This integer value specifies the height of a node in pixels. If the **Autosize** option is enabled, this option and the **NodeW** parameter are ignored.

This syntax shows how to specify the node height:

```

<datasource>->SetViewOption(<view>,
    "Diagrammer",
    "...";
    NodeH = <integer>;
    "...")
);

```

This code fragment sets the node height to 24 pixels—that is, 1/3 inch on a 72-pixel-per-inch display:

```

mGraphDs->SetViewOption(mDGRAMWGT,
    "Diagrammer", "NodeH = 24");

```

Type: **Int32**

Default: **NDDGram**-specific

Synonyms: **Height**

XGrid

This integer value specifies the size of the *x*-axis grid in pixels. This syntax shows how to specify the *x*-axis grid:

```

<datasource>->SetViewOption(<view>,
    "Diagrammer",

```

```

    "...;
    XGrid = <integer>;
    ..."
);

```

This code fragment sets the x -axis grid to 36 pixels—that is, 1/2 inch on a 72-pixel-per-inch display:

```

mGraphDs->SetViewOption(mDGRAMWgt,
    "Diagrammer", "XGrid = 36");

```

Type: Int32

Default: **NDDGram**-specific

Synonyms: None

YGrid

This integer value specifies the size of the y -axis grid in pixels. This syntax shows how to specify the y -axis grid:

```

<datasource>->SetViewOption(<view>,
    "Diagrammer",
    "...;
    YGrid = <integer>;
    ..."
);

```

This code fragment sets the y -axis grid to 36 pixels—that is, 1/2 inch on a 72-pixel-per-inch display:

```

mGraphDs->SetViewOption(mDGRAMWgt,
    "Diagrammer", "YGrid = 36");

```

Type: Int32

Default: **NDDGram**-specific

Synonyms: None

GridAlignment

With **GridAlignment** set to **FALSE** (the default), you can move nodes and links in an unconstrained manner within the **NDDGram** view. If you set **GridAlignment** to **TRUE**, the diagram contents snap to the grid. This syntax shows how to set the **GridAlignment** parameter:

```

<datasource>->SetViewOption(<view>,
    "Diagrammer",
    "...;
    GridAlignment = {FALSE|TRUE};
    ..."
);

```

This code fragment sets both the x -axis and y -axis grid to 36 pixels—that is, a grid of 1/2-inch squares on a 72-pixel-per-inch display—with diagram components snapping to the grid:

```

mGraphDs->SetViewOption(mDGRAMWgt,
    "Diagrammer", "YGrid = 36; XGrid = 36;
    GridAlignment = TRUE");

```

In the preceding example, the grid may or may not be visible.

Type: Boolean

Default: **FALSE**

Synonyms: **Align**, **Alignment**

ScaleFactor

The **ScaleFactor** parameter controls the zoom level of the diagram. The default for this parameter is 1.00. This syntax shows how to set the **ScaleFactor** parameter:

```
<datasource>->SetViewOption(<view>,
    "Diagrammer",
    "...;
    ScaleFactor = <double>;
    ..."
);
```

Note: The **ScaleFactor** setting is assumed to be 1.00 when you use pixels for dimensioning **NodeW**, **NodeH**, **XGrid**, and **YGrid**.

This code fragment sets the **ScaleFactor** to 2.00, which displays the diagram at twice its default size:

```
mGraphDs->SetViewOption(mDGRAMWGT,
    "Diagrammer", "ScaleFactor = 2.00");
```

Type: Double

Default: 1.00

Synonyms: **Scale**

Overview

The **NDDGRAM** overview is a high-level view of the entire graph. With the overview, you can position a virtual view window, rather than using scroll bars, to display the specific nodes and edges (links). This syntax shows how to set the **Overview** parameter:

```
<datasource>->SetViewOption(<view>,
    "Diagrammer",
    "...;
    Overview = {0|DEFAULT|NO|HIDE|NOOVERVIEW|
                1|TOP|
                2|LEFT
                };
    ..."
);
```

This code fragment enables an **Overview** display to the left of the diagram:

```
mGraphDs->SetViewOption(mDGRAMWGT,
    "Diagrammer", "Overview = LEFT");
// "Diagrammer", "Overview = 2");
```

Type: Enumerated

Default: **NOOVERVIEW**

Synonyms: None

Orientation

With **Orientation** set to **HORIZONTAL** (the default), a link emanates from the “center” of the node through the sides of the bounding box. If you set **Orientation** to **VERTICAL**, the links emanate from the bottom and top sides of the nodes. This syntax shows how to set the **Orientation** parameter for a center link:

```
<datasource>->SetViewOption(<view>,
    "Diagrammer",
    "...;
    Orientation = {0|DEFAULT|HORZ|HORIZONTAL},
```



```

        1 | VERT | VERTICAL } ;
    };
    ... "
);

```

This code fragment sets the **Orientation** to have the links emanate from the top and bottom sides of the pertinent nodes:

```

mGraphDs->SetViewOption(mDGRAMWgt,
    "Diagrammer", "Overview = VERTICAL");
// "Diagrammer", "Orientation = 1");

```

Type: Enumerated

Default: **HORIZONTAL**

Synonyms: None

Cycles

With **Cycles** set to **TRUE** (the default is **FALSE**), a warning is returned when a cyclic node reference is created. Two cyclic references are shown in Figure 4-13:

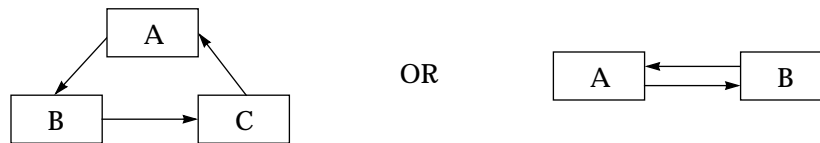


Figure 4-13 Cyclic Node Reference

If you are planning an itinerary for a trip on which you plan to return to your original departure point, a cyclic reference is very appropriate. In this case, **Cycles** should be set to **FALSE** to avoid unnecessary warnings.

However, if your application tracks family history, you can have your application warn you, by setting **Cycles** to **TRUE**, when an inappropriate parent-child relationship is established by a user.

This syntax shows how to set the **Cycles** parameter:

```

<datasource>->SetViewOption(<view>,
    "Diagrammer",
    "...";
    Cycles = {FALSE|TRUE};
    ... "
);

```

Type: Boolean

Default: **FALSE**

Synonyms: None

BitmapFile

You can place a bitmap image in the background of a diagram by specifying a filename for the **BitmapFile** parameter. Background images help when using a **NDDGram** view to display mapping information, such as floor plans for a computer network. This syntax shows how to set the **BitmapFile** parameter:

This code assigns the filename “network.gif” to the **BitmapFile** parameter:

```
BitmapFilemGraphDs->SetViewOption(mDGRAMWgt,
    "Diagrammer", "BitmapFile = network.gif");
```

All nodes and links in the diagram are superimposed on the bitmap file in the background.

Type: String

Default: None

Synonyms: None

BgColor

The **BgColor** parameter defines the color to be used in the background of the diagram. If a bitmap file is supplied as a background image, the **BgColor** setting is ignored. This syntax shows how to set the **BgColor** parameter:

```
<datasource>->SetViewOption(<view>,
    "Diagrammer",
    "...";
    BgColor = <color_resource>;
    "...";
);
```

This code assigns the color resource “res.blue” to the **BgColor** parameter:

```
BgColormGraphDs->SetViewOption(mDGRAMWgt,
    "Diagrammer", "BgColor = res.blue");
```

Type: Color

Default: **NDDGram**-specific

Synonyms: **Bg**

“Standard” View Settings for Nodes and Links

The “standard” view settings set the display characteristics for nodes and links that are not currently selected in the diagram. In addition to enabling and disabling the labels for these diagram components, these parameter set these characteristics:

- Shapes
- Colors
- Pens
- Fonts

Here is the syntax supporting these “standard” characteristics:

```
<datasource>->SetViewOption(<view>,
    "Diagrammer",
    "[NodeStandardDData.FrameColor =
    <color_resource>;] |
    [NodeStandardDData.BgColor =
    <color_resource>;] |
    [NodeStandardDData.LabelColor =
    <color_resource>;] |
    [NodeStandardDData.FramePen =
    <pen_resource>;] |
    [NodeStandardDData.LabelFont =
    <font_resource>;] |
    [NodeStandardDData.DrawLabel = {FALSE|TRUE};] |
    [NodeStandardDData.Shape =
    {0|DEFAULT|RECT|RECTANGLE|
    1|ROUNDRECT|ROUNDRECTANGLE|
    2|ELLIPSE|
```

```

        3 | DIAMOND |
        4 | HEXAGON |
        5 | TRIANGLE |
    }; |

[Link.Color =
    <color_resource>;] |
[LinkStandardDDData.LinkDirColor =
    <color_resource>;] |
[LinkStandardDDData.LinkUndirColor =
    <color_resource>;] |
[LinkStandardDDData.LabelColor =
    <color_resource>;] |
[LinkStandardDDData.LinkPen =
    <pen_resource>;] |
[LinkStandardDDData.LabelFont =
    <font_resource>;] |
[LinkStandardDDData.DrawLabel = {FALSE|TRUE};] |
[LinkStandardDDData.Shape =
    {0 | DEFAULT | DIAGONAL |
     1 | RIGHTANGLE
    };] |
);

```

NodeStandardDDData.FrameColor

NodeStandardDDData.FrameColor sets the color of the frame around the node. This syntax shows how to set the **NodeStandardDDData.FrameColor** parameter:

```

<datasource>->SetViewOption(<view>,
    "Diagrammer",
    "...;
    NodeStandardDDData.FrameColor =
        <color_resource>;
    ..."
);

```

Type: Color

Default: **NDDGram**-specific

Synonyms: **Node.FColor**, **Node.FrameColor**

NodeStandardDDData.BgColor

NodeStandardDDData.BgColor sets the color of the node. This syntax shows how to set the **NodeStandardDDData.BgColor** parameter:

```

<datasource>->SetViewOption(<view>,
    "Diagrammer",
    "...;
    NodeStandardDDData.BgColor = <color_resource>;
    ..."
);

```

Type: Color

Default: **NDDGram**-specific

Synonyms: **Node.Bg**, **Node.Color**, **Node.BgColor**

NodeStandardDDData.LabelColor

NodeStandardDDData.LabelColor sets the color of the node label. This syntax shows how to set the **NodeStandardDDData.LabelColor** parameter:

```

<datasource>->SetViewOption(<view>,
    "Diagrammer",
    "...;

```

```

        NodeStandardDData.LabelColor =
                                <color_resource>;
        ...
    );

```

Type: Color

Default: **NDDGram**-specific

Synonyms: **Node.Fg**, **Node.FgColor**, **Node.LabelColor**

NodeStandardDData.FramePen

NodeStandardDData.FramePen sets the pen to use for the node label. This syntax shows how to set the **NodeStandardDData.FramePen** parameter:

```

<datasource>->SetViewOption(<view>,
    "Diagrammer",
    ...;
    NodeStandardDData.FramePen = <pen_resource>;
    ...
);

```

Type: Pen

Default: **NDDGram**-specific

Synonyms: **Node.Pen**, **Node.FramePen**

NodeStandardDData.LabelFont

NodeStandardDData.LabelFont sets the font for the node label. This syntax shows how to set the **NodeStandardDData.LabelFont** parameter:

```

<datasource>->SetViewOption(<view>,
    "Diagrammer",
    ...;
    NodeStandardDData.LabelFont = <font_resource>;
    ...
);

```

Type: Font

Default: **NDDGram**-specific

Synonyms: **Node.Font**, **Node.LabelFont**

NodeStandardDData.DrawLabel

NodeStandardDData.DrawLabel specifies whether or not to display node labels. This syntax shows how to set the **NodeStandardDData.DrawLabel** parameter:

```

<datasource>->SetViewOption(<view>,
    "Diagrammer",
    ...;
    NodeStandardDData.DrawLabel = {TRUE|FALSE};
    ...
);

```

Type: Boolean

Default: TRUE

Synonyms: **Node.DrawLabel**

NodeStandardDDData.Shape

NodeStandardDDData.Shape specifies the default node shape. This syntax shows how to set the **NodeStandardDDData.Shape** parameter:

```
<datasource>->SetViewOption(<view>,
    "Diagrammer",
    "...";
    NodeStandardDDData.Shape =
        {0 | DEFAULT | RECT | RECTANGLE |
         1 | ROUNDRECT | ROUNDRECTANGLE |
         2 | ELLIPSE |
         3 | DIAMOND |
         4 | HEXAGON |
         5 | TRIANGLE
        };
    "...";
);
```

Type: Enumerated

Default: **RECTANGLE**

Synonyms: **Shape**, **Node.Shape**

Link.Color

Link.Color sets the link color if the **LinkStandardDDData.LinkDirColor** or **LinkStandardDDData.LinkUndirColor** parameter is not set. This syntax shows how to set the **Link.Color** parameter:

```
<datasource>->SetViewOption(<view>,
    "Diagrammer",
    "...";
    Link.Color = <color_resource>;
    "...";
);
```

Type: Color

Default: **NDDGram**-specific

Synonyms: None

LinkStandardDDData.LinkDirColor

LinkStandardDDData.LinkDirColor sets the default color of all directed links. This syntax shows how to set the **LinkStandardDDData.LinkDirColor** parameter:

```
<datasource>->SetViewOption(<view>,
    "Diagrammer",
    "...";
    LinkStandardDDData.LinkDirColor =
        <color_resource>;
    "...";
);
```

Type: Color

Default: **NDDGram**-specific

Synonyms: **Link.DirColor**

LinkStandardDData.LinkUndirColor

LinkStandardDData.LinkUndirColor sets the default color of undirected links. This syntax shows how to set **LinkStandardDData.LinkUndirColor**:

```
<datasource>->SetViewOption(<view>,
    "Diagrammer",
    "...";
    LinkStandardDData.LinkUndirColor =
        <color_resource>;
    ...")
);
```

Type: Color

Default: **NDDGram**-specific

Synonyms: **Link.UndirColor**

LinkStandardDData.LabelColor

LinkStandardDData.LabelColor sets the color of the node label. This syntax shows how to set the **LinkStandardDData.LabelColor** parameter:

```
<datasource>->SetViewOption(<view>,
    "Diagrammer",
    "...";
    LinkStandardDData.LabelColor =
        <color_resource>;
    ...")
);
```

Type: Color

Default: **NDDGram**-specific

Synonyms: **Link.LabelColor**

LinkStandardDData.LinkPen

LinkStandardDData.LinkPen sets the pen for the link label. This syntax shows how to set the **LinkStandardDData.LinkPen** parameter:

```
<datasource>->SetViewOption(<view>,
    "Diagrammer",
    "...";
    LinkStandardDData.LinkPen = <pen_resource>;
    ...")
);
```

Type: Pen

Default: **NDDGram**-specific

Synonyms: **Link.Pen**

LinkStandardDData.LabelFont

LinkStandardDData.LabelFont sets the font for the node label. This syntax shows how to set the **LinkStandardDData.LabelFont** parameter:

```
<datasource>->SetViewOption(<view>,
    "Diagrammer",
    "...";
    LinkStandardDData.LabelFont = <font_resource>;
    ...")
);
```

Type: Font

Default: **NDDGram-specific**

Synonyms: **Link.Font**

LinkStandardDData.DrawLabel

LinkStandardDData.DrawLabel specifies whether or not to display node labels. This syntax shows how to set the **LinkStandardDData.DrawLabel** parameter:

```
<datasource>->SetViewOption(<view>,
    "Diagrammer",
    "...";
    LinkStandardDData.DrawLabel = {TRUE|FALSE};
    "...";
);
```

Type: Boolean

Default: **FALSE**

Synonyms: **Link.DrawLabel**

LinkStandardDData.Shape

LinkStandardDData.Shape specifies the default shape for links. This syntax shows how to set the **LinkStandardDData.Shape** parameter:

```
<datasource>->SetViewOption(<view>,
    "Diagrammer",
    "...";
    NodeStandardDData.Shape =
        {0|DEFAULT|DIAGONAL|
         1|RIGHTANGLE
        };
    "...";
);
```

Type: Enumerated

Default: **DIAGONAL**

Synonyms: **Link.Shape**

“Focus” View Settings for Nodes and Links

The “focus” view settings set the display characteristics for nodes and links that are currently selected in the diagram. In addition to enabling and disabling the labels for diagram components, these parameter set these characteristics:

- Shapes
- Colors
- Pens
- Fonts

Here is the syntax supporting these “focus” characteristics:

```
<datasource>->SetViewOption(<view>,
    "Diagrammer",
    "[NodeFocusDData.FrameColor =
        <color_resource>]; |
 [NodeFocusDData.BgColor =
        <color_resource>]; |
 [NodeFocusDData.LabelColor =
        <color_resource>]; |
 [NodeFocusDData.FramePen = <pen_resource>]; |
```

```

[NodeFocusDData.LabelFont =
  <font_resource>;] |
[NodeFocusDData.DrawLabel = {FALSE|TRUE};] |
[NodeFocusDData.Shape =
  {0|DEFAULT|RECT|RECTANGLE|
  1|ROUNDRECT|ROUNDRECTANGLE|
  2|ELLIPSE|
  3|DIAMOND|
  4|HEXAGON|
  5|TRIANGLE};] |

[LinkFocus.Color =
  {Color.Red|<color_resource>;} |
[LinkFocusDData.LinkDirColor =
  <color_resource>;] |
[LinkFocusDData.LinkUndirColor =
  <color_resource>;] |
[LinkFocusDData.LabelColor =
  <color_resource>;] |
[LinkFocusDData.LinkPen = <pen_resource>;] |
[LinkFocusDData.LabelFont =
  <font_resource>;] |
[LinkFocusDData.DrawLabel = {FALSE|TRUE};] |
[LinkFocusDData.Shape =
  {0|DEFAULT|DIAGONAL|
  1|RIGHTANGLE};]"
);

```

NodeFocusDData.FrameColor

NodeFocusDData.FrameColor sets the color of the frame around the node. This syntax shows how to set the **NodeFocusDData.FrameColor** parameter:

```

<datasource>->SetViewOption(<view>,
  "Diagrammer",
  "...";
  NodeFocusDData.FrameColor = <color_resource>;
  "...")
);

```

Type: Color

Default: **Color.Red**

Synonyms: **NodeFocus.FColor**, **NodeFocus.FrameColor**

NodeFocusDData.BgColor

NodeFocusDData.BgColor sets the color of the node. This syntax shows how to set the **NodeFocusDData.BgColor** parameter:

```

<datasource>->SetViewOption(<view>,
  "Diagrammer",
  "...";
  NodeFocusDData.BgColor = <color_resource>;
  "...")
);

```

Type: Color

Default: **NDDGram-specific**

Synonyms: **NodeFocus.Bg**, **NodeFocus.Color**, **NodeFocus.BgColor**

NodeFocusDData.LabelColor

NodeFocusDData.LabelColor sets the color of the node label. This syntax shows how to set the **NodeFocusDData.LabelColor** parameter:

```
<datasource>->SetViewOption(<view>,
    "Diagrammer",
    "...";
    NodeFocusDData.LabelColor = <color_resource>;
    "...";
);
```

Type: Color

Default: **NDDGram**-specific

Synonyms: **NodeFocus.Fg**, **NodeFocus.FgColor**, **NodeFocus.LabelColor**

NodeFocusDData.FramePen

NodeFocusDData.FramePen sets the pen for the node label. This syntax shows how to set the **NodeFocusDData.FramePen** parameter:

```
<datasource>->SetViewOption(<view>,
    "Diagrammer",
    "...";
    NodeFocusDData.FramePen = <pen_resource>;
    "...";
);
```

Type: Pen

Default: **NDDGram**-specific

Synonyms: **NodeFocus.Pen**, **NodeFocus.FramePen**

NodeFocusDData.LabelFont

NodeFocusDData.LabelFont sets the font for the node label. This syntax shows how to set the **NodeFocusDData.LabelFont** parameter:

```
<datasource>->SetViewOption(<view>,
    "Diagrammer",
    "...";
    NodeFocusDData.LabelFont = <font_resource>;
    "...";
);
```

Type: Font

Default: **NDDGram**-specific

Synonyms: **NodeFocus.Font**, **NodeFocus.LabelFont**

NodeFocusDData.DrawLabel

NodeFocusDData.DrawLabel specifies whether or not to display node labels. This syntax shows how to set the **NodeFocusDData.DrawLabel** parameter:

```
<datasource>->SetViewOption(<view>,
    "Diagrammer",
    "...";
    NodeFocusDData.DrawLabel = {TRUE|FALSE};
    "...";
);
```

Type: Boolean

Default: **TRUE**

Synonyms: **NodeFocus.DrawLabel**

NodeFocusDData.Shape

NodeFocusDData.Shape specifies the default node shape. This syntax shows how to set the **NodeFocusDData.Shape** parameter:

```
<datasource>->SetViewOption(<view>,
    "Diagrammer",
    "...";
    NodeFocusDData.Shape =
        {0 | DEFAULT | RECT | RECTANGLE |
          1 | ROUNDRECT | ROUNDRECTANGLE |
          2 | ELLIPSE |
          3 | DIAMOND |
          4 | HEXAGON |
          5 | TRIANGLE |
        };
    "...";
);
```

Type: Enumerated

Default: **RECTANGLE**

Synonyms: **NodeFocus.Shape**

LinkFocus.Color

LinkFocus.Color sets the link color if the **LinkFocusDData.LinkDirColor** or **LinkFocusDData.LinkUndirColor** parameter is not set. This syntax shows how to set the **LinkFocus.Color** parameter:

```
<datasource>->SetViewOption(<view>,
    "Diagrammer",
    "...";
    LinkFocus.Color = <color_resource>;
    "...";
);
```

Type: Color

Default: **Color.Red**

Synonyms: None

LinkFocusDData.LinkDirColor

LinkFocusDData.LinkDirColor sets the default color of all directed links. This syntax shows how to set the **LinkFocusDData.LinkDirColor** parameter:

```
<datasource>->SetViewOption(<view>,
    "Diagrammer",
    "...";
    LinkFocusDData.LinkDirColor = <color_resource>;
    "...";
);
```

Type: Color

Default: **Color.Red**

Synonyms: **LinkFocus.DirColor**, **LinkFocus.LinkDirColor**

LinkFocusDData.LinkUndirColor

LinkFocusDData.LinkUndirColor sets the default color of undirected links. This syntax shows how to set **LinkFocusDData.LinkUndirColor**:

```
<datasource>->SetViewOption(<view>,
    "Diagrammer",
    "...";
    LinkFocusDData.LinkUndirColor =
        <color_resource>;
    "...")
);
```

Type: Color

Default: **Color.Red**

Synonyms: **LinkFocus.UndirColor**, **LinkFocus.LinkUndirColor**

LinkFocusDData.LabelColor

LinkFocusDData.LabelColor sets the color of the node label. This syntax shows how to set the **LinkFocusDData.LabelColor** parameter:

```
<datasource>->SetViewOption(<view>,
    "Diagrammer",
    "...";
    LinkFocusDData.LabelColor = <color_resource>;
    "...")
);
```

Type: Color

Default: **NDDGram-specific**

Synonyms: **LinkFocus.LabelColor**

LinkFocusDData.LinkPen

LinkFocusDData.LinkPen sets the pen for the link label. This syntax shows how to set the **LinkFocusDData.LinkPen** parameter:

```
<datasource>->SetViewOption(<view>,
    "Diagrammer",
    "...";
    LinkFocusDData.LinkPen = <pen_resource>;
    "...")
);
```

Type: Pen

Default: **NDDGram-specific**

Synonyms: **LinkFocus.Pen**, **LinkFocus.LinkPen**

LinkFocusDData.LabelFont

LinkFocusDData.LabelFont sets the font for the node label. This syntax shows how to set the **LinkFocusDData.LabelFont** parameter:

```
<datasource>->SetViewOption(<view>,
    "Diagrammer",
    "...";
    LinkFocusDData.LabelFont = <font_resource>;
    "...")
);
```

Type: Font

Default: **NDDGram**-specific

Synonyms: **LinkFocus.Font**, **LinkFocus.LabelFont**

LinkFocusDData.DrawLabel

LinkFocusDData.DrawLabel specifies whether or not to display node labels. This syntax shows how to set the **LinkFocusDData.DrawLabel** parameter:

```
<datasource>->SetViewOption(<view>,
    "Diagrammer",
    "...";
    LinkFocusDData.DrawLabel = {TRUE|FALSE};
    "...";
);
```

Type: Boolean

Default: **FALSE**

Synonyms: **LinkFocus.DrawLabel**

LinkFocusDData.Shape

LinkFocusDData.Shape specifies the default shape for links. This syntax shows how to set the **LinkFocusDData.Shape** parameter:

```
<datasource>->SetViewOption(<view>,
    "Diagrammer",
    "...";
    LinkFocusDData.Shape =
        {0 | DEFAULT | DIAGONAL |
         1 | RIGHTANGLE
        };
    "...";
);
```

Type: Enumerated

Default: **DIAGONAL**

Synonyms: **LinkFocus.Shape**

Custom Node and Link Options

You manage the basic node and link parameter using the **Diagrammer** option. You can also define *custom* node and link properties to which you can assign values and store them in the graph datasource. These “options”—*property:value*—support all of the same general, “standard,” and “focus” parameters as the **Diagrammer** option, except that you can define qualified node and link subsets.

For example, a mapping application might use a **City** node property with values of **Large**, **Medium**, and **Small**. In the same example, a **Road** link property might have values of **Fast**, **Moderate**, and **Slow** to indicate the relative speed grades of the roads between the cities.

Warning: The parser for the Elements Environment is case-sensitive. Verify that any strings you define for the properties and values in the **SetNodeProperty()** and **SetEdgeProperty()** functions are exactly the same as those used in the **SetViewOptions()** function.

These two sections discuss the node and link portions of this example:

- Custom Node Properties
- Custom Link Properties

Each *property:value* pair is a customized option of the graph datasource. It can have any number of associated assignment statements in its variable list; these can apply to all qualified nodes or links. Y

Warning: *You* must use value strings consistently, because the graph datasource does **not** check for the proper usage of custom properties and values.

Custom Node Properties

For the currently accessed node, you use the **SetNodeProperty()** function to set a custom node property and assign a value to it, as shown here:

```
Var <value_variable>;
<value_variable>->("<value>");
...
<datasource>->SetNodeProperty(<node_accessor>,
                              "<node_property>", &<value_variable>);
```

The **SetNodeProperty()** function assigns the variant value stored at the address of the *value_variable*. The custom node properties specified in the **SetViewOption()** function are meaningless unless you assign some custom node properties to the nodes in the graph datasource. Here is an example:

```
...
Var mCityLarge;
mCityLarge->("Large");
...
mGraphDs->SetNodeProperty(mNodeAccessor,
                          "City", &mCityLarge);
...
mGraphDs->SetViewOption(mDGRAMWgt,
                       "City:Large", "Width = 144; Height = 48");
```

Here is the **SetViewOption()** syntax for the custom node properties:

```
<datasource>->SetViewOption(<view>,
                            "<node_property>:<value>",
                            "[Width = <integer>;] |
                             [Height = <integer>;] |
                             [onFocus = {FALSE|TRUE};] |
                             [StandardDData.FrameColor = <color_resource>;] |
                             [StandardDData.BgColor = <color_resource>;] |
                             [StandardDData.LabelColor = <color_resource>;] |
                             [StandardDData.FramePen = <pen_resource>;] |
                             [StandardDData.LabelFont = <font_resource>;] |
                             [StandardDData.DrawLabel = {FALSE|TRUE};] |
                             [StandardDData.Shape =
                               {0|DEFAULT|RECT|RECTANGLE|
                                1|ROUNDRECT|ROUNDRECTANGLE|
                                2|ELLIPSE|
                                3|DIAMOND|
                                4|HEXAGON|
                                5|TRIANGLE};] |
                             [FocusDData.FrameColor = <color_resource>;] |
                             [FocusDData.BgColor = <color_resource>;] |
                             [FocusDData.LabelColor = <color_resource>;] |
                             [FocusDData.FramePen = <pen_resource>;] |
                             [FocusDData.LabelFont = <font_resource>;] |
                             [FocusDData.DrawLabel = {FALSE|TRUE};] |
                             [FocusDData.Shape =
                               {0|DEFAULT|RECT|RECTANGLE|
                                1|ROUNDRECT|ROUNDRECTANGLE|
```

```
2 | ELLIPSE |
3 | DIAMOND |
4 | HEXAGON |
5 | TRIANGLE}; ]
```

```
"
);
```

A geographical mapping application using **City** as a node property with values of **Large**, **Medium**, and **Small** might be implemented with this code fragment:

```
VarGrPtr mGraphDs = new NDVarGr;
VarGrNodeAccessorPtr mNodeAccessor = new NDVarGrNodeAccessor;
DGramPtr mDGRAMWgt = (NDDGramPtr)GetNamedWgt("DGRAM");
Str answer;
StrIVal len;

// Define value variables for node properties.
NDVarPtr citySmall = new NDVar;
NDVarPtr cityMedium = new NDVar;
NDVarPtr cityLarge = new NDVar;
enum CitySize {
    Small=1,
    Medium,
    Large
};
CitySize theSize;

// Define value strings for the custom node properties.
mCitySmall->SetStr("Small");
mCityMedium->SetStr("Medium");
mCityLarge->SetStr("Large");

// Set a node cursor using "mNodeAccessor."
mGraphDs->SetNodeCursor(mNodeAccessor);

// Set the cursor option to "CONTROLS."
mGraphDs->SetViewOption(mDGRAMWgt,
    "Cursor", "Controls");

// Set common node options.
mGraphDs->SetViewOption(mDGRAMWgt,
    "Diagrammer",
    "NodeStandardDDData.Shape = RECTANGLE");
// Set the parameters for the custom node options.
mGraphDs->SetViewOption(mDGRAMWgt,
    "City:Small", "NodeW = 72; NodeH = 12");
mGraphDs->SetViewOption(mDGRAMWgt,
    "City:Medium", "NodeW = 108; NodeH = 18");
mGraphDs->SetViewOption(mDGRAMWgt,
    "City:Large", "NodeW = 144; NodeH = 24");

// Define properties and values for individual nodes.
if (NDAaskW::AskQuestion("1=Small,
    2=Medium,
    3=Large", answer)) {
    len = NDStr::GetLen(answer);
    theSize = (enum CitySize)NDStr::GetDecInt(answer, &len);
    switch (theSize) {
        case Small:
            mGraphDs->SetNodeProperty(mNodeAccessor,
                "City", mCitySmall);
            break;
        case Medium:
            mGraphDs->SetNodeProperty(mNodeAccessor,
                "City", mCityMedium);
            break;
        case Large:
            mGraphDs->SetNodeProperty(mNodeAccessor,
                "City", mCityLarge);
            break;
    }
}
```

```

        default:
            NDAlrtW::Ok("Not a valid number");
            break;
    } /*End switch. */
} /* End if. */

```

Width

This integer value specifies the width of a node in pixels. If the **Autosize** option is enabled, this option and the **Height** parameter are ignored.

This syntax shows how to specify the node width:

```

<datasource>->SetViewOption(<view>,
    "<node_property>:<value>",
    "...;
    Width = <integer>;
    ..."
);

```

For nodes with options, **City:Large**, this code fragment sets the node width to 72 pixels—that is, 1 inch on a 72-pixel-per-inch display:

```

mGraphDs->SetViewOption(mDGRAMWgt,
    "City:Large", "width = 72");

```

Type: Int32

Default: **NDDGram**-specific

Synonyms: **W**

Height

This integer value specifies the width of a node in pixels. If the **Autosize** option is enabled, this option and the **Width** parameter are ignored.

This syntax shows how to specify the node height:

```

<datasource>->SetViewOption(<view>,
    "<node_property>:<value>",
    "...;
    Height = <integer>;
    ..."
);

```

For nodes with options, **City:Large**, this code fragment sets the node height to 24 pixels—that is, 1/3 inch on a 72-pixel-per-inch display:

```

mGraphDs->SetViewOption(mDGRAMWgt,
    "City:Large", "Height = 24");

```

Type: Int32

Default: **NDDGram**-specific

Synonyms: **H**

onFocus

With **onFocus** set to **FALSE** (the default), all nodes with the specified *node_property: value* combination are displayed using the “standard” graphic settings that apply to them. With **onFocus** set to **TRUE**, all nodes with the specified *node_property: value* combination are displayed using the “focus” graphic settings that apply to them.

In cases where multiple custom options are applied to a particular node with conflicting **onFocus** settings, the last **onFocus** setting applied to these nodes is in effect in the display.

This syntax shows how to set the **onFocus** parameter:

```
<datasource>->SetViewOption(<view>,
    "<node_property>:<value>",
    "...;
    onFocus = {TRUE|FALSE};
    ..."
);
```

Type: Boolean

Default: **FALSE**

Synonyms: None

StandardDData.FrameColor

StandardDData.FrameColor sets the color of the frame around the node. This syntax shows how to set the **StandardDData.FrameColor** parameter:

```
<datasource>->SetViewOption(<view>,
    "<node_property>:<value>",
    "...;
    StandardDData.FrameColor =
    <color_resource>;
    ..."
);
```

Type: Color

Default: **NDDGram**-specific

Synonyms: **FrameColor**, **FColor**

StandardDData.BgColor

StandardDData.BgColor sets the color of the node. This syntax shows how to set the **StandardDData.BgColor** parameter:

```
<datasource>->SetViewOption(<view>,
    "<node_property>:<value>",
    "...;
    StandardDData.BgColor = <color_resource>;
    ..."
);
```

Type: Color

Default: **NDDGram**-specific

Synonyms: **BgColor**, **Color**, **Bg**

StandardDData.LabelColor

StandardDData.LabelColor sets the color of the node label. This syntax shows how to set the **StandardDData.LabelColor** parameter:

```
<datasource>->SetViewOption(<view>,
    "<node_property>:<value>",
    "...;
    StandardDData.LabelColor =
    <color_resource>;
    ..."
);
```


Type: Color

Default: **NDDGram**-specific

Synonyms: **LabelColor, FgColor, Fg**

StandardDDData.FramePen

StandardDDData.FramePen sets the pen for the node label. This syntax shows how to set the **StandardDDData.FramePen** parameter:

```
<datasource>->SetViewOption(<view>,
    "<node_property>:<value>",
    "...;
    StandardDDData.FramePen = <pen_resource>;
    ..."
);
```

Type: Pen

Default: **NDDGram**-specific

Synonyms: **FramePen, Pen**

StandardDDData.LabelFont

StandardDDData.LabelFont sets the font for the node label. This syntax shows how to set the **StandardDDData.LabelFont** parameter:

```
<datasource>->SetViewOption(<view>,
    "<node_property>:<value>",
    "...;
    StandardDDData.LabelFont = <font_resource>;
    ..."
);
```

Type: Font

Default: **NDDGram**-specific

Synonyms: **LabelFont, Font**

StandardDDData.DrawLabel

StandardDDData.DrawLabel specifies whether or not to display node labels. This syntax shows how to set the **StandardDDData.DrawLabel** parameter:

```
<datasource>->SetViewOption(<view>,
    "<node_property>:<value>",
    "...;
    StandardDDData.DrawLabel = {TRUE|FALSE};
    ..."
);
```

Type: Boolean

Default: TRUE

Synonyms: **DrawLabel**

StandardDDData.Shape

StandardDDData.Shape specifies the default node shape. This syntax shows how to set the **StandardDDData.Shape** parameter:

```
<datasource>->SetViewOption(<view>,
    "<node_property>:<value>",
    "...;
    NodeStandardDDData.Shape =
```

```

{0 | DEFAULT | RECT | RECTANGLE |
1 | ROUNDRECT | ROUNDRECTANGLE |
2 | ELLIPSE |
3 | DIAMOND |
4 | HEXAGON |
5 | TRIANGLE |
};

```

```

... "
);

```

Type: Enumerated

Default: **RECTANGLE**

Synonyms: **Shape**

FocusDData.FrameColor

FocusDData.FrameColor sets the color of the frame around the node. This syntax shows how to set the **FocusDData.FrameColor** parameter:

```

<datasource>->SetViewOption(<view>,
    "<node_property>:<value>",
    "...";
    FocusDData.FrameColor = <color_resource>;
    ... "
);

```

Type: Color

Default: **Color.Red**

Synonyms: **Focus.FrameColor, Focus.FColor**

FocusDData.BgColor

FocusDData.BgColor sets the color of the node. This syntax shows how to set the **FocusDData.BgColor** parameter:

```

<datasource>->SetViewOption(<view>,
    "<node_property>:<value>",
    "...";
    FocusDData.BgColor = <color_resource>;
    ... "
);

```

Type: Color

Default: **NDDGram-specific**

Synonyms: **Focus.BgColor, Focus.Color, Focus.Bg**

FocusDData.LabelColor

FocusDData.LabelColor sets the color of the node label. This syntax shows how to set the **FocusDData.LabelColor** parameter:

```

<datasource>->SetViewOption(<view>,
    "<node_property>:<value>",
    "...";
    FocusDData.LabelColor = <color_resource>;
    ... "
);

```

Type: Color

Default: **NDDGram-specific**

Synonyms: **Focus.LabelColor, Focus.FgColor, Focus.Fg**

FocusDData.FramePen

FocusDData.FramePen sets the pen for the node label. This syntax shows how to set the **FocusDData.FramePen** parameter:

```
<datasource>->SetViewOption(<view>,
    "<node_property>:<value>",
    "...;
    FocusDData.FramePen = <pen_resource>;
    ..."
);
```

Type: Pen

Default: **NDDGram**-specific

Synonyms: **Focus.FramePen**, **Focus.Pen**

FocusDData.LabelFont

FocusDData.LabelFont sets the font for the node label. This syntax shows how to set the **FocusDData.LabelFont** parameter:

```
<datasource>->SetViewOption(<view>,
    "<node_property>:<value>",
    "...;
    FocusDData.LabelFont = <font_resource>;
    ..."
);
```

Type: Font

Default: **NDDGram**-specific

Synonyms: **Focus.LabelFont**, **Focus.Font**

FocusDData.DrawLabel

FocusDData.DrawLabel specifies whether or not to display node labels. This syntax shows how to set the **FocusDData.DrawLabel** parameter:

```
<datasource>->SetViewOption(<view>,
    "<node_property>:<value>",
    "...;
    FocusDData.DrawLabel = {TRUE|FALSE};
    ..."
);
```

Type: Boolean

Default: **TRUE**

Synonyms: **Focus.DrawLabel**

FocusDData.Shape

FocusDData.Shape specifies the default node shape. This syntax shows how to set the **FocusDData.Shape** parameter:

```
<datasource>->SetViewOption(<view>,
    "<node_property>:<value>",
    "...;
    FocusDData.Shape =
        {0|DEFAULT|RECT|RECTANGLE|
        1|ROUNDRECT|ROUNDRECTANGLE|
        2|ELLIPSE|
        3|DIAMOND|
        4|HEXAGON|
        5|TRIANGLE
        };
```

```
); ..."
```

Type: Enumerated

Default: **RECTANGLE**

Synonyms: **Focus.Shape**

Custom Link Properties

You control the standard link parameters with the **Diagrammer** option. You can also define *custom* link properties to which you can assign values. For example, a link property, **Road**, might have values of **Fast**, **Moderate**, and **Slow**.

For the currently accessed edge, use the **SetEdgeProperty()** function to define a property and assign a value to it, as shown here:

```
NDVarPtr <value_variable> = new NDVar;
<value_variable>->("<value>");
...
<datasource>->SetEdgeProperty(<edge_accessor>,
                             "<edge_property>", <value_variable>);
...
delete <value_variable>;
```

The **SetEdgeProperty()** function assigns the variant value stored at the address of the *value_variable*. The custom node properties specified in the **SetViewOption()** function are meaningless unless you assign some custom edge properties to the nodes in the graph datasource. Here is an example:

```
...
NDVarPtr mRoadFast = new NDVar;
mRoadFast->SetStr("Fast");
...
mGraphDs->SetNodeProperty(mEdgeAccessor,
                          "Road", mRoadFast);
...

mGraphDs->SetViewOption(mDGRAMWGT,
                       "Road:Fast",
                       "StandardDDData.LinkPen = Map.Fast");
...
delete mRoadFast;
```

Here is the **SetViewOption()** syntax for the custom node properties:

```
<datasource>->SetViewOption(<view>,
                            "<link_property>:<value>",
                            "[LinkColor = <color_resource>;] |
                             [onFocus = {FALSE|TRUE};] |
                             [StandardDDData.LinkDirColor =
                               <color_resource>;] |
                             [StandardDDData.LinkUndirColor =
                               <color_resource>;] |
                             [StandardDDData.LabelColor = <color_resource>;] |
                             [StandardDDData.LinkPen = <pen_resource>;] |
                             [StandardDDData.LabelFont = <font_resource>;] |
                             [StandardDDData.DrawLabel = {FALSE|TRUE};] |
                             [StandardDDData.Shape =
                               {0|DEFAULT|DIAGONAL|
                                1|RIGHTANGLE};] |
                             [FocusColor = <color_resource>;] |
                             [FocusDDData.LinkDirColor = <color_resource>;] |
                             [FocusDDData.LinkUndirColor = <color_resource>;]
                             |
                             [FocusDDData.LabelColor = <color_resource>;] |
                             [FocusDDData.LinkPen = <pen_resource>;] |
                             [FocusDDData.LabelFont = <font_resource>;] |
```

```

        [FocusDData.DrawLabel = {FALSE|TRUE};] |
        [FocusDData.Shape =
            {0|DEFAULT|DIAGONAL|
             1|RIGHTANGLE};]
    "
);

```

A mapping application using **Road** as an edge property with values of **Fast**, **Moderate**, and **Slow** might be implemented with this code fragment:

```

VarGrPtr mGraphDs = new NDVarGr;
VarGrEdgeAccessorPtr mEdgeAccessor = new NDVarGrEdgeAccessor;
DGramPtr mDGRAMWgt = (NDDGramPtr)GetNamedWgt("DGRAM");
Str answer;
StrIVal len;

// Define value variables for node properties.
NDVarPtr mRoadSlow = new NDVar;
NDVarPtr mRoadModerate = new NDVar;
NDVarPtr mRoadFast = new NDVar;
enum roadSpeed {
    Slow = 1,
    Moderate,
    Fast
};

roadSpeed mRoadSpeed;

// Define value strings for the custom edge properties.
mRoadSlow->SetStr("Slow");
mRoadModerate->SetStr("Moderate");
mRoadFast->SetStr("Fast");

// Set a edge cursor using "mEdgeAccessor."
mGraphDs->SetEdgeCursor(mEdgeAccessor);

// Set the cursor option to "CONTROLS."
mGraphDs->SetViewOption(mDGRAMWgt,
    "Cursor", "CONTROLS");
// Set common edge options.
mGraphDs->SetViewOption(mDGRAMWgt,
    "Diagrammer",
    "LinkStandardDDData.Shape = DIAGONAL");
// Set the parameters for the custom edge options.
mGraphDs->SetViewOption(mDGRAMWgt, "Road:Slow",
    "StandardDDData.LinkPen = Map.Slow");
mGraphDs->SetViewOption(mDGRAMWgt, "Road:Moderate",
    "StandardDDData.LinkPen = Map.Moderate");
mGraphDs->SetViewOption(mDGRAMWgt, "Road:Fast",
    "StandardDDData.LinkPen = Map.Fast");

// This sets the enumerated "mRoadSpeed" variable to the
// corresponding value. Define properties and values for
// individual edges.
if(NDAskW::AskQuestion("1=Slow, 2=Moderate, 3=Fast",answer)) {
    len=NDStr::GetLen(answer);
    mRoadSpeed =(enum roadSpeed)NDStr::GetDecInt(answer,&len);

    switch (theSpeed) {
        case Slow:
            mGraphDs->SetEdgeProperty(mEdgeAccessor,
                "Road", mRoadSlow);
            break;
        case Moderate:
            mGraphDs->SetEdgeProperty(mEdgeAccessor,
                "Road", mRoadModerate);
            break;
        case Fast:
            mGraphDs->SetEdgeProperty(mEdgeAccessor,
                "Road", mRoadFast);
            break;
        default:

```

```

        break;
    } /*End switch. */
} /* End if. */

```

LinkColor

LinkColor sets the link color if the **StandardDDData.LinkDirColor** or **StandardDDData.LinkUndirColor** parameter is not set. This syntax shows how to set the **LinkColor** parameter:

```

<datasource>->SetViewOption(<view>,
    "<link_property>:<value>",
    "...;
    LinkColor = <color_resource>;
    ..."
);

```

Type: Color

Default: **NDDGram**-specific

Synonyms: **Color**

onFocus

With **onFocus** set to **FALSE** (the default), all nodes with the specified *link_property:value* combination are displayed using the “standard” graphic settings that apply to them. With **onFocus** set to **TRUE**, all nodes with the specified *link_property:value* combination are displayed using the “focus” graphic settings that apply to them.

In cases where multiple custom options are applied to a particular link with conflicting **onFocus** settings, the last **onFocus** setting applied to these nodes is in effect in the display.

This syntax shows how to set the **onFocus** parameter:

```

<datasource>->SetViewOption(<view>,
    "<link_property>:<value>",
    "...;
    onFocus = {TRUE|FALSE};
    ..."
);

```

Type: Boolean

Default: **FALSE**

Synonyms: None

StandardDDData.LinkDirColor

StandardDDData.LinkDirColor sets the default color of all directed links.

This syntax shows how to set the **StandardDDData.LinkDirColor** parameter:

```

<datasource>->SetViewOption(<view>,
    "<link_property>:<value>",
    "...;
    StandardDDData.LinkDirColor =
    <color_resource>;
    ..."
);

```

Type: Color

Default: **NDDGram**-specific

Synonyms: **DirColor**

StandardDData.LinkUndirColor

StandardDData.LinkUndirColor sets the default color of undirected links. This syntax shows how to set **StandardDData.LinkUndirColor**:

```
<datasource>->SetViewOption(<view>,
    "<link_property>:<value>",
    "...;
    StandardDData.LinkUndirColor =
        <color_resource>;
    ..."
);
```

Type: Color

Default: **NDDGram**-specific

Synonyms: **UndirColor**

StandardDData.LabelColor

StandardDData.LabelColor sets the color of the node label. This syntax shows how to set the **StandardDData.LabelColor** parameter:

```
<datasource>->SetViewOption(<view>,
    "<link_property>:<value>",
    "...;
    StandardDData.LabelColor =
        <color_resource>;
    ..."
);
```

Type: Color

Default: **NDDGram**-specific

Synonyms: **LabelColor**

StandardDData.LinkPen

StandardDData.LinkPen sets the pen for the link label. This syntax shows how to set the **StandardDData.LinkPen** parameter:

```
<datasource>->SetViewOption(<view>,
    "<link_property>:<value>",
    "...;
    StandardDData.LinkPen = <pen_resource>;
    ..."
);
```

Type: Pen

Default: **NDDGram**-specific

Synonyms: **Pen**

StandardDData.LabelFont

StandardDData.LabelFont sets the font for the node label. This syntax shows how to set the **StandardDData.LabelFont** parameter:

```
<datasource>->SetViewOption(<view>,
    "<link_property>:<value>",
    "...;
    StandardDData.LabelFont = <font_resource>;
    ..."
);
```

Type: Font

Default: **NDDGram**-specific

Synonyms: **Font**

StandardDDData.DrawLabel

StandardDDData.DrawLabel specifies whether or not to display node labels. This syntax shows how to set the **StandardDDData.DrawLabel** parameter:

```
<datasource>->SetViewOption(<view>,
    "<link_property>:<value>",
    "...";
    StandardDDData.DrawLabel = {TRUE|FALSE};
    "...";
);
```

Type: Boolean

Default: **FALSE**

Synonyms: **DrawLabel**

StandardDDData.Shape

StandardDDData.Shape specifies the default shape for links. This syntax shows how to set the **StandardDDData.Shape** parameter:

```
<datasource>->SetViewOption(<view>,
    "<link_property>:<value>",
    "...";
    StandardDDData.Shape =
        {0|DEFAULT|DIAGONAL|
         1|RIGHTANGLE
        };
    "...";
);
```

Type: Enumerated

Default: **DIAGONAL**

Synonyms: **Shape**

FocusColor

FocusColor sets the link color if the **FocusDDData.LinkDirColor** or **FocusDDData.LinkUndirColor** parameter is not set. This syntax shows how to set the **FocusColor** parameter:

```
<datasource>->SetViewOption(<view>,
    "<link_property>:<value>",
    "...";
    Focus.Color = <color_resource>;
    "...";
);
```

Type: Color

Default: **Color.Red**

Synonyms: None

FocusDData.LinkDirColor

FocusDData.LinkDirColor sets the default color of all directed links. This syntax shows how to set the **FocusDData.LinkDirColor** parameter:

```
<datasource>->SetViewOption(<view>,
    "<link_property>:<value>",
    "...;
    FocusDData.LinkDirColor = <color_resource>;
    ..."
);
```

Type: Color

Default: **Color.Red**

Synonyms: **Focus.LinkDirColor**, **Focus.DirColor**

FocusDData.LinkUndirColor

FocusDData.LinkUndirColor sets the default color of undirected links. This syntax shows how to set **FocusDData.LinkUndirColor**:

```
<datasource>->SetViewOption(<view>,
    "<link_property>:<value>",
    "...;
    FocusDData.LinkUndirColor =
    <color_resource>;
    ..."
);
```

Type: Color

Default: **Color.Red**

Synonyms: **Focus.LinkUndirColor**, **Focus.UndirColor**

FocusDData.LabelColor

FocusDData.LabelColor sets the color of the node label. This syntax shows how to set the **FocusDData.LabelColor** parameter:

```
<datasource>->SetViewOption(<view>,
    "<link_property>:<value>",
    "...;
    FocusDData.LabelColor = <color_resource>;
    ..."
);
```

Type: Color

Default: **NDDGram-specific**

Synonyms: **Focus.LabelColor**

FocusDData.LinkPen

FocusDData.LinkPen sets the pen for the link label. This syntax shows how to set the **FocusDData.LinkPen** parameter:

```
<datasource>->SetViewOption(<view>,
    "<link_property>:<value>",
    "...;
    FocusDData.LinkPen = <pen_resource>;
    ..."
);
```

Type: Pen

Default: **NDDGram**-specific

Synonyms: **Focus.LinkPen**, **Focus.Pen**

FocusDData.LabelFont

FocusDData.LabelFont sets the font for the node label. This syntax shows how to set the **FocusDData.LabelFont** parameter:

```
<datasource>->SetViewOption(<view>,
    "<link_property>:<value>",
    "...;
    FocusDData.LabelFont = <font_resource>;
    ..."
);
```

Type: Font

Default: **NDDGram**-specific

Synonyms: **Focus.LabelFont**, **Focus.Font**

FocusDData.DrawLabel

FocusDData.DrawLabel specifies whether or not to display node labels. This syntax shows how to set the **FocusDData.DrawLabel** parameter:

```
<datasource>->SetViewOption(<view>,
    "<link_property>:<value>",
    "...;
    FocusDData.DrawLabel = {TRUE|FALSE};
    ..."
);
```

Type: Boolean

Default: **FALSE**

Synonyms: **Focus.DrawLabel**

FocusDData.Shape

FocusDData.Shape specifies the default shape for links. This syntax shows how to set the **FocusDData.Shape** parameter:

```
<datasource>->SetViewOption(<view>,
    "<link_property>:<value>",
    "...;
    FocusDData.Shape =
        {0 | DEFAULT | DIAGONAL |
         1 | RIGHTANGLE
        };
    ..."
);
```

Type: Enumerated

Default: **DIAGONAL**

Synonyms: **Focus.Shape**

Building a Graph Datasource

A graph datasource is a container of freely arranged nodes and edges. These may be related hierarchically or nonhierarchically—as defined by the edges relating them—or they may not be related at all (having no edge between

them). Each node has variant **ID** and **Value** properties, which you may set when you create the node or during a separate editing session.

Building a graph datasource involves these tasks:

1. Creating a Graph Datasource
2. Creating and Destroying an Edit Object
3. Creating Accessors
4. Creating Nodes
5. Creating Edges

The preceding list is somewhat simplified, but does explain the basic process, parts of which you may need to reiterate. **NDDGram** For more information about options for the **NDDGram** view, see “Options for the NDDGram View” on page 80.

Creating a Graph Datasource

Before you can begin creating node relationships in the graph datasource, your application must first create a graph datasource. This code fragment creates `mGraphDs` as a **VarGrPtr** variable and assigns an object of the **NDVarGr** object class to it:

```
VarGrPtr mGraphDs = new NDVarGr;
```

The preceding code fragment creates a graph datasource with the structure shown in Figure 4–14. The simple box in Figure 4–14 represents the memory location of the datasource object, `mGraphDs`.



Figure 4-14 Untitled Graph Datasource

The examples in the following sections build on this simple representation to construct a graph with relationships—neighbor, parent-child, or none—indicated by the shared edges. For more information about graph datasources, see “Graph Datasource” on page 59.

Creating and Destroying an Edit Object

Building a graph datasource requires modifying the initial structures. To modify the structure of a graph datasource—that is, to add and remove nodes and edges—you need a datasource edit object. This code fragment creates and destroys a datasource edit object, `mEditGraphDs`:

```
VarGrPtr mGraphDs = new NDVarGr;

// Declare a pointer variable for the edit object and assign
// the address of a datasource edit object to it.
VarGrEditPtr mEditGraphDs = mGraphDs->StartEdit();

// Edit operations defined.
...

// Commit edit operations to the datasource and destroy the edit
// object.
mEditGraphDs->End();
```

In this example, *mEditGraphDs* is a **VarGrEditPtr** variable and is assigned a datasource edit object—the value returned by the **NDVarGr::StartEdit()** function. When the **NDDsEdit::End()** function executes, all edit operations are committed to the datasource, and the edit object is destroyed. For more information about graph-datasource edit objects, see “Datasource Editing” on page 75.

As a first use of the datasource edit object, assign a title to the datasource, as shown here:

```
VarGrPtr mGraphDs = new NDVarGr;
VarGrEditPtr mEditGraphDs = mGraphDs->StartEdit();

// Set the title of the graph datasource.
mEditGraphDs->SetTitle("Graph Datasource");

// Commit edit operations to the datasource and destroy the edit
// object.
mEditGraphDs->End();
```

Note the use of the string literal in quotation marks. Unlike the node **ID** and **Value** properties, the **NDVarGrEdit::SetTitle()** function accepts a string as the datasource title. Building on the example from Figure 4–14, executing the preceding **NDVarGrEdit::SetTitle()** function adds a title to the graph datasource, as shown in Figure 4–15:

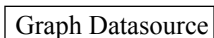


Figure 4–15 Titled Graph Datasource

Creating Accessors

The graph datasource use two basic types of accessors: node and edge accessors. At a minimum, building a graph datasource requires creating node accessors by which to access and operate on the nodes within it. Depending on the approach you take, you may also need an edge accessor when building a graph datasource.

Creating Node Accessors

You need at least one node accessor, and possibly two or more, when changing the structure of a graph datasource. For example, you might add or remove nodes, or simply update information about a particular node. Using a node accessor with the graph-datasource APIs, you can traverse the nodes in the graph.

The third line of this code fragment declares a node-accessor pointer, *mNodeAccessor*, creates a node-accessor object, and assigns the object to a node-accessor pointer:

```
VarGrPtr mGraphDs = new NDVarGr;
VarGrEditPtr mEditGraphDs = mGraphDs->StartEdit();

// Declare a node-accessor pointer and assign a node-accessor
// object to it.
VarGrNodeAccessorPtr mNodeAccessor = new NDVarGrNodeAccessor;
...
```

This creates *mNodeAccessor* as a **VarGrNodeAccessorPtr** variable and assigns an object of the **NDVarGrNodeAccessor** object class to it. For more information about node accessors, see “Node Accessor” on page 71.

Creating Edge Accessors

You need an edge accessor to update information—**ID** and **Value** properties—for a specific edge. Using an edge accessors with the graph-datasource APIs, you can traverse the edge in the graph. These edge-accessor types are supported:

- “All” Edge Accessors
- “In” and “Out” Edge Accessors
- “Undirected” Edge Accessors

“All” edge accessors are useful for traversing all of the edges in the graph datasource, regardless of whether they are directed or undirected. “In,” “out,” and “undirected” edge accessors are similar in that each of them accesses edges relative to a specified node.

“All” Edge Accessors

An “all” edge accessor is the only type of edge accessor that does not access edges relative to a particular node. That is, you do not create such an accessor using a node accessor as an argument. Because the “all” edge accessors are not based on a specific node as a navigational reference point, the API for navigating them is based on edge indexes. These, in turn, are based on the order in which you create the edges.

This code fragment declares an “all” edge-accessor pointer, `mAllEdgeAccessor`, creates an “all” edge-accessor object, and assigns the object to an edge-accessor pointer:

```
VarGrPtr mGraphDs = new NDVarGr;

// Declare a node-accessor pointer and assign an edge-accessor
// object to it based on the "all" edge accessor,
// mAllEdgeAccessor.
VarGrAllEdgeAccessorPtr mAllEdgeAccessor =
    new NDVarGrAllEdgeAccessor;
VarGrEditPtr mEditEdge =
    mGraphDs->StartEdgeEdit(mAllEdgeAccessor);
...

```

This declares a `VarGrAllEdgeAccessorPtr` variable, `mAllEdgeAccessor`, and assigns an `NDVarAllEdgeAccessor` object to it. For more information about “all” edge accessors, see “Edge Accessor” on page 71.

“In” and “Out” Edge Accessors

You define “in” and “out” edge accessors relative to a particular node. That is, you create them using a node accessor as an argument. Because “in” and “out” edge accessors are based on a specific node, the APIs for navigating them traverse only those edges with one end at the specified node.

The third executable line of the following code fragment declares an “in” edge-accessor pointer, `mInEdgeAccessor`, and assigns an “in” edge-accessor object to the “in” edge-accessor pointer. The fourth line declares an “out” edge-accessor pointer, `mOutEdgeAccessor`, and assigns an “out” accessor object to it.

```
VarGrPtr mGraphDs = new NDVarGr;

// Declare a node-accessor pointer and assign a node-accessor
// object to it.
VarGrNodeAccessorPtr mNodeAccessor = new NDVarGrNodeAccessor;

```

```

// Declare "in" and "out" edge-accessor pointers and assign an
// edge-accessor to each of them based on the node-accessor
// pointer, mNodeAccessor.
VarGrInEdgeAccessorPtr mInEdgeAccessor =
    new NDDVarGrInEdgeAccessor(mNodeAccessor);
VarGrOutEdgeAccessorPtr mOutEdgeAccessor =
    new NDDVarGrOutEdgeAccessor(mNodeAccessor);

// Assign an "in" edge-edit object to the edge-edit pointer
// based on the "in" edge accessor, mInEdgeAccessor.
VarGrEditPtr mEditEdge =
    mGraphDs->StartEdgeEdit(mInEdgeAccessor);
...
mEditEdge->End();

// Assign an "out" edge-edit object to the edge-edit pointer
// based on the "out" edge accessor, mOutEdgeAccessor.
VarGrEditPtr mEditEdge =
    mGraphDs->StartEdgeEdit(mOutEdgeAccessor);
...
mEditEdge->End();
...

```

This declares **VarGrInEdgeAccessorPtr** and **VarGrOutEdgeAccessorPtr** variables, *mInEdgeAccessor* and *mOutEdgeAccessor*, respectively, and assigns **NDDVarInEdgeAccessor** and **NDDVarOutEdgeAccessor** objects to them. For more information about “in” and “out” edge accessors, see “Edge Accessor” on page 71.

“Undirected” Edge Accessors

You define an “undirected” edge accessor, as with “in” and “out” edge accessors, relative to a particular node. That is, you create them using a node accessor as an argument. Because “undirected” edge accessors are based on a specific node, the APIs for navigating them traverse only those edges with one end at the specified node.

The third executable line of the following code fragment declares an “undirected” edge-accessor pointer, *mUndirEdgeAccessor*. It then creates an “undirected” edge-accessor object and assigns the object to the “undirected” edge-accessor pointer.

```

VarGrPtr mGraphDs = new NDDVarGr;

// Declare a node-accessor pointer and assign a node-accessor
// object to it.
VarGrNodeAccessorPtr mNodeAccessor = new NDDVarGrNodeAccessor;

// Declare an "undirected" edge-accessor pointer and, based on
// the node-accessor pointer, mNodeAccessor, assign an
// "undirected" edge-accessor object to the "undirected"
// edge-accessor pointer.
VarGrUndirEdgeAccessorPtr mUndirEdgeAccessor =
    new NDDVarGrUndirEdgeAccessor(mNodeAccessor);

// Assign an "undirected" edge-edit object to the edge-edit
// pointer based on the "undirected" edge accessor,
// mUndirEdgeAccessor.
VarGrEditPtr mEditEdge =
    mGraphDs->StartEdgeEdit(mUndirEdgeAccessor);
...
mEditEdge->End();
...

```

This declares a `VarGrUndirEdgeAccessorPtr` variable, `mUndirEdgeAccessor`, and assigns an `NDVarUndirEdgeAccessor` object to it. For more information about “undirected” edge accessors, see “Edge Accessor” on page 71.

Creating Nodes

The first node you must create in your graph datasource is the first root node. After doing so, you can use either of these two techniques to add nodes to the datasource:

- Creating Linked Nodes
- Creating Unlinked Nodes

Regardless of your overall scheme, the first node you add to your datasource is always an unlinked node. This is because there are no other nodes to which it can be linked. Figure 4–16 shows how to create a node accessor, add the first root node, and set its **ID** and **Value** properties:

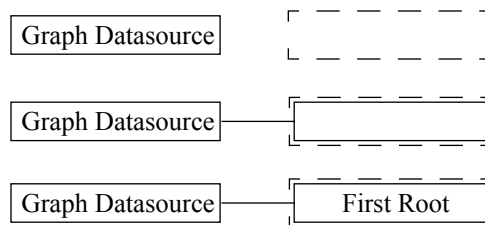


Figure 4–16 Creating the First Root Node in a Graph Datasource

This code fragment shows how to add the first node in the graph datasource:

```
VarGrPtr mGraphDs = new NDVarGr;;
VarPtr mVarID = new NDVar;
VarPtr mVarValue = new NDVar;

// Declare a node-accessor pointer variable, and assign a
// node-accessor object to it.
VarGrNodeAccessorPtr mNodeAccessor = new NDVarGrNodeAccessor;
...
// Declare an edit pointer variable, and assign a datasource
// edit object to it.
VarGrEditPtr mEditGraphDs = mGraphDs->StartEdit();

// Move mNodeAccessor to the first available node location.
mNodeAccessor->GoFirstRoot();

// Add the first root node to begin the graph.
mGraphDs->AddNode(mNodeAccessor);

// Set the node ID and Value properties.
mVarID->SetStr("0001");
mVarValue->SetStr("First Root");
mEditGraphDs->SetNodeID(mNodeAccessor, mVarID);
mEditGraphDs->SetNodeValue(mNodeAccessor, mVarValue);

mEditGraphDs->End();
```

You can also create the first root node using a “convenience” API, which creates and disposes of the edit object for you. This code fragment illustrates how to use the convenience functions to create the first root node in the datasource:

```

VarGrPtr mGraphDs = new NDVarGr;
VarGrNodeAccessorPtr mNodeAccessor = new NDVarGrNodeAccessor;
VarPtr mVarID = new NDVar;
VarPtr mVarValue = new NDVar;
...
// Move mNodeAccessor to the first available node location.
mNodeAccessor->GoFirstRoot();

// Use the convenience API to create and edit object, add the
// first root node, and dispose of the edit object.
mGraphDs->AddNode(mNodeAccessor);

// Set the node ID and Value properties using "convenience" API
// functions.
mVarID->SetStr("0001");
mVarValue->SetStr("First Root");
mGraphDs->SetNodeID(mNodeAccessor, mVarID);
mGraphDs->SetNodeValue(mNodeAccessor, mVarValue);
...

```

The “convenience” API functions:

1. Create an edit object.
2. Perform the specified operation.
3. Dispose of the edit object when the operation is complete.

Tip: Because these “convenience” functions create and dispose of an edit object for each operation, they are not very efficient when used to perform batches of edit operations.

Creating Linked Nodes

After creating the first root node, the simplest way to add nodes is to add them as child, parent, and neighbor nodes. This is because the edges connecting them are automatically created. This can save some time and effort initially, but you may want to later revisit the automatically created edges and label them.

By creating innately linked nodes, you can create these nodes relationships relative to the position of the node accessor:

- Child Nodes
- Parent Nodes
- Neighbor Nodes

Figure 4-17 shows how child, parent, and neighbor node relationships are depicted in the related sections that follow.

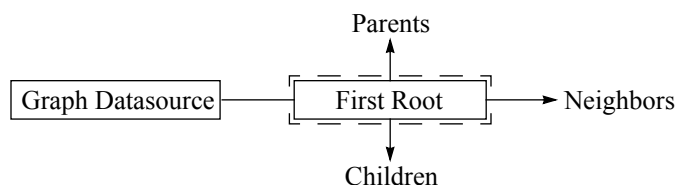


Figure 4-17 Illustration Scheme for Child, Parent, and Neighbor Nodes

This code fragment shows how to add the first node in the graph datasource and is assumed in the following sections:

```

// Create a graph datasource and the first root node.
VarGrPtr mGraphDs = new NDVarGr;
VarGrNodeAccessorPtr mNodeAccessor = new NDVarGrNodeAccessor;

```



```

VarPtr mVarID = new NDVar;
VarPtr mVarValue = new NDVar;

mNodeAccessor->GoFirstRoot();
mGraphDs->AddNode(mNodeAccessor);

// Set the node ID and Value properties using "convenience" API
// functions.
mVarID->SetStr("0001");
mVarValue->SetStr("First Root");
mGraphDs->SetNodeID(mNodeAccessor, mVarID);
mGraphDs->SetNodeValue(mNodeAccessor, mVarValue);
...

```

To add a linked node of a particular type to the end of the node list:

1. Create a graph datasource and the first root node.
2. Move the node accessor to the first node of the relationship type to be created.
3. Execute the **NDVarGrNodeAccessor::GoNext()** function repeatedly until an invalid node location is found.
4. Add a node.

Child Nodes

To add child nodes and the edges that connect them to the parent node:

1. Move the node accessor to the first child-node location.
2. Execute the **NDVarGrNodeAccessor::GoNext()** function repeatedly until an invalid node location is found.
3. Add a node.

Figure 4-18 illustrates this process for creating the first child node, which the code fragment that follows it also demonstrates:

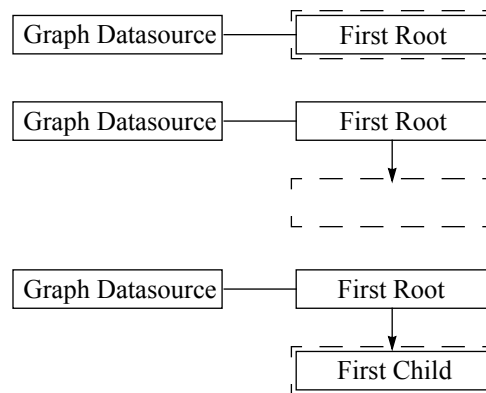


Figure 4-18 Adding the First Child Node

```

// Create a graph datasource and the first root node.
VarGrPtr mGraphDs = new NDVarGr;
VarGrNodeAccessorPtr mNodeAccessor = new NDVarGrNodeAccessor;
VarPtr mVarID = new NDVar;
VarPtr mVarValue = new NDVar;

mNodeAccessor->GoFirstRoot();
mGraphDs->AddNode(mNodeAccessor);

```

```

// Set the node ID and Value properties using "convenience" API
// functions.
mVarID->SetStr("0001");
mVarValue->SetStr("First Root");
mGraphDs->SetNodeID(mNodeAccessor, mVarID);
mGraphDs->SetNodeValue(mNodeAccessor, mVarValue);
...
// Move the node accessor to the first child-node location.
mNodeAccessor->GoFirstChild();

// Execute the GoNext() function repeatedly until an invalid
// node location is found.
while (mGraphDs->IsNodeValid(mNodeAccessor)) {
    mNodeAccessor->GoNext();
}

// The following GoNthChild() function would replace the
// preceding GoFirstChild() function and while loop.
// mNodeAccessor->GoNthChild(mGraphDs->
//                               GetNumChildren(mNodeAccessor));

// Add a node.
mGraphDs->AddNode(mNodeAccessor);

// Set the node ID and Value properties using "convenience" API
// functions.
mVarID->SetStr("0002");
mVarValue->SetStr("First Child");
mGraphDs->SetNodeID(mNodeAccessor, mVarID);
mGraphDs->SetNodeValue(mNodeAccessor, mVarValue);
...

```

Parent Nodes

To add parent nodes and the edges that connect them to the parent node:

1. Move the node accessor to the first parent node location.
2. Execute the **NDVarGrNodeAccessor::GoNext()** function repeatedly until an invalid node location is found.
3. Add a node.

When a parent node is added to the first root node, the node referenced by the node accessor is no longer a root node. The “First Parent” node in Figure 4-19 is now actually the first root node in the datasource, and the “First Parent” node is its child. In fact, the “First Root” node is the *firstchild* of the “First Parent” node.

Figure 4–19 illustrates this process for creating the first parent node, which the code fragment that follows it also demonstrates:

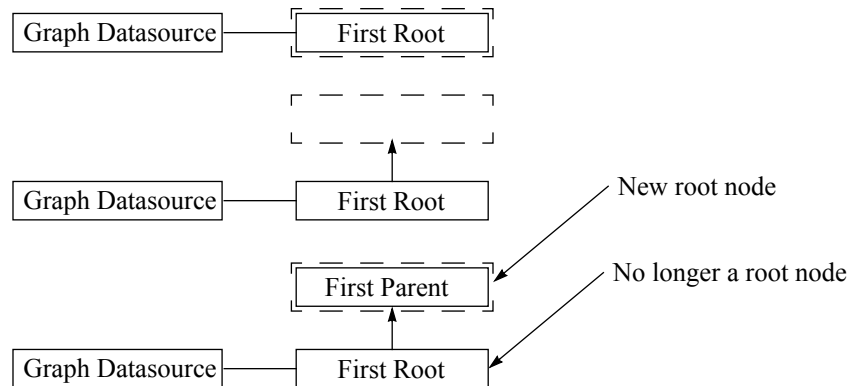


Figure 4–19 Adding the First Parent Node

```
// Create a graph datasource and the first root node.
VarGrPtr mGraphDs = new NDVarGr;
VarGrNodeAccessorPtr mNodeAccessor = new NDVarGrNodeAccessor;
VarPtr mVarID = new NDVar;
VarPtr mVarValue = new NDVar;

mNodeAccessor->GoFirstRoot();
mGraphDs->AddNode(mNodeAccessor);

// Set the node ID and Value properties using "convenience" API
// functions.
mVarID->SetStr("0001");
mVarValue->SetStr("First Root");
mGraphDs->SetNodeID(mNodeAccessor, mVarID);
mGraphDs->SetNodeValue(mNodeAccessor, mVarValue);
...
// Move the node accessor to the first parent-node location.
mNodeAccessor->GoFirstParent();

// Execute the GoNext() function repeatedly until an invalid
// node location is found.
while (mGraphDs->IsNodeValid(mNodeAccessor)) {
    mNodeAccessor->GoNext();
}

// The following GoNthParent() function would replace the
// preceding GoFirstParent() function and while loop.
// mNodeAccessor->GoNthParent(GetNumParents());

// Add a node.
mGraphDs->AddNode(mNodeAccessor);

// Set the node ID and Value properties using "convenience" API
// functions.
mVarID->SetStr("0003");
mVarValue->SetStr("First Parent");
mGraphDs->SetNodeID(mNodeAccessor, mVarID);
mGraphDs->SetNodeValue(mNodeAccessor, mVarValue);
...
```

Neighbor Nodes

To add neighbor nodes and the edges that connect them to the reference node:

1. Move the node accessor to the first neighbor-node location.
2. Execute the `NDVarGrNodeAccessor::GoNext()` function repeatedly until an invalid node location is found.
3. Add a node.

Figure 4-20 illustrates this process for creating the first neighbor node, which the code fragment that follows it also demonstrates:

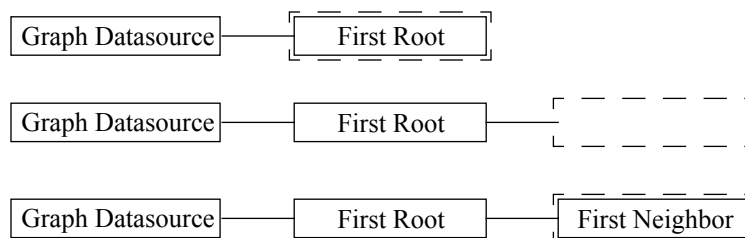


Figure 4-20 Adding the First Neighbor Node

```
// Create a graph datasource and the first root node.
VarGrPtr mGraphDs = new NDVarGr;
VarGrNodeAccessorPtr mNodeAccessor = new NDVarGrNodeAccessor;
VarPtr mVarID = new NDVar;
VarPtr mVarValue = new NDVar;

mNodeAccessor->GoFirstRoot();
mGraphDs->AddNode(mNodeAccessor);

// Set the node ID and Value properties using "convenience" API
// functions.
mVarID->SetStr("0001");
mVarValue->SetStr("First Root");
mGraphDs->SetNodeID(mNodeAccessor, mVarID);
mGraphDs->SetNodeValue(mNodeAccessor, mVarValue);
...
// Move the node accessor to the first neighbor-node location.
mNodeAccessor->GoFirstNeighbor();

// Execute the GoNext() function repeatedly until an invalid
// node location is found.
while (mGraphDs->IsNodeValid(mNodeAccessor)) {
    mNodeAccessor->GoNext();
}

// The following GoNthNeighbor() function would replace the
// preceding GoFirstNeighbor() function and while loop.
// mNodeAccessor->GoNthNeighbor(GetNumNeighbors());

// Add a node.
mGraphDs->AddNode(mNodeAccessor);

// Set the node ID and Value properties using "convenience" API
// functions.
mVarID->SetStr("0004");
mVarValue->SetStr("First Neighbor");
mGraphDs->SetNodeID(mNodeAccessor, mVarID);
mGraphDs->SetNodeValue(mNodeAccessor, mVarValue);
...
```

Creating Unlinked Nodes

To create unlinked nodes in the graph datasource, you simply add root nodes. You can add root nodes much like you add linked nodes, except that the node accessor traverses root-node locations.

To add unlinked nodes to the datasource:

1. Move the node accessor to the first root node location.
2. Execute the `NDVarGrNodeAccessor::GoNext()` function repeatedly until an invalid node location is found.
3. Add a node.

When you add a root node to the datasource, the new node is referenced by the datasource object. Figure 4-21 shows the creation of a second root node. You can add edges to an unlinked node as described in “Creating Unlinked Nodes” on page 123.

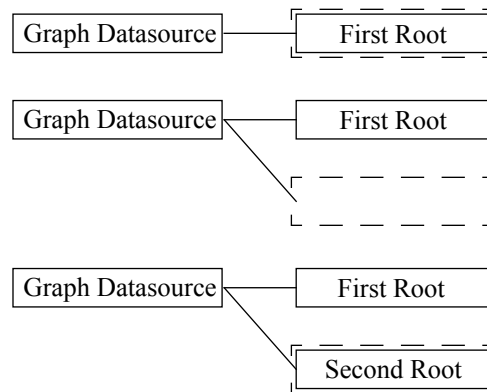


Figure 4-21 Adding an Unlinked Node

```
// Create a graph datasource and the first root node.
VarGrPtr mGraphDs = new NDVarGr;
VarGrNodeAccessorPtr mNodeAccessor = new NDVarGrNodeAccessor;
VarPtr mVarID = new NDVar;
VarPtr mVarValue = new NDVar;

mNodeAccessor->GoFirstRoot();
mGraphDs->AddNode(mNodeAccessor);

// Set the node ID and Value properties using "convenience" API
// functions.
mVarID->SetStr("0001");
mVarValue->SetStr("First Root");
mGraphDs->SetNodeID(mNodeAccessor, mVarID);
mGraphDs->SetNodeValue(mNodeAccessor, mVarValue);
...
// Move the node accessor to the first root-node location.
mNodeAccessor->GoFirstRoot();

// Execute the GoNext() function repeatedly until an invalid
// node location is found.
while (mGraphDs->IsValid(mNodeAccessor)) {
    mNodeAccessor->GoNext();
}

// The following GoNthRoot() function would replace the
// preceding GoFirstRoot() function and while loop.
// mNodeAccessor->GoNthRoot(GetNumRoots());
```

```

// Add a node.
mGraphDs->AddNode(mNodeAccessor);

// Set the node ID and Value properties using "convenience" API
// functions.
mVarID->SetStr("0003");
mVarValue->SetStr("First Parent");
mGraphDs->SetNodeID(mNodeAccessor, mVarID);
mGraphDs->SetNodeValue(mNodeAccessor, mVarValue);
...

```

Creating Edges

When adding linked nodes to a graph datasource, you do not have to manually define the edges that connect two related nodes. However, if you want to link two unlinked nodes, you must create an edge to define the relationship between them.

These two issues are of primary concern when creating edges:

- Node-Accessor and Edge-Accessor Requirements
- Adding Directed and Undirected Edges

Node-Accessor and Edge-Accessor Requirements

When adding linked nodes to the graph datasource, you only need one node accessor. However, when adding an edge to a pair of unlinked nodes, you have to have two node accessors. In addition, you must create an appropriate edge accessor to set the edge **ID** and **Value** properties.

This code fragment shows the declarations for the node and edge accessor that you need when adding edges between nodes:

```

// Create a graph datasource and node and edge accessors.
VarGrPtr mGraphDs = new NDVarGr;
VarGrNodeAccessorPtr mNodeAccessorFrom =
    new NDVarGrNodeAccessor;
VarGrNodeAccessorPtr mNodeAccessorTo =
    new NDVarGrNodeAccessor;
VarGrInEdgeAccessorPtr mInEdgeAccessor =
    new NDVarGrOutEdgeAccessor(mNodeAccessorFrom);
VarGrOutEdgeAccessorPtr mOutEdgeAccessor =
    new NDVarGrInEdgeAccessor(mNodeAccessorTo);
VarGrUndirEdgeAccessorPtr mUndirEdgeAccessor =
    new NDVarGrUndirEdgeAccessor(mNodeAccessorFrom);
...
// Move the "From" and "To" accessors to two unlinked nodes.
...

```

Adding Directed and Undirected Edges

There are some slight differences between the ways directed and undirected edges are handled. Directed edges are defined using a “source” node and a “target” node. For undirected edges, the terms “source” and “target” are irrelevant—that is, unless the **Directed** property of the edge is subject to change.

These two examples illustrate the differences between these types of edges:

- Directed Edges
- Undirected Edges

Directed Edges

The following example uses directed edges. The two node accessors are `mNodeAccessorFrom` and `mNodeAccessorTo`. The edge accessor used to

set the edge **ID** and **Value** properties is arbitrarily chosen to be an “out” edge accessor based on the node referenced by `mNodeAccessorFrom`. It could just as easily be an “in” edge accessor based on the node referenced by `mNodeAccessorTo`.

```
// Create a graph datasource and the first root node.
VarGrPtr mGraphDs = new NDVarGr;
VarGrNodeAccessorPtr mNodeAccessorFrom =
    new NDVarGrNodeAccessor;
VarGrNodeAccessorPtr mNodeAccessorTo =
    new NDVarGrNodeAccessor;
VarGrOutEdgeAccessorPtr mOutEdgeAccessor = NULL;

    new NDVarGrOutEdgeAccessor(mNodeAccessorFrom);
VarPtr mVarID = new NDVar;
VarPtr mVarValue = new NDVar;
...
// Move the "From" and "To" accessors to two unlinked nodes.
...

// Add a directed edge using mNodeAccessorFrom to identify the
// source of the edge, while using mNodeAccessorTo to identify
// the target node.
mGraphDs->AddDirEdge(mNodeAccessorFrom, mNodeAccessorTo);

// Create an "out" edge accessor for the node referenced by
// mNodeAccessorFrom, the source node.
mOutEdgeAccessor =
    VARGRNODEACCESSOR_CreateOutEdgeAccessor(mNodeAccessorFrom);

// Set the edge ID and Value properties using "convenience" API
// functions. This automatically creates an edge accessor for
// the edit operation and disposes of it for you.
mVarID->SetStr("0001");
mVarValue->SetStr("First Edge");
mGraphDs->SetEdgeID(mOutEdgeAccessor, mVarID);
mGraphDs->SetNodeValue(mOutEdgeAccessor, mVarValue);
...
```

Undirected Edges

This example uses undirected edges. The two node accessors are `mNodeAccessor1` and `mNodeAccessor2`. The edge accessor that is used to set the edge **ID** and **Value** properties is an “undirected” edge accessor based on the node referenced by `mNodeAccessor1`. It could just as easily be an “undirected” edge accessor based on the node referenced by `mNodeAccessor2`.

```
// Create a graph datasource and the first root node.
VarGrPtr mGraphDs = new NDVarGr;
VarGrNodeAccessorPtr mNodeAccessor1 =
    new NDVarGrNodeAccessor;
VarGrNodeAccessorPtr mNodeAccessor2 =
    new NDVarGrNodeAccessor;
VarGrUndirEdgeAccessorPtr mUndirEdgeAccessor = NULL;
VarPtr mVarID = new NDVar;
VarPtr mVarValue = new NDVar;

...
// Move the "1" and "2" accessors to two unlinked nodes.
...

// Add an undirected edge using mNodeAccessor1 and
// mNodeAccessor2 to identify its end points.
mGraphDs->AddUndirEdge(mNodeAccessor1, mNodeAccessor2);

// Create an "undirected" edge accessor for the node referenced
// by mNodeAccessor1.
mUndirEdgeAccessor =
```

```
VARGRNODEACCESSOR_CreateUndirEdgeAccessor(mNodeAccessor1);  
  
// Set the edge ID and Value properties using "convenience" API  
// functions. This automatically creates an edge accessor for  
// the edit operation and disposes of it for you.  
mVarID->SetStr("0001");  
mVarValue->SetStr("First Edge");  
mGraphDs->SetEdgeID(mUndirEdgeAccessor, mVarID);  
mGraphDs->SetNodeValue(mOutEdgeAccessor, mVarValue);  
...
```


This class is a facility to read arguments from the command-line.

Overview

It is similar to using the standard `argc/argv` except that:

- It can be used from any part of the application. The `argc/argv` needs to be specified only once at the program initialization.
- It supports response files. Response files can be used when the command line is too long for the operating system (command lines are limited to 128 characters on DOS, to 512 characters on VMS; there is no limitation on Mac/MPW; on Unix, there is no limitation in shell scripts but the input is limited to 512 characters when typing in an interactive shell). If one of the arguments starts with a `@`, then it is assumed to be `@<file>`. The given argument is replaced by the content of `<file>`.

Example:

If a file `myprog.opt` contains the following line:

```
-option1 -option2 -option3 -option4
```

Then the following command line:

```
myprog @myprog.opt -option5
```

will be equivalent to:

```
myprog -option1 -option2 -option3 -option4 -option5
```

API Overview

Your main routine should contain, before the initialization of the

Open Interface libraries:

```
main L2(int, argc, char**, argv)
{
    NDArgs::Init(argc, argv)
    ...
}
```

Then any class in your program can refer to the command-line arguments with:

```
Str    arg;
for (arg = NDArgs::GetFirst(); arg; arg = NDArgs::GetNext()) {
    ...
}
```

or with:

```
Int    i, nargs = NDArgs::GetNum();
for (i = 1; i < nargs; i++) {
    Str arg = NDArgs::GetNth(i);
    ...
}
```

or with:

```
ArrayPtr args = NDArgs::GetAll();
Int      index, len = args->GetLen();
for (index = 1; index < len; index++) {
    Str arg = (Str)args->GetElt(i);
    ...
}
```

If some class processes an option, it might also decide that no other class should process the same option. This can be done with:

```
NDArgs::RemoveNth(index);
```

Note: The 0th argument is usually not useful because it contains the name of the program itself. `ARGS_GetFirst()` is equivalent to `ARGS_GetNth(1)`;

Scanning the List of Command Arguments

Init

```
static void NDArgs::Init(CRTL_int argc, CRTL_char** argv);
```

Should be called from the main routine before any other initialization. Arguments like `@<file>` are replaced by the content of the given file. If `ARGS_Init` is called more than once, only the last attempt is considered. `ARGS_Init` MUST be called before any of the following calls can be used. It must also be called before the Open Interface Core library is initialized. All the other calls can only be used after the Open Interface Core library is initialized.

GetAll

```
static ArrayPtr NDArgs::GetAll(void);
```

Returns the list of all the arguments (including the application name itself). The list of arguments will be an array of strings.

GetNum

```
static ArgIVal NDArgs::GetNum(void);
```

Returns the number of arguments (including the application name itself).

GetNth

```
static CStr NDArgs::GetNth(ArgIVal n);
```

Returns the nth argument. It returns NULL if there are fewer arguments than n.

GetExecName

```
static CStr NDArgs::GetExecName(void);
```

Returns the application name.

GetFirst

static CStr NDArgs::GetFirst(void);

Returns the first argument after the application name. It returns NULL if there is no argument.

GetNext

static CStr NDArgs::GetNext(void);

Returns the next argument. GetNext can be called only after ARGV_GetFirst has been called at least once. It returns NULL when all arguments have been read.

RemoveNth

static void NDArgs::RemoveNth(ArgIVal n);

Extract the nth argument from the list.

InsertNth

static void NDArgs::InsertNth(ArgIVal n, CStr arg);

Inserts the new argument arg into the list at given index n.

The ArNum class implements a generic class corresponding to collections of numeric values.

Overview

An ArNum instance is a collection of possibly duplicate and possibly ordered elements expected to be numeric values of the same value.

The ArNum classes handle the dynamic allocation and deallocation of the internal structures that keep track of the items, with a limit in number of references being set to MAXINT32/sizeof(element) or the maximum available memory in the system.

Arnums grow as the number of elements stored in the array increases.

The ArNum classes perform deep copies: when an array object is copied into another, all the numeric values of the original are copied into the destination.

API Principle

This class implements a generic numeric values collection class.

The API is implemented in terms of macros that get compiled in the application when a particular reference collection class is defined.

The API is type-safe. If an arnum is an array of integers, all its items will be integers of the same time otherwise compiler warnings will be generated.

The following compile-time types are defined:

Type	Description
ARNUM	Type of the ArNum
ARNUM_ELT	Type of the ArNum element
ARNUM_KEY	Type of the search key

Macros

This API provides a set of macros that can be use to declare and implement collections of numeric values.

ARNUM_DECLARECLASS(ARNUM_ELT, ARNUM_KEY)

Declares the class ArNumOf<ARNUM_ELT>, the collection class that keeps track of ARNUM_ELT numeric values. The class needs to be implemented using the ARNUM_IMPLEMENTCLASS macro.

ARNUM_IMPLEMENTCLASS(ARNUM_ELT, ARNUM_KEY)

Implements the class `ArNumOf<ARNUM_ELT>`, which must have been declared using `ARNUM_DECLARECLASS`.

ARNUM_DEFCLASS(ARNUM_ELT, ARNUM_KEY)

Declares and provides an exclusively inline implementation for the `ArNumOf<ARNUM_ELT>` class, the collection class that keeps track of `ARNUM_ELT` numeric values.

ARNUM_DEFSTRUCT(STRUCT_NAME, ARNUM_ELT)

Defines the `<STRUCT_NAME>` structure implementing the collection of `<ARNUM_ELT>` numeric values.

Constructors and Destructor

Constructors

NDArNum::NDArNum(void);

Default ARNUM construction.

NDArNum::NDArNum(ArrayIVal len);

Constructs the ARNUM with 'len' elements. The contents of the ARNUM is initialized with NULL. The elements can then be set with `ARNUM_SetElt`.

Constructs the ARNUM with 0 elements but a buffer allocated for 'alloc' elements.

Then, you may fill the array by calling `ARNUM_AppendElt` and the array logic will not need to reallocate the buffer as long as the number of elements does not exceed 'alloc'.

Constructs the ARNUM as a copy of 'arnum2'. This is a deep copy.

Destructor

NDArNum::~NDArNum(void);

Default ARNUM destruction. All the values stored in the arnum are lost at this point.

Clone, Copy, Reset

Reset

void NDArNum::Reset(void);

Resets the contents of the ARNUM. After this call, the length of the ARNUM will be 0.

You are responsible for freeing the elements of the ARNUM.

Changing the Length of the Array

SetLen

void NDArNum::SetLen(ArrayIVal len);

Sets the number of elements of the ARNUM to 'len' and reallocates the contents of the ARNUM if necessary. If the ARNUM grows, the new elements are initialized with zeros.

SetAlloc

void NDArNum::SetAlloc(ArrayIVal alloc);

Reallocates the contents of the ARNUM for 'alloc' elements if necessary but does not change the number of elements in the ARNUM.

Global Queries

GetLen

ArrayIVal NDArNum::GetLen(void);

Returns the number of elements in the ARNUM.

IsEmpty

BoolEnum NDArNum::IsEmpty(void);

Returns whether the ARNUM is empty or not.

IsInRange

BoolEnum NDArNum::IsInRange(ArrayIVal i);

Returns whether 'i' is a valid index for the ARNUM (in the [0, len-1] range, where len is the length of the ARNUM).

Accessing Elements

GetNthElt

ARNUM_ELT NDArNum::GetNthElt(ArrayIVal i);

Returns the element at index 'i'. Fails if the index is not in the [0, len-1] range.

UnboundedGetNthElt

ARNUM_ELT NDArNum::UnboundedGetNthElt(ArrayIVal i);

Same as ARNUM_GetNthElt but returns 0 if 'i' is out of range instead of failing.

SetNthElt

void NDArNum::SetNthElt(ArrayIVal i, ARNUM_ELT elt);

Sets the element at index 'i'. Fails if the index is not in the [0, len-1] range. If you are replacing an existing element, you are responsible for freeing the old element (if needed).

UnboundedSetNthElt

void NDArNum::UnboundedSetNthElt(ArrayIVal i, ARNUM_ELT elt);

Same as ARNUM_SetNthElt but extends the array if 'i' is out of range and elt is not NULL ('i' must be positive).

Finding Elements

The comparison procedure used for ordering purposes is specified on a call-basis. It always takes the element as first argument, and a search key as second argument. The search key is a pointer to an object (or to void), and must not necessarily be of the same type as the reference to the stored object. You can implement and use as many ad-hoc comparison procedures as needed. The search (ARNUM_SortedLookup and ARNUM_SortedFind) uses a binary dichotomy, which makes it efficient even on large sorted arrays (search time grows in $O(\log(n))$).

ContainsElt

BoolEnum NDArNum::ContainsElt(ARNUM_ELT elt);

Returns whether or not the ARNUM contains 'elt'.

LookupElt

ArrayIVal NDArNum::LookupElt(ARNUM_ELTCPtr elt);

Returns the index of the first occurrence of 'elt' in the ARNUM. Returns -1 if the ARNUM does not contain 'elt'.

FindElt

ArrayIVal NDArNum::FindElt(ARNUM_ELTCPtr elt);

Same as ARNUM_Lookup routines but signal a failure if the ARNUM does not contain 'elt'.

SortedLookupElt

BoolEnum NDArNum::SortedLookupElt(CmpProc proc, ARNUM_KEY key, ArrayIValPtr result);

Searches element which matches key in the ARNUM. The ARNUM must be sorted in increasing order according to 'proc'. 'proc' will be called as (*proc)(elt, key) to determine how the elements of the array compare with 'key'.

Returns BOOL_TRUE and sets '*result' to the index of the matching entry if a match is found.

If no match is found, returns BOOL_FALSE and *result is set to the index at which key should be inserted if we had to insert it in the sorted array.

SortedFindElt**ArrayIVal NDArNum::SortedFindElt(CmpProc proc, ARNUM_KEY key);**

Searches element which matches 'key' in the ARNUM. The ARNUM must be sorted in increasing order according to 'proc'. 'proc' will be called as (*proc)(elt, key) to determine how the entries in the array compare with 'key'.

This routine returns the index of the element where the match occurred. If no match is found, this routine signals a failure.

Adding Elements

AppendElt**void NDArNum::AppendElt(ARNUM_ELT elt);**

Adds 'elt' at the end of the ARNUM. Does not modify the indices of the other elements in the ARNUM. The length of the ARNUM increases by one.

UniqAppendElt**void NDArNum::UniqAppendElt(ARNUM_ELT elt);**

Appends 'elt' to the ARNUM if 'elt' is not already in the ARNUM.

InsertNthElt**void NDArNum::InsertNthElt(ArrayIVal i, ARNUM_ELT elt);**

Inserts 'elt' at index 'i'. The elements which were at index 'i' or greater are moved one index further in the ARNUM. The relative order of the ARNUM elements is preserved by this call.

SortedInsertElt**ArrayIVal NDArNum::SortedInsertElt(CmpProc proc, ARNUM_ELT elt);**

Insert 'elt', using 'proc' to compare elements of the ARNUM. Returns the index at which the element was inserted.

ArrayIVal NDArNum::SortedUniqInsertElt(CmpProc proc, ARNUM_ELT elt);

Same as ARNUM_SortedXXX calls but do not insert if the element is already in the ARNUM. Return the index at which the element was inserted or found.

Removing Elements

RemoveNthElt**void NDArNum::RemoveNthElt(ArrayIVal i);**

Removes the element at index 'i'. In case 'i' is not the last index, the last element is moved to index 'i', so this routine does not preserve the ordering of the elements in the ARNUM. ARNUM_ExtractNthElt preserves the ordering but is less efficient.

RemoveElt**void NDArNum::RemoveElt(ARNUM_ELT elt);**

Removes the first occurrence of 'elt' in the ARNUM. Element 'elt' must be in the ARNUM. This call is less efficient than ARNUM_RemoveNthElt because it requires finding 'elt' in the ARNUM first. This call does not preserve the ordering of the elements in the ARNUM.

ExtractNthElt**void NDArNum::ExtractNthElt(ArrayIVal i);**

Removes the element at index 'i'. This call preserves the relative ordering of the ARNUM elements.

ExtractElt**void NDArNum::ExtractElt(ARNUM_ELT elt);****void ARNUM_ExtractElt(ArNumPtr arnum, ARNUM_ELT elt);**

Same as corresponding ARNUM_Remove calls but preserve the relative ordering of the elements in the ARNUM.

SortedExtractElt**ArrayIVal NDArNum::SortedExtractElt(CmpProc cmp, ARNUM_ELT elt);**

Extracts 'elt', using 'proc' to compare elements of the ARNUM. Returns the index at which the element was found.

Sorting

Sort**void NDArNum::Sort(CmpProc proc);**

Sorts the ARNUM. 'proc' is the procedure which will be used to compare the elements. (See basepub.h for the definition of CmpProc). This call uses the QuickSort algorithm which is very efficient on large arrays.

IsSorted**BoolEnum NDArNum::IsSorted(CmpProc proc) const;**

Returns whether a is sorted or not according to 'proc'.

Removing Duplicates

RemoveDupls**void NDArNum::RemoveDupls(void);**

Removes duplicate elements in the ARNUM.

SortedRemoveDupls

void NDArNum::SortedRemoveDupls(void);

Removes duplicates in the ARNUM, assumes that it is sorted. This routine is more efficient than ARNUM_RemoveDupls because duplicates are necessarily contiguous in this case.

ArObj Class

The ArObj class implements the generic collection of objects.

Overview

The ArObj class differs from ArPtr in that the stored elements are object values rather than pointers to external objects.

An ArObj is a collection of possibly duplicate and possibly ordered elements expected to be objects of all the same size.

The ArObj classes handle the dynamic allocation and deallocation of the internal structures that keep track of the items, with a limit in number of items being set to $\text{MAXINT32}/\text{sizeof}(\text{element})$ or the maximum available memory in the system. ArObjs grow as the number of elements stored in the array increases.

API Principle

The API is implemented in terms of macros that get compiled in the application when a particular object collection class is defined.

The API is type-safe. All objects stored in the array must be of the same type as the arrays element type or a subclass of the eleOment type, otherwise, compile-time warnings will be generated.

The following compile-time types are defined:

Identifiers	Description
AROBJ_ELT	Type of the array element
AROBJ_KEY	Type of the search

If key represents an object it must be a pointer not a value. Objects are cloned using the copy constructor for AROBJ_ELT when they are added to the array.

Array elements are destroyed using the destructor for AROBJ_ELT when the array is destroyed or made smaller or an element is set to a new value. Objects are compared using the == and != operators for AROBJ_ELT during comparison and lookup operations. Objects stored in the array must have at least the following public members:

Identifier	Description
Default constructor	AROBJ_ELT::AROBJ_ELT(void)
Copy constructor	AROBJ_ELT::AROBJ_ELT(const AROBJ_ELT&)
Destructor	AROBJ_ELT::~~AROBJ_ELT(void)

```

operator==          int AROBJ_ELT::operator==(const AROBJ_ELT&)
                    const
operator!=          int AROBJ_ELT::operator!=(const AROBJ_ELT&)
                    const

```

This class is only available in the C++ version of the Elements product.

Usage

To create arrays of a given object type the array of objects class for that type must first be declared and implemented using the macros described below.

An `AROBJ_DECLARE_xxx` macro is used in a header file to declare an array which holds a specific type of object.

An `AROBJ_IMPLEMENT_xxx` macro is used in a C++ source file to implement an array which holds a specific type of object.

`AROBJ_DECLARE_xxx` and `AROBJ_IMPLEMENT_xxx` should each appear once and only once in the files for a project for each element type (`AROBJ_ELT`).

Identifier	Description
<code>AROBJ_DECLARE_CLASS (AROBJ_ELT, AROBJ_KEY)</code>	Declares an array of objects class which holds elements of a class of type <code>AROBJ_ELT</code> . Search methods which accept a key will use a key of type <code>AROBJ_KEY</code> .
<code>AROBJ_IMPLEMENT_CLASS (AROBJ_ELT, AROBJ_KEY)</code>	Implements an array of objects class which holds elements of a class of type <code>AROBJ_ELT</code> . Search methods which accept a key will use a key of type <code>AROBJ_KEY</code> .

`AROBJ_ELT` and `AROBJ_KEY` should be the same as the values passed to corresponding `AROBJ_DECLARE_CLASS`.

Example

For example, if we want to use arrays of objects of the class `MyObj` which will be searched using a key of type `MyObj*`, we first need to declare the array in a header file (`myarray.h`). The class `MyObj` must have at least the public methods shown below.

```

class MyObj {
public:
    MyObj(void);
    MyObj(const MyObj& myObjToCopy);
    ~MyObj(void);
    int operator==(const MyObj& myObjToCompare) const;
    int operator!=(const MyObj& myObjToCompare) const;
    . };

```

Declare the array of `MyObj` objects class
`AROBJ_DECLARE_CLASS(MyObj, MyObj*)`

In a C++ source file (e.g. `myarray.cpp`) we need to implement the array of `MyObj` objects class:

```

#include "myarray.h"
AROBJ_IMPLEMENT_CLASS(MyObj, MyObj*)

```

Then we can use an array of objects of type MyObj as shown below:

```

#include "myarray.h"
void MyFunc(void)
{
    NDArObjOfMyObj arrayOfMyObjects;
    Construct empty array.
    ArrayIVal i;
    for (i = 0; i < 20; i++) {
        MyObj nextObj; Construct a instance of MyObj
        arrayOfMyObjects.AppendElt(nextObj);
        Append a copy of nextObj to the array.
    }
    MyObj* newObj = new MyObj;
    Construct a new instance of MyObj on the heap.
    arrayOfMyObjects.SetNthElt(10, *newObj);
    Set the 11'th element to a copy of newObj.
    delete newObj;
    Done with newObj.
    .
}

```

Identifier	Description
AROBJ_DECLARE_EXPORT_CLASS (AROBJ_ELT, AROBJ_KEY, LIB_DECLEXPORT)	Declares an array of objects class which holds elements of a class of type AROBJ_ELT. Search methods which accept a key will use a key of type AROBJ_KEY. LIB_DECLEXPORT can be used to specify an export/import directive to the Win16 or Win32 compilers for use when building DLL's. e.g. __declspec(dllexport)
AROBJ_IMPLEMENT_NESTED_CLASS (ENCL_SCOPE, AROBJ_ELT, AROBJ_KEY)	Like AROBJ_IMPLEMENT_CLASS except that the corresponding AROBJ_DECLARE_CLASS is placed inside a class scopedefined by ENCL_SCOPE.

For example:

```

Declares array of MyObj in scope of class Foo:
class Foo {
public:
    AROBJ_DECLARE_CLASS(MyObj, MyObjCPtr)
};

Implements array of MyObj in scope of class Foo:
Foo::NDArObjOfMyObj
AROBJ_IMPLEMENT_NESTED_CLASS(Foo, MyObj, MyObjCPtr)

```

Constructors and Destructor

Constructors

void NDArObjOfAROBJ_ELT::NDArObjOfAROBJ_ELT(void);

Default AROBJ construction.

void NDArObjOfAROBJ_ELT::NDArObjOfAROBJ_ELT(ArrayIVal len);

Constructs the array with `len' elements. The elements are initialized using the default constructor for AROBJ_ELT.

Destructor

void NDArObjOfAROBJ_ELT::~NDArObjOfAROBJ_ELT(void);

Destroys the array and all its elements..

Clone, Copy, Reset

**void NDArObjOfAROBJ_ELT::NDArObjOfAROBJ_ELT
(constNDArObjOfAROBJ_ELT arrayToCopy)'**

Constructs an array which is a copy of the array `arrayToCopy'. The elements are copied using the copy constructor for AROBJ_ELT.

Reset

void NDArObjOfAROBJ_ELT::Reset(void);

Resets the contents of the array. After this call, the length of the array will be 0.

Changing the Length

SetLen

void NDArObjOfAROBJ_ELT::SetLen(ArrayIVal len);

Sets the number of elements of the array to `len' . If the AROBJ grows, the new elements are created using the default constructor for AROBJ_ELT. If the array shrinks elements are destroyed using the destructor for AROBJ_ELT.

SetAlloc

void NDArObjOfAROBJ_ELT::SetAlloc(ArrayIVal alloc);

Reallocates the capacity of the array for `alloc' elements if necessary but does not change the number of elements in the array.

Global Queries

GetLen

ArrayIVal NDArObjOfAROBJ_ELT::GetLen(void);

Returns the number of elements in the array.

IsEmpty

BoolEnum NDArObjOfAROBJ_ELT::IsEmpty(void);

Returns whether the array is empty.

IsInRange

BoolEnum NDArObjOfAROBJ_ELT::IsInRange(ArrayIVal i);

Returns `BOOL_TRUE` if `i` is a valid index for the array (in the range `[0, len-1]` where `len` is the length of the array).

Accessing Elements

GetNthElt

AROBJ_ELT NDArObjOfAROBJ_ELT::GetNthElt(ArrayIVal i);

Returns a copy of the element at index `i`. Fails if the index is not in the `[0, len-1]` range.

GetNthEltRef

const AROBJ_ELT NDArObjOfAROBJ_ELT::GetNthEltRef(ArrayIVal i);

Returns a const reference to the element at index `i`. Fails if the index is not in the `[0, len-1]` range.

AROBJ_ELT NDArObjOfAROBJ_ELT::GetNthElt(ArrayIVal i);

Returns a reference to the element at index `i`. Fails if the index is not in the `[0, len-1]` range.

SetNthElt

void NDArObjOfAROBJ_ELT::SetNthElt(ArrayIVal i, const AROBJ_ELT elt);

Sets the element at index `i` to copy to object `elt`. Fails if the index is not in the `[0, len-1]` range.

Finding Elements

Note: The comparison procedure used for ordering purposes is specified on a call-basis.

It always takes the address of the stored object as first argument, and a search key as second argument. The search key is a pointer to an object (or to void), and must not necessarily be of the same type as the reference to the stored object. You can implement and use as many ad-hoc comparison procedures as needed.

The search (`NDArObjOfAROBJ_ELT::SortedLookup` and `NDArObjOfAROBJ_ELT::SortedFind`) uses a binary dichotomy, which makes it efficient even on large sorted arrays (search time grows in $O(\log(n))$).

ContainsElt

BoolEnum NDArObjOfAROBJ_ELT::ContainsElt(const AROBJ_ELT elt);

Returns `BOOL_TRUE` if the array contains an object equal to ``elt'`. Comparison is done using the `==` operator for `AROBJ_ELT`.

LookupElt

ArrayIVal NDArObjOfAROBJ_ELT::LookupElt(const AROBJ_ELT elt);

Returns the index of the first occurrence of an object which is equal to ``elt'`. Comparison is done using the `==` operator for `AROBJ_ELT`. Returns `-1` if the array does not contain ``elt'`.

FindElt

ArrayIVal NDArObjOfAROBJ_ELT::FindElt(const AROBJ_ELT elt);

Same as `AROBJ_Lookup` routine but signal a failure if ``elt'`.

SortedLookupElt

BoolEnum NDArObjOfAROBJ_ELT::SortedLookupElt(CmpProc proc, AROBJ_KEY key, ArrayIValPtr result);

Searches element which matches key in the array. The array must be sorted in increasing order according to ``proc'`. ``proc'` will be called as `(*proc)(addr, key)` to determine how the elements of the array compare with ``key'`. ``addr'` is the address of an element in the array. Returns `BOOL_TRUE` and sets ``*result'` to the index of the matching entry if a match is found. If no match is found, returns `BOOL_FALSE` and `*result` is set to the, index at which key should be inserted if we had to insert it in the sorted array.

SortedFindElt

ArrayIVal NDArObjOfAROBJ_ELT::SortedFindElt(CmpProc proc, AROBJ_KEY key);

Searches element which matches ``key'` in the array. The array must be sorted in increasing order according to ``proc'`. ``Proc'` will be called as `(*proc)(addr, key)`, where ``addr'` is the address of an element in the array, to determine how the entries in the array compare with ``key'`. This routine returns the index of the element where the mach occurred. If no match is found, this routine signals a failure.

Adding Elements

AppendElt

void NDArObjOfAROBJ_ELT::AppendElt(const AROBJ_ELT elt);

Adds ``elt'` at the end of the array. Does not modify the indices of the other elements in the array. The length of the array increases by one.

UniqAppendElt

void NDArObjOfAROBJ_ELT::UniqAppendElt(const AROBJ_ELT elt);

Appends `elt' to the array if `elt' is not already in the array. Comparison is done using the == operator for AROBJ_ELT.

InsertNthElt

void NDArObjOfAROBJ_ELT::InsertNthElt(ArrayIVal i, const AROBJ_ELT elt);

Inserts a copy of `elt' at index `i'. The elements which were at index `i' or greater are moved one index further in the array. The relative order of the array elements is preserved by this call.

SortedInsertElt

ArrayIVal NDArObjOfAROBJ_ELT::SortedInsertElt(CmpProc proc, const AROBJ_ELT elt);

Insert a copy of `elt', using `proc' to compare addresses the array elements. The key passed to `proc' is the address of the copy of `elt'. Returns the index at which the element was inserted. This call only applies to arrays of structures or scalars.

SortedUniqInsertElt

ArrayIVal NDArObjOfAROBJ_ELT::SortedUniqInsertElt(CmpProc proc, const AROBJ_ELT elt);

Same as NDArObjOfAROBJ_ELT::SortedInsertElt but does not insert if the element is already in the array. The key passed to `proc' is the address of `elt'. Returns the index at which the element was inserted or found.

Removing Elements

RemoveNthElt

void NDArObjOfAROBJ_ELT::RemoveNthElt(ArrayIVal i);

Removes the element at index `i'. In case `i' is not the last index, the last element is moved to index `i', so this routine does not preserve the ordering of the elements in the array. NDArObjOfAROBJ_ELT::ExtractNthElt preserves the ordering but is less efficient.

RemoveElt

NDArObjOfAROBJ_ELT::RemoveElt(const AROBJ_ELT elt);

Removes the first occurrence of `elt' in the array. Comparison is done using the == operator for AROBJ_ELT. Element `elt' must be in the array. This call does not preserve the ordering of the elements in the array.

ExtractNthElt

void NDArObjOfAROBJ_ELT::ExtractNthElt(ArrayIVal i);

Removes the element at index `i'. This call preserves the relative ordering of the array elements.

ExtractElt**void NDArObjOfAROBJ_ELT::ExtractElt(const AROBJ_ELT *elt*);**

Same as corresponding NDArObjOfAROBJ_ELT::RemoveElt call but preserves the relative ordering of the elements in the array.

SortedExtractElt**void NDArObjOfAROBJ_ELT::SortedExtractElt(CmpProc *proc*, elt)**

Extracts `elt', using `proc' to compare elements of the array. Element `elt' must be in the array. The key passed to `proc' is the address of `elt'.

Sorting

Sort**void NDArObjOfAROBJ_ELT::Sort(CmpProc *proc*)**

Sorts the array using `proc' to compare the elements. The key passed to `proc' is the address of an element.

IsSorted**BoolEnum NDArObjOfAROBJ_ELT::IsSorted(CmpProc *proc*);**

Returns BOOL_TRUE if the array is sorted according to `proc'. The key passed to `proc' is the address of an element.

Removing Duplicates

RemoveDupls**void NDArObjOfAROBJ_ELT::RemoveDupls(void);**

Removes duplicate elements in the array. The elements are compared using the == operator of AROBJ_ELT.

SortedRemoveDupls**void NDArObjOfAROBJ_ELT::SortedRemoveDupls(void);**

Removes duplicates in the array, assumes that it is sorted. The elements are compared using the != operator of AROBJ_ELT. This routine is more efficient than NDArObjOfAROBJ_ELT::RemoveDupls because duplicates are necessarily contiguous in this case.

The `ArPtr` class implements a generic class corresponding to collections of references to objects.

Technical Overview

An `ArPtr` instance is a collection of possibly duplicate and possibly ordered elements expected to be references to objects allocated and deallocated elsewhere in the application.

The `ArPtr` classes handle the dynamic allocation and deallocation of the internal structures that keep track of the items, with a limit in number of references being set to `MAXINT32/sizeof(ClientPtr)` or the maximum available memory in the system. `ArPtr`s grow as the number of elements stored in the array increases.

The `ArPtr` classes perform shallow copies: when an array object is copied into another, only the references to the objects are copied, and not the objects themselves.

API Principles

This class implements a generic reference collection class.

The API is implemented in terms of macros that get compiled in the application when a particular reference collection class is defined. The API is type-safe. If an `arptr` is an array of pointers to a given structure, its items have to be pointers to the given structure, otherwise, compile-time warnings will be generated.

The following compile-time types are defined:

Type	Description
<code>ARPTR</code>	Type of the <code>ArPtr</code>
<code>ARPTR_ELT</code>	Type of the <code>ArPtr</code> element
<code>ARPTR_KEY</code>	Type of the search key

Macros

This API provides a set of macros that can be used to declare and implement collections of pointers.

ARPTR_DECLARECLASS(ARPTR_ELT, ARPTR_KEY)

Declares the class `ArPtrOf<ARPTR_ELT>`, the collection class that keeps track of `ARPTR_ELT` pointers. The class needs to be implemented using the `ARPTR_IMPLEMENTCLASS` macro.

ARPTR_IMPLEMENTCLASS(ARPTR_ELT, ARPTR_KEY)

Implements the class `ArPtrOf<ARPTR_ELT>`, which must have been declared using `ARPTR_DECLARECLASS`.

ARPTR_DEFCLASS(ARPTR_ELT, ARPTR_KEY)

Declares and provides an exclusively inline implementation for the `ArPtrOf<ARPTR_ELT>` class, the collection class that keeps track of `ARPTR_ELT` pointers.

Constructors and Destructor

Constructors

NDArPtr::NDArPtr(void);

Default ARPTR constructor.

NDArPtr::NDArPtr(ArrayIVal len);

Constructs the ARPTR with `'len'` elements. The contents of the ARPTR is initialized with `NULL`. The elements can then be set with `ARPTR_SetElt`.

NDArPtr::ConstructAlloc (ArrayIVal alloc);

Constructs the ARPTR with 0 elements but a buffer allocated for `'alloc'` elements.

Then, you may fill the array by calling `NDArPtr::AppendElt` and the array logic will not need to reallocate the buffer as long as the number of elements does not exceed `'alloc'`.

NDArPtr::ConstructArPtr (ArPtrPtr arptr2);

Constructs the ARPTR as a copy of `'arptr2'`. This is a shallow copy. In case the elements are pointers to other objects, the ARPTR will contain the same pointers as `'arptr2'`.

Destructor

NDArPtr::~NDArPtr(void);

Default ARPTR destructor. If the ARPTR contains pointers to objects which have been allocated on the heap, only the ARPTR will be deallocated and the application is responsible for the deallocation of the objects referenced by the ARPTR.

Clone, Copy, Reset

Reset

void NDArPtr::Reset(void);

Resets the contents of the ARPTR. After this call, the length of the ARPTR will be 0. You are responsible for freeing the elements of the ARPTR.

Changing the length of the array

SetLen

void NDArPtr::SetLen(ArrayIVal len);

Sets the number of elements of the ARPTR to `len` and reallocates the contents of the ARPTR if necessary. If the ARPTR grows, the new elements are initialized with zeros.

SetAlloc

void NDArPtr::SetAlloc(ArrayIVal alloc);

Reallocates the contents of the ARPTR for `alloc` elements if necessary but does not change the number of elements in the ARPTR.

Global Queries

GetLen

ArrayIVal NDArPtr::GetLen(void);

Returns the number of elements in the ARPTR.

IsEmpty

BoolEnum NDArPtr::IsEmpty(void);

Returns whether the ARPTR is empty or not.

IsInRange

BoolEnum NDArPtr::IsInRange(ArrayIVal i);

Returns whether `i` is a valid index for the ARPTR (in the [0, len-1] range, where len is the length of the ARPTR).

Accessing Elements

GetNthElt

ARPTR_ELT NDArPtr::GetNthElt(ArrayIVal i);

Returns the element at index `i`. Fails if the index is not in the [0, len-1] range.

GetNthEltAddr

ARPTR_ELTPtr NDArPtr::GetNthEltAddr(ArrayIVal i);

Returns the address of the element at index `i`. Fails if the index is not in the [0, len-1] range.

UnboundedGetNthElt**ARPTR_EL** NDArPtr::UnboundedGetNthElt(ArrayIVal *i*);

Same as NDArPtr::GetNthElt but returns 0 if *i* is out of range instead of failing.

SetNthElt**void** NDArPtr::SetNthElt(ArrayIVal *i*, ARPTR_EL *elt*);

Sets the element at index *i*. Fails if the index is not in the [0, len-1] range. If you are replacing an existing element, you are responsible for freeing the old element (if needed).

UnboundedSetNthElt**void** NDArPtr::UnboundedSetNthElt(ArrayIVal *i*, ARPTR_EL *elt*);

Same as NDArPtr::SetNthElt but extends the array if *i* is out of range and *elt* is not NULL (*i* must be positive).

Finding Elements

The comparison procedure used for ordering purposes is specified on a call-basis.

It always takes the element as first argument, and a search key as second argument. The search key is a pointer to an object (or to void), and must not necessarily be of the same type as the reference to the stored object. You can implement and use as many ad-hoc comparison procedures as needed.

The search (NDArPtr::SortedLookup and NDArPtr::SortedFind) uses a binary dichotomy, which makes it efficient even on large sorted arrays (search time grows in $O(\log(n))$).

ContainsElt**BoolEnum** NDArPtr::ContainsElt(ARPTR_EL *elt*);

Returns whether or not the ARPTR contains *elt*.

LookupElt**ArrayIVal** NDArPtr::LookupElt(ARPTR_EL *elt*);

Returns the index of the first occurrence of *elt* in the ARPTR. Returns -1 if the ARPTR does not contain *elt*.

FindElt**ArrayIVal** NDArPtr::FindElt(ARPTR_EL *elt*);

Same as ARPTR_Lookup routines but signal a failure if the ARPTR does not contain *elt*.

SortedLookupElt

BoolEnum NDArPtr::SortedLookupElt(CmpProc proc, ARPTR_KEY key, ArrayIVaIPtr result);

Searches element which matches key in the ARPTR. The ARPTR must be sorted in increasing order according to `proc`. `proc` will be called as `(*proc)(elt, key)` to determine how the elements of the array compare with `key`.

Returns `BOOL_TRUE` and sets `*result` to the index of the matching entry if a match is found.

If no match is found, returns `BOOL_FALSE` and `*result` is set to the index at which key should be inserted if we had to insert it in the sorted array.

SortedFindElt

ArrayIVaIPtr NDArPtr::SortedFindElt(CmpProc proc, ARPTR_KEY key);

Searches element which matches `key` in the ARPTR. The ARPTR must be sorted in increasing order according to `proc`. `proc` will be called as `(*proc)(elt, key)` to determine how the entries in the array compare with `key`.

This routine returns the index of the element where the match occurred. If no match is found, this routine signals a failure.

Adding Elements

AppendElt

void NDArPtr::AppendElt(ARPTR_ELT elt);

Does not modify the indices of the other elements in the ARPTR. The length of the ARPTR increases by one.

UniqAppendElt

void NDArPtr::UniqAppendElt(ARPTR_ELT elt);

Appends `elt` to the ARPTR if `elt` is not already in the ARPTR.

InsertNthElt

void NDArPtr::InsertNthElt(ArrayIVaI i, ARPTR_ELT elt);

Inserts `elt` at index `i`. The elements which were at index `i` or greater are moved one index further in the ARPTR. The relative order of the ARPTR elements is preserved by this call.

SortedInsertElt

ArrayIVaIPtr NDArPtr::SortedInsertElt(CmpProc proc, ARPTR_ELT elt);

Insert `elt`, using `proc` to compare elements of the ARPTR. Returns the index at which the element was inserted.

SortedUniqInsertElt

ArrayIVal NDArPtr::SortedUniqInsertElt(CmpProc proc, ARPTR_ELT elt);

Same as NDArPtr::SortedXxx calls but do not insert if the element is already in the ARPTR. Return the index at which the element was inserted or found.

Removing elements

RemoveNthElt

void NDArPtr::RemoveNthElt(ArrayIVal i);

Removes the element at index `i`. In case `i` is not the last index, the last element is moved to index `i`, so this routine does not preserve the ordering of the elements in the ARPTR. NDArPtr::ExtractNthElt preserves the ordering but is less efficient.

RemoveElt

void NDArPtr::RemoveElt(ARPTR_ELT elt);

Removes the first occurrence of `elt` in the ARPTR. Element `elt` must be in the ARPTR. This call is less efficient than NDArPtr::RemoveNthElt because it requires finding `elt` in the ARPTR first. This call does not preserve the ordering of the elements in the ARPTR.

ExtractNthElt

void NDArPtr::ExtractNthElt(ArrayIVal i);

Removes the element at index `i`. This call preserves the relative ordering of the ARPTR elements.

ExtractElt

void NDArPtr::ExtractElt(ARPTR_ELT elt);

Same as corresponding NDArPtr::Remove calls but preserve the relative ordering of the elements in the ARPTR.

SortedExtractElt

ArrayIVal NDArPtr::SortedExtractElt(CmpProc cmp, ARPTR_ELT elt);

Extracts `elt`, using `proc` to compare elements of the ARPTR. Returns the index at which the element was found.

Sorting

Sort

void NDArPtr::Sort(CmpProc proc);

Sorts the ARPTR. `proc` is the procedure which will be used to compare the elements. (See basepub.h for the definition of CmpProc). This call uses the QuickSort algorithm which is very efficient on large arrays.

IsSorted

BoolEnum NDArPtr::IsSorted(CmpProc proc);

Returns whether a is sorted or not according to `proc`.

Removing Duplicates

RemoveDupls

void NDArPtr::RemoveDupls(void);

Removes duplicate elements in the ARPTR.

SortedRemoveDupls

void NDArPtr::SortedRemoveDupls(void);

Removes duplicates in the ARPTR, assumes that it is sorted. This routine is more efficient than `NDArPtr::RemoveDupls` because duplicates are necessarily contiguous in this case.

Overview

This module defines the base implementation for all Open Interface collection classes.

Collection classes can be:

- Collection of possibly duplicate items
 - (Bags)
- Indexable collections of possibly duplicate items
 - (Collections)
- Collection of unduplicated items
 - (Sets)
- Sequences of items with insert and remove operations at any index in the sequence
 - (Cover dequeues, queues, stacks)

Collection classes can be:

- Pointer-based collections:
 - Only reference to objects are stored
 - Copies are shallow copies
 - Objects are allocated, deallocated by the application.
- Value-based collections:
 - The actual value of the object is stored
 - Copies are deep copies
 - Copies are allocated, deallocated by the application.

The implementation of the collection classes takes care of all internal allocation issues. When an element is inserted, Open Interface takes care of adjusting the size of the structures used to keep track of the items in the collection.

In this version of Open Interface, the following generic collection classes are offered:

- ArPtr classes (see arptrpub.h):
 - Pointer-based collection, or integer-based collections
 - Duplicate or not items
 - Ordered or not items
 - Indexable
 - Insert/remove at any index
- ArRec classes (see arrecpub.h):
 - Uniform-size value-based collections
 - Duplicate or not items
 - Ordered or not items
 - Indexable
 - Insert/remove at any index

- ArNum classes (see arnumpub.h):
 - Uniform-size value-based collections for numeric values
 - Duplicate or not items
 - Ordered or not items
 - Indexable
 - Insert/remove at any index

Note: For compatibility reasons, this module also implements a set of macros that allow to directly manipulate instances of ARRAY. The programmer should avoid those macros, and use the classes implemented in arptrpub.h, arrecpub.h and arnumpub.h.

10 *ARRec Class*

The ArRec module implements the generic collection of records.

Overview

An ArRec is a collection of possibly duplicate and possibly ordered elements expected to be records of all the same size.

The ArRec classes handle the dynamic allocation and deallocation of the internal structures that keep track of the items, with a limit in number of items being set to `MAXINT32/sizeof(element)` or the maximum available memory in the system.

ArRecs grow as the number of elements stored in the array increases.

API Principle

The API is implemented in terms of macros that get compiled in the application when a particular record collection class is defined.

The API is type-safe. If an ArRec an array of records, what is stored is expected to be the same type of records, otherwise, compile-time warnings will be generated.

The following compile-time types are defined:

Type	Description
<code>ARREC</code>	Type of the array
<code>ARREC_ELT</code>	Type of the array element
<code>ARREC_KEY</code>	Type of the search key (must be a reference to an object).

Even though the collection classes defined through this module store object values and not references, the API calls take references to objects as arguments, and return references to objects.

Macros

This API provides a set of macros that can be use to declare and implement collections of records.

ARREC_DECLARECLASS(ARREC_ELT, ARREC_KEY)

Declares the class `ArRecOf<ARREC_ELT>`, the collection class that keeps track of `ARREC_ELT` records. The class needs to be implemented using the `ARREC_IMPLEMENTCLASS` macro.

ARREC_IMPLEMENTCLASS(ARREC_ELT, ARREC_KEY)

Implements the class `ArRecOf<ARREC_ELT>`, which must have been declared using `ARREC_DECLARECLASS`.

ARREC_DEFCLASS(ARREC_ELT, ARREC_KEY)

Declares and provides an exclusively inline implementation for the `ArRecOf<ARREC_ELT>` class, the collection class that keeps track of `ARREC_ELT` records.

Constructors and Destructor

Constructors

NDArRec::NDArRec(void);

Default ARREC construction

NDArRec::NDArRec(ArrayIVal len);

Constructs the ARREC with `len` elements. The contents of the ARREC is initialized with zeros.

Destructor

NDArRec::~~NDArRec(void);

Default ARREC destruction. All the records stored in the arrec are lost at this point.

Clone, Copy, Reset

Reset

void NDArRec::Reset(void);

Resets the contents of the ARREC. After this call, the length of the ARREC will be 0. You are responsible for freeing the elements of the ARREC.

Changing the length

SetLen

void NDArRec::SetLen(ArrayIVal len);

Sets the number of elements of the ARREC to `len` and reallocates the contents of the ARREC if necessary. If the ARREC grows, the new elements are initialized with zeros.

SetAlloc

void NDArRec::SetAlloc(ArrayIVal alloc);

Reallocates the contents of the ARREC for `alloc` elements if necessary but does not change the number of elements in the ARREC.

Global Queries

GetLen

ArrayIVal NDArRec::GetLen(void) const;

Returns the number of elements in the ARREC.

IsEmpty

BoolEnum NDArRec::IsEmpty(void) const;

Returns whether the ARREC is empty or not.

IsInRange

BoolEnum NDArRec::IsInRange(ArrayIVal i) const;

Returns whether *i* is a valid index for the ARREC (in the [0, len-1] range, where len is the length of the ARREC).

Accessing Elements

GetNthElt

ARREC_ELTPtr NDArRec::GetNthElt(ArrayIVal i) const;

Returns the address of the element at index *i*. Fails if the index is not in the [0, len-1] range.

SetNthElt

void NDArRec::SetNthElt(ArrayIVal i, ARREC_ELTPtr elt);

Sets the element at index *i*. Fails if the index is not in the [0, len-1] range.

Finding Elements

Note: The comparison procedure used for ordering purposes is specified on a call-basis. It always takes the address of the stored object as first argument, and a search key as second argument. The search key is a pointer to an object (or to void), and must not necessarily be of the same type as the reference to the stored object. You can implement and use as many ad-hoc comparison procedures as needed. The search (ARREC_SortedLookup and ARREC_SortedFind) uses a binary dichotomy, which makes it efficient even on large sorted arrays (search time grows in $O(\log(n))$).

ContainsElt

BoolEnum NDArRec::ContainsElt(ARREC_ELTPtr elt) const;

Returns whether or not the ARREC contains elt.

LookupElt**ArrayIVal NDARRec::LookupElt(ARREC_ELTCPtr elt) const;**

Returns the index of the first occurrence of elt in the ARREC. Returns -1 if the ARREC does not contain elt.

FindElt**ArrayIVal NDARRec::FindElt(ARREC_ELTCPtr elt) const;**

Same as ARREC_Lookup routine but signal a failure if elt.

SortedLookupElt**BoolEnum NDARRec::SortedLookupElt(CmpProc proc, ARREC_KEY key, ArrayIValPtr result) const;**

Searches element which matches key in the ARREC. the ARREC must be sorted in increasing order according to proc. Proc will be called as (proc)(addr, key) to determine how the elements of the ARREC compare with key. Addr is the address of an element in the ARREC.

Returns BOOL_TRUE and sets result to the index of the matching entry if a match is found.

If no match is found, returns BOOL_FALSE and result is set to the index at which key should be inserted if we had to insert it in the sorted ARREC.

SortedFindElt**ArrayIVal NDARRec::SortedFindElt(CmpProc proc, ARREC_KEY key) const;**

Searches element which matches key in the ARREC.

The ARREC must be sorted in increasing order according to proc. Proc will be called as (proc)(addr, key), where addr is the address of an element in the ARREC, to determine how the entries in the ARREC compare with key.

This routine returns the index of the element where the mach occurred. If no match is found, this routine signals a failure.

Adding elements

AppendElt**void NDARRec::AppendElt(ARREC_ELTPtr elt);**

Adds elt at the end of the ARREC. Does not modify the indices of the other elements in the ARREC. The length of the ARREC increases by one.

UniqAppendElt**void NDARRec::UniqAppendElt(ARREC_ELTPtr elt);**

Appends elt to the ARREC if elt is not already in the ARREC.

InsertNthElt**void NDArRec::InsertNthElt(ArrayIVal i, ARREC_ELTPtr elt);**

Inserts elt at index I. The elements which were at index i or greater are moved one index further in the ARREC.

The relative order of the ARREC elements is preserved by this call.

SortedInsertElt**ArrayIVal NDArRec::SortedInsertElt(CmpProc proc, ARREC_ELTPtr elt);**

Insert elt, using proc to compare addresses of the ARREC elements. Returns the index at which the element was inserted. This call only applies to ARRECs of structures or scalars.

SortedUniqInsertElt**ArrayIVal NDArRec::SortedUniqInsertElt(CmpProc proc, ARREC_ELTPtr elt);**

Same as Sorted calls but do not insert if the element is already in the ARREC. Return the index at which the element was inserted or found.

Removing Elements

RemoveNthElt**void NDArRec::RemoveNthElt(ArrayIVal i);**

Removes the element at index I. In case i is not the last index, the last element is moved to index i, so this routine does not preserve the ordering of the elements in the ARREC. ARREC_ExtractNthElt preserves the ordering but is less efficient.

RemoveElt**void NDArRec::RemoveElt(ARREC_ELTPtr elt);**

Removes the first occurrence of elt in the ARREC. Element elt must be in the ARREC. This call does not preserve the ordering of the elements in the ARREC.

ExtractNthElt**void NDArRec::ExtractNthElt(ArrayIVal i);**

Removes the element at index I. This call preserves the relative ordering of the ARREC elements.

ExtractElt**void NDArRec::ExtractElt(ARREC_ELTPtr elt);**

Same as corresponding ARREC_Remove calls but preserve the relative ordering of the elements in the ARREC.

SortedExtractElt

ArrayIVal NDArRec::SortedExtractElt(CmpProc cmp, ARREC_ELTPtr elt);

Extracts elt, using proc to compare elements of the ARREC. Returns the index at which the element was found.

Sorting

Sort

void NDArRec::Sort(CmpProc proc);

Sorts the ARREC by passing the address of the elements instead of the elements themselves to the comparison routine.

IsSorted

BoolEnum NDArRec::IsSorted(CmpProc proc) const;

Returns whether a is sorted or not according to proc.

Removing Duplicates

RemoveDupls

void NDArRec::RemoveDupls(void);

Removes duplicate elements in the ARREC.

SortedRemoveDupls

void NDArRec::SortedRemoveDupls(void);

Removes duplicates in the ARREC, assumes that it is sorted. This routine is more efficient than ARREC_RemoveDupls because duplicates are necessarily contiguous in this case.

This module implements the "Balanced Binary Tree" data structure.

Overview

This data structure is particularly adapted to hold a sorted collection of objects, especially when insertions, extractions and searches will be frequently performed and when the number of objects in the collection cannot be known in advance.

Some data structures (i.e. hash tables) may be more efficient for certain operations (i.e. search) but the balanced binary tree is a good compromise in which the three major operations (search, insertion and extraction) are reasonably efficient.

In a `AvlTree` (as implemented in this module), every node of the tree holds a key and may have two children nodes. The nodes belonging to the left branch of node `N`, if any, hold keys which are smaller than the key of `N` and the nodes belonging to its right branch, if any, hold keys which are larger than the key of `N`.

In addition, the tree is balanced, which means that the tree is reorganized when nodes are inserted or deleted so that on every node, the depths of the left and right branches do not differ by more than one. This rebalancing slows down insertion and extraction operations but guarantees that subsequent searches will be quasi optimal (searches will usually be necessary before insertions, so insertion performance is at stake too).

The API is organized around two data structures:

A `AvlNode` represents a node of the binary tree. A `AvlTree` represents the whole tree. It contains global information about the tree as well as a pointer to the root node.

The `AvlNode` structure is public so that you may subclass it and create a `AvlTree` of `MyNode` nodes (derived from `AvlNode`).

Data Structures

The following structure is mostly used to communicate information between `AVL_TreeLookupKey` and `AVL_TreeInsertNode`.

`NDAvlTreePos`

Data structure describing a position in a `AvlTree`. .

Identifiers	Descriptions
<code>Nearest</code>	closest node found
<code>NearCmp</code>	how close the closest node is

AvlTree and AvlNode Classes

The AvlTree class is the base class for AVL trees; the AvlNode class is the base class for nodes in an AVL tree.

AvlNode Class

Constructors and Destructor

Constructors

void NDAvlNode::NDAvlNode(void);

Default construction.

void NDAvlNode::NDAvlNode(ClientPtr key);

Constructs the node, and assigns 'key' to be its key.

Destructor

NDAvlNode::~~NDAvlNode(void);

Default destructor for AvlNodes.

Accessing the AvlNode Key

SetKey

void NDAvlNode::SetKey(ClientPtr key);

Changes the key in the AvlNode object. Normally you should set the key at the construction time, but you may want to use this call if you want to set the AvlNode as its own key, which may be interesting if you have subclassed the AvlNode and if the key information is in one (or several) of the subclasses fields. You are not allowed to use this call once the AvlNode has been inserted in the AvlTree.

GetKey

ClientPtr NDAvlNode::GetKey(void) const;

Returns the key stored in the AvlNode object.

Scanning AvlNodes

GetPrev

GetNext

AvlNodePtr NDAvlNode::GetPrev(void);

AvlNodePtr NDAvlNode::GetNext(void);

Return previous and next avlnode (sorted according to key comparison proc).

GetParent

GetLeftChild

GetRightChild

AvlNodePtr NDAvlNode::GetParent(void);

AvlNodePtr NDAvlNode::GetLeftChild(void);

AvlNodePtr NDAvlNode::GetRightChild(void);

Return respectively the parent, left child or right child AvlNode of the current node.

GetFirstLeaf

GetLastLeaf

AvlNodePtr NDAvlNode::GetFirstLeaf(void);

AvlNodePtr NDAvlNode::GetLastLeaf(void);

Return respectively the leftmost and rightmost descendant node of the AvlNode

AvlTree Class

Constructors and Destructor

Constructors

NDAvlTree::NDAvlTree(void);

Default constructor for Avl trees.

NDAvlTree::NDAvlTree(CmpProc cmp);

Constructs the Avl tree with 'cmp' as the key comparison procedure. It will be called as follows:

```
cmp = (*proc)(key1, key2)
```

key1 and key2 will be two keys (either AvlNodeKey field of a AvlNode or the key argument passed to AVL_TreeCurFindKeyKey or AVL_TreeLookupKey).

Destructor

NDAvlTree::~~NDAvlTree(void);

Default destructor for Avl trees.

Queries

GetLen

AvlIVal NDAvlTree::GetLen(void) const;

Returns the number of nodes in the tree.

GetFirstNode

GetLastNode

AvlNodePtr NDAvlTree::GetFirstNode(void) const;

AvlNodePtr NDAvlTree::GetLastNode(void) const;

Return the first/last AvlNodes in the tree (sorted according to the key comparison proc).

CurFindKeyKey

AvlNodePtr NDAvlTree::CurFindKeyKey(ClientCPtr key);

This call returns the AvlNode which matches 'key' and fails if no node matches key.

LookupKey

AvlNodePtr NDAvlTree::LookupKey(ClientCPtr key, AvlTreePosPtr pos);

This call returns the AvlNode which matches 'key' if 'key' is already in the Avltree, NULL otherwise. If 'pos' is not NULL, *pos describes the node next to where 'key' should be inserted. This information may be passed to AVL_TreeInsertNode.

InsertNode

void NDAvlTree::InsertNode(AvlNodePtr avlnode, AvlTreePosPtr pos);

This call inserts 'avlnode' in the Avltree at position 'pos' which should have been obtained through a call to AVL_TreeLookupKey. This call rebalances the tree if necessary.

ExtractNode

void NDAvlTree::ExtractNode(AvlNodePtr avlnode);

This call extracts 'avlnode' from 'avltree'. This call rebalances the tree if necessary.

Propagating an Action

PerfProc

PerfEnum NDAvlTree::PerfProc(AvlNodeCPtr avlnode, ClientPtr arg);

Callback function used when propagating an action.

PropagateAction

PerfEnum NDAvlTree::PropagateAction(AvlTreePerfProc proc, ClientPtr arg) const;

Calls

```
ret = (*proc)(avlnode, arg)
```

for each 'avlnode' in the tree, as long as 'ret' is PERF_CONTINUE. The nodes will be visited in the order defined by the key comparison proc. Propagating an action with this call is usually more efficient than iterating through the nodes with AVL_NodeGetNext.

Current Node API

The original design of the AvlTree API was oriented around the concept of a "current node," like many other Open Interface APIs (i.e. list box). Our experience with this API (and others) demonstrated that although this type of API has advantages (separation between queries and actions), it is somewhat heavy and unnatural to use. Also, the fact that there is only one current node complexifies the coding when queries on related nodes need to be done in the middle of an iteration loop (cursor contention).

So, the new API described above is more classic, but we have kept a cursor-oriented API for compatibility.

In this API, the current node of a AvlTree may be positioned through calls to AVL_TreeGoFirstNode routines, or by calling AVL_TreeCurFindKey. Then, the current node may be queried by calling AVL_TreeCurGetNode. The AVL_TreeCurInsertNode and AVL_TreeCurExtractNode routines are cursor-oriented version of AVL_TreeInsertNode and AVL_TreeExtractNode.

GoFirstNode

GoLastNode

void NDAvlTree::GoFirstNode(void);

void NDAvlTree::GoLastNode(void);

Positions the current node on the first or last node of the tree (sorted according to the key comparison proc).

GoPrevNode

GoNextNode

void NDAvlTree::GoPrevNode(void);

void NDAvlTree::GoNextNode(void);

Changes the current node to the previous/next node. The current node becomes NULL when these calls are applied to the first/last nodes of the tree.

GoNode

void NDAvlTree::GoNode(AvlNodePtr avlnode);

Sets the current node to be 'avlnode' (which must belong to the AvlTree).

CurGetNode**AvlNodePtr NDAvlTree::CurGetNode(void) const;**

Returns the current AvlNode in the tree. The current node must have been previously positioned by a call to AVL_TreeGoFirstNode, ..., or by a call to AVL_TreeCurFindKey.

CurGetNearestNode**AvlNodePtr NDAvlTree::CurGetNearestNode(CmpEnumPtr *cmp*) const;**

This call should be issued after a call to AVL_TreeCurFindKey and returns the node which is nearest to the key passed to AVL_TreeCurFindKey. **cmp* is set to CMP_EQUAL if the node matches the key exactly, to CMP_UNDER or CMP_OVER otherwise to indicate how the nearest node is positioned relative to the key.

CurFindKey**AvlNodePtr NDAvlTree::CurFindKey(ClientCPtr *key*);**

Searches `key' in the AvlTree. This call sets the current node to the node matching `key', to NULL if `key' is not already in the AvlTree. Information about the "nearest" node may also be obtained by calling AVL_TreeCurGetNearestNode after this call.

CurInsertNode**void NDAvlTree::CurInsertNode(AvlNodePtr *avlnode*);**

Inserts `avlnode' in the AvlTree. A call to AVL_TreeCurFindKey MUST have been done just before this call. `avlnode' becomes the current node of the tree.

CurExtractNode**void NDAvlTree::CurExtractNode(void);**

Extracts a node from the AvlTree. A call to AVL_TreeCurFindKey MUST have been done just before.

12 *Base Class*

The Base class implements a number of basic Open Interface tools, macros, and data structures.

Technical Summary

The Base is an unusual class in that it is mostly enumerated types and macros that are used throughout the Open Interface libraries. The class is composed of tools for booleans, comparisons, debugging, memory manipulation, general enumerated types, and miscellaneous macros and constants.

Of those groupings, only the debugging and memory manipulation macros are unusual. In the debugging tools, there are macros that will aid in setting up a debugging environment. This includes debugging flags, generating code only when debugging flags are on, indicating the file name and line number that source is on, not implemented yet tools, and assertion checking.

The memory manipulation tools are designed to work on a contiguous block of memory. This includes API's to clear, copy, move or set a block of memory.

The Base class is divided into the following categories.

- Standard constant definitions
- Debugging
- Memory manipulation
- Maximum integer values
- Miscellaneous Enumerated Types
- Miscellaneous Macros.

See also:

Mch, Str classes.

Basic Data Types

Double
Long

Defines portable data types for long integers and double floats.

Portable data types for long integers and double floating point numbers. These data types are described below:

Identifier	Description
Long	4-byte integer.
Double	Double floating point number (usually 8 bytes, but this is machine dependent).

Use these data types to insure cross platform portability.

Int
Int8
Int16
Int32
Int64

Data types for integers. These data types are described below:

Identifier	Description
Int	Same as int (you cannot assume that an int can hold more than 16 bits if you want your code to be portable). Note: On the Macintosh, it is defined as “short” for the THINK C environment. This lets you use the 4-byte integers option in your project and still call Open Interface libraries built with the 2-byte option.
Int8	8 bit integer (may be 16 bits if compiler does not support signed keywords)
Int16	16 bit integer
Int32	32 bit integer
Int64	64 bit integer (not supported by all operating systems).

See also

UInt/UInt8/UInt16/UInt32

UInt
UInt8
UInt16
UInt32
UInt64

Data types for unsigned integers.

These data types are described below:

Identifier	Description
UInt	Unsigned integer (may be 16 or 32 bits).
UInt8	8 bit unsigned integer
UInt16	16 bit unsigned integer
UInt32	32 bit unsigned integer
UInt64	64 bit unsigned integer (not supported by all operating systems)

See also

`Int/Int8/Int16/Int32`

MAXINT8
MAXINT16
MAXINT32
MAXINT64
MAXUINT8
MAXUINT16
MAXUINT32
MAXUINT64

Maximum integer and unsigned integer constants.

These constants represent the maximum values for each of the signed and unsigned integer types.

Identifier	Description
<code>MAXINT8</code>	Maximum 8 bit signed integer (127).
<code>MAXINT16</code>	Maximum 16 bit signed integer (32,767).
<code>MAXINT32</code>	Maximum 32 bit signed integer (2,147,483,647).
<code>MAXINT64</code>	Maximum 64 bit signed integer (2,147,483,647). Not supported by all operating systems, specifically DOS.
<code>MAXUINT8</code>	Maximum 8 bit unsigned integer (255).
<code>MAXUINT16</code>	Maximum 16 bit unsigned integer (65535).
<code>MAXUINT32</code>	Maximum 32 bit unsigned integer (4,294,967,295).
<code>MAXUINT64</code>	Maximum 64 bit unsigned integer (4,294,967,295). Not supported by all operating systems, specifically DOS.

If you want the minimum value of each of these types, use the negative of these constants for the signed types and zero for the unsigned types.

ClientPtr

Pointer that can contain 32 bits or less of client information.

typedef void C_FAR * ClientPtr;

`ClientPtr` is a data type for storing 32 bits or less of client information. On 64 bit machines with the appropriate operating system, `ClientPtr` can be a 64 bit item.

It is perfectly legal to use `ClientPtr` to hold any pointer or any integer type.

If you store less than 32-bit integer values in a `ClientPtr`, you must use the following typecasting to avoid warnings on PC compilers:

```
ClientPtr    x;
Int16       y;
x = (ClientPtr)y; // no problem with Int16
y = (Int16)(Int32)x; // cast with (Int32) to avoid warning.
```

See also

`Res Ptr`

HugePtr

Data type for a huge pointer.

typedef void C_HUGE *HugePtr;

This type of pointer is only required if you are porting to the PC (MS Windows or PM) and require structures larger than 64K.

On the PC, most buffers are less than 64K and therefore fit into a single segment. Most of the library functions (from Open Interface or from your compiler) make the assumption that all pointers arguments are contained in one segment.

Huge buffers can cross over segment boundaries and therefore require special functions to handle operations on them; if you are doing pointer arithmetic on those buffers (like p++), it is also necessary to declare the pointers as huge.

The memory manager has a 16 bytes overhead.

See also

HugeStr, HUGELIMIT

**Byte
BytePtr**

Pointer and data type for a byte.

BytePtr is a pointer to Byte, which is an unsigned 8-bit quantity.

When working with binary data, you should use the void*, Byte or BytePtr types. To work with strings, use the Char or Str types.

See also

Char, Str

HugeStr

Pointer to a huge string

HugeStr is a pointer to a huge string. This type of pointer is only required if you are porting to the PC (MS Windows or PM) and require structures larger than 64K.

On the PC, most buffers are less than 64K and therefore fit into a single segment. Most of the library functions (from Open Interface or from your compiler) make the assumption that all pointers arguments are contained in one segment.

Huge buffers can cross over segment boundaries and therefore require special functions to handle operations on them; if you are doing pointer arithmetic on those buffers (like p++), it is also necessary to declare the pointers as huge.

The memory manager has a 16 bytes overhead.

See also

HugePtr, HUGELIMIT

HUGELIMIT

Defines an upper limit for a non-huge pointer.

HUGELIMIT is a constant defining an upper limit for a non-huge pointer.

See also

HugePtr, HugeStr

BoolEnum

BoolEnum

Defines boolean values.

BoolEnum is the constant indicating a boolean (true or false) value. Many Open Interface routines return a code of this type.

Identifier	Description
BOOL_FALSE	False.
BOOL_TRUE	True.

You can take advantage of the specific values of BOOL_FALSE and BOOL_TRUE through the use of the BOOL_OF macro.

BOOL_OF

Converts an integer to a boolean.

BoolEnum BOOL_OF (Int *number*);

BOOL_OF converts the number passed to a boolean and returns a BoolEnum. It considers all integers not equal to zero to be BOOL_TRUE and all integers equal to zero to be BOOL_FALSE.

BOOL_OF is defined as:

```
#define BOOL_OF(b) ((b) ? BOOL_TRUE : BOOL_FALSE)
```

CpyEnum

CpyEnum

Defines codes for the result of a copy process.

CpyEnum is the enumerated type indicating the result of a copy process. Copy processes may be either successful or have to truncate part of the result. This enumerated type is indicating which took place.

Identifier	Description
CPY_OK	Copy was successful.
CPY_TRUNC	Truncation occurred during copy.

CmpEnum

CmpEnum

Defines codes for the result of a comparison.

CmpEnum is an enumerated type indicating the result of a comparison. Comparisons yield one of three results: $a > b$, $a < b$ and $a = b$. This enumerated type is used to indicate which took place.

Identifier	Description
CMP_UNDER	First entity was shorter/smaller/less than second entity.
CMP_EQUAL	Entities were equal.
CMP_OVER	First entity was longer/larger/greater than second entity.

INT_Compare

Compares two integers and returns a CmpEnum to indicate the result.

CmpEnum INT_Compare (Int *number1*, Int *number2*);

INT_Compare compares number1 to number2 and returns a CmpEnum to indicate the result. If number1 is less than number2 then CMP_UNDER is returned. If number1 is greater than number2 then CMP_OVER is returned. If number1 is equal to number2 then CMP_EQUAL is returned.

INT_Compare will work for any numeric type.

INT_Compare is defined as:

```
#define INT_Compare(a, b) (((a) < (b)) ? CMP_UNDER : (((b) <
(a)) ?
                        CMP_OVER : CMP_EQUAL))
```

INT_ToCmp

Converts the integer passed into a CmpEnum.

CmpEnum INT_ToCmp (Int *number*);

INT_ToCmp converts the number passed into a CmpEnum. All integers that are greater than 0 are CMP_OVER. Those equal to zero are CMP_EQUAL. And those less than zero are CMP_UNDER.

INT_ToCmp will also work for non-integer numerics as well.

INT_ToCmp is defined as:

```
#define INT_ToCmp(i) ((i) > 0) ? CMP_OVER : ((i) == 0) ?
CMP_EQUAL :
                        CMP_UNDER)
```

PerfEnum

PerfEnum

Defines codes for how a routine should propagate an action.

PerfEnum is the enumerated type indicating whether an action should be propagated or not.

Identifier	Description
PERF_STOP	Stop propagation.
PERF_CONTINUE	Continue propagation.

See also

WinPerfProc

CmpProc

Type definition for comparison functions.

CmpProc is the type definition for a comparison function that you will write. Your function must be formally declared the same as this type definition. Your function will return a CmpEnum as the result of the comparison performed on the two ClientPtr's passed.

See also

CmpEnum

VertEnum and HorzEnum

HorzEnum

Defines codes for the horizontal direction.

HorzEnum is the enumerated type indicating horizontal direction. Possible directions are left and right.

Identifier	Description
HORZ_LEFT	Horizontal direction to the left.
HORZ_RIGHT	Horizontal direction to the right.

VertEnum

Defines codes for vertical direction.

VertEnum is the enumerated type indicating vertical direction. Possible directions are up and down.

Identifier	Description
VERT_UP	Vertical upward direction.
VERT_DOWN	Vertical downward direction.

Version Enum

VersEnum

Defines codes for version numbers.

VersEnum is the enumerated type indicating the version numbers.

Identifier	Description
VERS_MAJOR	Major version number.
VERS_MINOR	Minor version number.

Debugging Macros

DBG_CHECK

Signals a failure if an expression is false.

void DBG_CHECK (*xpr*);

DBG_CHECK is a debugging macro used to determine whether an expression is valid. If the expression is true, nothing will happen. If the expression is false, DBG_CHECK will signal a failure. This macro is only active if DBG_ON is defined.

DBG_CHECK is defined as:

```
#ifdef DBG_ON
#define DBG_CHECK(t) ERR_CHECK(t)
```

See also

DBG_CHECKSTR, DBG_ERROR, DBG_FILE, DBG_LINE, DBG_NIY, DBG_ON, DBG_SOURCE

DBG_CHECKSTR

Signals a specific failure and generates a message if an expression is false.

```
#ifdef DBG_ON
#   define DBG_CHECKSTRERR_CHECKSTR (xpr, str)
#else
#   define DBG_CHECKSTR (xpr, str)
#endif
```

DBG_CHECKSTR is a debugging macro used to determine whether an expression is valid. If the expression is true, nothing will happen. If the expression is false, DBG_CHECKSTR will signal a failure and generate the error message:

```
assertion <str> failed file ... line ...
```

This macro is only active if DBG_ON is defined.

DBG_CHECKSTR is identical to DBG_CHECK but should be used in the special case when *xpr* is too long to fit on one line or it contains a quote (") symbol.

See also

DBG_CHECK, DBG_ERROR, DBG_FILE, DBG_LINE, DBG_NIY, DBG_ON, DBG_SOURCE

DBG_ERROR

Invokes `ERR_FailAssert` with the current file name and line number.

```
#ifndef DBG_ON
#   define DBG_ERRORERR_FailError ( (str) DBG_FILE,
DBG_LINE)
#else
#   define DBG_ERROR
#endif
```

`DBG_ERROR` is used to invoke an error. It makes a call to `ERR_FailAssert` with the current file name and line number (`DBG_FILE` and `DBG_LINE`).

One of the more effective places to use `DBG_ERROR` is in the default case of a switch statement. If you know that the default should never be reached, place a `DBG_ERROR` to signal a failure.

See also

`DBG_CHECK`, `DBG_FILE`, `DBG_LINE`, `DBG_NIY`, `DBG_ON`,
`DBG_SOURCE`

**DBG_FILE
DBG_LINE**

Determines the current file name and line number.

`DBG_FILE` and `DBG_LINE` are define as the compiler directives `__FILE__` and `__LINE__`, respectively. Use them to determine the current file name and line number for the line they are called from.

Statement	Description
<code>DBG_FILE</code>	Current file name.
<code>DBG_LINE</code>	Current line number.

See also

`DBG_CHECK`, `DBG_ERROR`, `DBG_NIY`, `DBG_ON`, `DBG_SOURCE`

DBG_NIY

Signals a warning to the error handler.

```
#ifndef DBG_ON
#   define DBG_NIYERR_WarnNiy ( (str)DBG_FILE, DBG_LINE)
#else
#   define DBG_NIY
#endif
```

`DBG_NIY` signals a warning to the error handler. If the default error handler is installed, a dialog will appear on the screen indicating that that routine is not implemented yet. This macro does not signal a failure, only a warning.

See also

`DBG_CHECK`, `DBG_ERROR`, `DBG_FILE`, `DBG_LINE`, `DBG_ON`,
`DBG_SOURCE`

DBG_ON

Defines whether the debugging/assertion macros are active.

DBG_ON;

DBG_ON needs to be defined in the command line or in the makefile for your compiler. This flag needs to be set if you want the debugging/assertion monitoring macros to be active. You set DBG_ON by passing it to the compiler.

For the THINK_C compiler on the Macintosh, you must define DBG_ON in the mchpub.h header file since there is no command line interface.

See also

DBG_CHECK, DBG_ERROR, DBG_FILE, DBG_LINE, DBG_NIY,
DBG_SOURCE

DBG_REQUIRE

Checks that assertion *t* is true.

```
#ifdef DBG_ON
#   define DBG_REQUIRE (t, msg)if (! (t)) ERR_Fail
(S_ModuleName, msg)
DBG_LINE)
#else
#   define DBG_REQUIRE (t, msg)
#endif
```

If assertion fails, it generates a failure with message #num loaded from current module S_ModuleName (see errpub.h).

DBG_SCCS

Holds the SCCS (Source Code Control System) name and version number of a file.

DBG_SCCS (*str*)

Macro to hold the SCCS (Source Code Control System) name and version number of a file.

See also

DBG_FILE, DBG_LINE, DBG_CHECK(expr), DBG_ERROR, DBG_NIY,
DBG_SOURCE

DBG_SOURCE

Activates source code if DBG_ON is defined.

DBG_SOURCE (source code *source*);

When DBG_ON is defined, the DBG_SOURCE macro is evaluated into the source indicated. When DBG_ON is not defined, DBG_SOURCE evaluates to nothing.

DBG_SOURCE is defined as:

```
#ifdef DBG_ON
#define DBG_SOURCE(code)   code
#else
```

```
#define DBG_SOURCE(code)
#endif
```

See also

DBG_CHECK, DBG_ERROR, DBG_FILE, DBG_LINE, DBG_NIY, DBG_ON

Exit Status

EXIT_FAIL

EXIT_OK

Returned by the “main” function to indicate whether it has completed successfully.

Flags that should be returned by the “main” function to indicate whether it has completed successfully.

Although ANSI defines 2 similar constants (EXIT_SUCCESS and EXIT_FAILURE), these constants are used differently on VMS, so use EXIT_OK and EXIT_FAIL instead for maximum portability.

BASE_NOMINMAX

Allows overriding of the MIN, MAX, EVEN, ODD, and ABS macros.

```
ifndef BASE_NOMINMAX

#ifdef MIN
#undef MIN
#endif

#ifdef MAX
#undef MAX
#endif

#ifdef EVEN
#undef EVEN
#endif

#ifdef ODD
#undef ODD
#endif

#ifdef ABS
#undef ABS
#endif

#define MIN (x, y) ((x) < (y) ? (x) : (y))
#define MAX (x, y) ((x) > (y) ? (x) : (y))

#define EVEN (x) (( (x) % 2) == 0)
#define ODD (x) (( (x) % 2) != 0)

#define ABS (x) (( (x) >= 0 ? x : - (x))

#endif /* BASE_NOMINMAX */
```

BASE_NOMINMAX is a flag that allows you to override the MIN, MAX, EVEN, ODD, and ABS macros. You might want to set this flag if you have a conflict with other definitions by the same name that are included after the basepub.h file. If set, the definitions for MIN, MAX, EVEN, ODD, and ABS override any previous definitions for these identifiers.

See also

MIN, MAX, EVEN, ODD, ABS

Miscellaneous Basic Macros

ABS

Computes the absolute value of an integer.

Int ABS (integer *number*);

ABS takes a number and returns the same number if the number is positive and returns the negative of the number if it is negative.

ABS will work on any integer type and many other numerics.

ABS is defined as:

```
#define ABS(x) ((x) >= 0 ? x : -(x))
```

C_INITOFFSET

C_OFFSET

Provides the offset in a C structure.

C_OFFSET is a macro for the offset in a C structure. Use it instead of `offsetof` to avoid problems with unsigned integer arithmetic.

C_INITOFFSET is the same macro, but without the (int) cast. Use it for static initializations only. This was mainly introduced due to a MPW 3.2 bug.

EVEN

Determines whether an integer is even.

BoolEnum EVEN (Int *number*);

EVEN determines whether the number passed is even. It returns `BOOL_TRUE` if it is even and `BOOL_FALSE` if it is odd.

EVEN is defined as:

```
#define EVEN (x) (((x) % 2) == 0) ? BOOL_TRUE : BOOL_FALSE
```

MAX

Returns the greater of the two numbers passed.

Int MAX (numeric *number1*, numeric *number2*);

MAX returns the greater of two numbers passed.

MAX is defined as:

```
#define MAX(x, y) ((x) > (y) ? (x) : (y))
```

MIN

Returns the lesser of the two numbers received.

Int MIN (numeric *number1*, numeric *number2*);

MIN returns the lesser of `number1` and `number2`. This macro will work for any numeric data type.

MIN is defined as:

```
#define MIN(x, y) ((x) < (y) ? (x) : (y))
```

NULL

Null value for pointers.

```
#ifndef NULL
#   undef NULL
#endif

#ifdef (C_ISANSI) && !defined (_WATCOMC_)
#   define NULL ( (void*)0)
#else
#   define NULL0
#endif
```

NULL is the null or empty value for pointers.

ODD

Determines whether an integer is odd.

BoolEnum ODD (integer *number*);

ODD determines whether the number passed is odd. It returns `BOOL_TRUE` if it is odd and `BOOL_FALSE` if it is even.

ODD is defined as:

```
#define ODD (x) (((x) % 2) != 0) ? BOOL_TRUE : BOOL_FALSE)
```


13 *BBuf Class*

Overview

This class provides a portable and very efficient way to read/write binary data from/to a memory buffer or a file.

It is extremely portable since it can accommodate any byte order (i.e. order of bytes inside an integer) in both the source (a file or a memory buffer) and the destination (machine-specific representation).

Examples:

- In a GIF file, numeric values are always stored in LSB format (Least Significant Byte first)
- In a TIFF file, numeric values are stored in either MSB or LSB format. The actual format is stored as a special flag at the beginning of the file.

In all these cases, you also have to consider what the natural order is on your machine (LSB format on Intel-based machines, MSB format on Sun, Mac, HP, IBM RS/6000, ..).

The main design issue in this class is to provide the kind of flexibility described above while preserving a reasonable performance.

[A] Paging Mechanism and Data Source

Rather than invoking the native File I/O routines for every read/write operation, it is usually faster to work on a local buffer and to read/write the buffer from/to the file only when necessary. This is known as a paging mechanism.

Using a paging mechanism has another advantage since it allows the same API to be used for data loaded from a file, or from a memory buffer, or from any other source (like an inter-process communication or a database query). All the source-specific functions are defined as call-back methods. You can adapt this code by choosing between several pre-defined sets of methods or by implementing your own custom methods.

Examples:

- If all the data is already loaded in a memory buffer (which should be a `BytePtr`), you can just make `BBuf` point to this buffer. Your code will look like this:

```
#define BBUF_ENDIAN BBUF_ENDIANNATIVE
#define BBUF_OPTNOPAGING
#include <bbufpub.h>
static void S_ParseBuf L2(BytePtr, buf, Int, buflen)
{
    NdBBuf          bbuf(buf, buflen);
    Int8            int8;
    Int16           int16;
    Uint32          uint32;
    bbuf->ReadInt8(&int8);
    bbuf->ReadInt16(&int16);
}
```

```

        bbuf->ReadUInt32(&int32);
        S_DoSomething(int8, int16, uint32); ..
    }

```

- If the input source is a file but you do not want to load the whole file into memory, you should open the file and make the BBuf point to the file:

```

#define BBUF_ENDIAN BBUF_ENDIANBIG
#include <filepub.h>
#include <bbufpub.h>
static void S_ParseFile Ll(Str, name)
{
    FilePtr          file = new NdFile(name);
    UInt32\          uint32;
    file->Open(FILE_IOREAD, FILE_FMTBINARY);
    NdBBuf          bbuf(file, 1, 512);
    bbbuf->ReadInt32(&uint32);
    S_DoSomething(uint32); ..
    file->Close();
    delete file;
}

```

- If the data does not come from a file but from some other input source (from inter-process communication for instance), you can install your own methods. Your code will look like this:

```

#define BBUF_ENDIAN BBUF_ENDIANNATIVE
#include <bbufpub.h>

static BBufMethodsRec S_ChannelMethods = {
    S_ChannelSeek,
    S_ChannelRead,
    S_ChannelWrite,
    S_ChannelFlush,
    S_ChannelEnd
};

static void S_ReadChannel Ll(MyChannelPtr, channel)
{
    NdBBu          bbuf((ClientPtr)channel);
    UInt32          uint32;
    bbuf->SetMethods(&S_ChannelMethods);
    ..
    bbuf->ReadUInt32(&uint32);
    S_DoSomething(uint32); ..
}

```

[B] Data Format

In the general case, it is not always possible to read numeric values from a file with a simple C assignment between integers. There are 2 things which can prevent it:

4. File endianness:

When reading multi-bytes integers (Int16, Int32, UInt16 or UInt32), the order of the bytes in a file is not necessarily the same as what the machine architecture expects. For instance, Windows bitmap files are always in Little-Endian format (also known as MSB: the Least Significant Byte is stored first). If the local machine architecture is also Little-Endian, then no conversion will be necessary. However if the current machine architecture is Big-Endian (or MSB: Most Significant Byte first), then numeric values need to be converted (bytes are swapped).

5. Data alignment:

Some machine architectures request that 2-bytes integers be always stored in memory at an even address and that 4-bytes integers be

always stored at an address which is a multiple of 4. This alignment constraint allows the Arithmetic & Logic Unit to perform some optimization on arithmetic operations. Unfortunately, this means that you can not read an integer value with a single C assignment if the value is not aligned on a normal boundary. This happens frequently when reading values from a memory buffer loaded from a file. In this occurs, the integer value must be read byte by byte.

BBuf Class

The BBuf class is the base class for I/O buffered operations.

Specialization Flags

The following flags are used to specialize the API defined in this file. These flags need to be defined before including `bbufpub.h`. Flags which are not defined explicitly will take a default value.

Type	Description
<code>BBUF_ENDIAN</code>	Should always be defined. It can be used to improve I/O performance in case the order of bytes in numeric values read from the BBuf is known at compile-time. Must be one of:
<code>BBUF_ENDIANBIG</code>	Most significant byte is stored first (example: MacPaint).
<code>BBUF_ENDIANLITTLE</code>	Least significant byte is stored first (example: Gif).
<code>BBUF_ENDIANNATIVE</code>	Bytes are stored in the same order as on the local machine (order given by <code>MCH_ENDIAN</code>).
<code>BBUF_ENDIANREVERSE</code>	Bytes are stored in the reverse order (relative to the native order given by <code>MCH_ENDIAN</code>).
<code>BBUF_ENDIANVARIABLE</code>	Bytes order is not known at compile-time. The real order (probably specified somewhere in the file) must be set at run-time with an explicit call to <code>BBUF_SetEndianness</code> (example: Tiff).

Data Structures

NDBBufMethods

Structure containing the paging methods (which will be called only if an operation can not be performed on the current page)

Type	Description
<code>SeekProc</code>	<code>SeekProc(bbuf, pos)</code> should move the current position to <code>pos</code> and load the page containing the byte at current position. The <code>PageBeginPos</code> , <code>PageBeginPtr</code> , <code>PageEndPtr</code> and <code>CurPtr</code> should be updated.
<code>ReadProc</code>	should just skip <code>n</code> bytes. The method should fail if an attempt to read past the end of data is made. The <code>PageBeginPos</code> , <code>PageBeginPtr</code> , <code>PageEndPtr</code> and <code>CurPtr</code> should be updated.

WriteProc	WriteProc(bbuf, buf, len) should write len bytes of buf to the bbuf, starting at position BBUF_CurPos. If buf is NULL, WriteProc should just skip n bytes in bbuf and leave the skipped bytes unchanged. If an attempt to write past the end of data is made, the BBuf should be expanded and TotalSize updated. If buf is NULL, extra bytes are set to 0. WriteProc should then update TotalSize, PageBeginPos, PageBeginPtr, CurPtr and PageEndPtr.
FlushProc	FlushProc(bbuf) should write any unsaved data and flush the changes. FlushProc is called by an explicit call to BBUF_Flush. FlushProc should then update PageModified.
EndProc	EndProc(bbuf) should write unsaved data, flush changes and close/terminate/deallocate anything which has been opened / initialized/allocated during or after the Init.. method. EndProc is called by an explicit call to BBUF_Destruct.

For all of these methods, if the PageModified is set to BOOL_TRUE, the current page should be saved before being paged out.

Constructors and Destructor

Constructors

NDBinBuf

NDBinBuf::NDBinBuf(EndianEnum *endianness*);

NDBBuf::NDBBuf(void);

Default construction.

NDBinBuf

NDBinBuf::NDBinBuf(BBufBytePtr *data*, BBufOffsetVal *len*, EndianEnum *endianness*);

Constructs the bbuf to point to data. The size of data must be len. For best performance, you can declare BBUF_HASSMALLBUF if you only used buffer smaller than HUGELIMIT, and you can also declare BBUF_OPTNOPAGING if you only use this type of BBuf.

NDBinBuf

NDBinBuf::NDBinBuf(FilePtr *file*, BBufPageVal *maxbufs*, BBufOffsetVal *bufsize*, EndianEnum *endianness*);

Constructs the bbuf to point to file (which must have been opened before). To improve performance, the paging methods will use several buffers of bufsize bytes. maxbufs is the maximum number of buffers. maxbufs and bufsize must be > 0. The appropriate paging methods are installed. The file should be opened in Binary mode. The file is not closed by BBUF_Destruct.

NDBinBuf

NDBinBuf::NDBinBuf(ClientPtr *data*, EndianEnum *endianness*);

NDBBuf::NDBBuf(ClientPtr *data*);

Constructs a BBuf and attaches to it some custom data (which can be eventually NULL). This data can be accessed/changed afterward with the Get/SetMethodData calls BBUF_SetMethodData.

After this call, you should probably install your custom paging methods (with `BBUF_SetMethods`) and set explicitly the total size (with `BBUF_SetTotalSize`).

Destructor

NDBinBuf::~NDBinBuf(void);

NDBBuf::~NDBBuf(void);

Destructs the bbuf. In particular, it calls the `End` method. For instance, if `Init` method had allocated some buffers, the `End` method will free them.

Read and Write Operations

ReadNBytes

void NDBBuf::ReadNBytes(HugeBytePtr ptr, BBufOffsetVal len);

Reads `len` bytes from the bbuf and puts result into `ptr`. `ptr` should be allocated for at least `len` bytes.

ReadIntx

ReadUIntx

void NDBinBuf::ReadInt8(Int8Ptr valptr);

void NDBinBuf::ReadUInt8(UInt8Ptr valptr);

void NDBinBuf::ReadInt16(Int16Ptr valptr);

void NDBinBuf::ReadUInt16(UInt16Ptr valptr);

void NDBinBuf::ReadInt32(Int32Ptr valptr);

void NDBinBuf::ReadUInt32(UInt32Ptr valptr);

Reads an `Int8`, `Int16`, `Int32`, `UInt8`, `UInt16` or a `UInt32` respectively and writes it into `valptr`.

WriteNBytes

void NDBBuf::WriteNBytes(HugeByteCPtr ptr, BBufOffsetVal len);

Writes `len` bytes of `ptr` to the bbuf. If the current position is past the end of data and if `BBuf` was initialized with `BBUF_ConstructFile`, the `Write` method will be called and `TotalSize` will be updated.

WriteIntx

WriteUIntx

void NDBBuf::WriteInt8(Int8 val);

void NDBBuf::WriteUInt8(UInt8 val);

void NDBBuf::WriteInt16(Int16 val);

void NDBBuf::WriteUInt16(UInt16 val);

void NDBBuf::WriteInt32(Int32 val);

void NDBBuf::WriteUInt32(UInt32 val);

Writes an `Int8`, `Int16`, `Int32`, `UInt8`, `UInt16`, `UInt32` respectively into the bbuf.

Flush

void NDBBuf::Flush(void);

Calls the FlushProc method. For instance, if BBuf was initialized with a file (file must be writable), the Flush method will save any local buffer which has been modified and flush the changes to the file.

Seek Operations

CurPos

BBufOffsetVal NDBBuf::CurPos(void);

Returns current position.

SeekTo

void NDBBuf::SeekTo(BBufOffsetVal pos);

Sets position to absolute offset. The new position must be between 0 and TotalSize-1.

SeekBy

void NDBBuf::SeekBy(BBufOffsetVal pos);

Sets position to offset relative to current position. The new position must be between 0 and TotalSize-1.

SkipRead

void NDBBuf::SkipRead(BBufOffsetVal pos);

Skips <n> bytes from current position. The new position must stay between 0 and TotalSize-1. Same as BBUF_SeekBy except that offset must be > 0.

SkipWrite

void NDBBuf::SkipWrite(BBufOffsetVal pos);

Skips <n> bytes from current position. If new position is beyond the end of data, the Write method is called to write zeros at the end and update the TotalSize field.

LoadCurPage

void NDBBuf::LoadCurPage(void);

Loads the current page (if needed). Although we do not encourage direct memory access, you may read directly as many as (PageEndPtr-CurPtr) bytes starting from the current position (address returned by BBUF_GetCurPtr(bb)).

After a Read, a Write or a Seek operation, CurPtr is always between PageBeginPtr and PageEndPtr, inclusive. If CurPtr is left at PageEndPtr, BBUF_LoadCurPage loads the next page and CurPtr is set to PageBeginPtr. If CurPtr is between PageBeginPtr and PageEndPtr-1, BBUF_LoadCurPage does nothing.

Accessing Private Fields

GetClientData

SetClientData

ClientPtr NDBBuf::GetClientData(void);

void NDBBuf::SetClientData(ClientPtr data);

Respectively, returns user-defined data set by `BBUF_SetClientData` and sets the `ClientData`. The `ClientData` should be used only by the client, and not by the paging methods.

GetEndianness

SetEndianness

EndianEnum NDBBuf::GetEndianness(void);

void NDBBuf::SetEndianness(EndianEnum endianness);

Respectively, returns the real order of bytes in integers and sets the real order of bytes in integers for the `bbuf`.

These calls can be used only if `BBUF_ENDIAN` is set to `BBUF_ENDIANVARIABLE`. The real order of bytes should be set at run-time by `BBUF_SetEndianness` to either `ENDIAN_BIG` or `ENDIAN_LITTLE`.

GetTotalSize

SetTotalSize

BBufOffsetVal NDBBuf::GetTotalSize(void);

void NDBBuf::SetTotalSize(BBufOffsetVal len);

Respectively, returns the total size of data and sets the total size of data for the `bbuf`. If the `BBuf` is constructed with a buffer, `Size` is initialized to the specified buffer size. If the `BBuf` is initialized with `BBUFConstructFile`, `Size` is initialized to the size of the file.

If the `BBuf` is initialized with `BBUF_ConstructData`, `Size` should be set explicitly with `BBUF_SetTotalSize`.

`TotalSize` is updated if an attempt to write past the end of the data is made. `TotalSize` can not decrease.

The following fields should not be accessed or modified by the client code, but only by the paging methods:

GetPagingData

SetPagingData

ClientPtr NDBBuf::GetPagingData(void);

void NDBBuf::SetPagingData(ClientPtr data);

Respectively, returns `PagingData` and modifies `PagingData`. If `BBuf` is initialized with `BBUF_ConstructBuf`, `PagingData` is set to `NULL`.

If `BBuf` is initialized with `BBUF_ConstructFile`, `PagingData` is set to the specified file. If `BBuf` is initialized with `BBUF_ConstructData`, `PagingData` is set to the specified `ClientPtr`.

IsPageModified
SetPageModified

BoolEnum NDBBuf::IsPageModified(void);
void NDBBuf::SetPageModified(BoolEnum *mod*);

Respectively, returns BOOL_TRUE if current page has been modified, BOOL_FALSE otherwise, and sets/unsets the PageModified flag.

GetPageBeginPos
SetPageBeginPos

BBufOffsetVal NDBBuf::GetPageBeginPos(void);
void NDBBuf::SetPageBeginPos(BBufOffsetVal *pos*);

Respectively, returns the offset to the first byte in current page, and sets the offset to the first byte in current page.

GetPageBeginPtr
SetPageBeginPtr

BBufBytePtr NDBBuf::GetPageBeginPtr(void);
void NDBBuf::SetPageBeginPtr(BBufBytePtr *pageBeg*);

Respectively, returns a pointer to the first byte of current page, and sets the pointer to the first byte of current page.

GetPageEndPtr
SetPageEndPtr

BBufBytePtr NDBBuf::GetPageEndPtr(void);
void NDBBuf::SetPageEndPtr(BBufBytePtr *pageEnd*);

Respectively, returns a pointer to the first byte after current page, and sets the pointer to the first byte after current page. The page size can be computed with:

`PageSize = PageEndPtr - PageBeginPtr.`

GetCurPtr
SetCurPtr

BBufBytePtr NDBBuf::GetCurPtr(void);
void NDBBuf::SetCurPtr(BBufBytePtr *cur*);

Respectively, returns a pointer to the byte at current position, and modifies the pointer to the byte at current position. The CurPtr should always be between PageBeginPtr and PageEndPtr-1.

The current position offset can be computed with:

`CurPos = CurPtr - PageBeginPtr + PageBeginPos.`

Installing Custom Paging Methods

QueryMethods

void NDBBbuf::QueryMethods(BBufMethodsPtr methods);

Fills methods with the methods installed in the bbuf.

SetMethods

void NDBBbuf::SetMethods(BBufMethodsPtr methods);

Installs the methods in methods in the bbuf.

14 *Cell Class*

The Cell class implements the Open Interface cell and range data structures and tools.

Technical Summary

More specifically, this class implements the CellPtr, CellRec, RangePtr, and RangeRec data structures as well as a utility to determine whether a cell is within a specified range.

The cell and range structures are similar to the Point16 and Rect16 structure, the difference being that the names of the cell structure fields (Col, Row) are better suited to represent cells in a table than the (x,y) fields of the Point16 data structure.

See also

Rect, LBox classes

Data Structures

CellPtr
CellRec

Pointer and data structure for cells.

CellPtr is a pointer to CellRec, a data structure that stores the row and column indices of a cell.

This structure is the same as Point16Rec but for use with cells.

See also

RANGE_ContainsCell.

RangePtr
RangeRec

Pointer and data structure for ranges.

RangePtr is a pointer to RangeRec, the data structure that stores the origin and extent of a range. The fields of this structure are described below.

Field	Description
Ori	Coordinates of the top left cell of the table.
Ext	Ext.Col determines the width of the table and Ext.Row determines its height.

See also

`RANGE_ContainsCell`

Cell Range Operations

ContainsCell

Determines whether a cell is within a range.

`RANGE_ContainsCell` determines whether a cell is within a range. Returns `BOOL_TRUE` if the cell is within the limits established by the range, otherwise it returns `BOOL_FALSE`.

15 *Char Class*

The Char class implements the Open Interface character data structures and utilities.

Technical Summary

The functions in this class offers support for English, European, and Asian languages by providing functions which handle single-byte and multibyte characters.

Languages

The culturally dependent rules to control collation, case conversions, word delimitation, and so on are encapsulated in a language environment object. The LgEnv class gives more detailed information about language environments and the resources which parameterize them.

The APIs which do not take any LgEnvPtr argument perform operations without taking into account cultural specificities (i.e. case conversions limited to the ASCII range).

The APIs which take into account cultural specificities take a LgEnvPtr argument. If you pass NULL in this argument, the default language environment (as defined by the ND_CHARLANG environment variable) will be assumed.

Character Types

Open Interface APIs allow your application to support a single native language or a more general environment with more than one language or character set. Writing your application using the Open Interface APIs enables you to switch language environments by resetting an environment variable.

Open Interface offers two basic data types, Native and UNICODE.

The Native Character Type

If you intend your application to operate in one language at a time, you can use native types for your data. Many systems dedicated to a specific locale already have a native code type specified.

When you use the Native calls, write your application with `if` statements and include separate pieces of code for each language. For example, if the native code type is SJIS, the application will perform different operations than if the native language is English and it will require a different set of APIs.

While this process can result in duplicated code, switching languages is as easy as setting an environment variable (ND_CHARNATIVE, see below) to change the native language.

The UNICODE Character Type

This type supports UNICODE characters. UNICODE strings contain UNICODE characters.

Conversion

Open Interface provides APIs which enable you to convert strings and characters from one type to another.

Character Encoding

The ChCode and NatCode types encode multibyte characters in an unsigned 32 bit integer. ChCode and NatCode types contain four bytes: Byte1, Byte2, Byte3 and Byte4, with Byte1 being the least significant byte and Byte4 being the most significant.

Multibyte character encoding is shown in the following table:

Byte Number	Contents
Byte1	First byte of the multibyte character.
Byte2	Second byte of the multibyte character, or NULL.
Byte3	Third byte of the multibyte character, or NULL.
Byte4	NULL.

Char and ChCode values are always identical for pure ASCII characters, but differ for multibyte characters.

Environment Variables and Flags

The ND_CHARNATIVE environment variable defines the native language for the application. When you want to change from one native language to another, you must reset this environment variable. This cannot be done dynamically.

Open Interface Character API's

The APIs in Open Interface Char class enable you to manipulate characters and obtain information about them. The APIs let you get a character code, obtain an ASCII character's classification, convert ASCII characters, convert characters between data types, convert between ASCII and EBCDIC, get a character length, and get a specified byte of a character.

The basic character classification APIs enable you to obtain information such as whether the character is alphanumeric, hexadecimal, a control character, or a space. The NDChar::AsciiIs APIs assume that the character is in the C RTL classification specified.

Char Class Operations

- Testing whether a character is ASCII.
- Converting between ASCII and EBCDIC.
- Getting character information (such as length).

Char and NatChar Data Types

The Char and NatChar data types are defined in the base class.

Environment Variables

ND_CHARNATIVE

Defines the native code type in which NatStr and NatChar objects are encoded.

On an ASCII-based machine, you cannot choose an EBCDIC-based native code type. Similarly, on an EBCDIC-based machine, you cannot choose an ASCII-based native code type. Character constants (e.g., 'a') have been set to their ASCII or EBCDIC values at compile time. As a result, code which has been compiled on an EBCDIC host assumes that the Char type is EBCDIC based.

ND_CHARLANG

Defines the default language environment which defines the precise set of rules for string collation, case conversion, word delimitation, and so on.

Data Structures

CharPtr

Data type for a global character pointer.

See also

Char

ChCodePtr

Data type for a character code pointer.

See also

ChCode

NatCharPtr

Data type for a native character pointer.

See also

NatChar

NatCodePtr

Data type for a native character code pointer.

See also

NatCode

UnicodePtr

Data type for a UNICODE character pointer.

See also

Unicode

UniStrPtr

Data type for a UNICODE string pointers.

See also

UniStr

ChCode

Data type for a multibyte character code. ChCode is an unsigned 32-bit integer.

See also

ChCodePtr

NatCode

Data type for a native character code. NatCode is an unsigned 32-bit integer.

See also

NatCodePtr

Unicode

Data type for a UNICODE character. A Unicode character is an unsigned 16-bit integer.

See also

UnicodePtr

UniStr

Data type for a UNICODE string.

See also

UniStrPtr

CharInfoVal

A 32-bit integer used for character classification information.

StrIVal

A 32-bit integer used for indexing strings and characters.

Character Length

GetLen

Returns the length of the character whose first byte contains the specified 8-bit character.

static StrIVal NDChar::GetLen(Char *char*);

NDChar::GetLen returns the length in bytes of the character whose first byte contains the specified 8-bit character.

The length result depends on the values of the ND_CHARNATIVE environment variable.

See also

NDChar::CodeGetLen, NDChar::NatGetLen

CodeGetLen

Returns the length of the character whose first byte contains the specified 8-bit character code.

static StrIVal NDChar::CodeGetLen(ChCode *chcode*);

Returns the length in bytes of the character whose first byte contains the specified 8-bit character code. The length result depends on the values of the ND_CHARNATIVE environment variable.

See also

NDChar::GetLen, NDChar::NatGetLen

NatGetLen

Returns the length of the native character whose first byte contains the specified 8-bit character code.

static StrIVal NDChar::NatGetLen(NatChar *natchar*);

NDChar::NatGetLen returns the length in bytes of the character whose first byte contains the specified 8-bit character code. The length result depends on the values of the ND_CHARNATIVE environment variable.

See also

NDChar::GetLen, NDChar::CodeGetLen

Character Code

The ``ChCode'` and ``NatCode'` types encode multibyte characters in a unsigned 32 bit integer.

If `b1`, `b2`, `b3` and `b4` are the bytes of a ``ChCode'` or ``NatCode'`, `b1` begin the least significant byte and `b4` the most significant, the multi-byte

character is encoded as follows:

- b1 : first byte of the multi byte character.
- b2 : second byte of the multi byte character, or 0.
- b3: third byte of the multi byte character, or 0.
- b4: 0 (for now).

With this encoding, the Char and ChCode values are always identical for pure ASCII characters, but will differ on multi-byte characters.

You can extract information from a multi-byte character with the following API:

GetByte...

Returns the contents the specified byte of a multibyte character.

```
static Char NDChar::GetByte(ChCode chcode, Int bytenum);  
static Char NDChar::GetByte1(ChCode chcode);  
static Char NDChar::GetByte2(ChCode chcode);  
static Char NDChar::GetByte3(ChCode chcode);
```

Returns the contents of the specified byte of a multibyte character.

The NDChar::GetByte function takes a byte number as an argument. You can specify a byte number between zero (the first byte) and 2 (the third byte).

NDChar::GetByte1 obtains the first byte, NDChar::GetByte2 obtains the second byte, and NDChar::GetByte3 obtains the third byte of a multibyte character.

See also

NDChar::NatGetByte

NatGetByte...

Returns the contents the specified byte of a multibyte character.

```
static NatChar NDChar::NatGetByte(NatCode natcode, Int byteEnum);  
static NatChar NDChar::NatGetByte1(NatCode natcode);  
static NatChar NDChar::NatGetByte2(NatCode natcode);  
static NatChar NDChar::NatGetByte3(NatCode natcode);
```

Returns the contents of the specified byte of a native multibyte character.

The NDChar::GetByte function takes a byte number as an argument. You can specify a byte number between zero (the first byte) and 2 (the third byte). NDChar::GetByte1 obtains the first byte, NDChar::GetByte2 obtains the second byte, and NDChar::GetByte3 obtains the third byte of a native multibyte character.

See also

NDChar::GetByte

Basic Character Classification

IsAscii...

Determines whether the character is ASCII.

```
static BoolEnum NDChar::IsAscii(ChCode chcode);
static BoolEnum NDChar::IsAsciiAlpha(ChCode chcode);
static BoolEnum NDChar::IsAsciiUpper(ChCode chcode);
static BoolEnum NDChar::IsAsciiLower(ChCode chcode);
static BoolEnum NDChar::IsAsciiAlNum(ChCode chcode);
static BoolEnum NDChar::IsAsciiDigit(ChCode chcode);
static BoolEnum NDChar::IsAsciiHexDigit(ChCode chcode);
static BoolEnum NDChar::IsAsciiOctDigit(ChCode chcode);
static BoolEnum NDChar::IsAsciiSpace(ChCode chcode);
static BoolEnum NDChar::IsAsciiPunct(ChCode chcode);
static BoolEnum NDChar::IsAsciiControl(ChCode chcode);
static BoolEnum NDChar::IsAsciiPrint(ChCode chcode);
static BoolEnum NDChar::IsAsciiGraph(ChCode chcode);
```

The NDChar::IsAscii... macros classify characters according to the C RTL standard rules. Use these macros if you need to classify ASCII characters only. The return value is FALSE if the given character is not an ASCII character.

On an EBCDIC system, the NDChar::IsAscii... macros automatically assume that the chcode argument is an EBCDIC character code, not an ASCII code.

The various versions of the NDChar::IsAscii... macros are described in the following table:

Macro	Inquiry
NDChar::IsAscii(chcode)	Does the character belong to the ASCII set?
NDChar::IsAsciiAlpha(chcode)	Is the character an ASCII letter?
NDChar::IsAsciiUpper(chcode)	Is the character an ASCII upper case letter?
NDChar::IsAsciiLower(chcode)	Is the character an ASCII lower case letter?
NDChar::IsAsciiAlNum(chcode)	Is the character an ASCII letter or a digit?
NDChar::IsAsciiDigit(chcode)	Is the character an ASCII digit?
NDChar::IsAsciiHexDigit(chcode)	Is the character an ASCII hexadecimal digit?
NDChar::IsAsciiOctDigit(chcode)	Is the character an ASCII octal digit?
NDChar::IsAsciiSpace(chcode)	Is the character an ASCII space character?
NDChar::IsAsciiPunct(chcode)	Is the character an ASCII punctuation?

<code>NDChar::IsAsciiControl(chcode)</code>	Is the character an ASCII control character?
<code>NDChar::IsAsciiPrint(chcode)</code>	Is the character an ASCII printable character?
<code>NDChar::IsAsciiGraph(chcode)</code>	Is the character an ASCII "graph" character?

The `ChCode` value corresponds to the `Char` value on the ASCII range, so you can pass either `ChCode` or `Char` values to the these calls.

See also

`NDChar::AsciiIs...`

AsciiIs...

Same as `NDChar::IsAscii...` macros except that an error is generated if the character is not ASCII.

```
static BoolEnum NDChar::AsciiIsAlpha(Char char);
static BoolEnum NDChar::AsciiIsUpper(Char char);
static BoolEnum NDChar::AsciiIsLower(Char char);
static BoolEnum NDChar::AsciiIsAlNum(Char char);
static BoolEnum NDChar::AsciiIsDigit(Char char);
static BoolEnum NDChar::AsciiIsHexDigit(Char char);
static BoolEnum NDChar::AsciiIsOctDigit(Char char);
static BoolEnum NDChar::AsciiIsSpace(Char char);
static BoolEnum NDChar::AsciiIsPunct(Char char);
static BoolEnum NDChar::AsciiIsControl(Char char);
static BoolEnum NDChar::AsciiIsPrint(Char char);
static BoolEnum NDChar::AsciiIsGraph(Char char);
```

The `NDChar::AsciiIs...` functions are similar to the corresponding `NDChar::IsAscii...` macros except that they assume that the character is ASCII and they signal an error if the character is not ASCII (for debugging libraries only).

See also

`NDChar::IsAscii`

Basic Character Conversion

```
AsciiDigitGetInt
AsciiHexDigitGetInt
AsciiOctDigitGetInt
```

Returns the integer value of an ASCII digit.

static Int NDChar::AsciiDigitGetInt(Char digit);
static Int NDChar::AsciiHexDigitGetInt(Char digit);
static Int NDChar::AsciiOctDigitGetInt(Char digit);

Returns the integer value of an ASCII digit. The digit argument must be a decimal, hexadecimal, or octal digit or else the result is unreliable.

In the debugging library, this function signals an error if the input is not an ASCII digit.

See also

NDChar::AsciiDigitGetBase

AsciiAlphaGetBase

Returns the base value of an ASCII letter.

static Int NDChar::AsciiAlphaGetBase(Char char);

NDChar::AsciiAlphaGetBase returns the base value of an ASCII letter. The base value is an integer between 0 and 25. The char argument must be an ASCII letter or else the result is unreliable.

In the debugging library, this function signals an error if the input is not an ASCII character.

See also

NDChar::AsciiDigitGetInt

AsciiGetLower

Converts an ASCII character to lower case.

static Char NDChar::AsciiGetLower(Char char);

NDChar::AsciiGetLower converts an ASCII character to lower case. The char argument must be an ASCII character or else the result is unreliable.

In the debugging library, this function signals an error if the input is not an ASCII character.

See also

NDChar::AsciiGetUpper

AsciiGetUpper

Converts an ASCII character to upper case.

static Char NDChar::AsciiGetUpper(Char char);

NDChar::AsciiGetUpper converts an ASCII character to upper case. The char argument must be an ASCII character or else the result is unreliable.

In the debugging library, this function signals an error if the input is not an ASCII character.

See also

NDChar::AsciiGetLower

AsciiGetControl

Converts a character to a control character.

static Char NDChar::AsciiGetControl(Char *char*);

NDChar::AsciiGetControl converts a character to a control character. An ASCII character is converted to control characters in the [00-1f] + 7f range. EBCDIC characters are converted to EBCDIC control codes.

Converts A and a to ^A.

Converts [and { to ^[.

Converts ? to DEL.

With the debugging library, this function signals an error if the input is not an ASCII or EBCDIC character.

See also

NDChar::AsciiGetGraph

AsciiGetGraph

Converts a control character into a character.

static Char NDChar::AsciiGetGraph(Char *controlchar*);

NDChar::AsciiGetGraph converts control characters into characters. For ASCII, converts a control character into the corresponding ASCII character in the [0x3f-0x5f] range. Converts EBCDIC control characters to EBCDIC codes. Converts DEL to ?.

In the debugging library, this function signals an error if the input is not an ASCII or EBCDIC control character.

See also

NDChar::AsciiGetControl

AsciiGetEbcDic

Converts an ASCII character to an EBCDIC character.

static Byte NDChar::AsciiGetEbcDic(Byte *byte1*);

NDChar::AsciiGetEbcDic converts an ASCII character to the first byte of an EBCDIC character.

See also

NDChar::EbcDicGetAscii

Conversions between ASCII and EBCDIC

EbcdicGetAscii

Converts an EBCDIC character to an ASCII character.

static Byte NDChar::EbcdicGetAscii(Byte *byte1*);

NDChar::EbcdicGetAscii converts an EBCDIC character to an ASCII character. If the EBCDIC character does not belong to the ASCII set, the value returned is in the [80-ff] range.

You rarely need to convert EBCDIC codes to ASCII, but in some cases you may need to. For example, you might want to compare strings according to the ASCII order, or use `lex` and `yacc` tables which were generated on an ASCII host.

See also

NDChar::AsciiGetEbcdic

ToAscii

Converts a native character to ASCII.

static Byte NDChar::ToAscii(Char *natchar*);

Converts a native character to ASCII. On an ASCII host, this function does nothing.

See also

NDChar::FromAscii

FromAscii

Converts an ASCII code to a native character.

static Char NDChar::FromAscii(Byte *byte*);

NDChar::FromAscii converts an ASCII code to a native character. On an ASCII host, this function does nothing.

See also

NDChar::ToAscii

16 *Cs Class*

The Cs module defines a generic "code set" data structure. See the definition of code types, code sets, and code mappings in charpub.h.

Overview

A "code set" must define 3 methods:

- GetCharInfo()

which should return character information of the code set;

- CvtChar()

which should convert a character of the code set; and

- TransChar()

which should translate a character from the specified code set.

These member functions vary depending on the code set that is specified at creation time. This module is used mainly from the Ct module.

Code Sets

Unfortunately, there are a fairly large number of standard code sets, and many manufacturers have "extended" the standard code sets in proprietary ways. To minimize the amount of overlap between reference code sets, we will consider that the overall coding scheme combines code sets in such cases.

For example, the Microsoft Windows ANSI 1252 code set combines the ISO 8859-1 code set (a0-ff range) and MS/Windows extensions in the 80-9f range (special quotes, bullet). So, a code set is actually defined by a combination of several code sets and a code mapping.

Some coding schemes assign glyphs to control characters in the 00-1f range (i.e. Macintosh lozenge). This will be handled by defining a coding scheme with a special code set which partially covers the 00-1f range.

Also, some coding schemes assign non-standard glyphs to some ASCII characters. (For example, Japanese fonts have a Yen sign instead of a backslash, and an overbar instead of tilde). We will still consider these characters to be the ASCII character, because most existing software treats them according to their ASCII semantics, not according to their actual glyphs.

CsIdEnum

Data type for code set id.

ISO Code Set

These are the strict ISO code sets, which only cover the a0-ff range.

Type	Description
CS_ASCII	
CS_ISO_LATIN1	
CS_ISO_LATIN2	
CS_ISO_LATIN3	
CS_ISO_LATIN4	
CS_ISO_CYRILLIC	
CS_ISO_ARABIC	
CS_ISO_GREEK	
CS_ISO_HEBREW	
CS_ISO_LATIN9	
CS_EMPTY_809f	covers the 80-9f range by not associating any character to such codes

Various extensions for the 80-9f range (i.e. ANSI 1252) are considered as separate code sets.

ADOBE Code Sets

Type	Description
CS_ADOBE_STD	covers the a0-ff range
CS_ADOBE_LATIN1	extends ISO_LATIN1 in the 80-9f range
CS_ADOBE_SYMBOL	covers the 20-7f and a0-ff ranges
CS_ADOBE_ZAPF_DB	covers the 00-ff range (to be verified)

Macintosh Code Sets

The Macintosh Roman character set is completely different from ISO_LATIN1 and covers the 80-ff range.

The Macintosh defines extensions to the ISO_ARABIC and 8 for Arabic and Hebrew. These extensions cover at least the 80-9f range (R2L variants of corresponding ASCII punctuations), but also fill empty slots of the a0-ff range.

We still have to investigate whether there are significant differences or not between the Macintosh Greek code set and the ISO_GREEK (the UNICODE document says that they are identical) and between the Macintosh symbol

font and the ADOBE symbol font (the UNICODE document gives Mac addition in the 00-1f range).

Type

CS_MAC_ROMAN
CS_MAC_ARABIC
CS_MAC_HEBREW

MS/Windows Code Sets

The MS/Windows code sets are not simply related to the ISO code sets, except the 1252 code which extends the ISO_LATIN1 code set and is limited to the 80-9f range.

Type

CS_MSW_EASTEUR
O
CS_MSW_CYRILLI
C
CS_MSW_ANSI
CS_MSW_GREEK
CS_MSW_TURK
CS_MSW_HEBREW
CS_MSW_ARABIC

PC Code Pages

The PC code pages cover the 80-ff range, and also assign glyphs to the 00-1f range. They coincide on many characters, but we consider them as separate code sets for the whole 0-ff range.

Note: The 1004 code set described in the UNICODE document seems identical to ISO_LATIN1, so it is not listed here.

Type

CS_PC_850
CS_PC_857
CS_PC_863
CS_PC_437
CS_PC_860
CS_PC_861
CS_PC_865
CS_PC_852
CS_PC_869
CS_PC_855
CS_PC_864
CS_PC_M4

CJK Code Sets

JIS_0201, JIS_0208 and JIS_0212 are two-code sets for Japanese characters.

Type	Description
CS_JIS_0201	covers only the half-width katakana
CS_JIS_0208	the primary Japanese code set; contains full width katakana, hiragana, kanji, CJK, punctuation, full-width Latin, Greek, Cyrillic letters, symbols
CS_JIS_0212	not very widespread. We will not distinguish the different variants of these JIS standards (i.e. 1978, 1990)
CS_KSC_5601	the standard encoding for Hangul (Korean)
CS_GB_2312	the standard encoding for Mainland China
CS_BIG5	the standard encoding for Taiwan

UNICODE

Some portions of UNICODE map more or less directly to existing code sets, so we could unify specific portions of UNICODE with standard code sets.

Type

CS_UNICODE

The problem with this approach is that UNICODE pages have holes because some characters are unified and so do not quite map to standard code sets.

Unifying between UNICODE and old style code sets would introduce quite some complexities, so we will avoid it except in the ASCII and ISO_LATIN1 cases.

Also, UNICODE is special in many respects (diacritical marks, directionality), so it is better to consider it as a separate code set overall than to try to unify parts of it with other standards.

The ASCII and ISO_LATIN1 portions of the UNICODE code set will be unified with ASCII, ISO_LATIN1 and EMPTY_809f (first page of UNICODE). The rest of UNICODE will be treated as a separate code set.

EBCDIC

The EBCDIC code sets only contain EBCDIC characters which do not map to pure ASCII characters. In the CsChar representation, EBCDIC characters which map to ASCII are unified with ASCII and coded as CS_ASCII.

Type

CS_EBCDIC

For now, we have a generic EBCDIC brand, but we may distinguish several later, when we get more precise documentation (UNICODE documentation describes 037, 500V1, 1026 and 875 variants of the EBCDIC standard).

HP**Type**

CS_HP_ROMAN8

CNS**Type**CS_CNS11643_
1CS_CNS11643_
2CS_CNS11643_
3

Creating and Destroying

Constructors

inline NDCs::NDCs(void);

Default code set object constructor.

inline NDCs::NDCs(CsIdEnum csid);

Constructs the code set object from the `csid` information.

Destructor

NDCs::~NDCs(void);

Default code set object destructor.

Convenience Macros

The following code set functions can be called for any code set. All the following operations are implemented as macros which use member functions defined for this class.

GetCsId

CsIdEnum NDCs::GetCsId(void);

Get the code set's id.

GetCharLen

StrIVal NDCs::GetCharLen(void);

Get the character length for the code set.

GetCharInfo

CharInfoVal NDCs::GetCharInfo(CsCode code);

Get the 'charinfo' value of the character `code`.

CvtChar

Converts a character within the code set.

BoolEnum NDCs::CvtChar(CsCode in, CharCvtSet flags, LgEnvCPtr lgenv, CsCodePtr out);

Convert the character in `in` described in `flags` and set the result to `out`. `lgenv` specifies a language environment. `flags` specifies the ways of translation. If bool is true, it indicates the conversion is reversible; otherwise, not reversible.

The 'flags' could be:

Type	Description
CHAR_CVT_DOWNCASE	
CHAR_CVT_UPCASE	
CHAR_CVT_STRIPDIACR	
CHAR_CVT_SPLITDIGRAPHS	
CHAR_CVT_HIRAGANA	(CS_JIS_0208 only)
CHAR_CVT_KATAKANA	(CS_JIS_0208 only)
CHAR_CVT_PRECOMPOSE	(CS_UNICODE only)
CHAR_CVT_DECOMPOSE	(CS_UNICODE only)
CHAR_CVT_NOCOMPAT	(CS_UNICODE only)

If the 'flags' is NULL, it translates as much as possible. UNICODE conversion can be done by specifying UNICODE to 'cs' code set.

TransChar

Converts a character between two code sets.

BoolEnum NDCs::TransChar(CsCode code, CharCvtSet flags, CsCPtr incs, CsCodePtr chcodeptr);

Translates the character of specified code set to the character within this code set. 'flags' specifies the ways of translation. If bool is true, it indicates the conversion is reversible; otherwise, not reversible.

The 'flags' could be:

Type	Description
CHAR_CVT_STRIPDIACR	(in' code set = CS_ISO_LATIN1, CS_ADOBE_STD... etc.)
CHAR_CVT_SPLITDIGRAPHS	(out' code set = ASCII)
CHAR_CVT_FULLWIDTH	(in code set = CS_ASCII, CS_ISO_GREEK, CS_ISO_CYRILLIC ...) (out code set = CS_JIS_0208, maybe CS_KSC_5601 ...)
CHAR_CVT_HALFWIDTH	(in code set = CS_JIS_0208, maybe CS_KSC_5601 ...) (out code set = CS_ASCII, CS_ISO_GREEK, CS_ISO_CYRILLIC ...)

If the 'flags' is NULL, it translates as much as possible. UNICODE conversion can be done by specifying UNICODE to 'cs' code set.

If the character cannot be translated, 'out' is set to NULL.

ToUni

BoolEnum NDCs::ToUni(CsCode cscode, UniCodePtr uni);

Converts cscode to unicode. If it cannot be converted, returns false; otherwise, set the unicode to uni and return true.

FromUni

BoolEnum NDCs::FromUni(CsCode cscode, CsCodePtr uni);

Converts unicode to cscode. If it cannot be converted, returns false otherwise set the cscode value to cscode and return true.

Predefined Code Sets**GetCsNative**

static CsPtr NDCs::GetCsNative(void);

Returns a pointer to the native code set.

GetCsUnicode

static CsPtr NDCs::GetCsUnicode(void);

Returns a pointer to the Unicode code set.

GetCsGlobal

static CsPtr NDCs::GetCsGlobal(void);

Returns a pointer to the global code set.

Local Macros

Type	Definition
CHARINFO_UNKNOWN	CHAR_DOM_UNKNOWN CHAR_LEVEL_BASIC CHAR_LEX_UNKNOWN CHAR_CASE_NONE
CHARINFO_UNKNOWN_FULLWIDTH	CHARINFO_UNKNOWN CHAR_WIDTH_FULL
CHARINFO_UNKNOWN_HALFWIDTH	CHARINFO_UNKNOWN CHAR_WIDTH_FULL

ISO LATIN1 Character Information Definition

Type	Definition
IS1_COM	CHAR_DOM_LATIN CHAR_LEVEL_BASIC CHAR_WIDTH_HALF

ASCII Character Information Definition

Type	Definition
ASCII_COM	CHAR_DOM_GENERIC CHAR_LEVEL_BASIC CHAR_WIDTH_HALF CHAR_LOOSE_ASCII_MASK

JIS0208 Character Information Definition

Type	Definition
JIS0208_COM	CHAR_WIDTH_FULL
JIS0208_1KU_COM	JIS0208_COM CHAR_DOM_MISC CHAR_LEVEL_EXTENDED CHAR_CASE_NONE
JIS0208_2KU_COM	JIS0208_COM CHAR_DOM_MISC CHAR_LEVEL_EXTENDED CHAR_CASE_NONE
JIS0208_3KU_COM	JIS0208_COM CHAR_DOM_GENERIC CHAR_LEVEL_BASIC CHAR_LOOSE_ASCII_MASK
JIS0208_4KU_COM	JIS0208_COM CHAR_DOM_HIRAGANA CHAR_LEVEL_BASIC CHAR_CASE_NONE CHAR_LEX_BASE_LETTER
JIS0208_5KU_COM	JIS0208_COM CHAR_DOM_KATAKANA CHAR_LEVEL_BASIC CHAR_CASE_NONE CHAR_LEX_BASE_LETTER
JIS0208_6KU_COM	JIS0208_COM CHAR_DOM_GREEK CHAR_LEVEL_BASIC CHAR_LEX_BASE_LETTER
JIS0208_7KU_COM	JIS0208_COM CHAR_DOM_CYRILLIC CHAR_LEVEL_BASIC CHAR_LEX_BASE_LETTER
JIS0208_8KU_COM	JIS0208_COM CHAR_DOM_MISC CHAR_LEVEL_EXTENDED CHAR_CASE_NONE CHAR_LEX_SPECIAL
JIS0208_16TO84KU_COM	JIS0208_COM CHAR_DOM_KANJI CHAR_LEVEL_BASIC CHAR_CASE_NONE CHAR_LEX_BASE_LETTER

JIS0201 Character Information Definition

Type	Definition
JIS0201_COM	CHAR_WIDTH_FULL CHAR_DOM_KATAKANA CHAR_LEVEL_BASIC CHAR_CASE_NONE CHAR_WIDTH_HALF
JIS0201_CODE_MASK	
JIS0201_DIACR_DOUBLEDOTS_MASK	
JIS0201_DIACR_CIRCLE_MASK	
JIS0201_LOSTINFO_MASK	

17 *Ct Class*

his Ct class implements the Open Interface code type structures and utilities.

Technical Summary

The functions in this class offers support for English, European, and Asian languages by providing support for many different character code types.

Multibyte characters require the use of code sets, code mappings, and code types. These represent the characters in an alphabet as numeric codes and determine how these codes are placed within a multibyte character structure.

Code Sets

A code set (or character set) represents each character in an alphabet by a numeric code. The numeric codes in each code set vary in their hexadecimal range.

Most code sets are extensions to the ASCII character set. EBCDIC is an exception. Code sets are combined with mappings to form a code type.

The current version of Open Interface supports the ASCII, ISO_LATIN1, JIS_0201, and JIS_0208 code sets.

Code Mapping

A code mapping determines the representation of the encoded character within a multibyte character, which is an unsigned 32-bit integer. A mapping includes the placement of bytes within the character and any manipulation that might be needed for each byte.

Sometimes mapping is more complex than simple byte placement. The JIS code set defines codes where the first and second bytes are in the 0x21 - 0x7e range. JIS bytes cannot be inserted into a string regardless of the byte order because the JIS code would be indistinguishable from the ASCII codes.

Several mappings address this problem:

- The JEUC mapping transposes a JIS code in the 0xa1-0xfe range by adding 0x80 to each byte.
- The SJIS mapping is more complex and includes a transposition of JIS code in the (0x80-0x9f, 0x40-0xfe) or (0xe0-0xff, 0x40-0xfe) ranges.

Code Types

A code type (or "coding scheme") combines one or more code sets with a code mapping.

For single-byte ASCII or extended ASCII characters, the byte value maps directly to the code value. For these alphabets, the code set and the code type are identical.

For multibyte characters, different code types can be based on the same code set but on different code mappings. For example, the Japanese EUC code type offered by Sun and the SJIS code type offered by Sony are two different mappings of the JIS code set. The UNICODE code type consists of mappings of existing standard code sets.

Open Interface provides two levels of support for code sets, tested and untested. The CT_ASCII, CT_SJIS, and CT_JEUC code types are fully supported and tested. Also, a wide range of other standard code types are implemented but not tested.

Code types supported and tested under the current version of Open Interface include the following:

- ASCII Code Type. The CT_ASCII code type contains the CS_ASCII code set.
- CJK Code Types. In the CJK code type group, Open Interface offers the CT_SJIS and CT_JEUC. The CT_SJIS code type is a combination of CS_ASCII and the CS_JIS_0201 and CS_JIS_0208 code sets. CT_JEUC combines CS_ASCII with CS_JIS_0201, CS_JIS_0208, and CS_JIS_0212. The remainder of the CJK code types are supported but are not fully tested in the current version.

See also

Char class.

Data Types

ChCode

Data type for a code value within a code type.

Ct

Defines the code type data record.

Private data elements in the record are the code type id, the code set pointer, the maximum character length for the code type, and the pointer to a code set.

See also

NDct::GetCharLen, NDct::GetFwr, NDct::GetBwr, NDct::GetInfo, NDct::CvtChar, NDct::CvtCtToCs, NDct::CvtCsToCt

Enumerated Types

CtIdEnum

Data type for a code type id.

CT_ID

Code type ids identify a complete character coding system.

Code type ids identify a complete character coding system. They are built from an association of code sets and a code mapping. Code type categories include: ASCII, ISO 8859-X, Adobe, Macintosh, Microsoft Windows, PC, CJK, UNICODE, EBCDIC, Global, and HP code types.

ASCII Code Type

Code Type Id	Description
CT_ASCII	CT_ASCII characterizes pure ASCII: single byte strings and fonts which only provide glyphs for the ASCII range.

ISO 8859-X Code Types.

Code Type Id	Description
CT_ISO_LATIN1	The ISO-8859 code types characterize single byte strings and fonts which combine the ASCII, the EMPTY_809f and the ISO_8859_X code sets.
CT_ISO_LATIN2	
CT_ISO_LATIN3	
CT_ISO_LATIN4	
CT_ISO_CYRILLIC	
CT_ISO_ARABIC	
CT_ISO_GREEK	
CT_ISO_HEBREW	
CT_ISO_LATIN9	

ADOBE Code Types

Code Type Id	Description
CT_ADOBE_STD	STD is ASCII + EMPTY_809f + ADOBE_STD
CT_ADOBE_LATIN1	LATIN1 is ASCII + ADOBE_LATIN1 + ISO_LATIN1
CT_ADOBE_SYMBOL	SYMBOL is ASCII (00-1f only) + EMPTY_809f + ADOBE_SYMBOL.
CT_ADOBE_ZAPFDB	ZAPFDB is ADOBE_ZAPFDB only.

Macintosh Code Types

Code Type Id	Description
CT_MAC_ROMAN	ROMAN is ASCII + MAC_ROMAN
CT_MAC_ARABIC	ARABIC is ASCII + ISO_ARABIC + MAC_ARABIC
CT_MAC_HEBREW	HEBREW is ASCII + ISO_HEBREW + MAC_HEBREW

Microsoft Windows Code Types

Code Type Id	Description
CT_MSW_EASTEURO	1252 is ASCII + MSW_ANSI + ISO_LATIN1
CT_MSW_CYRILLIC	125X is ASCII + MSW_125X
CT_MSW_ANSI	
CT_MSW_GREEK	
CT_MSW_TURK	
CT_MSW_HEBREW	
CT_MSW_ARABIC	

PC Code Types

Code Type Id	Description
CT_PC_850	PC_XXX ASCII + PC_XXX
CT_PC_857	
CT_PC_863	
CT_PC_437	
CT_PC_860	
CT_PC_861	
CT_PC_865	
CT_PC_852	
CT_PC_869	
CT_PC_855	
CT_PC_864	
CT_PC_M4	

CJK Code Types

Code Type Id	Description
CT_SJIS	SJIS is ASCII + JIS_0201 (a0-df range) + JIS_0208 (80-9f + e0-ff / 40-ff)
CT_JEUC	EUC is ASCII + JIS_0201 (8e / a1-fe) + JIS_0208 (a1-fe / a1-fe) +
CT_KSC	JIS_0212 (8f / a1-fe / a1-fe) (not implemented)
CT_GB	KSC is ASCII + KSC_5601 (a1-fe / a1-fe)
CT_BIG5	GB is ASCII + GB_2312 (a1-fe / a1-fe)
	BIG5 is ASCII + BIG5 (a1-fe / 40-7e, a1-fe)

UNICODE Code Type

Code Type Id	Description
CT_UNICODE	UNICODE is ASCII + EMPTY_809f + ISO_LATIN1 + UNICODE

EBCDIC Code Type

Code Type Id	Description
CT_EBCDIC	EBCDIC replaces ASCII and ASCII extensions completely.

HP Code Type

Code Type Id	Description
CT_HP_ROMAN8	The Hewlett-Packard Roman code type.

UTF8 Code Type

Also called FSS-UTF or UTF2. Characters can be 1, 2, or 3 byte. 1 byte characters are the same as ASCII.

Code Type Id	Description
CT_UTF8	UNICODE transformation format.

CNS Code Type

This is EUC based encoding which can mix up to 16 code sets. In this version we only support CNS11643-1, CNS11643-2, and CNS11643-3.

Code Type Id	Description
CT_UTF8	Yet another code type for Traditional Chinese.

```
ASCII(CNS11643-0) +
CNS11643-1 (a1-fe / a1-fe) +
CNS11643-2 (8e / a1 / a1-fe / a1-fe )
CNS11643-3 (8e / a2 / a1-fe / a1-fe )
```

In general,

```
CNS11643-X (8e / a0 + X / a1-fe / a1-fe)
where: 1 <= X <= 16
```

As of October 15, 1995, only $1 \leq X \leq 7$ are defined.

Creating and Disposing

Constructor

Constructor for the Ct class.

```
void NDCt::NDCt (void);
```

```
void NDCt::NDCt (CtIdEnum ctid);
```

Constructs the Ct with the specified code type `ctid' type data and member functions. It fills the Ct with specified code type data/member functions, by calling a initialization routine. In each initialization proc it may overwrite default values in the Ct and set its own data and member functions. Each code type should have its initialization proc, which this function will call.

Destructor

Destructor for the Ct class.

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

Member Functions

GetCtId

Gets the code type id.

```
CtIdEnum NDCt::GetCtId(void);
```

NDCt::GetCtId returns the code type id from the code type data record structure.

See also

Ct.

GetFwr

Returns the value of the character found at the beginning of a string.

CharCode **NDct::GetFwr(CStr str, StrIVaIPtr lenptr);**

Returns the value of the character found at the beginning of global string. The lenptr is set to the length of the character.

See also

NDct::GetBwr, Ct.

GetBwr

Returns the character code for the character found in front of a given index in a string.

CharCode **NDct::GetBwr(CStr str, StrIVaI index, StrIVaIPtr lenptr);**

Returns the character code for the character found in front of a given index in a global string. The lenptr is set to the length of the character.

The NDct::GetBwr macro can be called for any code type. For more information, see Ct.

See also

NDct::GetFwr, Ct

GetInfo

Returns the CharInfoVal for a character.

CharInfoVal **NDct::GetInfo(CharCode chcode);**

Returns the CharInfoVal for a character. For more information on CharInfoVal, see the Char class.

See also

CharInfoVal, Ct

CvtChar

Converts a character and sets the result.

BoolEnum **NDct::CvtChar(CharCode chcode, CharCvtSet flag, LgEnvCPtr lgenv, ChCodePtr chcodeptr);**

Converts a character to a character code in a given language environment. Converts the character given by the character code in the language environment specified, applies the flag, and sets the chcodeptr with the result. If the Boolean return value is true, then the conversion is reversible, otherwise it is irreversible. The flag options are:

CHAR_CVT_DOWNCASE

CHAR_CVT_UPCASE

CHAR_CVT_STRIPDIACR

CHAR_CVT_SPLITDIGRAPHS

CHAR_CVT_HIRAGANA (CS_JIS_0208 only)

<code>CHAR_CVT_KATAKANA</code>	(CS_JIS_0208 only)
<code>CHAR_CVT_PRECOMPOSE</code>	(CS_UNICODE only)
<code>CHAR_CVT_DECOMPOSE</code>	(CS_UNICODE only)
<code>CHAR_CVT_NOCOMPAT</code>	(CS_UNICODE only)

If the flag is `NULL`, the macro converts the character as completely as possible. UNICODE conversion can be done by specifying `UNICODE` as the codeset. If the character cannot be converted, `chcodeptr` is set to `NULL`. For more information on flags and language environments, see the `Char` class.

If the Boolean return value is true, it indicates the conversion is reversible, otherwise the conversion is irreversible.

See also

`Ct`, `NDct::CvtCtToCs`, `NDct::CvtCsToCt`

`CvtCtToCs`

Converts a character code from its code type form to its code set form.

CSCode `NDct::CvtCtToCs(ChCode chcode, CsPtr* codesetptr);`

Converts a code type character code to the code set character code and gets a pointer to the code set structure of the character.

See also

`NDct::CvtCsToCt`, `Ct`

`CvtCsToCt`

Converts a character code from its code set form to its code type form.

ChCode `NDct::CvtCsToCt(CsCode cscode, CsCPtr codesetptr);`

Converts a code set character code to the code type character code and sets a pointer to the code set structure of the character.

See also

`NDct::CvtCtToCs`, `Ct`

`ToUni`

Converts `chcode` to unicode.

BoolEnum `NDct::ToUni(ChCode ch, UniCodePtr uni);`

If the `chcode` cannot be converted to UNICODE it returns `FALSE`; otherwise, `TRUE`.

`FromUni`

Converts unicode to `chcode`.

BoolEnum `NDct::FromUni(UniCode uni, ChCodePtr ch);`

If the UNICODE cannot be converted to `chcode` it returns `FALSE`; otherwise, `TRUE`.

GetMaxCharLen

Returns the maximum character length supported by a code type.

StrIVal NDct::GetMaxCharLen(void);

NDct::GetMaxCharLen returns the maximum character length supported by a code type.

IsSingleOnly

Determines whether a code type defines single-byte characters only.

BoolEnum NDct::IsSingleOnly(void);

NDct::IsSingle determines whether a code type supports single-byte characters only. If the BoolEnum return value is TRUE, the code type supports single-byte characters only. If the code type is not single-byte only, the macro returns FALSE.

GetUpper

Returns the upper case form of a character.

ChCode NDct::GetUpper(ChCode *chcode*);

NDct::Get Upper returns the upper case form of a character.

See also

NDct::GetLower

GetLower

Returns the lower case form of a character.

ChCode NDct::GetLower(ChCode *chcode*);

NDct::GetLower returns the lower case form of a character.

See also

NDct::GetUpper

18 *Ds Module*

This module specifies the virtual Data Source.

Design Overview

A data source is an object that can be used as an intermediary between the data itself and the different views on this data. The classes defined in this module are pure virtual. A number of subclasses are described in other modules.

Classes

The DataSource data structure is private. It is a subclass of Res.

Class

static RClasPtr NDDs::Class(void);

Returns a pointer to the DataSource class.

View Interface

RegisterView

void NDDs::RegisterView(ResPtr view);

Register the resource view with the data source.

SetViewOption

void NDDs::SetViewOption(ResPtr view, CStr option, CStr info);

Set info as the option for the view registered in the data source.

GetViewOption

CStr NDDs::GetViewOption(ResCPtr view, CStr option);

CStr (DsCPtr ds, ResCPtr view, CStr option);

Returns the string corresponding to option for the view registered with the data source.

UnregisterView

void NDDs::UnregisterView(ResPtr view);

void (DsPtr ds, ResPtr view);

Unregisters view from the data source.

ViewGetDs

```
static DsPtr NDDs::ViewGetDs(ResPtr view);  
DsPtr DS_ViewGetDs(ResPtr view);
```

Returns the data source, if any, associated to the view.

Edition Interface

DsEditCompletionEnum

Enumerated type describing the success of the edition.

Methods	Description
DSEEDIT_COMPLETIONOK	The edition was successfully completed
DSEEDIT_COMPLETIONFAILED	The edition failed for some reason
DSEEDIT_COMPLETIONPREEMPTED	The edition was preempted

StartEdit

```
DsEditPtr NDDs::StartEdit(void);
```

Opens an edition on the whole data source. The operations are done through the edition object returned by this call.

End

```
DsEditCompletionEnum NDDsEdit::End(void);
```

Commit the edition on the whole of the data source. The edition object is destroyed and deleted.

Abort

```
void NDDsEdit::Abort(void);
```

Abort the edition on the data source. The edition object is destroyed and delete.

AddOperation

```
DsEditOpPtr NDDsEdit::AddOperation(void);
```

Add an operation to the edition

SetOwner

```
void NDDsEdit::SetOwner(ResPtr owner);
```

Set owner of the edition. Results normally not re-propagated by to the owner (useful for asynchronous updates to avoid confusion between current view and current value).

GetOwner**ResPtr NDDsEdit::GetOwner(void);**

Retrieve owner (if any) of the edition.

Default constructors and destructors for the base DsEdit and DsEditOp classes.

Update Interface

StartUpdateEdit**DsUpdateEditPtr NDDs::StartUpdateEdit(void);**

Opens an update on the whole data source. The operations are done through the edit object returned.

End**void NDDsUpdateEdit::End(void);**

Commit the update on the whole of the data source. The edit object is destroyed and deleted.

Abort**void NDDsUpdateEdit::Abort(void);**

Abort the update on the data source. The edit object is destroyed and delete.

Contained/Container Data Source Interface

Data source can be contained in another. For example, a table data source may decide to instantiate a value data source to allow manipulation of the data in a particular cell of the table.

AddContDs**void NDDs::AddContDs(DsPtr contDs);**

Adds contDs as a contained data source to the data source.

RemoveContDs**void NDDs::RemoveContDs(DsPtr contDs);**

Removes contDs from the data source.

Creating and Disposing

Constructor

```
NDDs(CStr modName, CStr NULL, ResRunTimeClassArrayPtr NULL);
NDDs(RClasPtr DsGetClass(), RClasCreateCPtr NULL);
```

Destructor

```
~NDDs(void);
```

Class

Edition Operation

DsEditOpEnum

Methods	Description
DSEEDIT_OPENUMINHERIT	(DSEEDITOP)

DsEditTypeEnum

Methods	Description
DSEEDIT_TYPEENUMINHERIT	(DSEEDITTYPE)

DsEditStateEnum

Methods	Description
DSEEDIT_STATECLOSED	
DSEEDIT_STATEOPEN	
DSEEDIT_STATEPREEMPTED	

Modifications Implementation

DsModsSetEnum

Methods	Description
DS_MODSBITSETINHERIT	(DSMODS)

Data Source

```
extern "C" RClasPtr DsGetClass(void);
extern "C" void DsConstruct(ResPtr res, RClasCPtr rclas,
RClasCreateCPtr rCreate);
extern "C" void DsDestruct(ResPtr res);
```

19 *Err Class*

This class provides support for error handling and error reporting.

Overview

Open Interface uses an exception based error handling mechanism, following the "disciplined exceptions" model described by B Meyer in Object Oriented Software Construction (OOSC, pg 144).

Actually, error handling is only a part of a global programming philosophy. The "disciplined exceptions" model can only be fully understood in the context of the "contracting metaphor" described in detail in OOSC. The reader is encouraged to read this difficult but enlightening book. The fundamental idea of the "contracting metaphor" is that a contract is associated with every routine that you write or that you use.

The contract states what the client of the routine (the caller) should guarantee at the time he calls the routine (preconditions). It also states what the implementer of the routine guarantees the routine will do (postconditions) in case it was called in acceptable conditions (with the preconditions satisfied).

For example, the contract behind the `strlen(char* str)` function is:

Item	Description
Preconditions	Str must be a valid zero terminated C string.
Postconditions	Strlen will return the length of str in bytes.

If you pass an invalid address (i.e. 0) to `strlen`, you violate the preconditions. The key idea is that it is crucial to define precisely who is responsible for what in a program, so that if anything goes wrong one can know who must be blamed and fix the problem. Then, there are no half-successes, half-failures which are so confusing, only successes (the contract has been fulfilled) or failures (the contract could not be fulfilled).

Disciplined Exceptions

In this context, a failure (I prefer using "failure" than "error") is defined by the fact that a routine cannot fulfill its contract, either because the caller did not meet the preconditions or because the routine cannot meet the postconditions (i.e. because it does an I/O operations which fails or because there is a bug).

Failures are not reported through special return values (as is usual in C) but by an out of band mechanism (exceptions). If a failure occurs in a routine, the routine simply does not return, the execution continues elsewhere (in a recovery clause of one of the calling routines, see later). As a result, procedures should be declared with the void return type.

In the "disciplined exceptions" scheme, a routine may handle failures in one of two ways:

- Return to its caller without fulfilling its contract. If necessary, the routine should clean up its state (restore the class invariant in OOSC terms) before returning. The failure is propagated to the caller.
- Try to fulfill its contract by another mean. The error state will be cleared and another path will be tried. If the retry is successful, the caller won't notice that the routine had failed in the first place.

Error Handling And Reporting

The error handling class provides mechanisms to:

- Set up a recovery environment where the execution will resume in case of failure.
- Clear the error state and attempt a retry in a routine.
- Generate (signal) a failure.
- Report warning and failure messages to the user.

The error handling mechanisms (recovery, retry, signalling) are based on the "disciplined exception" model, as described above. The error reporting scheme is not described in OOSC. Reporting errors is a complex issue because failures are usually detected in low level routines which do not know where to report the error (is it a windows based, terminal based, batch application). Also, reporting only the low-level failure is usually insufficient. The user also wants to know the high level context in which the error occurred (it is not very interesting to know that an assertion of the memory manager failed if we do not know in which context the memory manager was being used).

This implies several things:

- We need a mechanism to keep context information in intermediate procedures.
- Error reporting must be initiated by the low level.
- The "user-interface aware" high level needs a way to set-up the procedure which will display the error.
- Reporting procedures may be nested. The most specific reporting procedure (closest to the current procedure in the stack frame) will be the one which reports the error or warning.

The idea is that the reporting procedure will be called from the low level. At this time, all the context information is available. Our scheme also provides "warnings" in addition to "failures". When a failure occurs, the failure is reported to the user and then the execution continues in the recovery clauses of the routines which are on the stack until one routine attempts a retry.

On the other hand, warnings are reported to the user but then the execution continues normally in the routine. With those mechanisms, we should also be able to design "smart" warning procedures which gives several options to the user:

- Abort operation (an exception will be generated), continue or continue and discard subsequent warnings (in case the same warning keeps being repeated).

Entry/Exit Macros

Every procedure or function which uses error handling mechanisms should start with an `ERR_XIN` macro (immediately after the declaration of the automatic variables) and end with an `ERR_XOUT` (procedures) or `ERR_XRET` (functions) macro.

```
void  MODUL_Proc (Type1 arg1)
Type2  autovar1;
ERR_XIN;
...
ERR_XOUT;
```

Where X is one of the following : TRACE, MSG(id), RECOV, RETRY.

Note: All the paths exiting from the routine must end with an `ERR_XOUT` or `ERR_XRET` statement. You are not allowed to use a "return" in such a routine, you should use `ERR_XOUT` or `ERR_XRET` instead. Forgetting an `ERR_XOUT` or `ERR_XRET` clause on one of the paths will confuse the error handling mechanism and generate a fatal error if the `DBG_ON` compilation flag is set.

If you use these macros in a file, you must define a static `char*` variable called `S_ModuleName` and initialize it to the name of the module to which the source file belongs. This module name is used by the error reporting procedure to generate traceback information or to load error messages.

You can use the `ERR_INMODULE` macro to define this variable. After the `#include` directives at the top of the file, you should add the following statement:

```
ERR_INMODULE("Modul ")
```

where "Modul" is the name of the module. Which translates into:

```
static const char S_ModuleName[] = "Modul";
```

Error Recovery

Every routine may have an error recovery label where execution will resume in case there is a failure (in the routine or in a subroutine it called). If you do not provide an error recovery label, the execution will resume in the error recovery clause of one of the callers of your routine. All the information which was on the stack of your routine at the time of the failure will be lost.

Usually, you should use the recovery clause to put your program back in a stable state. For example, you will release resources which had been allocated by the routine or reset a global variable which had been temporarily modified by your routine.

Retry

In some cases, your routine can retry to fulfill its contract by another way. In such cases, you should use the `ERR_RETRYIN/OUT/RET` macros:

```
void MODUL_Proc (void)
Int  attempts= 0;
ERR_RETRYIN;
MODUL_ReadFromConnection();// might fail
ERR_RETRYOUT;// success, execution will continue normally in
//caller.
err_catch:
```

```
if (++attempts <= 5) ERR_RETRY;// will branch to ERR_RETRYIN
ERR_RETRYOUT;// failure will be propagated to caller.
```

Another use of the `RETRY` mechanism is to convert a routine which signals its errors through the exception mechanism into a routine which returns a status code (if you do not like our exception based error handling, you can write a little wrapper around every API call in the following way):

```
BoolEnum  MODUL_ProcWithRetStatus (void)
BoolEnum  success = BOOL_TRUE;
ERR_RETRY;
if (success) MODUL_ProcWithExceptions();
ERR_RETRYRET(success);
err_catch:
success = BOOL_FALSE;
ERR_RETRY;
```

Note: This routine never fails!

Important: You should be careful and not put a function call in the argument of `ERR_CATCHRET` and `ERR_RETRYRET`. The function would be called after the error recovery environment has been unlinked (see implementation of `ERR_RET` below). The following code is incorrect because the error recovery will not resume at the `err_catch` label of the current routine but in one of its callers.

```
ERR_CATCHIN;
ERR_CATCHRET(MyFunc(myarg));
```

The following code is correct:

```
MyType  result;
ERR_CATCHIN;
result = MyFunc(myarg);
ERR_CATCHRET(result);
```

Signalling A Failure

Above, we have described how to recover from failures. Now, how do we signal a failure, for example if we notice some abnormal condition or if some system call fails?

Three calls are provided to signal failures:

- `ERR_Fail`
- `ERR_FailStr` and
- `ERR_FailSilent`

Item	Description
<code>ERR_Fail</code>	Signals a failure. The error message will be loaded from the "list of strings" resource called <code>Modul.Errors</code> where "Modul" is the first argument passed to <code>ERR_Fail</code> (usually you pass <code>S_ModuleName</code> , the name of the current module). The second argument is the index of the message in the list of strings" resource (from 0 to n-1 where n is the number of messages in the resource. The message may contain some printf formatting directives (<code>%d</code> , <code>%lx</code> , ...) in which case you pass additional arguments.
<code>ERR_FailStr</code>	Signals a failure. Instead of loading the error message from a resource, the error message is hard-coded and passed as first argument. For now, printf like formatting is not supported by that routine.

`ERR_FailSilent` Signals a failure silently (by disabling the error reporting mechanism).

A typical use of `ERR_Fail` will be the following:

```
#define MODUL_FAILFILEOPEN2
FILE* MODUL_FileOpen(char* name)
FILE* file;
file = fopen(name, "r");
if (file == NULL) ERR_Fail("Modul", MODUL_FAILFILEOPEN, name);
return file;
```

The `modul.rc` resource file will contain the following resource definition:

StrL.Compile

Name	Modul.Errors
Text:	"error #0".
Text:	"error #1".
Text:	"cannot open file %s".
Text:	"error #3".

WARNINGS

If you want to generate a warning, you can use one of the following calls:

Item	Description
<code>ERR_Warn</code>	Generates a warning. The warning message will be loaded from the "list of strings" resource called <code>Modul.Warnings</code> where "Modul" is the first argument passed to <code>ERR_Warn</code> (usually you pass <code>S_ModuleName</code> , the name of the current module). The second argument is the index of the message in the list of strings" resource (from 0 to n-1 where n is the number of messages in the resource. The message may contain some <code>printf</code> formatting directives (<code>%d</code> , <code>%lx</code> , ...) in which case you pass additional arguments.
<code>ERR_WarnStr</code>	Generates a warning. Instead of loading the message from a resource, the message is hard-coded and passed as first argument. For now, <code>printf</code> like formatting is not supported by that routine.

The major difference between warnings and failures is that execution will continue normally after a warning instead of resuming in recovery code as is the case with failures.

Fatal Errors

A fatal error is an error which will cause the program to terminate without attempting any recovery. Usually, you should not use fatal errors but signal failures instead to give a chance to continue. Fatal errors will be generated internally by the error handler in case we are completely lost (error recovery data corrupted, failure while recovering from a failure, ...).

But if you really want to terminate the program, you can use `ERR_Fatal` which will terminate the program with a message and dump a core file on

UNIX or call `ERR_Exit` which will display a message and terminate the program. The message is hardcoded in this case because we do not want to risk failing while loading the error message.

Error Contexts

As mentioned previously, we want our error reporting to include high level context information as well as a low level message describing the failure detected at the low level.

The way to achieve this is to set up some "error context messages" in high level procedures. If a failure is detected in a lower level routine, the procedure which reports the failure can display the error message and can also scan the "error contexts" which are active at that time and display the "error context messages".

Usually, the error context messages begin with a present participle (an ing form), for example: "loading ...", "opening ...", "compiling ..." whereas the error message usually starts with "unexpected ..." or "cannot ...". Error contexts should indicate the state the program is in, not a particular error or abnormal condition. Then the whole error message (with contexts) will be something like:

Item	Description
ERROR	Unexpected end of file.
While	Reading file header loading file "myapp.dat" initializing application.

The first message (unexpected end of file) is specified in the `ERR_Fail` call which signalled the failure (probably in a low level call like `FILE_Read`).

The other messages have been set up at a higher level, for example in calls like `RLibReadHeader`, `RLIB_LoadFile`, `MYAPP_Init`.

To set up an error context, you use the `ERR_MSGIN/OUT/RET` macros:

```
define RLIB_MSGREADHEADER1
define RLIB_MSGLOADFILE3
void      RLibReadHeader(RLibPtr rlib)
ERR_MSGIN(RLIB_MSGREADHEADER);
RLibReadHeader code
FILE_Read(...);
```

more `RLibReadHeader` code.

```
ERR_MSGOUT;
void RLIB_LoadFile(char* filename)
RLibPtr  rlib;
ERR_MSGIN(RLIB_MSGLOADFILE);
ERR_SETOPTSTR(filename);
RLIB_LoadFile code
RLibReadHeader(rlib);
more RLIB_LoadFile code
ERR_MSGOUT;
```

The message numbers (`RLIB_MSGXXX` codes) are indices in a resource called "RLib.Messages". The `rlib.rc` resource file will contain the following resource:

(StrL.Compile

Name	RLib.Messages
Text :	"message #0".
Text :	"reading file header".
Text :	"message #2".
Text :	"loading file \"%s\"".

In the first case (RLibReadHeader), the message does not contain any formatting directive. In the second case, the ERR_SETOPTSTR macro sets up 'filename' as the parameter of the formatting directive contained in the message.

Three macros are currently provided for formatting context messages:

Item	Description
ERR_SETOPTSTR	Sets up a string parameter.
ERR_SETOPTVSTR	Sets up a variable string (see vstrpub.h) parameter.
ERR_SETOPTINT	Sets up an integer parameter.

It is possible to implement other formatting directives by using the FormatProc field of the error frame. This will be documented later in the "advanced error reporting" section.

Note: Context messages are formatted at the time failures are reported, not at the time the routine is entered, so there is very little overhead in setting up context messages.

Error Tracing

In debugging versions of your program, it may be interesting to get a complete traceback of the execution stack in addition to the context messages. Context messages are for the end user of your application, the complete traceback will give the name of the source files and the line numbers and is for the developer of the application.

If you use the ERR_TRACEIN/OUT/RET macros in all your routines, you will get a complete traceback in case of failure. These macros are controlled by the ERR_TRACEALL compilation flag. If ERR_TRACEALL is set, the macros will turn into effective code and the traceback mechanism will be fully operational. If this flag is not set, ERR_TRACEIN is a NOP and ERR_TRACEOUT/RET become simple return statements. Then you get optimal performance but you lose the traceback information.

We recommend that you turn on the ERR_TRACEALL compilation flag when producing debugging and prerelease versions of your software. Once you are confident in the stability of your software, you can turn that flag off to get optimal performance.

Global Variables And Initialization

As a general rule, Open Interface does not define any global variables. The error handling is the exception because we need a global variable for performance reasons. This raises problems when software has to be

packaged in DLLs on MS/Windows and OS/2 PM) because global variables cannot be exported by DLLs.

So, instead of having one global variable for the whole application, we use one global variable per linking unit. A linking unit is either a library DLL, shareable image, shared library, object library) or the set of object files which are linked with the `main` routine but are not part of any library (on systems which support DLLs or shared libraries, a linking unit is a set of files which get linked together).

The error handling uses the `ERR_LIB` global variable. As each linking unit must define its own global variable and we do not want to have multiply defined symbols, `ERR_LIB` must be redefined in every C file so that all the C files belonging to the same linking unit define `ERR_LIB` the same way (and not the same way as C files belonging to other linking units).

This `ERR_LIB` redefinition must be done before including any Open Interface header file.

So every C file using Open Interface headers should start as follows:

■ Files in linking unit MYLIB.

```
define ERR_LIB MYLIB
include <wgtpub.h>
// for example
```

■ Files in linking unit MYAPP (`main` linking unit).

```
define ERR_LIB MYAPP
include <errpub.h>
// for example
```

Note: If you forget to redefine `ERR_LIB`, `errpub.h` won't compile properly.

This error handling global variable must be declared and initialized once in every linking unit. Usually, every library should have one initialization entry point where you will initialize `ERR_LIB`. You will also need to initialize `ERR_LIB` in your `main` routine.

The following macros are provided to declare and initialize `ERR_LIB`:

Item	Description
<code>ERR_DECLARE</code>	Declares <code>ERR_LIB</code> .
<code>ERR_LIBCREATEINIT</code>	Allocates, and initializes <code>ERR_LIB</code> for a library.
<code>ERR_ISLIBCREATED</code>	Is <code>BOOL_FALSE</code> if <code>ERR_LIB</code> has not been allocated for a library.
<code>ERR_MAININIT</code>	Initializes <code>ERR_LIB</code> for the `main` routine.

and also:

Item	Description
<code>ERR_EXTERN</code>	Special macro used by the Macintosh version to allow precompiled headers.

`ERR_LIBINIT` is a macro that checks whether `ERR_LIB` for a library has been allocated, and, if it has not, creates and initializes it by using `ERR_LIBCREATEINIT` and if it has, returns (using a `return` statement. It is always very dangerous to hide a `return` statement in a macro which does not

make it obvious. A better way would be to actually use the `ERR_ISLIBCREATED` and `ERR_LIBCREATEINIT` macros.

A library initialization routine will look like:

```
ERR_DECLARE
void MYLIB_Init()
if (!ERR_ISLIBCREATED) {
ERR_LIBCREATEINIT;
```

Your `'main'` routine will look like:

```
ERR_DECLARE
int main(int argc, char**argv)
ERR_MAININIT;
```

Note: If you are not using any error handling macro in your linking unit, you do not need to use `ERR_DECLARE` and `ERR_LIBCREATEINIT/ERR_MAININIT`

Advanced Error Reporting

This section will cover two topics:

- Formatting context messages (beyond what the `ERR_SETOPTXXX` macros provide).
- Installing a custom error reporting procedure.

These topics will be documented later.

The only documented routines for now are:

- `ERR_TraceBack()` Outputs error traceback starting from the top error frame for the current exception.
- `ERR_FrameTraceBack(ErrFramePtr frame)` Outputs error traceback starting from the error frame specified by `frame`'.

Summary Of Error Handling And Reporting

To set up an error recovery label in a routine, use `ERR_CATCHIN/OUT/RET`. If the routine attempts a retry, you should use `ERR_RETRYIN/OUT/RET` instead.

The `ERR_RETRY` macro will clear the error state and branch at the beginning of the routine.

- To signal a failure, use `ERR_Fail`, `ERR_FailStr` or `ERR_FailSilent`.
- To generate a warning, use `ERR_Warn`, `ERR_WarnStr`.
- To set up context messages, use `ERR_MSGIN/OUT/RET`. You can parameterize context messages with the `ERR_SETOPTSTR/VSTR/INT` macros.
- To set up traceback information, use `ERR_TRACEIN/OUT/RET`.

The error reporting messages are loaded from "list of string" resources see the `StrL` module). Every module may define three "list of string" resources:

Item	Description
<code>Modul.Errors</code>	Messages associated with failures.
<code>Modul.Warnings</code>	Messages associated with warnings.
<code>Modul.Messages</code>	Context messages.

Where "Modul" is the name of the module.

Note: If you have an error recovery label in a function (not a procedure), the value returned by `ERR_CATCHRET(val)` after the `err_catch` label will be meaningless because the execution will not continue normally in the caller) at that point. You should nevertheless use `ERR_CATCHRET` rather than `ERR_CATCHOUT` to keep lint (and some smart C compilers) happy.

The error recovery mechanism uses the `setjmp/longjmp` calls, so you should be careful about using 'register' variables. Any variable which may be modified in the body of the routine (before the `err_catch` label) should be declared with the `C_VOLATILE` keyword so that the compiler does not assign it to a register.

The (non documented) `ERR_Signal` procedure is called by `ERR_Fail`, `ERR_FailStr`, `ERR_Warn` and `ERR_WarnStr`. When you are running your application from a debugger, you are encouraged to set a breakpoint on this procedure (`ERR_Signal`), so that you can examine the contents of the stack and of your variables when a failure occurs. You should also set a breakpoint on `ERR_Fatal` to trap fatal errors.

Reporting Errors for Calls to Third Party APIs

Certain of the modules in Open Interface provide a portable interface to third party APIs, such as calls to the underlying operating system (eg. the `FILE` and `FMGR` modules.) The contracting metaphor used by Open Interface's error mechanism is not always appropriate for these interfaces, because the underlying operating system call or third party product may not have been designed with the contracting metaphor in mind. There are two issues which are important here:

1. The contracting metaphor is inherently boolean in that the called routine either fulfils its contract or it does not and fails. Certain third party API calls fit this boolean model (eg. memory allocation), whereas others do not (eg. file access) because they may return an error status for a number of different reasons which may be of interest to the upper levels of the software (for the forming of dialog boxes, determining a retry mechanism, etc.)
2. Assertions should be used to indicate controllable programming errors that reflect the underlying contract, eg. a pointer must not be `NULL`, a positive extent must be supplied for a buffer, etc. However if one of the input parameters to a routine is derived from input typed by a user, for instance from the contents of a text edit in a dialog box, then that input is not under the control of the program. In such instances it might be more appropriate to return that information to the caller instead of asserting because the system call failed.

For this reason for those API calls which interface to the operating system or other third party product, and for which the contracting metaphor may not be appropriate there are usually two versions of the call provided, one of which asserts and another which does not but returns a boolean value indicating whether or not the call succeeded. The calls usually take the same arguments, with the asserting version being declared as `void` and the non-asserting version being declared as `BoolEnum`, and the non-asserting version takes the name of the asserting version and appends "Try" to the beginning (eg. `FILE_Open` and `FILE_TryOpen` in the `FILE` module.) An

error reporting structure is maintained by the ERR module which can be used to store information about the call which failed. It is the responsibility of the module which made the call to the routine which failed to write to this structure, and the it is the responsibility of the caller of the module to check this structure if an error has occurred.

The error reporting structure contains the following three fields:

Item	Description
Code	A enumeration of the error code, which is defined by the module which writes to the structure ie. there is one set of constants for module A, antother set for module B, etc.).
SysCode	The system specific error code.
FuncName	The name of the API call which returned an error status.

An API call is provided, `ERR_GetErrFuncCallPtr`, to obtain a pointer to this structure.

Data Structures

NDErrFuncCall

Item	Description
ErrCodeVal	Code;
ErrSysVal	SysCode;
CStr	FuncName;

GetErrFuncCallPtr

ErrFuncCallPtr NDErr::GetErrFuncCallPtr (void)

Returns a pointer to the error reporting structure.

ErrFrame API for Error Reporting and Discrimination

FrameGetTop

ErrFramePtr NDErr::FrameGetTop (void)

Returns topmost error frame.
This call is only valid inside catch blocks. The frame returned by this call may be used to discriminate among exception types, or to report the exception in a custom fashion.

FrameQueryMessage

void NDErr::FrameQueryMessage (ErrFrameCPtr *frame*, Str *buf*, StrIVal *size*)

Formats the frame's end user message into ``buf'`, not writing more than `size'` characters.

FrameQueryTraceback**void NDErr::FrameQueryTraceback (ErrFrameCPtr *frame*, Str *buf*, StrIVal *size*)**Formats the frame's traceback message into `buf`, not writing more than *size* characters.**FrameQueryFullTraceback****void NDErr::FrameQueryFullTraceback (ErrFrameCPtr *frame*, VStrPtr *vstr*)**

Formats the full traceback message into `vstr`.

FrameSetReported**void NDErr::FrameSetReported (void)**

Marks the error frames to indicate that the error has been reported to the user.

FrameIsReported**BoolEnum NDErr::FrameIsReported (void)**

Returns whether or not the error has already been reported to the user.

FrameReport**void NDErr::FrameReport (ErrFrameCPtr *frame*)**

Invokes the global error reporting routine.

FrameDefReport**void NDErr::FrameDefReport (ErrFrameCPtr *frame*)**

Invokes the default error reporting routine.

GetReportProc**ErrReportProc NDErr::GetReportProc (void)**

Returns the global error reporting procedure.

SetReportProc**void NDErr::SetReportProc (ErrReportProc *proc*)**

Overrides the global error reporting procedure.

ErrFrame Class**ErrFrame**

Private stuff.

Item	Description
ErrFramePtr	Next
ErrJumpPtr	Env
BoolEnum	Failed; for propagate
Traceback	CStr File
ErrLineVal	Line

Context	CStr Module
ErrIdVal	Id
ErrTypeVal	Type

Handlers and Client Data.

Item	Description
ErrFormatProc	FormatProc;
ErrReportProc	ReportProc;
ClientCPtr	ClientData;
ErrGblPtr	ErrLib;

Macros

Recovery and retry.

Item	Description
ERR_CATCHIN	Entry of routine with error cleanup code.
ERR_CATCHOUT	Exit of procedure with error cleanup code.
ERR_CATCHRET(val)	exit of function with error cleanup code.
ERR_RETRYIN	Entry of routine with error retry code.
ERR_RETRYOUT	Exit of procedure with error retry code.
ERR_RETRYRET(val)	Exit of function with error retry code.
ERR_RECOVER	Recover directive. (will be documented later).
ERR_RETRY	Retry directive.
ERR_RECOVER_SILENT	Recover with no error reporting.
ERR_RETRY_SILENT	Retry with no error reporting.

Context Messages and Tracing

The context messages and tracing mechanisms can be turned off or on by using the `ERR_TRACEALL` compilation flag. If `ERR_TRACEALL` is defined, then the following macros will generate messages and tracing information. If not, they won't do anything.

Item	Description
ERR_IN(e00); e00.Id = id	Entry of routine with error context message
ERR_OUT(e00)	Exit of procedure with error context message
ERR_RET(e00, val)	Exit of function with error co

Misc Macros For Error Reporting

Item	Description
<code>ERR_SETOPTINT(val)</code>	Sets up an optional integer argument for error reporting.
<code>ERR_SETOPTSTR(str)</code>	Sets up an optional string argument for error reporting.
<code>ERR_SETOPTVSTR(vstr)</code>	Sets up an optional variable string argument for error reporting.

SetReportPrint
SetReportSilent
Print

void NDErr::SetReportPrint (ErrFramePtr *errframe*)

void NDErr::SetReportSilent (ErrFramePtr *errframe*)

void NDErr::Print (ErrFrameCPtr *errframe*)

Default REPORT procedures.

Do not install 'Print' with `ERR_SETREPORT`, but use "SetReportPrint".

'Print' must not be directly installed but can be called from a normal report procedure.

Format

void NDErr::Format (ErrFrameCPtr *errframe*, Str *str*, StrIVal *len*)

Default FORMAT procedure (used by `ERR_Print`).

'Format' loads a message with `ERR_MsgLoad` and then calls the user-defined format procedure (if any).

LoadMsg

void NDErr::LoadMsg (ErrFrameCPtr *errframe*, Str *str*, StrIVal *len*)

Loading the error message from resource file.

ERR_LIB, ERR_EXTERN

ERR_LIB

Global variable for error handling. All the files which belong to the same linking unit should define `ERR_LIB` the same way. `ERR_LIB` must be defined BEFORE including any Open Interface header file.

MAC_HEADERS

Special macro to declare `ERR_LIB`, used in the case `MAC_HEADERS` is defined, i.e. on Mac when using precompiled headers. `ERR_EXTERN` must be in all your C files except the main module which has `ERR_DECLARE` (See your Macintosh manual for more information on precompiled headers with THINK C or MPW)

Initialization Macros

Item	Description
<code>ERR_LIBDECLARE</code>	Declares global variable for error handling.
<code>ERR_ISLIBCREATED</code>	Has the library been created.
<code>ERR_LIBCREATEINIT</code>	Create and initialize the library.
<code>ERR_MAININIT</code>	Initialization of error handling in <code>'main'</code> .

The following macro is preserved for compatibility purposes only.

Note: Is this still needed or can we just forget about the DS issues.

Fatal Errors

Fatal

void NDErr::Fatal (CStr msg)

Exits with a message and produces a `'core dump'` (on UNIX).

Exit

void NDErr::Exit (CStr msg)

Exits with a message.

Signaling Failures

Fail

void NDErr::Fail (CStr modname, ErrIdVal msgId, ...)

Generates a failure and displays the error message which is in `"modname.Errors"` at index `msgId`. If the message contains conversion specifications like `"%d"` or `"%.2s"`, the conversion will apply on the additional arguments (it works like `printf`).

FailStr

void NDErr::FailStr (CStr str, ...)

Generates a failure and displays `'str'` instead of loading a message from the resource file).

FailSilent

void NDErr::FailSilent (void)

Generates a `'silent'` failure (without message).

FailAssert

void NDErr::FailAssert (CStr cstr, CStr fileName, ErrLineVal line)

Use `DBG_CHECK` or `ERR_CHECK` instead.

FailError

void NDErr::FailError (CStr fileName, ErrLineVal line)

Use `DBG_ERROR` instead.

Generating Warnings

Warn

void NDErr::Warn (CStr modname, ErrIdVal msgId, ...)

Generates a warning and displays a warning message loaded from resource. Conversion specifications, if any, will apply.

WarnStr

void NDErr::WarnStr (CStr str, ...)

Generates a warning and displays ``str'` instead of loading a message from the resource file).

Querying the Error State

InError

BoolEnum NDErr::InError (void)

Returns whether we are currently executing error. Recovery code (`BOOL_TRUE`) or whether we are executing normally (no failure signalled, or last failure was cleared by a `RETRY`).

Assertions

Item	Description
<code>ERR_CHECK(t)</code>	Checks that <code><t></code> is true. If <code><t></code> is false, generates the error message: <code>assertion <t> failed file ... line ... "</code> This assertion is not controlled by the <code>DBG_ON</code> compilation flag. Usually, you will use <code>DBG_CHECK</code> rather than <code>ERR_CHECK</code> because you want assertions to disappear in production code. (See <code>basepub.h</code> for <code>DBG_XXX</code> macros).
<code>ERR_CHECKSTR(t, str)</code>	Same as <code>ERR_CHECK</code> except that it generates the message: <code>assertion <str> failed file ... line</code> This is to be used in the special cases when <code><t></code> is too long to fit on one line or if it contains a <code>("</code>). You should use <code>DBG_CHECKSTR</code> instead.
<code>ERR_ASSERT(t)</code>	Is a synonym for <code>ERR_CHECK</code> ifdef <code>DBG_NOCHECKSTR</code> .

Error Reporting

TraceBack

void NDErr::TraceBack (void)

Outputs error traceback starting from the top error frame for the current exception.

FrameTraceBack

void NDErr::FrameTraceBack (ErrFramePtr *frame*)

Outputs error traceback starting from the error frame specified by *frame*'.

Error Conditions Signaled by Error Module

The error module signals the following error conditions:

Item	Description
WARNNIY	Not implemented yet warning (see DBG_NIY macro).
FAILINTR	Failure generated when the program receives an interrupt from the keyboard.
FAILQUIT	Failure generated when the program receives a `quit' signal.
FAILASSERT	Failure generated when an assertion was not satisfied see DBG_CHECK and ERR_CHECK macros).
FAILEXCEPTION	Failure generated by DBG_ERROR.
FAILEXCEPTION	Failure raised by an exception.

Exiting from the Application

ModExit

void NDErr::ModExit (void)

Broadcasts an “exit” message to all modules. This is called implicitly when the application is exited and need not be called.

UNIX Exception Handling

On UNIX, Open Interface installs signal handlers to catch system exceptions such as bus error, floating point exceptions, ... The handlers are installed during the initialization of the Open Interface libraries (usually GW_LibInit) from a static table describing which handler should be installed for which signal. By default, Open Interface catches the following signals: INT, ILL, FPE, SEGV, TERM, HUP, QUIT, TRAP, EMT, BUS, SYS, PIPE and ALRM. If your program relies on signals, you may want to prevent Open Interface from installing some signal handlers (typically SIGPIPE or SIGALRM). To do so, you can query and modify the table of system handlers with the following calls.

SysExceptProc**void NDErr::SysExceptProc (ErrSigVal);**

Data type for UNIX exception handler.

SetSysExceptionHandler**void NDErr::SetSysExceptionHandler (ErrSigVal sig, ErrSysExceptProc proc)**

Records `proc` as the signal handler for signal `sig` in Open Interface's exception handler table. You can pass 0 as `proc` to prevent Open Interface from trapping signal `sig`. This call may be called before initializing the Open Interface libraries.

GetSysExceptionHandler**ErrSysExceptProc NDErr::GetSysExceptionHandler (ErrSigVal sig)**

Returns the signal handler for signal `sig` currently installed in Open Interface's exception handler table. This call may be called before initializing the Open Interface libraries.

SysException**void NDErr::SysException (ErrSigVal sig)**

Default exception handler installed by Open Interface. You may call this procedure from your own signal handler. This procedure reinstalls the exception handler through a call to the `signal` system call and then triggers the Open Interface exception mechanism (`ERR_CATCHXXX`, `ERR_RETRYXXX` directives, see above).

Note: This call does not return to its caller. The exception handler which is reinstalled by this call is the handler in OI's exception handler table, so you need to modify OI's table through `SetSysExceptionHandler` if you want your custom handler to be reinstalled correctly.

W16 Exceptions Handling

Under W16 API, only one interrupt handler per application can be registered by calling `InterruptRegister()`. By default, Open Interface always registers a native interrupt handler to trap the error signals such as GP Fault, divided by zero, and etc.. Open interface will un-register installed handler when program terminated.

For any reason, user can use the following function to disable or enable this interrupt registration mechanism.

Item	Description
<code>ERR_MswRegisterInterrupt (bool)</code>	Enable/Disable the native interrupt registration mechanism and set default action for <code>ERR_LIBINIT</code> , <code>ERR_MAININIT</code> .
<code>ERR_MswRegisterInterruptOnInit (bool)</code>	Set default action for <code>ERR_LIBINIT</code> , <code>ERR_MAININIT</code> .

MswRegisterInterrupt**void NDErr::MswRegisterInterrupt (BoolEnum bool)**

Enable/Disable the ND native interrupt mechanism according to *bool*. Also calls `ERR_MswRegisterInterruptOnInit(bool)` to set the default action of `ERR_LIBINIT`, `ERR_MAININIT`.

It can be called at any time to enable/disable the ND native interrupt mechanism.

If it is called before the Error module initialization, i.e. before `ERR_LIBINIT` or `ERR_MAININIT` it will determine whether the initialization installs the ND native interrupt handler.

MswIsInterruptRegistered**BoolEnum NDErr::MswIsInterruptRegistered (void)**

Return `BOOL_TRUE` if a ND native interrupt handler is registered.

MswRegisterInterruptOnInit**void NDErr::MswRegisterInterruptOnInit (BoolEnum bool)**

If *bool* is set to `BOOL_FALSE` the ND native interrupt handler will be not be enabled by `ERR_LIBINIT` or `ERR_MAININIT`. The default setting is `BOOL_TRUE`. It must be called before the Error module initialization, i.e. before `ERR_LIBINIT` or `ERR_MAININIT`.

Mac Exceptions Handling

Open Interface installs low level error handlers to recover from 680X0 exceptions such as bus error, address error or illegal instruction. If, for some reason, you want to prevent this in your application you need to call `'NoMacSignals'()` at the very beginning of your `main()` routine. The current exception handlers are restored when the application quits or is switched to the background).

NoMacSignals**void NDErr::NoMacSignals (void)**

Disables the low level signal mechanism. It must be called before the Error module initialization, i.e. before `ERR_LIBINIT` or `ERR_MAININIT`.

20 *File Class*

This class provides a portable File I/O API.

Technical Summary

File management and file I/O is a rather complex issue. The main differences between operating systems are the following:

File names: Different syntaxes.

Text files: Special record-oriented format on some systems (mainframes).
Different 'newline' delimiters.

File attributes: Creator and type signatures on the Macintosh.

The File API provides the following services:

- Checking the existence and attributes (owner, access mode) of files.
- Opening (or creating) and closing files.
- Reading/writing data from/to a file (binary and text files).
- Seek and tell operations.
- Filename conversions.
- Reading directories.

This class provides a portable API to open a text file or a binary file and perform I/O operations (i.e. Read and Write) on files.

This class is built on top of the FName and FMgr classes. File names are automatically converted to native syntax if necessary. Use FName if you need more advanced file name conversions. Use FMgr for advanced queries and modifications on the file manager.

Quick Overview of Various File I/O Packages

The various systems have several differences, especially with text files and with record-oriented files, making generic file I/O a complex issue. The following table summarizes some of the differences between systems:

File System	Comments
Unix	On Unix, everything is simple: all files are unstructured. If the file contains text, the lines are separated by a \n character in the byte stream.
Macintosh	On the Macintosh, files have some additional attributes (Type, Creator, VolId). Files are unstructured but lines are separated by a \r character in the text files.
DOS, OS/2, NT	On the PC, files are also unstructured. In text files, lines are separated by a \r\n on the disk but the C runtime library allows you optionally to open the file in TEXT mode, in which case the C RTL maps \r\n into \n on reads and \n into \r\n on writes.

VMS	On VMS, RMS supports many file organizations and record attributes, but everything behaves as on UNIX if the file is converted to Stream_LF, which is the recommended format for binary files. The normal native format for text files is record-oriented, but Stream_LF is also accepted by most native text editors as long as the lines are not too long (less than 512 characters). In record-oriented files, records can be fixed size or variable size.
IBM Mainframe	On the mainframe, a large number of formats are supported, including a flat format for binary files. The native representation for text files is record-oriented (i.e. line-oriented). The SAS/C RTL includes four libraries: First, a Unix-like I/O library (open, read, write, lseek, etc.). This library is compliant to Unix specifications, but is very inefficient (each file is entirely copied to a large memory buffer where all the I/O operations are then performed). Second, a Standard I/O library (fopen, fread, fwrite, etc.). This library is efficient but does not fully comply to the ANSI standard. Third, an 'Augmented Standard I/O' library (afopen, afread, afwrite, etc.) which is a supplement to the Standard I/O library to support features which are not supported by ANSI standard (like record-oriented I/O). And finally, a very complete and very efficient native I/O library. This library is non-portable.
Tandem Mainframes	Text files are also special record-oriented files.

Overview of Open Interface File I/O

Open Interface File I/O is very similar to that provided by the Standard (ANSI) I/O library. It actually extends its functionality to address some of the portability issues. Here is a summary of the features of Open Interface File I/O:

Record-oriented file I/O: This is particularly important because it is the native representation of text files on VMS and CMS and it is not supported by the ANSI library. A typical CMS application will have to use `afopen`, `afread`, `afreadh`, etc. instead of `fopen`, `fread`, etc.

Text files with incomplete last line: This occurs very frequently on Unix because, with some text editors (gnu-emacs for instance), it is possible to save a text file which does not end with a final `\n`. The problem is that other applications (compilers, other text editors, SCCS, ftp, etc.) will complain or will fail when reading such a file. A well-behaved application should not complain but should always save the file with a line terminator at the end of the file. This is handled automatically if a file is opened in Open Interface's `FILE_FMTLINE` format.

Better specifications: In the ANSI specifications, the opening flags for `fopen` are confusing and incomplete. The POSIX standard committee actually had to introduce a new 'fdopen' call which tries to provide modes not covered by `fopen`.

Performance: The VMS implementation of the Standard ANSI library can be inefficient. A typical VMS or CMS application might use instead system-specific calls, and maybe use some tricks like preloading files in the global section.

Macintosh signatures: This class supports Macintosh Creator and Type signatures. It also supports file names which contain Volumes (i.e. logical

disk). This is not supported by the ANSI library. A typical Mac application will have to use `FSOpen`, `FSRead`, etc. instead of `fopen`, `fread`, etc.

PC limitations: This class removes an important limitation on PC where `fread` and `fwrite` are limited to a buffer of 32k bytes. In this class, `NDFile::ReadNBytes` and `NDFile::WriteNBytes` are limited to `MAXINT32`.

Search paths: A search path is a list of directories the application should look through when trying to open a file which is not in the current directory. This feature is available on DOS but requires using `DOSFileOpen` instead of `fopen`. This feature is not usually available on any other system.

File name conversion: File names are automatically converted to the appropriate syntax if they are specified in a foreign syntax (for instance, DOS file names are converted to Unix names when running on Unix).

If some native features are still not covered by this API (for instance, opening a file in Shared access on the Macintosh), it is still possible to retrieve the native file descriptor and call the native API directly. Beware that such calls might not be portable and should be clearly identified, and if possible isolated in one central place in your code.

General Principles for the File API

FilePtr: Most API calls in this File class take a `FilePtr` as their first argument. A `FilePtr` is a pointer to a private `FileRec` structure which is similar to a `FILE` structure (usually defined in `stdio.h`) and which serves as an handle to the actual system file. Several `FilePtrs` could point to the same system file, although this is not recommended.

Checking existence and access rights: Once a `FilePtr` has been created (with `NDFile::New`), you can either check that the corresponding system file actually exists (with `NDFile::Find`) and that it has the appropriate access rights (with `NDFile::GetAccess`), before actually trying to open the file, or you can try to open the file directly (with `NDFile::Open` or `NDFile::TryOpen`).

Opening modes: `NDFile::Open` takes two extra parameters: a `FileIOEnum` and a `FileFmtEnum`. The `FileIOEnum` modes control the Read-Write access. They are the same as the mode argument of `fopen` (`READ`, `WRITE`, `APPEND`, etc.). The `FileFmtEnum` mode specifies the expected format of the file. This can be one of the following three formats: The first is `BINARY` format, in which files are read and written exactly as they appear in the physical storage device, without any conversion. The next is `TEXT` format, in which line separators (`\n` on Unix, `\r` on Mac, `\r\n` on PC, separate records on IBM Mainframes, Tandem and VMS) are automatically translated into a unique and a portable representation which is `'\n'`. The last is `LINE` format. The `TEXT` format can be very inefficient on some systems, so Open Interface introduces this line-oriented file I/O.

Read/Write: The API for Read and Write operations is completely different depending on the file format: In Binary and Text format, the API is very similar to the standard ANSI routines (although the implementation might use machine-specific calls). In Line format, `NDFile::ReadLine` and `NDFile::WriteLine` are slightly different from the standard gets and puts.

Seek/Tell: Here also the API is completely different depending on the file format: In Binary format, the current position is a numeric offset and can be

set arbitrarily. In Text format, it is only possible to set the position to a place which has already been visited. The current position is not kept as a numeric offset (because of the line terminators). In Line format, the current position is always at the beginning of a line. The file system may also support special files (like FIFO, pipes, terminal on Unix) in which it is not possible at all to change the current position.

Examples of Using this API

Open a binary file “data” and read 200 bytes starting at offset 300:

```
{
    NDFilePtr file = new NDFile("data");
    Byte    buf[200];

    file->Open(FILE_IOREAD, FILE_FMTBINARY);
    file->SeekBinaryTo(300);
    file->ReadNBtyes(buf, 200);
    file->Close();
    delete file;
}
```

Open a text file “myapp.rc”, or “defaults.rc” if myapp.rc does not exist, then print all the lines which start with “Definition”:

```
{
    NDFilePtr file = new NDFile("myapp.rc");
    Str        line;

    if (!file->Find()) file->SetSpecName("defaults.rc");
    file->Open(FILE_IOREAD, FILE_FMTLINE);

    while (line = file->ReadLine()) {
        if (NDStr::EqualsSub(line, 10, "Definition", 10))
        {
            NDStr::Printf("%d: %s\n",
file->GetLineOffset(), line);
        }

        file->Close(file);
        delete file;
    }
}
```

Open a text file in READWRITE mode if possible, or in READ mode if it is read-only, or create the file if it does not exist.

```
{
    if (file->Find()) {
        if (file->IsWritable()) {
            file->Backup();
            file->Open(FILE_IOREADWRITE,
FILE_FMTTEXT);
        } else {
            file->Open(FILE_IOREAD, FILE_FMTTEXT);
        }
    } else {
        FMgrCreateFileRec    info;
        info.Access = FMGR_ACCESSDEFAULTS;
        info.MacIds.Creator = FMGR_MACCREATOROIT;
        info.MacIds.Type = FMGR_MACTYPETEXT;
        file->CreateOpen(&info, FILE_FMTTEXT);
    }
    ...
}
```

Summary

The File class does the actual opening, reading, writing, and closing of files — the manipulation of data within a file. It uses the FMgr class for performing operations on files as a whole — copying them, moving them, setting file attributes, etc., and it uses the FName class to do string manipulation when converting file names between the syntax of the various systems.

See also

FName class and FMgr class.

Data Structures

FilePtr

Pointer to the private structure that stores information for performing file I/O.

FilePtr is a pointer to a file object. The file data structure is kept private, but some fields can be accessed indirectly through the API.

See also

NDFile::New, NDFile::Dispose

FileLinePosPtr FileLinePosRec

Position type for files opened in line format mode.

FileLinePosRec is the position type for files opened in line format mode. The NatPos is a machine-specific opaque type. LineNumber is the current line number. The first line of the file is line 0. This type is only used for files opened in FILE_FMTLINE mode.

See also

NDFile::CurLineNumber, NDFile::QueryLinePos, NDFile::SetLinePos

FileNatRefPtr FileNatRefRec

The structure for storing native representation of a file handle and/or file pointer.

FileNatRefRec is the structure for storing the native representation of a file handle and/or file pointer. On ANSI systems, the FileID == fileno (StdioFile).

See also

NDFile::QueryNatRef, NDFile::SetNatRef

FileOffsetVal

Data type for storing file size and offset values.

FileOffsetVal is the data type for storing file size and offset values.

See also

FMgrSizeVal (FMgr class), NDFile::CurSize, NDFile::CurBinaryOffset, NDFile::CurTextOffset

FileTextPosPtr**FileTextPosRec**

The structure for storing generic and machine specific text file positions.

FileTextPosRec is the structure for storing generic and machine specific file positions for files opened in text format. The NatPos is a machine-specific opaque type. The TextOffset is the number of characters before the current position, where each line terminator counts for one character. This type is only used for files opened in FILE_FMTTEXT mode.

See also

NDFile::QueryTextPos, NDFile::SetTextPos

Enumerated Types

FileErrEnum

Enumerated type for specifying the errors reported by this class.

FileErrEnum is the enumerated type for specifying the errors reported by this class. These error codes are also stored in the ErrCodeEnum field of the ErrFuncCallRec defined in the err class.

The various errors are described below.

Identifier	Description
FILE_ERRNONE	No error.
FILE_ERRCVTNAME	File name could not be converted to the target syntax.
FILE_ERRNOTFOUND	File could not be found.
FILE_ERRBADTYPE	File of this type cannot be opened.
FILE_ERRBADACCESS	File access privileges set in operating system denies opening the file in specified I/O mode.
FILE_ERRBADNAME	File could not be created because file name syntax is not allowed.
FILE_ERRNOSPACE	File could not be created or extended because no space is available.
FILE_ERRNOTDIRECTORY	File name is not the name of a directory.
FILE_ERROSSPECIFIC	This error is operating system specific. The actual error code returned by the system call is stored in the ErrCode field of the ErrFuncCallRec structure.

See also

NDFile::GetError, NDFile::Open, ERR_GetErrFuncCallRec

FileFmtEnum

Enumerated type for specifying the format in which a file can be opened.

FileFmtEnum is an enumerated type for specifying the format in which a file can be opened.

The various file formats are described below.

Identifier	Description
FILE_FMTBINARY	<p>This format should be used for binary files. Its ANSI equivalent is using fopen with the "b" flag. In this format, physical bytes are read and written as they actually appeared in the file, without any kind of conversion.</p> <p>This format should not be used with text files because the physical representation of line separators is not portable: "\n" on Unix, "\r" on Mac, "\r\n" on PC, no physical delimiter on VMS or Mainframes. If you open a Record-Oriented file in this format (VMS, Mainframe), the record structure is not accessible.</p> <p>This format is the most flexible for querying and changing the current position (position is a numeric offset on which you can perform arithmetic operations).</p>
FILE_FMTTEXT	<p>This format should be used for a text file when character per character access is required. It is typically used when parsing a text file where performance is not especially important. Its ANSI equivalent is using fopen with no "b" flag.</p> <p>In this format, you can use the same Read/Write calls as in the FILE_FMTBINARY format, the difference is that native line delimiters ("\r" on Mac, "\r\n" on PC, record breaks on VMS and Mainframes) are automatically converted to "\n".</p> <p>You can not use a numeric offset to query and set the current position. You must use NDFile::QueryTextPos and NDFile::SetTextPos instead.</p>
FILE_FMTLINE	<p>This format is similar to FILE_FMTTEXT and should only be used with Text files. The difference is that Read/Write operations are done line by line instead of character by character. By doing this, performance can be significantly improved on systems like VMS and CMS. The ANSI equivalent uses only fgets and fputs. On VMS and CMS, it is mapped to record-oriented file I/O.</p> <p>For this mode, the Read/Write API is completely different and consists of two calls: NDFile::ReadLine and NDFile::WriteLine. The current position can only be at the beginning of a line. You can query and set the current position with NDFile::QueryLinePos and NDFile::SetLinePos. The current line number is managed automatically and can be queried or set by NDFile::CurLineNumber and NDFile::SetLinePos. Also, in this format the current line number is automatically maintained.</p>

File I/O on Record-Oriented Binary files is not supported in this version of the library.

See also

NDFile::Open, NDFile::TryOpen, NDFile::CreateOpen,
NDFile::GetOpenFormat, NDFile::IsOpen...

FileIOEnum

Enumerated type for specifying Input/Output file open modes.

FileIOEnum is an enumerated type for specifying Input/Output file open modes.

The various file I/O modes are described below.

Identifier	Description
FILE_IOREAD	File is opened in read-only mode.
FILE_IOWRITE	File is opened in write-only mode. The content of the file is reset.
FILE_IOAPPEND	File is opened in append mode. The content is preserved and the current position is set at the end of the file.
FILE_IOREADWRITE	File is opened in read-write mode. The content of the file is preserved and the position is set at the beginning of the file.
FILE_IOCLEARREADWRITE	File is opened in read-write mode. The content of the file is reset. The current position is set at the beginning of the file.
FILE_IOREADAPPEND	File is opened in read-append mode. The content of the file is preserved and the current position is set at the end of the file.

The correspondence between these modes and the standard ANSI modes is described below:

FileIOEnum	read	write	write at end	create	clear content	ANSI equivalent
READ	y	n	n	n	n	fopen (f, "r")
WRITE	n	y	n	y	y	fopen (f, "w")
APPEND	n	y	y	y	n	fopen (f, "a")
READWRITE	y	y	n	n	n	fopen (f, "r+")
CLEARREADWRITE	y	y	n	y	y	fopen (f, "w+")
READAPPEND	y	y	y	y	n	fopen (f, "a+")

In all cases except FILE_IOREAD, a backup file is created if the AutoBackup flag is set.

In all cases except FILE_IOREAD and FILE_IOREADWRITE, a new file is created if none is found, except if FailIfNotFound is set (in which case NDFile::Open fails and NDFile::TryOpen returns BOOL_FALSE).

See also

NDFile::Open, NDFile::TryOpen, NDFile::GetOpenMode

Accessing File Attributes

GetSpecName

Get the file name used by `NDFile::Open`.

CStr NDFile::GetSpecName (void);

`NDFile::GetSpecName` returns the “specified name” of the fileobj. The “spec name” is the name used by `NDFile::Open` to open the file. The spec name is initialized to the value passed to `NDFile::New`. The spec name can be relative or absolute. It may be expressed in a foreign syntax, in which case `NDFile::Open` will convert it to the native syntax when it creates the “real name” for a file.

See also

`FilePtr`, `NDFile::SetSpecName`, `NDFile::GetRealName`, `NDFile::New`

SetSpecName

Set the file name used by `NDFile::Open`.

void NDFile::SetSpecName (CStr specname);

`NDFile::SetSpecName` changes the specname of the fileobj. The spec name is the name used by `NDFile::Open` to open the file. The spec name is initialized to the value passed to `NDFile::New`. The spec name can be relative or absolute. It may be expressed in a foreign syntax, in which case `NDFile::Open` will convert it to the native syntax when it creates the “real name” for a file.

See also

`FilePtr`, `NDFile::SetSpecName`, `NDFile::GetRealName`, `NDFile::New`

GetRealName

Get the real file name as set by `NDFile::Open`.

CStr NDFile::GetRealName (void);

`NDFile::GetRealName` returns the “real name” of the fileobj. The real name is set by `NDFile::Find` or `NDFile::Open` to the full, absolute path name of the native file which matches the spec name description. If the spec name is a relative file name and several files are found in the search path, the first matching file will be considered as the real name. The real name might differ from the spec name if the spec name was a relative path name or if it was in a foreign syntax.

Since the real name of the file is something which is returned by the operating system, there is no API call provided to change the value of this field.

See also

`FilePtr`, `NDFile::GetSpecName`, `NDFile::SetSpecName`, `NDFile::New`

GetSearchPath

Get the search path used by `NDFile::Open`.

CStr NDFile::GetSearchPath (void);

`NDFile::GetSearchPath` returns the search path used by `NDFile::Open`. The search path is a list of directory names where `NDFile::Open` (or `NDFile::Find`) should look for the file in case a relative file name is specified. The directories should be separated by a `'|'` character. The search path does not need to contain the current directory — the current directory is always searched first. Each file has its own search path. If the search path is set to `NULL`, which is the default, `NDFile::Open` will use the global `DefSearchPath` defined by `NDFile::SetDefSearchPath` instead.

See also

`NDFile::SetSearchPath`, `NDFile::GetDefSearchPath`,
`NDFile::SetDefSearchPath`

SetSearchPath

Set the search path used by `NDFile::Open`.

void NDFile::SetSearchPath (CStr path);

`NDFile::SetSearchPath` sets the search path used by `NDFile::Open`. The search path is a list of directory names where `NDFile::Open` (or `NDFile::Find`) should look for the file in case a relative file name is specified. The directories should be separated by a `'|'` character. The search path does not need to contain the current directory — the current directory is always searched first. Each file has its own search path. If the search path is set to `NULL`, which is the default, `NDFile::Open` will use the global `DefSearchPath` instead. Use `NDFile::SetSearchPath (file, "")` if `NDFile::Open` should look only in the current directory.

See also

`NDFile::GetSearchPath`, `NDFile::GetDefSearchPath`,
`NDFile::SetDefSearchPath`

GetAutoBackup

Get the value of a file object's auto backup flag.

BoolEnum NDFile::GetAutoBackup (void);

`NDFile::GetAutoBackup` gets the value of a file object's auto backup flag. The auto backup flag is used by `NDFile::Open` if the file is opened in a non read-only mode. If the flag is set to `BOOL_TRUE`, a backup copy of the file will automatically be created. By default, the auto backup flag is set to `BOOL_FALSE`.

See also

`NDFile::SetAutoBackup`

SetAutoBackup

Set the value of a file object's auto backup flag.

void NDFile::SetAutoBackup (BoolEnum flag);

NDFile::SetAutoBackup sets the value of a file object's auto backup flag. The auto backup flag is used by NDFile::Open if the file is opened in a non read-only mode. If the flag is set to `BOOL_TRUE`, a backup copy of the file will automatically be created. By default, the auto backup flag is set to `BOOL_FALSE`.

See also

NDFile::GetAutoBackup

GetFailIfNotFound

Get the value of a file structure's FailIfNotFound flag.

BoolEnum NDFile::GetFailIfNotFound (void);

NDFile::GetFailIfNotFound gets the value of a file object's FailIfNotFound flag. This flag is used by the NDFile::Open and NDFile::TryOpen routines. If the specified file cannot be found after looking up in the search path, and if this flag is set, NDFile::TryOpen returns `BOOL_FALSE` and NDFile::Open fails. If the file can not be found but the flag is not set, the file will automatically be created.

By default, the FailIfNotFound flag is set to `BOOL_FALSE`.

See also

NDFile::SetFailIfNotFound

SetFailIfNotFound

Set the value of a file structure's FailIfNotFound flag.

void NDFile::SetFailIfNotFound (BoolEnum flag);

NDFile::SetFailIfNotFound sets the value of a file object's FailIfNotFound flag. This flag is used by the NDFile::Open and NDFile::TryOpen routines. If the specified file cannot be found after looking up in the search path, and if this flag is set, NDFile::TryOpen returns `BOOL_FALSE` and NDFile::Open fails. If the file can not be found but the flag is not set, the file will automatically be created.

By default, the FailIfNotFound flag is set to `BOOL_FALSE`.

See also

NDFile::GetFailIfNotFound

SetFailOnEof

Set the value of a file structure's FailOnEOF flag.

void NDFile::SetFailOnEof (BoolEnum flag);

NDFile::SetFailOnEof sets the value of a file object's FailOnEOF flag. This flag is used by NDFile::Read and any other read command. If set, the read command will fail if it tries to read past the end of the file.

By default, the `FailOnEOF` flag is set to `BOOL_FALSE`.

See also

`NDFile::GetFailOnEof`

GetFailOnEof

Get the value of a file structure's `FailOnEOF` flag.

BoolEnum NDFile::GetFailOnEof (void);

`NDFile::GetFailOnEof` gets the value of a file object's `FailOnEOF` flag. This flag is used by `NDFile::ReadNBytes` and any other read command. If set, the read command will fail if it tries to read past the end of the file.

By default, the `FailOnEOF` flag is set to `BOOL_FALSE`.

See also

`NDFile::SetFailOnEof`

GetClientData

Gets the client data attached to a file object.

ClientPtr NDFile::GetClientData (void);

`NDFile::GetClientData` gets the client data attached to a file object.

SetClientData

Sets the client data attached to a file object.

void NDFile::SetClientData (ClientPtr data);

`NDFile::SetClientData` sets the client data attached to a file object.

Checking Existence and Access Rights of a File

Find

Searches for a file specified by its spec name.

BoolEnum NDFile::Find (void);

`NDFile::Find` searches for a file specified by its "spec name." The spec name can be absolute or relative. If the spec name is relative, `NDFile::Find` uses the search path to locate the file. The spec name can be specified in a foreign file name syntax (for instance, you can use a DOS file name when running on Unix). The name is converted automatically.

If the file is found, its "real name" is set to the full path name of the file and `NDFile::Find` returns `BOOL_TRUE`. `NDFile::Find` returns `BOOL_FALSE` if the file name can not be converted to the current target syntax, or if the file can not be found. Use `NDFile::GetError` to determine the exact cause of the error.

Even if the file exists, you may not be able to open it. For instance, this file may not be readable (use `NDFile::IsReadable` to check this) or may not be a

normal file (it could be a directory for instance. Use `NDFile::GetNodeType` to check this).

IsReadable

Determines whether the given file has read access.

BoolEnum NDFile::IsReadable (void);

`NDFile::IsReadable` determines whether the current user has read access to the file specified by the `fileobj`.

IsWritable

Determines whether the given file has write access.

BoolEnum NDFile::IsWritable (void);

`NDFile::IsWritable` determines whether the current user has write access to the file specified by the `fileobj`.

GetNodeType

Determines the type of a node.

FMgrNodeEnum NDFile::GetNodeType (void);

`NDFile::GetNodeType` determines the type (file, directory, link, etc.) of the node specified by the `fileobj`.

Opening and Closing a File

Open

Open a file with the specified I/O mode and format.

void NDFile::Open (FileIOEnum *iomode*, FileFmtEnum *format*);

`NDFile::Open` opens the file in the specified I/O mode and format. `NDFile::Open` uses the Search Path to locate the file. Once the file has been found, the `RealName` is set to the absolute name of the file.

Before calling `NDFile::Open`, you can check that the file exists with `NDFile::Find` and then check the file access rights with `NDFile::IsReadable` and `NDFile::IsWritable`. You can also check the file type (normal file or directory) with `NDFile::GetNodeType`.

If the file does not exist, a new empty file is created (unless the I/O mode is `IOREAD`, or unless it is `IOREADWRITE` and the `FailIfNotFound` flag is not set). A better mechanism to create a new file is to call `NDFile::CreateOpen`, which lets you to specify creation parameters.

See also

`FileIOEnum`, `FileFmtEnum`, `NDFile::TryOpen`, `NDFile::CreateOpen`, `NDFile::Close`, `NDFile::IsOpen...`, `NDFile::Find`, `NDFile::IsReadable`, `NDFile::IsWritable`, `NDFile::SetFailIfNotFound`

CreateOpen

Creates a new file and opens it in the given format.

void NDFile::CreateOpen (FMgrCreateFileCPtr *createptr*, FileFmtEnum *format*);

NDFile::CreateOpen creates a new file and opens it in the given format. The file name is specified by the fileobj's "spec name". The "search path" is not used, and the file should not already exist. Once the file has been created, it is opened in FILE_IOREADWRITE mode, and according to the given format. The createptr argument points to a structure which contains additional parameters necessary to create the file. See fmgrpub.h for the description of the FMgrCreateFileRec structure.

On OpenVMS systems if the createptr argument is set to NULL, then the file will be created with the process default protection flags.

See also

FMgrCreateFileRec (FMgr class), FileFmtEnum, NDFile::TryCreateOpen, NDFile::Open, NDFile::TryOpen, NDFile::Close

Close

Close a file.

void NDFile::Close (void);

NDFile::Close closes the file referenced by the fileobj. This function does not release the memory used by the fileobj. Call NDFile::Dispose to release that memory.

See also

NDFile::TryClose, NDFile::Open, NDFile::Dispose

TryClose
TryCreateOpen
TryOpen

Non-asserting versions of the file close, create, and open functions.

BoolEnum NDFile::TryClose (void);

BoolEnum NDFile::TryCreateOpen (FMgrCreateFileCPtr *createptr*, FileFmtEnum *format*);

BoolEnum NDFile::TryOpen (FileIOEnum *iomode*, FileFmtEnum *format*);

The NDFile::TryXXX set of functions perform the same actions as their corresponding NDFile::Close, NDFile::CreateOpen, and NDFile::Open counterparts, except that the NDFile::TryXXX functions all return a BoolEnum value to indicate whether the function succeeded or failed. Refer to the NDFile::XXX functions listed in the See Also section below for details about a corresponding NDFile::TryXXX function.

The rationale for providing two sets of calls (NDFile::TryXXX and NDFile::XXX) that perform the same actions, lies in the way Open Interface handles errors. Because third party APIs may not be designed based upon the contracting metaphor used by Open Interface's error mechanism, it would not be valid for calls in Open Interface that interact with third party APIs (by making calls to the underlying operating system for example) to apply this metaphor.

For this reason, the `NDFile::TryXXX` set of functions are designed to fail without making an assertion. However, the error reporting structure `ErrFuncCallRec` can be used to store information about the `NDFile::TryXXX` function that failed. It is the responsibility of the class which made the call to the routine which failed to write to this structure, and it is the responsibility of the caller of the class to check the structure when an error occurs.

See also

`ErrFuncCallRec`, `ERR_GetErrFuncCallPtr`, `NDFile::Close`,
`NDFile::CreateOpen`, `NDFile::Open`, `NDFile::SetFailIfNotFound`

IsOpen...

Determine if a file is open, and in which modes it is open.

BoolEnum NDFile::IsOpen (void);

BoolEnum NDFile::IsOpenRead (void);

BoolEnum NDFile::IsOpenWrite (void);

#define NDFile::IsOpenBinary (void)
(NDFile::GetOpenFormat (void) ==FILE_FMTBINARY)

#define NDFile::IsOpenText (void) (NDFile::GetOpenFormat (void) ==FILE_FMTTEXT)

#define NDFile::IsOpenLine (void) (NDFile::GetOpenFormat (void) ==FILE_FMTLINE)

The various `NDFile::IsOpen...` functions and macros determine if a file is open, and in which modes it is open.

`NDFile::IsOpen` returns `BOOL_TRUE` if the `fileobj` is open.

`NDFile::IsOpenRead` returns `BOOL_TRUE` if the `fileobj` is open with read access (mode is `READ`, `READWRITE`, `CLEARREADWRITE`, or `READAPPEND`)

`NDFile::IsOpenWrite` returns `BOOL_TRUE` if the `fileobj` is open with write access (mode is `CLEARWRITE`, `OVERWRITE`, `APPEND`, `READWRITE`, `CLEARREADWRITE` or `READAPPEND`)

`NDFile::IsOpenBinary` returns `BOOL_TRUE` if the `fileobj` is open in binary mode.

`NDFile::IsOpenText` returns `BOOL_TRUE` if the `fileobj` is open in text mode.

`NDFile::IsOpenLine` returns `BOOL_TRUE` if the `fileobj` is open in line mode.

See also

`FileIOEnum`, `FileFmtEnum`, `NDFile::GetOpenMode`,
`NDFile::GetOpenFormat`

GetOpenFormat
GetOpenMode

Determine the file format and I/O mode in which a file is open.

FileFmtEnum NDFile::GetOpenFormat (void);

FileIOEnum NDFile::GetOpenMode (void);

NDFile::GetOpenFormat determines the file format in which a file has been opened.

NDFile::GetOpenMode determines the I/O mode in which a file has been opened.

See also

NDFile::IsOpen..., **FileIOEnum**, **FileFmtEnum**

Querying and Changing Position in a File

CurSize

Return the current size of a file.

FileOffsetVal NDFile::CurSize (void);

NDFile::CurSize returns the current size of the file referenced by **fileobj**. This function is valid for all **FileFmtEnum** formats.

GotoBeg

Seek to the beginning of a file.

void NDFile::GotoBeg (void);

NDFile::GotoBeg seeks to the beginning of the file referenced by **fileobj**. This function is valid for all **FileFmtEnum** formats.

GotoEnd

Seek to the end of a file.

void NDFile::GotoEnd (void);

NDFile::GotoEnd seeks to the end of the file referenced by **fileobj**. This function is valid for all **FileFmtEnum** formats.

IsAtEnd

Return whether the current position is the end of the file.

BoolEnum NDFile::IsAtEnd (void);

NDFile::IsAtEnd returns **BOOL_TRUE** if file **fileobj**'s current position is the end of the file. This function is valid for all **FileFmtEnum** formats.

CurBinaryOffset

Return the current position offset in a binary file.

FileOffsetVal NDFile::CurBinaryOffset (void);

NDFile::CurBinaryOffset returns the current position offset in the file referenced by fileobj. This function is similar to the ANSI ftell function. The first byte of the file is at offset zero. This function is only valid for files opened in FILE_FMTBINARY mode.

SeekBinaryTo

Set the current absolute position in a binary file.

void NDFile::SeekBinaryTo (FileOffsetVal position);

NDFile::SeekBinaryTo sets the current position in fileobj to the specified position. This function is similar to fseek (... , SEEK_SET) in ANSI. If the specified value is bigger than the current size of the file, the file will be extended to (at least) the new position. If position is -1, the function goes to the end of the file. This function is only valid for files opened in FILE_FMTBINARY mode.

SeekBinaryBy

Set the file position relative to the current position for a binary file.

void NDFile::SeekBinaryBy (FileOffsetVal position);

NDFile::SeekBinaryBy sets fileobj's position relative to the current position. This function is similar to fseek (... , SEEK_CUR) in ANSI. This function is only valid for files opened in FILE_FMTBINARY format.

CurTextOffset

Return the current position offset in a text file.

FileOffsetVal NDFile::CurTextOffset (void);

NDFile::CurTextOffset returns the current text offset in the file referenced by fileobj. The first byte of the file is at offset zero, and line separators count as one character. This function is only valid for files opened in FILE_FMTTEXT mode.

QueryTextPos

Query the current position structure for a text file.

void NDFile::QueryTextPos (FileTextPosPtr posptr);

NDFile::QueryTextPos queries the current position structure for the text file referenced by fileobj. This posptr structure can later be passed to NDFile::SetTextPos to restore the position to a saved position. This function is only valid for files opened in FILE_FMTTEXT mode.

SetTextPos

Set the current position in a text file.

void NDFile::SetTextPos (FileTextPosCPtr posptr);

NDFile::SetTextPos sets the current position in fileobj to the posptr position saved by a previous call to NDFile::QueryTextPos. This function is only valid for files opened in FILE_FMTBINARY mode.

CurLineNumber

Return the current line number in a file.

FileLineNbVal NDFile::CurLineNumber (void);

NDFile::CurLineNumber returns the current line number in a file opened in FILE_FMTLINE format. First line is line 0. This function is only valid for files opened in FILE_FMTLINE mode.

QueryLinePos

Query the current position for a file opened in line format.

void NDFile::QueryLinePos (FileLinePosPtr posptr);

NDFile::QueryLinePos queries the current position structure for the file referenced by fileobj. This posptr structure can later be passed to NDFile::SetLinePos to restore the position to a saved position. This function is only valid for files opened in FILE_FMTLINE mode.

SetLinePos

Set the current position in a file opened in line format.

void NDFile::SetLinePos (FileLinePosCPtr posptr);

NDFile::SetLinePos sets the current position in fileobj to the posptr position saved by a previous call to NDFile::QueryLinePos. This function is only valid for files opened in FILE_FMTLINE mode.

See also

NDFile::CurLineNumber, FileLinePosRec, NDFile::QueryLinePos

Reading and Writing

The routines to read from and write to a file are rather straightforward. Two sets of routines are provided: one for Binary and Text formats and one for Line format.

Notes:

- Writing works in 'overstrike' mode, not in 'insert' mode, which means that if the position in the file is not at the end, the n bytes following the current position will be overwritten. There is no practical and efficient way to insert data in the middle of an existing file.
- The file will be automatically expanded when writing past the end.

ReadByte

Return the next byte in a binary file.

Int NDFile::ReadByte (void);

NDFile::ReadByte returns the next byte in the binary file referenced by the fileobj. It returns EOF at end of file. This function is similar to fgetc. This function is used for files opened in FILE_FMTBINARY mode.

See also

NDFile::ReadChar, NDFile::WriteByte, NDFile::ReadNBytes,
NDFile::WriteNBytes

WriteByte

Writes a byte to a binary file.

void NDFile::WriteByte (Int byte);

NDFile::WriteByte writes a byte to a binary file. This function is similar to fputc. This function is used for files opened in FILE_FMTBINARY mode.

Writing works in 'overstrike' mode, not in 'insert' mode, which means that if the position in the file is not at the end, then bytes following the current position will be overwritten. There is no practical and efficient way to insert data in the middle of an existing file.

The file will be automatically expanded when writing past the end.

See also

NDFile::WriteChar, NDFile::ReadByte, NDFile::ReadNBytes,
NDFile::WriteNBytes

ReadNBytes

Read a number of bytes from a binary file.

FileOffsetVal NDFile::ReadNBytes (VoidPtr buffer, FileOffsetVal n);

NDFile::ReadNBytes reads n bytes from the fileobj into buffer. The buffer must be allocated for at least n bytes. NDFile::ReadNBytes returns the number of bytes actually read. This number should be the same as n unless the end of the file has been reached. Then, when trying to read past the end of file, a value r (which is less than n) will be returned. The n-r bytes at the end of the buffer will be cleared (set to '\0'). After the read, the position is on the next byte to be read. This function is used for files opened in FILE_FMTBINARY mode.

This class removes an important limitation on the PC where fread and fwrite are limited to a buffer of 32k bytes. In this class, NDFile::ReadNBytes and NDFile::WriteNBytes are limited to MAXINT32.

See also

NDFile::ReadNChars, NDFile::WriteByte, NDFile::ReadByte,
NDFile::WriteNBytes

WriteNBytes

Writes a number of bytes to a binary file.

void NDFile::WriteNBytes (VoidCPtr *buffer*, FileOffsetVal *n*);

NDFile::WriteNBytes writes *n* bytes from *buffer* into *fileobj*. After the write, the file position is just after the last byte written. This function is used for files opened in FILE_FMTBINARY mode.

Writing works in 'overstrike' mode, not in 'insert' mode, which means that if the position in the file is not at the end, then bytes following the current position will be overwritten. There is no practical and efficient way to insert data in the middle of an existing file.

The file will be automatically expanded when writing past the end.

This class removes an important limitation on the PC where *fread* and *fwrite* are limited to a buffer of 32k bytes. In this class, NDFile::ReadNBytes and NDFile::WriteNBytes are limited to MAXINT32.

See also

NDFile::WriteNChars, NDFile::WriteByte, NDFile::ReadByte,
NDFile::ReadNBytes

ReadChar

Return the next character in a text file.

Int NDFile::ReadChar (void);

NDFile::ReadChar returns the next character in the text file referenced by the *fileobj*. It returns EOF at end of file. This function is similar to *fgetc*. This function is used for files opened in FILE_FMTTEXT mode.

See also

NDFile::ReadByte, NDFile::WriteChar, NDFile::ReadNChars,
NDFile::ReadStr, NDFile::ReadTextLine, NDFile::Printf

WriteChar

Writes a character to a text file.

void NDFile::WriteChar (Int *char*);

NDFile::WriteChar writes a *char* to a text file, and increments the *fileobj*'s text offset by one. This function is similar to *fputc*. This function is used for files opened in FILE_FMTTEXT mode.

Writing works in 'overstrike' mode, not in 'insert' mode, which means that if the position in the file is not at the end, then bytes following the current position will be overwritten. There is no practical and efficient way to insert data in the middle of an existing file.

The file will be automatically expanded when writing past the end.

See also

NDFile::WriteByte, NDFile::ReadChar, NDFile::WriteNChars,
NDFile::WriteStr, NDFile::WriteTextLine, NDFile::Printf

ReadNChars

Read a number of characters from a text file.

FileOffsetVal NDFile::ReadNChars (Str buffer, FileOffsetVal n);

NDFile::ReadNChars reads *n* characters from the fileobj into buffer. The buffer must be allocated for at least *n* characters. The result is not necessarily terminated by '\0'. This function is identical to NDFile::ReadNBytes except that the file must be opened in FILE_FMTTEXT format, and all the line separators are converted to '\n' upon reading. The fileobj's text offset is incremented by *n*.

See also

NDFile::ReadNBytes, NDFile::WriteNChars, NDFile::ReadChar,
NDFile::ReadStr

WriteNChars

Writes a number of characters to a text file.

void NDFile::WriteNChars (CStr buffer, FileOffsetVal n);

NDFile::WriteNChars writes *n* characters from buffer into fileobj. This function is identical to NDFile::WriteNBytes except that the file must be opened in FILE_FMTTEXT format and that all occurrences of '\n' are converted to a physical line separator. The fileobj's text offset is incremented by *n*.

Writing works in 'overstrike' mode, not in 'insert' mode, which means that if the position in the file is not at the end, then bytes following the current position will be overwritten. There is no practical and efficient way to insert data in the middle of an existing file.

The file will be automatically expanded when writing past the end.

See also

NDFile::WriteNBytes, NDFile::WriteChar, NDFile::ReadNChars

ReadStr

Read and return a string of characters from a text file.

CStr NDFile::ReadStr (FileOffsetVal n);

NDFile::ReadStr reads a string of *n* characters from the fileobj text file, and returns it. It returns NULL if it was already at the end of the file. The string returned by NDFile::ReadStr is terminated by '\0'. This function is identical to NDFile::ReadNChars except that it does not need any buffer and that the resulting string is terminated by '\0'. Also, NDFile::ReadStr does not return how many characters have been read (use NDStr::Len on the string instead).

See also

NDFile::ReadNChars, NDFile::WriteNChars, NDFile::ReadChar,
NDFile::ReadTextLine

WriteStr

Write a null terminated string to a text file.

void NDFile::WriteStr (CStr string);

NDFile::WriteStr writes a NULL-terminated string into the fileobj FILE_FMTTEXT mode. This function is identical to NDFile::WriteNChars except that the string must be NULL terminated, and you do not need to specify the number of characters to write.

Writing works in 'overstrike' mode, not in 'insert' mode, which means that if the position in the file is not at the end, then bytes following the current position will be overwritten. There is no practical and efficient way to insert data in the middle of an existing file.

The file will be automatically expanded when writing past the end.

ReadTextLine

Read and return a line of characters from a text file.

CStr NDFile::ReadTextLine (void);

NDFile::ReadTextLine reads the next line from a file opened in FILE_FMTTEXT format and returns it as a string. The '\n' is not included. It returns NULL if the position was already at the end of the file. If the file is not properly terminated (i.e. the last line is missing a line terminator), the end of file might occur before the Read is completed. In this case, NDFile::ReadTextLine will return the characters read up to this point as a normal line. At the next attempt, NDFile::ReadTextLine will return NULL. If you want to check for this particular situation, you can call NDFile::IsAtEnd to detect the end of file.

After the read, the position is at the beginning of the next line to be read or at the end of file. The fileobj's text offset is updated. The fileobj's line number is NOT updated.

WriteTextLine

Write a string and line terminator to a text file.

void NDFile::WriteTextLine (CStr string);

NDFile::WriteTextLine writes a string to a text mode file and goes to the next line. This function is identical to NDFile::WriteStr except that it always adds a line terminator. After the write, the position is at the beginning of the next line (in FILE_IOREADWRITE mode, some lines might have been totally or partially overwritten by the operation) or at the end of the file. The fileobj's text offset is updated. The fileobj's line number is NOT updated.

Writing works in 'overstrike' mode, not in 'insert' mode, which means that if the position in the file is not at the end, then bytes following the current position will be overwritten. There is no practical and efficient way to insert data in the middle of an existing file.

The file will be automatically expanded when writing past the end.

ReadLine

Read one line of text from a line format file.

CStr NDFile::ReadLine (void);

NDFile::ReadLine reads the next line from a file in FILE_FMTLINE format and returns it as a string. The '\n' is not included. NDFile::ReadLine returns NULL if it was already at the end of the file. The fileobj's line number is incremented by 1. This function is identical to NDFile::ReadTextLine except that the file must be in FILE_FMTLINE format and that it updates the fileobj's line number. On some systems (CMS, VMS), this mode can be faster than using FILE_FMTTEXT.

WriteLine

Write a string and line terminator to a text file.

void NDFile::WriteLine (CStr string);

NDFile::WriteLine writes a string to a file and goes to the next line. The LineNumber is incremented by 1. This function is identical to NDFile::WriteTextLine except that the file must be in FILE_FMTLINE format and that it updates the fileobj's line number.

Writing works in 'overstrike' mode, not in 'insert' mode, which means that if the position in the file is not at the end, then bytes following the current position will be overwritten. There is no practical and efficient way to insert data in the middle of an existing file.

The file will be automatically expanded when writing past the end.

Miscellaneous Functions

Backup

Create a backup of a file.

void NDFile::Backup (void);

NDFile::Backup creates a backup of the fileobj. The file must not be open at the time of the call.

See also

NDFName::MakeBackupName (FName class)

Flush

Flushes file output buffer.

void NDFile::Flush (void);

NDFile::Flush causes any buffered but unwritten data to be written to the fileobj. The file must be open at the time of the call.

See also

NDFile::Close, NDFile::Write...

Truncate

Truncate a file at the current position.

void NDFile::Truncate (void);

NDFile::Truncate truncates the fileobj at the current position. The file must be open at the time of the call.

See also

NDFile::Close, NDFile::Write...

Default Search Path

SetDefSearchPath

Set the default search path used by NDFile::Open.

static void NDFile::SetDefSearchPath (CStr path);

NDFile::SetDefSearchPath sets the default search path used by NDFile::Open. The default search path is the default value for the search path used by NDFile::Find and NDFile::Open to locate files specified as relative path names. The search path can be overridden globally with NDFile::SetDefSearchPath or only for a specific file with NDFile::SetSearchPath.

As for any search path, the default search path should be a list of directory names separated by '|' or by the native search path separator (':' on Unix, ';' on PC). The default search path is set initially to the value of the SearchPathName environment variable, except on the Macintosh, where the default search path is set to the content of the resource string STR# 10000.

See also

NDFile::GetSearchPath, NDFile::SetSearchPath,
NDFile::GetDefSearchPath, NDFile::GetDefSearchPathName,
NDFile::SetDefSearchPathName

GetDefSearchPath

Get the default search path used by NDFile::Open.

static CStr NDFile::GetDefSearchPath (void);

NDFile::GetDefSearchPath gets the default search path used by NDFile::Open. The default search path is the default value for the search path used by NDFile::Find and NDFile::Open to locate files specified as relative path names. The search path can be overridden globally with NDFile::SetDefSearchPath or only for a specific file with NDFile::SetSearchPath.

As for any search path, the default search path should be a list of directory names separated by '|' or by the native search path separator (':' on Unix, ';' on PC). The default search path is set initially to the value of the SearchPathName environment variable, except on the Macintosh, where the default search path is set to the content of the resource string STR# 10000.

See also

`NDFile::GetSearchPath`, `NDFile::SetSearchPath`, `NDFile::SetDefSearchPath`,
`NDFile::GetDefSearchPathName`, `NDFile::SetDefSearchPathName`

GetDefSearchPathName

Get the name of the environment variable containing the default search path.

static CStr NDFile::GetDefSearchPathName (void);

`NDFile::GetDefSearchPathName` gets the name of the environment variable containing the default search path.

See also

`NDFile::GetSearchPath`, `NDFile::SetSearchPath`,
`NDFile::GetDefSearchPath`, `NDFile::SetDefSearchPath`,
`NDFile::SetDefSearchPathName`

SetDefSearchPathName

Set the name of the environment variable containing the default search path.

static void NDFile::SetDefSearchPathName (CStr name);

`NDFile::SetDefSearchPathName` sets the name of the environment variable containing the default search path.

See also

`NDFile::GetSearchPath`, `NDFile::SetSearchPath`,
`NDFile::GetDefSearchPath`, `NDFile::SetDefSearchPath`,
`NDFile::GetDefSearchPathName`

Direct access to native File I/O

These calls are not implemented on OpenVMS systems because the OpenVMS file system does not represent access to a file via a single 32-bit handle.

QueryNatRef

void NDFile::QueryNatRef (FileNatRefPtr nat);

Returns in ``nat'` the native file handlers for the file.

SetNatRef

void NDFile::SetNatRef (FileNatRefCPtr nat);

Attaches a different native file to an existing file. It does not close the old native file.

Errors

GetError

Return the last error generated.

FileErrEnum NDFile::GetError (void);

NDFile::GetError returns the last error generated by a call to NDFile::Find or NDFile::TryOpen.

See also

FileErrEnum, NDFile::Find, NDFile::TryOpen

SetError

Sets an error for a file.

void NDFile::SetError (FileErrEnum *fileerr*);

Portable API to access native file managers.

Technical Summary

This class provides a portable API to get information from the native file system and change it. It provides queries about existence and access permissions for individual files. It also allows creation, deletion or renaming of files or directories, as well as searching in a directory.

The class assumes that its file name arguments are compatible with the native syntax. Use the `FName` class to convert non-compatible file names to the native syntax. Use the `File` class to open a file and perform File I/O operations (i.e. Read or Write).

The class uses the term `node` to refer to any file system entity (a file, a directory, a link, a device, etc.).

See also:

`FName` class and `File` class.

Data Types

`FMgrNodePtr`
`FMgrNodeRec`

The structure for containing all the file manager information for a node.

`FMgrNodeRec` is the structure for containing all the file manager information for a node. This structure is filled in by the `NDFMgr::QueryNodeInfo` function.

The various fields of this structure are summarized below:

Identifier	Description
<code>NodeType</code>	The type (file, directory, link, etc.) of the node.
<code>RefsNb</code>	The number of references (or hard links) to this node.
<code>Owner</code>	The user ID and group ID of the node's owner.
<code>Times</code>	The node's various creation and modification times.
<code>Access</code>	The node's allowed access rights.
<code>MacIds</code>	The node's Macintosh Type and Creator signatures.
<code>TotalSize</code>	The total size of the node.

See also

`FMgrNodeEnum`, `FMgrRefsVal`, `FMgrOwnerRec`, `FMgrTimesRec`, `FMgrAccessSet`, `FMgrMacIdsRec`, `FMgrSizeVal`, `NDFMgr::QueryNodeInfo`

FMgrAccessSet

Data type for specifying access rights.

FMgrAccessSet is the data type used for specifying access rights for a node. It consists of access bits ored (or added) together. The access bit constants are defined by the `#define FMGR_ACCESS...` statements. The **FMgrAccessSet** type is used by the **FMgrNodeRec** structure, the **FMgrCreateFileRec** structure, and the **FMgrCreateDirRec** structure.

See also

`FMGR_ACCESS...`, **FMgrNodeRec**, **FMgrCreateFileRec**, **FMgrCreateDirRec**

FMgrCreateDirPtr
FMgrCreateDirRec

The structure for storing information necessary for creating a new directory.

FMgrCreateDirRec is the structure for containing information necessary for creating a new directory. It contains a subset of the information in a full **FMgrNodeRec** — only the access information. This structure is used by the `NDFMgr::CreateDir` function.

See also

FMgrNodeRec, **FMgrAccessSet**, `NDFMgr::CreateDir`, **FMgrCreateFileRec**

FMgrCreateFilePtr
FMgrCreateFileRec

The structure for storing information necessary for creating a new file.

FMgrCreateFileRec is the structure for containing information necessary for creating a new file. It contains a subset of the information in a full **FMgrNodeRec** — only the access information and the Macintosh signature information. This structure is used by the `NDFMgr::CreateFile` function and the `NDFFile::CreateOpen` function (in the `File` class).

See also

FMgrNodeRec, **FMgrAccessSet**, **FMgrMacIdsRec**, `NDFMgr::CreateFile`, `NDFFile::CreateOpen` (`File` class), **FMgrCreateDirRec**

FMgrMacIdsPtr
FMgrMacIdsRec

Structure for storing Macintosh type and creator signatures.

FMgrMacIdsRec is the structure for storing a Macintosh file's Creator and Type signatures. The **FMgrMacIdsRec** is used by the **FMgrNodeRec** structure and the **FMgrCreateFileRec** structure. Several common type and creator values are defined by the `FMGR_MAC...` constants.

See also

FMgrNodeRec, **FMgrMacIdVal**, **FMgrCreateFileRec**, `NDFMgr::GetMacCreator`, `NDFMgr::GetMacType`, `FMGR_MAC...`

FMgrMacIdVal

Data type for storing a Macintosh signature.

FMgrMacIdVal is the data type for storing a Macintosh signature. One Id is used to store the Mac file Creator, and one Id is used to store the Mac file Type. The FMgrMacIdVal type is used by the FMgrMacIdsRec structure. Several common type and creator values are defined by the FMGR_MAC... constants.

See also

FMgrMacIdsRec, FMGR_MAC..., NDFMgr::GetMacCreator, NDFMgr::GetMacType

**FMgrOwnerPtr
FMgrOwnerRec**

Structure for storing the owner information for a node.

FMgrOwnerRec is the structure for specifying the user Id and group Id of the owner of a node. The FMgrOwnerRec structure is used by the FMgrNodeRec structure.

See also

FMgrNodeRec

FMgrRefsVal

Data type for representing the number of references (or hard links) to a node.

FMgrRefsVal is the data type for representing the number of references (or hard links) to a node. The FMgrRefsVal type is used by the FMgrNodeRec structure.

See also

FMgrNodeRec

FMgrSizeVal

Data type for representing node size.

FMgrSizeVal is the data type for representing the size of a node. The FMgrSizeVal type is used by the FMgrNodeRec structure.

See also

FMgrNodeRec

**FMgrTimesPtr
FMgrTimesRec**

Structure for storing the various modification and creation times for a node.

FMgrTimesRec is the structure for storing the various modification and creation times for a node. The FMgrTimesRec structure is used by the FMgrNodeRec structure.

The various fields are described below:

Identifier	Description
Creation	The time that the file was originally created.
LastAccess	The last time that the file was opened.
LastModifData	The last time that the file's contents were modified.
LastModifInfo	The last time that the file's description was modified.

On OpenVMS systems, the Creation and LastModifData fields both represent the creation time of the file, and the LastAccess and LastModifData fields both represent the revision time of the file.

See also

FMgrNodeRec, FMgrTimeVal

FMgrTimeVal

Data type for specifying modification and creation times.

FMgrTimeVal is the data type for specifying the various modification and creation times for a node. The FMgrTimeVal type is used by the FMgrTimesRec structure.

See also

FMgrTimesRec

Enumerated Types

FmgrErrEnum

Enumerated type for specifying the errors reported by this class.

FmgrErrEnum is the enumerated type for specifying the errors reported by this class. These error codes are also stored in the ErrCodeEnum field of the ErrFuncCallRec defined in the err class.

The various errors are described below.

Identifier	Description
FILE_ERRNONE	No error.
FILE_ERRNOTFOUND	File could not be found.
FILE_ERRBADACCESS	File access privileges set in operating system denies opening the file in specified I/O mode.
FILE_ERRBADNAME	File could not be created because file name syntax is not allowed.
FILE_ERRNOSPACE	File could not be created or extended because no space is available.
FILE_ERRNOTDIRECTORY	File name is not the name of a directory.
FILE_ERROSSPECIFIC	This error is operating system specific. The actual error code returned by the system call is stored in the ErrCode field of the ErrFuncCallRec structure.

See also

`ERR_GetErrFuncCallRec`

FMgrFileTypeEnum

Identifier	Description
<code>FMGR_FILETYPEUNKNOWN</code>	type unknown
<code>FMGR_FILETYPESTATICLIB</code>	static libraries (Unix:.a , DOS: .lib ..)
<code>FMGR_FILETYPEEDYNAMICLIB</code>	dynamic libraries (.sl .so, .dll ..)
<code>FMGR_FILETYPEAPPLICATION</code>	executables (.exe on DOS and VMS)
<code>FMGR_FILETYPEOBJECT</code>	object file (.o, .obj)
<code>FMGR_FILETYPEASSEMBLY</code>	assembly source file (.asm, .s)
<code>FMGR_FILETYPELINKOPTIONS</code>	special options (.lnk, .opt)
<code>FMGR_FILETYPESCRIPT</code>	shell scripts (.sh .csh .bat .com)
<code>FMGR_FILETYPEPLUSPLUS</code>	C++ source file (.cc .cxx .cpp .C)
<code>FMGR_FILETYPECFILE</code>	C source file (.c)
<code>FMGR_FILETYPEHFILE</code>	C/C++ header file (.h)
<code>FMGR_FILETYPELINT</code>	Lint file (.ln)
<code>FMGR_FILETYPEYACC</code>	Yacc source file (.y)
<code>FMGR_FILETYPELEX</code>	Lex source file (.l)
<code>FMGR_FILETYPESED</code>	Sed source file (.sed)
<code>FMGR_FILETYPEAWK</code>	Awk source file (.awk)
<code>FMGR_FILETYPEEMACSLISP</code>	Emacs-Lisp source file (.el)
<code>FMGR_FILETYPEEMACSLISPOBJ</code>	Emacs-Lisp object file (.elc)
<code>FMGR_FILETYPEARCHIVE</code>	Archive file (.tar)
<code>FMGR_FILETYPECOMPRESSED</code>	Compressed file (.Z, .zip)
<code>FMGR_FILETYPEHELP</code>	Help document (.doc, .man, .doc)
<code>FMGR_FILETYPERC</code>	Rescomp source file (.rc)
<code>FMGR_FILETYPERCO</code>	Rescomp object file (.rco)
<code>FMGR_FILETYPEPCDAT</code>	Rescomp library file (.dat)
<code>FMGR_FILETYPETEXT</code>	Text file (.txt)
<code>FMGR_FILETYPEENXPTKB</code>	Nexpert Text Knowledge-Base (.tkb)
<code>FMGR_FILETYPEENXPCKB</code>	Nexpert Compiled Knowledge-Base (.ckb)
<code>FMGR_FILETYPEENXPEKB</code>	Nexpert Text Knowledge-Base (.ekb)
<code>FMGR_FILETYPEENXPDB</code>	Nexpert Text Database (.npx)
<code>FMGR_FILETYPEESYLK</code>	Sylk database file (.slk)
<code>FMGR_FILETYPEEDBASE</code>	DBase database file (.dbf)
<code>FMGR_FILETYPELOTUS</code>	Lotus database file (.wks)
<code>FMGR_FILETYPEORACLE</code>	Oracle database file (.ora)
<code>FMGR_FILETYPEESYBASE</code>	Sybase database file (.syb)
<code>FMGR_FILETYPEINFORMIX</code>	Informix database file (.inf)
<code>FMGR_FILETYPEINGRES</code>	Ingres database file (.ing)
<code>FMGR_FILETYPEPICT</code>	MacDraw file
<code>FMGR_FILETYPEEMOVIE</code>	QuickTime Movie file
<code>FMGR_FILETYPEEMACPAIN</code>	MacPaint file (.mcp)
<code>FMGR_FILETYPEEMSPAIN</code>	Microsoft Paint file (.msp)

FMGR_FILETYPEGIF	GIF image file (.gif)
FMGR_FILETYPETIFF	TIFF image file (.tif .tiff .TIF .TIFF)
FMGR_FILETYPEXWD	X-Windows Dump image file (.xwd)
FMGR_FILETYPEWINDOWSBMP	MS-Windows DIB image file (.bmp)
FMGR_FILETYPEXBITMAP	X-Windows Bitmap file (.bm)
FMGR_FILETYPEXPIXMAP	HP-Vue Pixmap file (.pm)
FMGR_FILETYPEINDIMAGE	Neuron Data image file (.ndi)
FMGR_FILETYPESUNRASTER	Sun Raster image file (.im8 .ras .rs)
FMGR_FILETYPEXPIIMAGE	Nexpert image file (.nbm)

FMgrNodeEnum

Enumerated type for specifying the type of a node.

FMgrNodeEnum is an enumerated type for specifying the type of a node (file, directory, volume, etc.). This is used as one of the fields of the FMgrNodeRec structure, and is the return value for the NDFMgr::GetNodeType () function.

The various value types are described below.

Identifier'	Description
FMGR_NODEBAD	Node not found by file manager.
FMGR_NODEFILE	Node is a normal file.
FMGR_NODEDIR	Node is a directory.
FMGR_NODEVOLUME	Node is a volume (or storage device).
FMGR_NODECHR	Node is a character special file (unix terminal device).
FMGR_NODEBLOCK	Node is a block special file (unix storage device).
FMGR_NODEFIFO	Node is a pipe or a FIFO structure.

See also

FMgrNodeRec, NDFMgr::GetNodeType

ACCESS...

Defines the various access rights for a node.

The FMGR_ACCESS... constants define the bits that can be ored (or added) together to define a set of access rights for a node. The set of bits for a node is stored in variable of type FMgrAccessSet.

The access bit constants use the following naming convention:

```
FMGR_ACCESS... FMGR_ACCESS<XXX><YYY>
```

where <XXX> is one of:

USER: access rights for the owner of the file

GROUP: access rights for users in the same group

OTHERS: access rights for other users.

and <YYY> is one of:

READ: Read access

WRITE: Write access

EXEC: Exec access (executable for a file or accessible for a directory)

The first six constants in the #define statements above are only used to generate the actual access constants. The actual access constants are described below:

Identifier	Description
FMGR_ACCESSREAD	The owner of the file has read access
FMGR_ACCESSWRITE	The owner of the file has write access
FMGR_ACCESSEXEC	The owner has execute access (if the node is a file) or access to the directory (if the node is a directory).
FMGR_ACCESSGROUPREAD	Members of the owner's group have read access
FMGR_ACCESSGROUPWRITE	Members of the owner's group have write access
FMGR_ACCESSGROUPEXEC	Members of the owner's group have execute access (if the node is a file) or access to the directory (if the node is a directory).
FMGR_ACCESSOTHERSREAD	Other users have read access
FMGR_ACCESSOTHERSWRITE	Other users have write access
FMGR_ACCESSOTHERSEXEC	Other users have execute access (if the node is a file) or access to the directory (if the node is a directory).
FMGR_ACCESSDEFAULTS	The is the default sets of access rights for a node. This is an example of a set of ored (added) access bits.
FMGR_ACCESSSHIFTUSER	
FMGR_ACCESSSHIFTUSER	
FMGR_ACCESSSHIFTOOTHERS	

See also

`FMgrAccessSet`

MAC...

Defines various common Macintosh creator and type signatures.

The Macintosh file system provides a mechanism to give a type to files by the means of signatures. Signatures are four-character (UInt32) values. For

each file, you can define two signatures: one to identify the application which created the file (creator), and one to identify the type of the file itself (type). The FMGR_MAC... constants define several commonly used Macintosh creator and type signatures.

Mac signatures are normally written using a special C syntax when using Macintosh C compilers. This syntax allows defines such as the following:

```
#define FMGR_MACCREATORXCEL 'XCEL'
```

Unfortunately, this single-quote syntax is not portable to many other compilers, so the fmgrpub.h file defines these constants using standard C notation hex constants instead, with the Macintosh notation format inside comments. Also, since ANSI compilers interpret consecutive question marks as the beginning of a trigraph sequence, and some compilers even complain about having this sequence in a comment, '????' is written '<?><?><?><?>' in the comments in the header file.

The creator constants are described below:

Identifier	Signature	Description
FMGR_MACCREATORNONE	????	Unknown creator.
FMGR_MACCREATORSYSTEM	MACS	Macintosh system.
FMGR_MACCREATORXCEL	XCEL	Microsoft Excel.
FMGR_MACCREATORJMND	JMND	Nexpert or SE
FMGR_MACCREATORLDND	LDND	Nextpert or SE
FMGR_MACCREATOROIT	NDOI	Open Editor.
FMGR_MACCREATOROIAP	NDOI	Default signature for an NDOI based applications
FMGR_MACCREATORMPW	MPS	MPW Shell (Apple's Macintosh Programming Workshop).
FMGR_MACCREATORTHINK	KAHL	Symantec Think C.

The file type constants are described below:

Identifier	Signature	Description
FMGR_MACTYPENONE	N/A	Invalid type.
FMGR_MACTYPEUNKNOWN		Unknown type
FMGR_MACTYPEAPPL	APPL	Application program (executable).
FMGR_MACTYPEFLDR	Fldr	Folder (directory).
FMGR_MACTYPEFLDRALIAS	fdrp	Alias of a folder.
FMGR_MACTYPETEXT	TEXT	Text file.
FMGR_MACTYPETKB	TEXT	Text knowledge-base for Neuron Data NEXPERT OBJECT (source format).
FMGR_MACTYPECKB	KBND	Compiled knowledge-base for Neuron Data NEXPERT OBJECT.
FMGR_MACTYPEEKB	NXPE	Nextpert text file (usually a knowledgeable base)
FMGR_MACTYPEDAT	.DAT	.dat file (resource file for Open Interface Elements -based applications).
FMGR_MACTYPEPAINT	PNTG	MacPaint format graphics file.
FMGR_MACTYPEPICT	PICT	PICT format graphics file.

FMGR_MACTYPESYLK	SYLK	Text based spreadsheet interchange format.
FMGR_MACTYPEPNTG	PNTG	MacPaint format graphics file.
FMGR_MACTYPEMOVIE	MooV	QuickTime Movie file.
FMGR_MACTYPETIFF	TIFF	TIFF format graphics file.

See also

FMgrMacIdVal, FMgrMacIdsRec, FMgrNodeRec,
NDFMgr::GetMacCreator, NDFMgr::GetMacType

Querying and Changing File/Directory Attributes

Exists

Determines whether the specified node exists.

static BoolEnum NDFMgr::Exists (CStr name);

NDFMgr::Exists returns TRUE if the given node name (file, directory or volume) exists. It returns FALSE otherwise.

See also

NDFMgr::GetNodeType, NDFMgr::Is...

Is...

Functions for checking the access permissions of a node.

static BoolEnum NDFMgr::IsExecutable (CStr name);

static BoolEnum NDFMgr::IsReadable (CStr name);

static BoolEnum NDFMgr::IsWritable (CStr name);

These functions test whether a given node can be read from, written to, or executed. For directories, NDFMgr::IsExecutable means that the directory can be searched.

Note, on the Macintosh a file will be considered as non-writable in 3 cases:

- The file is locked.
- The file is in use by another user or another application.
- The file is in a shared folder to which you do not have write access.

See also

FMgrAccessSet, FMGR_ACCESS...

IsDevConcealed

Checks whether a concealed device is present for the file or directory passed.

static BoolEnum NDFMgr::IsDevConcealed (CStr name);

The OpenVMS file system has the concept of a concealed device. NDFMgr::IsDevConcealed determines whether the file specification passed contains a concealed device. This function checks whether the top level

directory contains the directory file 000000.DIR;1 which will be present in the case of a physical device, but absent in the case of a concealed device.

On all other platforms `NDFMgr::IsDevConcealed` returns `BOOL_FALSE`.

See also

`NDFMgr::CheckDir`, `NDFMgr::CheckFile`

QueryNodeInfo

Queries all the information for a node.

static BoolEnum NDFMgr::QueryNodeInfo (CStr name, FMgrNodePtr info);

`NDFMgr::QueryNodeInfo` collects all the information about the given node, and fills in the info structure. It returns `TRUE` if the given node (file, directory or volume) exists. It returns `FALSE` otherwise.

See also

`NDFMgr::GetNodeType`, `FMgrNodePtr`

GetNodeType

Determines the type of the specified node.

static FMgrNodeEnum NDFMgr::GetNodeType (CStr name);

`NDFMgr::GetNodeType` returns the type of the specified node. It returns type `FMGR_NODEBAD` if the specified string does not refer to a valid node.

See also

`NDFMgr::Is...`, `FMgrNodeEnum`

GetMac...

Returns the Macintosh signatures of a file.

static FMgrMacIdVal NDFMgr::GetMacCreator (CStr name);

static FMgrMacIdVal NDFMgr::GetMacType (CStr name);

`NDFMgr::GetMacCreator` returns the Macintosh creator signature of the specified file. It returns type `FMGR_MACCREATORNONE` on non-Macintosh platforms. `NDFMgr::GetMacType` returns the Macintosh type signature of the specified file. It returns type `FMGR_MACTYPENONE` on non-Macintosh platforms.

See also

`FMGR_MAC...`, `FMgrMacIdVal`, `FMgrMacIdsRec`

Is...

Macros for checking the type of a node.

static BoolEnum NDFMgr::IsDir (CStr name);

static BoolEnum NDFMgr::IsFile (CStr name);

static BoolEnum NDFMgr::IsVolume (CStr name);

These macros test whether a given node is of a particular type. Their return values are compatible with type BoolEnum.

On VMS systems, you should use NDFMgr::CheckFile or NDFMgr::CheckDir. These calls will not perform I/O operations, resulting in a significant performance increase over calls to NDFMgr::IsFile or NDFMgr::IsDir.

See also

NDFMgr::GetNodeType, FMgrNodeEnum, NDFMgr::CheckDir, NDFMgr::CheckFile

Check...

Checks the type of a node without performing I/O operations on VMS.

static BoolEnum NDFMgr::CheckDir (CStr name);

static BoolEnum NDFMgr::CheckFile (CStr name);

VMS file and directory names have a particular syntax (they must not end in .DIR;1) and so in situations in which you are sure that the file or directory already exists (such as calling NDFMgr::PerfDirFiles to trigger a callback procedure for a particular file or directory), you can call NDFMgr::CheckXXX to determine whether or not the file or directory is valid.

On VMS systems these functions will not perform I/O operations, resulting in a significant performance increase over their NDFMgr::IsXXX counterparts.

These functions are intended for use on VMS systems. On all platforms except VMS these functions are equivalent to NDFMgr::IsXXX.

See also

NDFMgr::IsDevConcealed, NDFMgr::PerfDirFiles, NDFMgr::IsDir, NDFMgr::IsFile, Finding File Type by MacType or FileExtRec

Finding File Type by Mac Type or by File Extension

FMgrFileTypeEnum

Enumerated type identifying one of the file types pre-defined in Open Interface.

Identifier	Description
FMGR_FILETYPEUNKNOWN	type unknown
FMGR_FILETYPESTATICLIB	static libraries (Unix:.a , DOS: .lib ..)

FMGR_FILETYPE DYNAMICLIB	dynamic libraries (.sl .so, .dll ..)
FMGR_FILETYPE APPLICATION	executables (.exe on DOS and VMS)
FMGR_FILETYPE OBJECT	object file (.o, .obj)
FMGR_FILETYPE ASSEMBLY	assembly source file (.asm, .s)
FMGR_FILETYPE LINKOPTIONS	special options (.lnk, .opt)
FMGR_FILETYPE SCRIPT	shell scripts (.sh .csh .bat .com)
FMGR_FILETYPE CPLUSPLUS	C++ source file (.cc .cxx .cpp .C)
FMGR_FILETYPE CFILE	C source file (.c)
FMGR_FILETYPE HFILE	C/C++ header file (.h)
FMGR_FILETYPE LINT	Lint file (.ln)
FMGR_FILETYPE YACC	Yacc source file (.y)
FMGR_FILETYPE LEX	Lex source file (.l)
FMGR_FILETYPE SED	Sed source file (.sed)
FMGR_FILETYPE AWK	Awk source file (.awk)
FMGR_FILETYPE EMACSLISP	Emacs-Lisp source file (.el)
FMGR_FILETYPE EMACSLISPOBJ	Emacs-Lisp object file (.elc)
FMGR_FILETYPE ARCHIVE	Archive file (.tar)
FMGR_FILETYPE COMPRESSED	Compressed file (.Z, .zip)
FMGR_FILETYPE HELP	Help document (.doc, .man, .doc)
FMGR_FILETYPE RC	Rescomp source file (.rc)
FMGR_FILETYPE RCO	Rescomp object file (.rco)
FMGR_FILETYPE DAT	Rescomp library file (.dat)
FMGR_FILETYPE TEXT	Text file (.txt)
FMGR_FILETYPE NXP TKB	Nexpert Text Knowledge-Base (.tkb)
FMGR_FILETYPE NXP CKB	Nexpert Compiled Knowledge-Base (.ckb)
FMGR_FILETYPE NXP EKB	Nexpert Text Knowledge-Base (.ekb)
FMGR_FILETYPE NXP DB	Nexpert Text Database (.nxp)
FMGR_FILETYPE SYLK	Sylk database file (.slk)
FMGR_FILETYPE DBASE	DBase database file (.dbf)
FMGR_FILETYPE LOTUS	Lotus database file (.wks)
FMGR_FILETYPE ORACLE	Oracle database file (.ora)
FMGR_FILETYPE SYBASE	Sybase database file (.syb)
FMGR_FILETYPE INFORMIX	Informix database file (.inf)
FMGR_FILETYPE INGRES	Ingres database file (.ing)
FMGR_FILETYPE PICT	MacDraw file
FMGR_FILETYPE MOVIE	QuickTime Movie file
FMGR_FILETYPE MACPAINT	MacPaint file (.mcp)
FMGR_FILETYPE MSPAINT	Microsoft Paint file (.msp)
FMGR_FILETYPE GIF	GIF image file (.gif)
FMGR_FILETYPE TIFF	TIFF image file (.tif .tiff .TIF .TIFF)
FMGR_FILETYPE XWD	X-Windows Dump image file (.xwd)
FMGR_FILETYPE WINDOWS BMP	MS-Windows DIB image file (.bmp)
FMGR_FILETYPE XBITMAP	X-Windows Bitmap file (.bm)
FMGR_FILETYPE XPIXMAP	HP-Vue Pixmap file (.pm)
FMGR_FILETYPE NDI	Neuron Data image file (.ndi)

FMGR_FILETYPESUNRASTER	Sun Raster image file (.im8 .ras .rs)
FMGR_FILETYPEENXPIIMAGE	Nexpert image file (.nbm)

NDFMgrFileExt

Describes a file extension

Identifier	Description
ExtText	text representation of the file extension (without the '.').
Syntaxes	Set of file name syntaxes in which this extension is valid (see fnamepub.h). Use FNAME_STXALL if valid on all machines.

NDFMgrFileType

Describes a file type

Identifier	Description
FileTypeId	Enumerated constant identifying the file type (see FMgrFileTypeEnum).
MacType	Identifies a Mac type signature. Should be 0 or FMGR_MACTYPENONE if this type does not define any Mac signature. If several file types share the same signature, FMGR_FindFileType only considers the first one (so do not use FMGR_MACTYPETEXT even if it is a Text file, because being a text file is not a good discriminant; use FMGR_MACTYPENONE instead).
Extensions [FMGR_FILEEXTMAX]	Array of up to 5 FMgrFileExtRec. This allows you to define several possible extensions for the same type, or to use different extensions on different platforms.

AddFileType

Adds a file type.

```
static void NDFMgr::AddFileType (FMgrFileTypeCPtr fmgrfiletype);
```

RemoveFileType

Removes a file type.

```
static void NDFMgr::RemoveFileType (FMgrFileTypeCPtr fmgrfiletype);
```

GetNumFileTypes

Returns the number of registered file types.

```
static ArrayIVal NDFMgr::GetNumFileTypes (void);
```

GetNthFileType

Returns the nth register file type description.

```
static FMgrFileTypeCPtr NDFMgr::GetNthFileType (ArrayIVal n);
```

FindFileTypeId

Returns the FileTypeId for the given file.

static FMgrFileTypeEnum NDFMgr::FindFileTypeId (CStr name);

This call fails if specified name does not exist. Returns FMGR_FILETYPEUNKNOWN if file exists but its type can not be determined.

FindFileTypeInfo

Returns the full file description for the given file.

static FMgrFileTypeCPtr NDFMgr::FindFileTypeInfo (CStr name);

Same as NDFMgr::FindFileTypeId but returns the full type description instead of just the FileTypeId. Returns NULL if type can not be determined.

Creating

CreateDir

Create a directory with the specified permission rights.

static void NDFMgr::CreateDir (CStr name, FMgrCreateDirCPtr access);

NDFMgr::CreateDir creates a directory called name with the specified permission rights. The parent of the directory must already exist but the name directory itself should not exist. The new directory is created with the access rights specified in the FMgrCreateDir structure passed, or set to FMGR_ACCESSDEFAULTS if the parameter is NULL.

On OpenVMS systems the name of the directory can either be specified as a file name (for example, [A]B.DIR or just [A]B, since .DIR is the default extension), or as a directory specification (for example, [A.B]). If there is no parent or top level directory in the specification, then the directory will be created in the current working directory.

See also

FMgrCreateDirRec, FMGR_ACCESS..., NDFMgr::TryCreateDir,
NDFMgr::CreateFile

CreateFile

Create a file with the specified permission rights.

static void NDFMgr::CreateFile (CStr name, FMgrCreateFileCPtr access);

NDFMgr::CreateFile creates a file called name with the specified permission rights and Macintosh signatures. The target directory must already exist but the file itself should not exist.

The new file is created with the access rights and signatures specified in the FMgrCreateDir structure passed. If access is NULL, then the new file's access rights are set to FMGR_ACCESSDEFAULTS, its Macintosh Creator signature is set to FMGR_MACCREATORNONE, and its Macintosh Type signature is set to FMGR_MACTYPENONE.

On OpenVMS systems if the file already exists, then a new version of the file is created.

See also

FMgrCreateFileRec, FMGR_ACCESS..., NDFMgr::TryCreateFile, NDFMgr::CreateDir

Copying

CopyFile

Copy a file.

static void NDFMgr::CopyFile (CStr original, CStr copy);

NDFMgr::CopyFile makes a copy of a file. *original* is the name of the original file. *copy* is the name of the copy. If *copy* is NULL, the copy will be created in the same directory and the name will be generated by NDFName::MakeBackupName. NDFMgr::CopyFile can also be used for links.

Under Windows, Windows NT, and OS/2 the default is to copy the file without checking whether a copy of the same name exists first.

See also

NDFName::MakeBackupName (FName class), NDFMgr::TryCopyFile, NDFMgr::CopyDir, NDFMgr::CopyNode, NDFMgr::MoveFile

CopyDir

Copy a directory and all of its content.

static void NDFMgr::CopyDir (CStr original, CStr copy);

NDFMgr::CopyDir makes a copy of a directory and all of its content.

Under Windows, Windows NT, and OS/2 the default is to copy the directory without checking whether a copy of the same name exists first.

See also

NDFMgr::TryCopyDir, NDFMgr::CopyFile, NDFMgr::CopyNode, NDFMgr::MoveDir

CopyNode

Copy a node.

static void NDFMgr::CopyNode (CStr original, CStr copy);

NDFMgr::CopyNode makes a copy of a node (file, link or directory). NDFMgr::CopyNode is more general than NDFMgr::CopyFile and NDFMgr::CopyDir, but may be slower.

See also

NDFMgr::TryCopyNode, NDFMgr::CopyFile, NDFMgr::CopyDir, NDFMgr::MoveNode

Moving

MoveFile

Rename and/or move a file.

static void NDFMgr::MoveFile (CStr *original*, CStr *move*);

NDFMgr::MoveFile renames a file and/or moves it to another directory. The move argument can be an existing directory, in which case the original is moved to this directory, or it can be the new path name for the original file. NDFMgr::MoveFile can also be used for links.

NDFMgr::MoveFile will fail if a file of the same name already exists in the location passed.

See also

NDFMgr::TryMoveNode, NDFMgr::MoveDir, NDFMgr::MoveNode,
NDFMgr::CopyFile

MoveDir

Rename and/or move a directory.

static void NDFMgr::MoveDir (CStr *original*, CStr *move*);

NDFMgr::MoveDir renames a directory and/or moves it to another directory. The move argument can be an existing directory, in which case the original is moved to this directory, or it can be the new path name for the original directory.

NDFMgr::MoveDir will fail if a directory of the same name already exists in the location passed. Note, that it is not possible to move an entire directory between physical drives; this is a limitation imposed by the operating system. Alternatively, you can use NDFMgr::CopyDir to copy the directory to the new location and then NDFMgr::DeleteDir to delete the original directory.

Although DOS and Windows do not support moving directories, NDFMgr::MoveDir is implemented using the alternative method just described.

See also

NDFMgr::TryMoveDir, NDFMgr::MoveFile, NDFMgr::MoveNode,
NDFMgr::CopyDir

MoveNode

Rename and/or move a node.

static void NDFMgr::MoveNode (CStr *original*, CStr *move*);

NDFMgr::MoveNode renames a node and/or moves it to another directory. The move argument can be an existing directory, in which case the original is moved to this directory, or it can be the new path name for the original node. NDFMgr::MoveNode can be used for either files or directories. It is more general than NDFMgr::MoveDir and NDFMgr::MoveFile, but may be slower.

See also

NDFMgr::TryMoveNode, NDFMgr::MoveDir, NDFMgr::MoveFile,
NDFMgr::CopyNode

Deleting

DeleteFile

static void NDFMgr::DeleteFile (CStr name);

Delete a file.

NDFMgr::DeleteFile deletes the specified file. The file must exist.

See also

NDFMgr::TryDeleteFile, NDFMgr::DeleteDir, NDFMgr::DeleteNode,
NDFMgr::DeleteDirContent

DeleteDir

Delete a directory and all its content.

static void NDFMgr::DeleteDir (CStr name);

NDFMgr::DeleteDir deletes the specified directory and all its content.

See also

NDFMgr::TryDeleteDir, NDFMgr::DeleteFile, NDFMgr::DeleteNode,
NDFMgr::DeleteDirContent, NDFMgr::PurgeDir

DeleteNode

Delete a node.

static void NDFMgr::DeleteNode (CStr name);

NDFMgr::DeleteNode deletes the specified node. The node can be a file or a directory. This call is more general than NDFMgr::DeleteFile and NDFMgr::DeleteDir, but it is slower because it needs to test the type of the node first.

See also

NDFMgr::TryDeleteNode, NDFMgr::DeleteFile, NDFMgr::DeleteDir

DeleteDirContent

Delete the contents of a directory.

static void NDFMgr::DeleteDirContent (CStr name);

NDFMgr::DeleteDirContent deletes the content of the specified name directory (including sub-directories), but leaves the name directory itself intact.

See also

NDFMgr::TryDeleteDirContent, NDFMgr::DeleteDir, NDFMgr::PurgeDir

Try...

Non-asserting versions of the copy, create, delete, and move functions.

```

static BoolEnum NDFMgr::TryCopyDir (CStr original, CStr copy);
static BoolEnum NDFMgr::TryCopyFile (CStr original, CStr copy);
static BoolEnum NDFMgr::TryCopyNode (CStr original, CStr copy);
static BoolEnum NDFMgr::TryCreateDir (CStr name, FMgrCreateDirCPtr access);
static BoolEnum NDFMgr::TryCreateFile (CStr name, FMgrCreateFileCPtr access);
static BoolEnum NDFMgr::TryDeleteDir (CStr name);
static BoolEnum NDFMgr::TryDeleteFile (CStr name);
static BoolEnum NDFMgr::TryDeleteNode (CStr name);
static BoolEnum NDFMgr::TryDeleteDirContent (CStr name);
static BoolEnum NDFMgr::TryMoveDir (CStr original, CStr move);
static BoolEnum NDFMgr::TryMoveDir (CStr original, CStr move);
static BoolEnum NDFMgr::TryMoveFile (CStr original, CStr move);
static BoolEnum NDFMgr::TryMoveNode (CStr original, CStr move);

```

The NDFMgr::TryXXX set of functions perform the same actions as their corresponding NDFMgr::XXX counterparts, except that the NDFMgr::TryXXX functions all return a BoolEnum value to indicate whether the function succeeded or failed. Refer to the NDFMgr::XXX functions listed in the See Also section below for details about a corresponding NDFMgr::XXX function.

The rationale for providing two sets of calls (NDFMgr::TryXXX and NDFMgr::XXX) that perform the same actions, lies in the way Open Interface handles errors. Because third party APIs may not be designed based upon the contracting metaphor used by Open Interface's error mechanism, it would not be valid for calls in Open Interface that interact with third party APIs (by making calls to the underlying operating system for example) to apply this metaphor.

For this reason, the NDFMgr::TryXXX set of functions are designed to fail without making an assertion. However, the error reporting structure ErrFuncCallRec can be used to store information about the NDFMgr::TryXXX function that failed. It is the responsibility of the class which made the call to the routine which failed to write to this structure, and it is the responsibility of the caller of the class to check the structure when an error occurs.

See also

```

ErrFuncCallRec, ERR_GetErrFuncCallPtr, NDFMgr::CopyDir,
NDFMgr::CopyFile, NDFMgr::CopyNode, NDFMgr::CreateDir,
NDFMgr::CreateFile, NDFMgr::DeleteDir, NDFMgr::DeleteFile,
NDFMgr::DeleteNode, NDFMgr::DeleteDirContent, NDFMgr::MoveDir,
NDFMgr::MoveFile, NDFMgr::MoveNode

```

PurgeDir

Purge specified files from a directory.

static void NDFMgr::PurgeDir (CStr dir, CStr pattern);

NDFMgr::PurgeDir purges from the specified directory dir, all the files that match the specified pattern. If pattern is NULL, a “default” purge will be performed as follows:

- On Unix, delete “core”, “*~” and “#*” files;
- On PC, delete “*.?\$?”;
- On VMS, delete lower numbered versions.

Wildcard specifications used in the pattern argument are interpreted literally, unlike DOS where the pattern *.* yields all files and directories, in this case, NDFMgr::PurgeDir therefore purges only those files with a “.” in the name. Use the functions NDFMgr::DirWildCard or NDFMgr::AllFilesWildCard to return the pattern that obtains either the directories or directories plus files respectively.

On OpenVMS systems, the pattern should be a standard wildcard file specification (for example, *.RCO would cause the .RCO files within a directory to be purged). Also a directory specification of the form [A.B...] will cause a purge of files within the directory tree starting at directory [A.B] to be initiated.

See also

NDFMgr::DeleteDir, NDFMgr::DeleteDirContent

Performing an Action

PerfDirFiles

Call a user function for each matching file in a directory.

static PerfEnum NDFMgr::PerfDirFiles (CStr dir, CStr pattern, FMgrPerfFileProc func, ClientPtr data);

typedef PerfEnum (*FMgrPerfFileProc) (CStr, CStr, ClientPtr);

NDFMgr::PerfDirFiles performs an action on all the entries of a directory which match a given pattern. The first argument (dir) is the pathname of the directory to scan. The second argument (pattern) is a wildcard expression which will be used to filter the entries. No filtering will be done (all entries will be processed) if pattern is NULL.

Note on OpenVMS systems, the second argument (pattern) should be a standard wildcard file specification (for example, *.RCO would cause the .RCO files within a directory to be processed). Also a directory specification of the form [A.B...] will cause all the files within the directory tree starting at directory [A.B] to be processed.

Wildcard specifications used in the pattern argument are interpreted literally, unlike DOS where the pattern *.* yields all files and directories, in this case, NDFMgr::PerfDirFiles therefore limits its search to those files with a “.” in the name. Use the functions NDFMgr::DirWildCard or

NDFMgr::AllFilesWildcard to return the pattern that obtains either the directories or directories plus files respectively.

Wildcard specifications used in the pattern argument are interpreted literally, unlike DOS where the pattern *.* yields all files and directories, FMgr functions will limit its search to files and directories with a "." in the name.

The third argument (func) is a user function, of type FMgrPerfFileProc, which will be called for each matching entry. The fourth argument (data) is a pointer to client data which will be passed to func at each iteration.

FMgrPerfFileProc is the type to use for the function which will be called for matching entries in a directory. The first argument to the user function is the pathname of the directory being scanned. The second argument is the name of the entry (file or subdirectory) being processed. The third argument is the client data which was passed to NDFMgr::PerfDirFiles. The user function should return PERF_CONTINUE if the scanning should continue, or PERF_STOP otherwise.

ClientPtr, PerfEnum and the PERF_... constants are described in the Base class.

See also

NDFMgr::DirWildcard, NDFMgr::AllFilesWildcard,
NDFMgr::PerfVolumes, Base class

DirWildcard

Return wildcard pattern that matches only directories.

static CStr NDFMgr::DirWildcard (void);

NDFMgr::DirWildcard returns a wildcard expression which matches only directories. It returns "*.DIR;1" on VMS systems, and "*" on other systems (in which directories cannot be distinguished from regular files by just the syntax of their names). The returned wildcard string can be used for filtering files to process with the FMgrPerfFileProc procedure.

See also

FMgrPerfFileProc, NDFMgr::AllFilesWildcard

AllFilesWildcard

Return wildcard pattern that matches all the files in a directory.

static CStr NDFMgr::AllFilesWildcard (void);

NDFMgr::AllFilesWildcard returns a wildcard expression which matches all the files in a directory. It returns "*.*;*" on VMS systems, and "*" on other systems (in which files cannot be distinguished from directories by just the syntax of their names). The returned wildcard string can be used for filtering files to process with the FMgrPerfFileProc procedure.

See also

FMgrPerfFileProc, NDFMgr::DirWildcard

PerfVolumes

Call a user function for each volume in the system.

static void NDFMgr::PerfVolumes (FMgrPerfVolProc *func*, ClientPtr *data*);

NDFMgr::PerfVolumes performs an action on all the volumes of the system. The first argument (*func*) is a user function, of type FMgrPerfVolProc, which will be called for each volume. The second argument (*data*) is a pointer to client data which will be passed to *func* at each iteration.

FMgrPerfVolProc is the type to use for the function which will be called for each volume in the system. The first argument to the user function is the name of the volume being processed. The second argument is the client data which was passed to NDFMgr::PerfVolumes. The user function should return PERF_CONTINUE if the scanning should continue, or PERF_STOP otherwise.

ClientPtr, PerfEnum and the PERF_... constants are described in the Base class.

Under Windows, Windows NT, or OS/2 the volume name to pass to the callback function is the upper case drive letter followed by a semi-colon (C: or E: for example). Removeable diskette drives (usually A: and B:) cannot be passed explicitly nor scanned. Logical drives for CD-ROM, hard drives, RAM drives, and network drives can be passed explicitly by their drive letter or scanned.

See also

NDFMgr::PerfDirFiles, Base class

22 *FName Class*

This class provides file name conversion between DOS, Mac, Unix and VMS.

Technical Summary

This class provides utility routines for manipulating file names and for converting file names between the syntaxes of various machines.

Most of these routines are pure string manipulations. Use the File class to open one file and perform File I/O operations (i.e. Read or Write). Use the FMgr class to query the file system about the existence and access rights for a given file, or to delete, rename, find, move or copy a file or a directory.

Design Overview

This class is useful because unfortunately, the different operating systems have different syntaxes for filenames. The most general syntax would include all of the following components:

```
host volume directories basename extension version
```

Terminology:

host identifies a machine on a network (also called node on VMS). volume identifies a physical disk or a logical partition of a disk (also called a device on VMS. the term device was not used so as to prevent confusion with PC or Unix devices.). extension is usually '.' followed by one or more characters. version is supported only in VMS. It consists of ';' and a number. The path of a file is defined as the host + volume + directories. The full path name is defined as the path + base + ext + version.

Syntax Rules:

Some systems only support certain components (for example, the UNIX syntax does not have any volume component). Some components are built-in on some systems but are just a matter of convention on others. For example, extensions are built-in on DOS and VMS, but are just a convention on UNIX and Macintosh. The version component is built-in on VMS. The same syntax can be used on Unix or Macintosh, but usually file names do not have any version component on these systems.

Also, some systems impose restrictions on the length on some components (i.e. 8 characters max for basename and 3 max for extension on DOS). Some systems impose that some components must be present (basename cannot be empty on DOS; extension must at least contain '.' on VMS). Another complicating factor is that file names can be specified either as absolute or relative names (with variants such as the '~' on UNIX).

The following table summarizes the different syntaxes for absolute file names:

System	Host	Volume	Directories	Base	Ext	Vers
UNIX			/dir1/dir2/	file	.ext	
DOS, OS/2, NT		A:	\dir1\dir2\	file	.ext	
Macintosh		volume:	dir1:dir2:	file	.ext	
VMS	host::	volume:	[dir1.dir2]	file	.ext	;vers

Here are a few examples of relative file names (some with alternate relative forms inside parentheses) for the different systems:

System	In Current Dir	In Subdirectory	In Grandparent Dir
UNIX	file (./file)	dir1/file (./dir1/file)	.././dir1/file
DOS, OS/2, NT	file (.\file)	dir1\file (.\dir1\file)	..\..\dir1\file
MAC	file	:dir1:file	:::dir1:file
VMS	file [file]	[.dir1]file	[--.dir1]file

UNIX shell scripts have the following additional features:

~	Designates the home directory of the current user.
~user	Designates the home directory of the specified user.
\$VAR	Designates the value of the environment variable VAR.

VMS syntax also has a few additional features:

[000000]	Is used for the top directory of a volume, so [000000.dir1] is equivalent to [dir1].
[dir1.dir2]	Is used when referring to the directory as a path.
[dir1]dir2. dir	Is used when referring to the directory as a file.
lognam	VMS also defines logical names (similar but not quite identical to UNIX environment variables) with lots of semantic subtleties (i.e. concealed volumes).

The characters allowed in a directory or file name vary from system to system, as shown in the following table:

System	Character Set
UNIX	Any character except '/'.
DOS, OS/2, NT	Any character except the following: space " * + , . / : ; < = > ? [\] \t \n
MAC	Any character except ':'.
VMS	Only a-z, A-Z, 0-9, '-', '_', and '\$'.

Note that on UNIX, you can create a file named "*" or "?" or even " ", but then manipulating this file in the shell will not be easy. Also on Unix, "host:" is sometimes used as a prefix to specify a host name but this notation is used only by `rcp` and `mount`.

Finally, UNIX file names are case sensitive (Makefile and makefile can coexist in the same directory). The other systems are not.

Portability Information:

If you want your filenames to be as portable as possible, you should use the following guidelines:

- Do not use absolute filenames. Use relative filenames or names with an environment variable to define a root directory (i.e. \$ND_HOME/xxx).
- Use the most restrictive character set you will be porting to (VMS is the most restrictive).
- Use the most restrictive component lengths you will be porting to (DOS is the most restrictive).

Determining the Syntax of a Name:

It is not always possible to tell the syntax of a file name because there are many cases when names are ambiguous. For example, "c:main.c" is most likely a DOS file name, but it could also be a valid VMS name, or a Mac name, or even a Unix name. Unix is the most extreme case since any sequence of characters makes a valid Unix name (provided the name is embedded inside single quotes when used from command shells).

Therefore, we cannot provide a function which returns the syntax of a name, but we do provide a function which returns the MOST LIKELY syntax according to an ad-hoc algorithm (see below for NDFName::FindSyntax).

Syntax Conversions:

Converting file names is not completely straightforward. The main problem is that certain syntactic elements are only supported by some systems:

Element	Supported Systems
host	VMS
volume	DOS, MAC and VMS
default directory on volume	DOS and VMS
home directories	UNIX (VMS with SYS\$LOGIN)
version numbers	VMS

The other problems come from the fact that there might be non portable ways to fully specify the top directory of a path.

Also, there is very little chance that two systems have the same volume names, so we need a flexible scheme for volume name translation. We also provide flexible translations for host names.

Also, it would be nice to support embedded variables in paths, like \$ (ND_HOME)/bin/resed. VMS has a built-in support with logical names, other systems don't but some programs perform translation of environment variables embedded in file names (like make on UNIX). Environment variables are not supported by the MAC (except in MPW, otherwise resources replace them advantageously). We can use the MPW syntax to emulate environment variables and have a portable scheme even with environment variables.

In case the initial syntax contained some elements which can not be translated naturally in the target syntax, the ill-defined conversion is solved according to the following algorithm:

We first check whether the faulty component can be replaced by the value of an environment variable according to the following convention:

FNAME_WHOSTXVOLYDEF	W is a letter identifying the originating machine (U: UNIX, D: DOS, M:MAC, V:VMS). X is the host name, Y the volume name. The <code>_HOSTX</code> might be absent as well as the <code>_VOLY</code> . The optional DEF indicates that the default directory of the volume was specified.
or FNAMEWHOMEX	Where X is a user name or empty if the home directory of the current user was specified.
or FNAME_ROOT	when translating the root directory to the Macintosh, which does not have any syntax for it.

Examples:

```

UNIX  ~/dir1/dir2/file
VMS   NDFName::UHOME:[dir1.dir2]file
DOS   $ (NDFName::UHOME)\dir1\dir2\file
MAC   $ (NDFName::UHOME):dir1:dir2:file

DOS   a:\dir1\dir2\file
UNIX  $ (NDFName::DVOLa)/dir1/dir2/file
VMS   NDFName::DVOLA:[dir1.dir2]file
MAC   $ (NDFName::DVOLA):dir1:dir2:file

DOS   a:dir1\dir2\file
UNIX  $ (NDFName::DVOLADEF)/dir1/dir2/file
VMS   NDFName::DVOLADEF:[dir1.dir2]file
MAC   $ (NDFName::DVOLADEF):dir1:dir2:file

DOS   \dir1\dir2\file
UNIX  /dir1/dir2/file
VMS   [dir1.dir2]file
MAC   $ (NDFName::ROOT):dir1:dir2:file

DOS   dir1\dir2\file
UNIX  dir1/dir2/file
VMS   [.dir1.dir2]file
MAC   :dir1:dir2:file

```

If there is no such environment variable, we apply one of the following methods (according to parameters set by `NDFName::StxQueryForeignCvt`):

- We can leave the faulty component unchanged, eventually resulting in an invalid file name.
- Or we can skip the faulty component.
- Or we can transform it into something which is valid in the target syntax. For instance, a VMS host name can be translated to Unix as a directory name. The root directory can be translated to Macintosh as the name of the current volume.
- Or the conversion can abort and fail.

FNameBuf structure

One of the problems with this API is that we have to process strings and thus we should ideally use a variable string (`VStr` or `GStr`) data type to get rid of any limitation in string lengths. On the other hand, we have to explicitly create and destroy variable strings in C (C++ would help here) and also we have to be careful about error recovery if we use variable strings. Fortunately, file names never get too long in realistic cases so fixed

strings (Str) were chosen instead of variable strings for this API. The FNAME_MAXLEN constant defines the maximum length that can be considered reasonable for a file name. The idea is that you should use buffers of size FNAME_MAXLEN allocated on the stack when using this API. Then, once you have obtained the result that you want, you can store it in a variable string.

Current, Parent and Top directories

On DOS, the operating system maintains a separate current directory for every volume. On MAC, there is only one current path and when you switch from one volume to another, you end up on the root directory of the new volume. On UNIX, there is only one volume and only one global current directory. On VMS, there is only one current path and when you switch from one volume (or drive) to another, you may end up in a non-existent directory.

Summary

The FName class is only used for string manipulation of file names. Management of files (deleting, copying, moving, changing permissions, etc.) is done by the FMgr class, actual file I/O is done by the File class, and file selection windows are done by the FileW class.

See also

FMgr class, File class, FileW class.

Data Types

FNameBuf

File name storage type.

FNameBuf is the data type used for storing path names for processing by the FName class.

One of the problems with this API is that we have to process strings and thus we should ideally use a variable string (VStr or GStr) data type to get rid of any limitation in string lengths. On the other hand, we have to explicitly create and destroy variable strings in C (C++ would help here) and also we have to be careful about error recovery if we use variable strings. Fortunately, file names never get too long in realistic cases, so fixed strings (Str) were chosen instead of variable strings for this API. The FNAME_MAXLEN constant defines the maximum length that can be considered reasonable for a file name. The idea is that you should use buffers of size FNAME_MAXLEN allocated on the stack when using this API. Then, once you have obtained the result that you want, you can store it in a variable string.

See also

FNAME_MAXLEN, FNAME_STATUSCHARTRUNCATED

FNameCompSet

Data type for specifying file name components.

```
typedef UInt16   FNameCompSet;
```

FNameCompSet is the type used for specifying a set of file name component, such as the volume name or file extension.

See also

FNAME_COMP...

FNameParamsPtr**FNameParamsRec**

Data type for storing conversion parameters.

FNameParamsRec is the structure used for storing various parameters for the file syntax conversion process.

The fields of this structure are described below:

Type	Description
CurSyntax	The default naming syntax. By default, it is set to the system syntax.
SourceSyntaxes	The set of input syntaxes which are looked for by NDFName::FindSyntax or NDFName::Convert . It should always contain at least FNAME_STXMASKOF (CurSyntax). By default, it is set to FNAME_STXMASKALL .
TargetSyntaxes	The set of target syntaxes which are checked by NDFName::IsPortable . It should always contain at least FNAME_STXMASKOF (CurSyntax). By default, it is set to FNAME_STXMASKALL .
CvtEvaluate	If TRUE , NDFName::Convert automatically evaluates variable expressions (like "\$ (VAR)") and replaces them by their values or by some reasonable defaults. By default, it is set to BOOL_TRUE .
CvtMakeValid	If TRUE , NDFName::Convert eventually truncates and/or modifies some parts of the result name to make it valid in the target syntax. By default, it is set to BOOL_TRUE .

See also

FNameStxMaskVal, **NDFName::FindSyntax**, **NDFName::Convert**, **NDFName::IsPortable**

FNameStxMaskVal

Constants that identify particular system's file name syntaxes.

Some of the functions in the **FName** class refer to individual system syntaxes, and some refer to sets of syntaxes. The **FNameStxMaskVal** type is used to store a set of one or more syntax flags. **FNameStxMaskVal** is set by oring (or adding) one or more of the **FNAME_STXMASK...** values.

The **FNAME_STXMASK...** constants are used for setting the values of variables of type **FNameStxMaskVal**. These constants can be ored (or added) together to create a set of system syntaxes. This set is used for identifying the possible syntaxes of a file name.

The syntaxes are described below:

Constant	Description
<code>FNAME_STXMASKDOS</code>	The name could be a DOS, OS/2 or NT path name.
<code>FNAME_STXMASKMAC</code>	The name could be a Macintosh path name.
<code>FNAME_STXMASKUNIX</code>	The name could be a Unix path name.
<code>FNAME_STXMASKVMS</code>	The name could be a VMS path name.
<code>FNAME_STXMASKW32</code>	Name could be a Windows 95 or Windows NT path name
<code>FNAME_STXMASKALL</code>	The name could be in any syntax.

See also

`FNameStxEnum`, `NDFName::StxGetName`

Enumerated Types

`FNameStatusEnum`

Enumerated type that indicates the status of the previous conversion

`FNameStatusEnum` is the enumerated type for indicating the status of the previous conversion. The various status types are described below:

Type	Description
<code>FNAME_STATUSOK</code>	Conversion has worked normally without any loss of information.
<code>FNAME_STATUSCHARTRUNCATED</code>	Result string has been truncated because it was over <code>FNAME_MAXLEN</code> .
<code>FNAME_STATUSCOMPTRUNCATED</code>	Some components have been dropped because there were more than 32 components.
<code>FNAME_STATUSCHARSKIPPED</code>	Some characters have been skipped because otherwise a component would be too long.
<code>FNAME_STATUSCOMP_SKIPPED</code>	Some components have been skipped because they could not be translated.
<code>FNAME_STATUSCHARALTERED</code>	Some characters have been replaced by '_'. These characters would not have been valid.
<code>FNAME_STATUSCOMPALTERED</code>	Some components have been changed into components of different types. These components were not supported in the target syntax.

See also

`NDFName::GetStatus`, `NDFName::SetStatus`, `NDFName::StatusGetMsg`

`FNameStxEnum`

Enumerated type that identifies a particular system's file name syntax.

The functions in the `FName` class refer to individual system syntaxes. For example, a function may be given an individual syntax, and try to convert a file name into that syntax. The `FNameStxEnum` enumerated values are used for specifying individual syntaxes

The syntaxes are described below:

Type	Description
<code>FNAME_STXBAD</code>	Illegal or ambiguous syntax.
<code>FNAME_STXDOS</code>	The syntax in DOS, OS/2 and NT.
<code>FNAME_STXMAC</code>	The syntax in Macintosh.
<code>FNAME_STXUNIX</code>	The syntax in Unix.
<code>FNAME_STXVMS</code>	The syntax in VMS.
<code>FNAME_STXW32</code>	Syntax in Windows 95 or NT

See also

`NDFName::StxGetName`, `FNameStxMaskVal`

FAIL...

Errors signaled by this class.

The `FNAME_FAIL...` constants represent the errors signaled by this class.

The error codes are described below:

Constant	Description
<code>FNAME_FAILUNIX</code>	Invalid UNIX filename syntax.
<code>FNAME_FAILDOS</code>	Invalid DOS filename syntax.
<code>FNAME_FAILMAC</code>	Invalid Mac filename syntax.
<code>FNAME_FAILVMS</code>	Invalid VMS filename syntax.
<code>FNAME_FAILVAR</code>	Invalid variable substitution syntax.
<code>FNAME_FAILAMBIGUOUS</code>	Ambiguous syntax.
<code>FNAME_FAILDOSLEN</code>	File name exceeds DOS length.
<code>FNAME_FAILTOODEEP</code>	Too many levels of directories.
<code>FNAME_FAILVMSCLOSEBKT</code>	Missing ']' character in VMS filename.
<code>FNAME_FAILNOTPATH</code>	Directory name not in its path syntax.
<code>FNAME_FAILSPLITPATH</code>	Failed to split a path name.
<code>FNAME_FAILFILETOPATH</code>	Failed to convert a directory name into its path syntax.

See also

`FNameStatusEnum`, `NDFName::GetStatus`, `NDFName::SetStatus`, `NDFName::StatusGetMsg`

MAXLEN

File name buffer size.

`FNAME_MAXLEN` is the string length of the `FNameBuf` string.

One of the problems with this API is that we have to process strings and thus we should ideally use a variable string (`VStr` or `GStr`) data type to get rid of any limitation in string lengths. On the other hand, we have to explicitly create and destroy variable strings in C (C++ would help here) and also we have to be careful about error recovery if we use variable strings. Fortunately, file names never get too long in realistic cases so fixed

strings (Str) were chosen instead of variable strings for this API. The FNAME_MAXLEN constant defines the maximum length that can be considered reasonable for a file name. The idea is that you should use buffers of size FNAME_MAXLEN allocated on the stack when using this API. Then, once you have obtained the result that you want, you can store it in a variable string.

FNameCompSetEnum

Defines the various components of a file name.

The FNAME_COMP... constants define the bits that can be ored (or added) together to define a set of file name components. A set of component bits is stored in variable of type FNameCompSet.

Usually, the two components which are extracted from a file name are the pathname (host + volume + directory) and the filename (base + ext + vers), which is why we introduce the FNAME_COMPPATH and FNAME_COMPFILE combinations.

File name components are defined so that the full file name can be reobtained by concatenating all its components. Thus, the extension part includes the "." which is normally between the base and extension.

The component bit constants are described below:

Identifier	Description
FNAME_COMPHOST	A single component bit representing the host.
FNAME_COMPVOL	A single component bit representing the volume.
FNAME_COMPDIR	A single component bit representing the directory (or directories).
FNAME_COMPBASE	A single component bit representing the base file name.
FNAME_COMPEXT	A single component bit representing the extension.
FNAME_COMPVERS	A single component bit representing the version.
FNAME_COMPVERSBIT	
FNAME_COMPPATH	A component bit set representing the path. The path consists of the host, the volume and the directory.
FNAME_COMPFILE	A component bit set representing the file name. The file name consists of the base name, the extension and the version.
FNAME_COMPFILENOVERS	A component bit set representing the file name without the version component. It consists of the base name and the extension.
FNAME_COMPALL	A component bit set representing the entire pathed file name. It consists of the full path and the file name.

See also

NDFNameCompSet, NDFName::GetCompSet, NDFName::ReduceComps, NDFName::QueryComps

File Name Syntax

StxGetName

Returns the name of the specified syntax.

status CStr NDFName::StxGetName (FNameStxEnum *syntax*);

NDFName::StxGetName returns the name of the specified syntax.

GetSysSyntax

Determine the syntax of the native system.

status FNameStxEnum NDFName::GetSysSyntax (void);

NDFName::GetSysSyntax returns the syntax of the system on which the program is currently running. This is referred to as the native file system.

GetCurSyntax

Determine the current syntax.

status FNameStxEnum NDFName::GetCurSyntax (void);

NDFName::GetCurSyntax returns the current syntax. The current syntax is used by functions such as NDFName::Convert.

SetCurSyntax

Set a particular syntax as the current syntax.

status void NDFName::SetCurSyntax (FNameStxEnum *syntax*);

NDFName::SetCurSyntax sets the given syntax to be the current syntax. The current syntax is used by functions such as NDFName::Convert.

QueryCurParams

Query the current syntax conversion parameters.

status void NDFName::QueryCurParams (FNameParamsPtr *params*);

NDFName::QueryCurParams queries the currently set syntax conversion parameters and stores the results in *params*.

SetCurParams

Set the current syntax conversion parameters.

status void NDFName::SetCurParams (FNameParamsCPtr *params*);

NDFName::SetCurParams sets the current syntax conversion parameters to the ones in the given *params* structure. These parameters are used by the syntax conversion functions.

ResetCurParams

Reset the current syntax conversion parameters to the default parameters.

status void NDFName::ResetCurParams (void);

NDFName::ResetCurParams resets the current syntax conversion parameters to the default parameters. These parameters are used by the syntax conversion functions.

Find Path Name Syntax

FindSyntax

Returns the most likely syntax for the given name.

status FNameStxEnum NDFName::FindSyntax (CStr name);

NDFName::FindNameSyntax returns the most likely syntax for the given file name.

Since the syntax of file names is sometimes ambiguous, this function uses the following algorithm:

- First, it considers only the syntaxes specified in SourceSyntaxes.
- If SourceSyntaxes contains only one syntax, the choice is easy.
- If the name contains spaces, it is most likely a Mac name.
- Else if the name starts with a '.', it is most likely a Mac name.
- Else if the name starts with a '~', it is most likely a Unix name.
- Else if the name contains '/', it is most likely a Unix name.
- Else if the name contains '\', it is most likely a DOS name.
- Else if the name contains '::', it is most likely a VMS name.
- Else if the name contains ';', it is most likely a VMS name.
- Else if the name contains '[' or ']', it is most likely a VMS name.
- Else if the name contains several ':', it is most likely a Mac name.
- Else if the name contains one or several '\$', then ':' and then no other ':', it is most likely a VMS name.
- Else if the first character is a letter and the second one is ':' and then no other ':', it is most likely a DOS name.
- Else if the name contains one ':', it is most likely a Mac name.
- Else if the name ends with a '.', it is most likely a VMS name.
- Else if the first character is '.' and the second letter is a letter, then it is most likely a Unix name.
- Otherwise, if there is still ambiguity between two or more syntaxes, NDFName::FindSyntax returns the first one (looking first for Unix, then Dos, then Mac, then VMS).

See also

NDFNameParamsRec, FNameStxEnum

Checking Path Name Validity

IsValidIn

Determine whether a file name is valid in a given file system syntax.

status BoolEnum NDFName::IsValidIn (CStr name, FNameStxEnum syntax);

NDFName::IsValidIn returns `BOOL_TRUE` if the given file name is valid in the specified syntax. If the name still contains variable expressions (like “\$ (VAR)”), these expressions are not evaluated.

IsValid

Determine whether a file name is valid in the current syntax.

status BoolEnum NDFName::IsValid (CStr name);

NDFName::IsValid returns `BOOL_TRUE` if the given file name is valid in the current syntax. If the name still contains variable expressions (like “\$ (VAR)”), these expressions are not evaluated.

MakeValidIn

Modify a name to make it valid in a specified syntax

status void NDFName::MakeValidIn (FNameBuf name, FNameStxEnum syntax);

NDFName::MakeValidIn modifies the given name to make it valid in the specified syntax

MakeValid

Modify a name to make it valid in the current syntax

status void NDFName::MakeValid (FNameBuf name);

NDFName::MakeValid modifies the given name to make it valid in the current syntax

Evaluating Variable Expressions

EvaluateIn

Replace each variable expression by its value, using the specified syntax.

status void NDFName::EvaluateIn (FNameBuf name, FNameStxEnum syntax);

NDFName::EvaluateIn replaces each variable expression (such as \$ (VAR)) in the given name by its value. The format of these expressions depends on the specified syntax:

- On DOS: \$ (VAR) or \${VAR}
- On Mac: \$ (VAR), \${VAR} or {VAR}
- On Unix: \$ (VAR), \${VAR} or \$VAR. ~ is also evaluated.
- On VMS: \$ (VAR) or \${VAR}

See also

NDFNameStxEnum, NDFName::Evaluate

Evaluate

Replace each variable expression by its value, using the current syntax.

status void NDFName::Evaluate (FNameBuf name);

NDFName::Evaluate replaces each variable expression (such as \$ (VAR)) in the given name by its value. The format of these expressions depends on the current syntax:

- On DOS: \$ (VAR) or \${VAR}
- On Mac: \$ (VAR), \${VAR} or {VAR}
- On Unix: \$ (VAR), \${VAR} or \$VAR. ~ is also evaluated.
- On VMS: \$ (VAR) or \${VAR}

Conversion between Syntaxes

Convert

Determine the syntax of a name and convert it to the current syntax.

status void NDFName::Convert (CStr source, FNameBuf dest);

NDFName::Convert determines the source name syntax and converts it to the current syntax. The result is stored in dest.

This procedure calls NDFName::Evaluate and NDFName::MakeValid if the flags CvtEvaluate and CvtMakeValid are TRUE.

If CvtMakeValid is set to TRUE, some characters or components might be altered or truncated so that the final result would be valid in the target syntax. This condition can be checked with NDFName::GetStatus.

See also

FNameBuf, NDFName::Evaluate, NDFName::MakeValid,
NDFName::GetStatus, NDFName::SetCurSyntax,
NDFName::ConvertFromTo, NDFName::ConvertInPlace

ConvertFromTo

Convert a name from one given syntax to another.

**status void NDFName::ConvertFromTo (CStr source, FNameBuf dest,
FNameStxEnum syntax1, FNameStxEnum syntax2);**

NDFName::ConvertFromTo converts the given source name from syntax1 to syntax2. The result is stored in dest.

This procedure calls NDFName::Evaluate and NDFName::MakeValid if the flags CvtEvaluate and CvtMakeValid are TRUE.

If CvtMakeValid is set to TRUE, some characters or components might be altered or truncated so that the final result would be valid in the target syntax. This condition can be checked with NDFName::GetStatus.

See also

NDFNameStxEnum, FNameBuf, NDFName::Evaluate,
NDFName::MakeValid, NDFName::GetStatus, NDFName::SetCurSyntax,
NDFName::Convert, NDFName::ConvertInPlace

ConvertInPlace

Determine the syntax of a name and convert it to the current syntax.

status void NDFName::ConvertInPlace (FNameBuf name);

NDFName::ConvertInPlace determines the file name syntax and converts it to the current syntax. The input name string is replaced by the result of the conversion.

This procedure calls NDFName::Evaluate and NDFName::MakeValid if the flags CvtEvaluate and CvtMakeValid are TRUE.

If CvtMakeValid is set to TRUE, some characters or components might be altered or truncated so that the final result would be valid in the target syntax. This condition can be checked with NDFName::GetStatus.

See also

NDFNameBuf, NDFName::Evaluate, NDFName::MakeValid,
NDFName::GetStatus, NDFName::SetCurSyntax,
NDFName::ConvertFromTo, NDFName::Convert

IsConvertible

Determine if a name can be completely converted.

status BoolEnum NDFName::IsConvertible (CStr name);

NDFName::IsConvertible returns BOOL_TRUE if name can be converted without any loss of information (no truncation nor alteration).

See also

NDFName::IsPortable, NDFName::Convert

IsPortable

Determine if a name can be completely converted to all target syntaxes.

status BoolEnum NDFName::IsPortable (CStr name);

NDFName::IsPortable returns BOOL_TRUE if name can be converted to all syntaxes in TargetSyntaxes with no loss of information (no truncation nor alteration).

See also

NDFNameParamsRec, NDFName::SetCurParams,
NDFName::IsConvertible, NDFName::Convert

Conversion Status

GetStatus

Get the status of the most recent conversion.

status FNameStatusEnum NDFName::GetStatus (void);

NDFName::GetStatus returns the status of the most recent file name conversion.

See also

FNameStatusEnum, NDFName::SetStatus, NDFName::StatusGetMsg

SetStatus

Set the status flag to the given value.

status void NDFName::SetStatus (FNameStatusEnum status);

NDFName::SetStatus sets the status flag to the given status value.

See also

NDFNameStatusEnum, NDFName::GetStatus, NDFName::StatusGetMsg

StatusGetMsg

Get the text description of the given status value.

status CStr NDFName::StatusGetMsg (FNameStatusEnum status);

NDFName::StatusGetMsg returns the text description of the given status value.

See also

NDFNameStatusEnum, NDFName::GetStatus, NDFName::SetStatus

Extracting File Components

Usually, the two components which are extracted from a file name are the pathname (host + volume + directory) and the filename (base + ext + vers), which is why we introduce the FNAME_COMPPATH and FNAME_COMPFILE combinations.

Note: File name components are defined so that the full file name can be reobtained by concatenating all its components. Thus, the extension part includes the “.” which is normally between the base and extension. The name is assumed to be in the current syntax.

GetCompSet

Return the set of components which are present in a file name.

status FNameCompSet NDFName::GetCompSet (CStr name);

NDFName::GetCompSet returns the set of components which are present in the file name string name. The name is assumed to be in the current syntax.

See also

NDFNameCompSet, FNAME_COMP..., NDFName::ReduceComps, NDFName::QueryComps

QueryComps

Extract specified file name components and copy to a string.

status void NDFName::QueryComps (CStr name, FNameCompSet components, Str output);

NDFName::QueryComps extracts the specified set of components from the given file name and puts the result into the output string. The name is assumed to be in the current syntax.

See also

NDFNameCompSet, FNAME_COMP..., NDFName::GetCompSet, NDFName::ReduceComps

ReduceComps

Reduce a file name to a specified set of components.

status void NDFName::ReduceComps (Str name, FNameCompSet components);

NDFName::ReduceComps reduces the file name string to the specified set of components. The name is assumed to be in the current syntax.

See also

NDFNameCompSet, FNAME_COMP..., NDFName::GetCompSet, NDFName::QueryComps

Directories Specified as Paths or as Files

IsDirAsFile

Determine if a directory is specified as a directory name or as a file name.

status BoolEnum NDFName::IsDirAsFile (CStr directory);

A Directory can be specified in one of two ways: Either as a path (to which a file name can be appended), or as a final file name. For example, in UNIX, /dir1/dir2 represents a complete file name, and /dir1/dir2/ represents a directory component to which a file name can be appended. In VMS, [dir1]dir2.dir represents a complete file name, and [dir1.dir2] represents the directory component of a complete name. NDFName::IsDirAsFile returns BOOL_TRUE if the given directory name is represented as a file name, and it returns BOOL_FALSE if the given directory name is specified as a directory component. The directory is assumed to already be in the current syntax.

CvtDirPathToFile

Convert a directory string from path syntax to file syntax.

status void NDFName::CvtDirPathToFile (CStr path, FNameBuf file);

A directory can be specified in one of two ways: Either as a path (to which a file name can be appended), or as a final file name. For example, in UNIX, /dir1/dir2 represents a complete file name, and /dir1/dir2/ represents a directory component to which a file name can be appended. In VMS, [dir1]dir2.dir represents a complete file name, and [dir1.dir2] represents the directory component of a complete name. NDFName::CvtDirPathToFile converts a directory string from path syntax to file syntax. It generates an error if the initial syntax is not a path syntax. The path is assumed to already be in the current syntax.

CvtDirFileToPath

Convert a directory string from file syntax to path syntax.

status void NDFName::CvtDirFileToPath (CStr *file*, FNameBuf *path*);

A Directory can be specified in one of two ways: Either as a path (to which a file name can be appended), or as a final file name. For example, in UNIX, /dir1/dir2 represents a complete file name, and /dir1/dir2/ represents a directory component to which a file name can be appended. In VMS, [dir1]dir2.dir represents a complete file name, and [dir1.dir2] represents the directory component of a complete name. NDFName::CvtDirFileToPath converts a directory string from file syntax to path syntax. It generates an error if the initial syntax is not a file syntax. The file is assumed to already be in the current syntax.

SplitFile

Split a file name into path and file components.

status void NDFName::SplitFile (CStr *name*, FNameBuf *path*, FNameBuf *file*);

NDFName::SplitFile splits the file name string into path and file component set strings. The path string consists of the host, the volume and the directory. The file string consists of the base name, the extension and the version. NULL may be passed in to either the path or file argument if the particular string result is not desired. The name is assumed to already be in the current syntax.

On UNIX, /dir/foo would be split as “/dir/” and “foo”. On VMS, [dir]foo would be split as “[dir]” and “foo”.

SplitPath

Split a path into parent path and child subdirectory components.

status BoolEnum NDFName::SplitPath (CStr *name*, FNameBuf *path*, FNameBuf *subdirectory*);

NDFName::SplitPath splits the path name string into parent path and child subdirectory component strings. NULL may be passed in to either the path or subdirectory argument if the particular string result is not desired. The name is assumed to already be in the current syntax. This function returns `BOOL_FALSE` if the path cannot be split (because it is already at the top level).

On UNIX, /a/b/ would be split as “/a/” and “b”. On VMS, [a.b] would be split as “[a]” and “b”.

MergeFile

Merge a path component and a file component into a full file name.

status void NDFName::MergeFile (CStr *path*, CStr *file*, FNameBuf *name*);

NDFName::MergeFile merges a path component string and a file component string into a concatenated file name string. The path string consists of the host, the volume and the directory. The file string consists of the base name, the extension and the version.

Although current directory ‘.’ or parent directory ‘..’ designations can be used in the path component, NDFName::MergeFile removes these

designations from the resultant file name in order to simplify the full file name. For example, on Unix, a directory path component “a/b/” and a file name component “./foo” return the resultant file name “a/b/foo”. Whereas, “a/b/” and “../foo” return the resultant file name “a/foo”.

MergePath

Merge a path and a subdirectory into a full path for the subdirectory.

status void NDFName::MergePath (CStr path, CStr subdirectory, FNameBuf name);

NDFName::MergePath merges a parent directory path component string and a subdirectory string into a concatenated path name string for the subdirectory.

Although current directory ‘.’ or parent directory ‘..’ designations can be used in the path and subdirectory components, NDFName::MergePath removes these designations from the resultant path name in order to simplify the full path name. For example, on Unix, a directory path component “a/b/” and a subdirectory component “./sub” return the resultant path name “a/b/sub/”. Whereas, “a/b/” and “../sub” return the resultant path name “a/sub/”.

Top Directory

TopDirStr, QueryTopDir, IsTopDir

Returns the string representation of the top directory of the current volume.

status CStr NDFName::TopDirStr (void);

NDFName::TopDirStr returns the string representation of the top directory of the current volume.

The strings returned for the various systems are shown below:

System	String
PC	“\\”
Macintosh	“”
Unix	“/”
VMS	“[000000]”

QueryTopDir

Queries the current top directory.

static void NDFName::QueryTopDir (FNameBuf fnamebuf);

IsTopDir

Returns whether a directory path name is at the top level.

static BoolEnum NDFName::IsTopDir (CStr name);

Current Volume / Current Directory

On DOS, the operating system maintains a separate current directory for every volume. On MAC, there is only one current path and when you switch from one volume to another, you end up on the root directory of the new volume. On UNIX, there is only one volume and only one global current directory. On VMS, there is only one current path and when you switch from one volume (or drive) to another, you may end up in a non-existent directory.

QueryCurDir

Query the full current directory string.

status void NDFName::QueryCurDir (FNameBuf directory);

NDFName::QueryCurDir queries the full current directory, and passes the string back in the directory argument. This string consists of the current volume and the current directory for this volume.

See also

NDFName::QueryCurVolume, NDFName::VolumeQueryCurDir,
NDFName::VolumeSetCurDir, NDFName::SetCurDir,
NDFName::QueryHomeDir

SetCurDir

Set the current directory.

status void NDFName::SetCurDir (CStr directory);

NDFName::SetCurDir sets the current directory to the given directory name. If a volume is specified in the directory string, the current volume is changed if necessary, and the current directory for this volume is changed. If no volume is specified, the current volume is assumed.

See also

NDFName::QueryCurVolume, NDFName::VolumeQueryCurDir,
NDFName::VolumeSetCurDir, NDFName::QueryCurDir,
NDFName::QueryHomeDir

QueryCurVolume

Query the current volume string.

status void NDFName::QueryCurVolume (FNameBuf volume);

NDFName::QueryCurVolume queries the current volume string. It copies the string into the volume buffer argument.

VolumeQueryCurDir

Query the current directory of a given volume.

status void NDFName::VolumeQueryCurDir (CStr volume, FNameBuf directory);

NDFName::VolumeQueryCurDir queries the current directory of the given volume. The volume name is passed as the first argument, and the directory string is returned in the second argument.

VolumeSetCurDir

Set the current directory of a given volume.

status void NDFName::VolumeSetCurDir (CStr volume, CStr directory);

NDFName::VolumeSetCurDir sets the current directory of the given volume to the given directory name. The volume name is passed as the first argument, and the directory string is passed as the second argument.

CurDirStr

Returns the string representation of the current directory.

status CStr NDFName::CurDirStr (void);

NDFName::CurDirStr returns the string representation of the current directory. The returned string is ":" on Macintosh, "[]" on VMS, and "." on all other systems.

Parent Directory

QueryParentDir

Query the parent directory string of the current directory.

status void NDFName::QueryParentDir (FNameBuf directory);

NDFName::QueryParentDir queries the parent directory, and passes the string back in the directory argument. This string consists of the parent directory of the current directory.

See also

NDFName::QueryCurVolume, NDFName::VolumeQueryCurDir,
NDFName::VolumeSetCurDir, NDFName::SetCurDir,
NDFName::QueryHomeDir

ParentDirStr

Returns the string representation of the parent directory.

status CStr NDFName::ParentDirStr (void);

NDFName::ParentDirStr returns the string representation of the parent of the current directory. The returned string is ".." on Unix and PC, [-] on VMS, and "::" on Macintosh.

DirQueryParent

Returns the parent directory of the specified directory.

status void NDFName::DirQueryParent (CStr *dir*, FNameBuf *parent*);

NDFName::DirQueryParent queries the parent directory, and passes the string back in the parent argument. This string consists of the parent directory of the specified directory.

Home Directory

HomeDirStr

Returns the string representation of the top directory of the system.

status CStr NDFName::HomeDirStr (void);

NDFName::HomeDirStr returns the string representation of the home directory of the system. The returned string is “~” on UNIX, “SYS\$LOGIN:” on VMS, and “” on all other systems.

QueryHomeDir

Query the home directory of the current user.

status void NDFName::QueryHomeDir (FNameBuf *home*);

NDFName::QueryHomeDir queries the home directory of the current user, and passes it back in the home string. This string consists of a volume and a directory.

See also

NDFName::QueryCurVolume, NDFName::VolumeQueryCurDir,
NDFName::VolumeSetCurDir, NDFName::QueryCurDir,
NDFName::SetCurDir, NDFName::QueryTopDir,
NDFName::QueryParentDir

QueryTopDir

Query the top directory of the current volume.

status void NDFName::QueryTopDir (FNameBuf *top*);

NDFName::QueryTopDir queries the top directory of the current volume, and passes it back in the top string. This string consists of a volume and a directory.

See also

NDFName::QueryCurVolume, NDFName::VolumeQueryCurDir,
NDFName::VolumeSetCurDir, NDFName::QueryCurDir,
NDFName::SetCurDir, NDFName::QueryHomeDir,
NDFName::QueryParentDir

QueryParentDir

Query the parent directory of the current directory.

status void NDFName::QueryParentDir (FNameBuf parent);

NDFName::QueryParentDir queries the parent directory of the current directory, and passes it back in the parent string. This string consists of a volume and a directory.

See also

NDFName::QueryCurVolume, NDFName::VolumeQueryCurDir,
NDFName::VolumeSetCurDir, NDFName::QueryCurDir,
NDFName::SetCurDir, NDFName::QueryHomeDir,
NDFName::QueryTopDir

Absolute / Relative Parts

IsTopDir

Determine whether a directory path name is at the top level.

status BoolEnum NDFName::IsTopDir (CStr name);

NDFName::IsTopDir returns `BOOL_TRUE` if the given directory path name is at the top level, and it returns `BOOL_FALSE` otherwise.

IsAbsolute

Determine if a file name is specified as absolute or as relative.

status BoolEnum NDFName::IsAbsolute (CStr name);

NDFName::IsAbsolute returns `BOOL_TRUE` if the given file name is an absolute file name, and it returns `BOOL_FALSE` if the given file name is a relative file name.

CvtToAbsolute

Convert a file name to an absolute file name.

status void NDFName::CvtToAbsolute (CStr name, FNameBuf absolute);

NDFName::CvtToAbsolute converts the given file name to an absolute file name.

Comparing File Names

Cmp

Compare two file names.

status CmpEnum NDFName::Cmp (CStr name1, CStr name2);

NDFName::Cmp compares the two file names `name1` and `name2`. The comparison is case sensitive if the current syntax is UNIX.

Equal

Compare two file names.

status BoolEnum NDFName::Equal (CStr name1, CStr name2);

NDFName::Equal compares the two file names name1 and name2 and returns BOOL_TRUE if file names identify the same file.

Generating Temporary and Backup File Names

GetTmpPath

Return the path where temporary file names are created.

status CStr NDFName::GetTmpPath (void);

NDFName::GetTmpPath returns the path where temporary file names are created.

SetTmpPath

Change the path where temporary file names will be created.

status void NDFName::SetTmpPath (CStr directory);

NDFName::SetTmpPath changes the path where temporary file names will be created to the given directory. This call fails if directory does not exist.

SysTmpPath

Return the native system's default path for temporary files.

status CStr NDFName::SysTmpPath (void);

NDFName::SysTmpPath returns the native system's default path for temporary files. The returned path is "/tmp/" on AIX; "/var/tmp/" on Sony and Univel; "/usr/tmp/" on HPUX, SunOS, Mips, Sco, SG, and Ultrix; and "SYSSCRATCH:" on VMS.

MakeTmpFileName

Generate a temporary file name.

status void NDFName::MakeTmpFileName (CStr prefix, FNameBuf buffer);

NDFName::MakeTmpFileName generates a temporary file name. The prefix argument is the user's desired prefix for the generated file name. The prefix is truncated to its first five characters. The buffer argument receives the result. The result is based on NDFName::GetTmpPath, on the prefix, and on a number generated by the system library.

MakeBackupName

Generate a name for a backup file.

status void NDFName::MakeBackupName (CStr filename, FNameBuf backupname);

NDFName::MakeBackupName generates a backupname backup file name for the given input filename.

The Hash class implements a general purpose hash table manager.

Overview

Hash is totally portable and has built-in optimizations for a certain number of standard key types.

A hash table is an object which keeps track of associations between keys and data through a hash function which converts the key into an index in an array of bins. When a pair (key, value) is inserted in the hash table, the hash function is applied to the key and an entry containing (key, value) is inserted in the corresponding bin. The best performance will be obtained by the use of a hash function which is fast enough and spreads the indexes across the whole set of bins. When two keys give the same index, the entries are double-linked together, thus allowing for easier deletion. See Knuth Vol 3, p 514.

Open Interface uses a default hash function that gives reasonable results for strings, Int32s and (far) pointers. In the future, specialized hashing functions will also be implemented.

Of course when the hash table is created, programmers can install their own hash function as well as their own comparison function. This could be necessary in the case where a set of keys is in use for which specialized and very fast hash and comparison functions can be defined.

The hash tables that can be created in this class can grow automatically when the average number of entries in the bins of the table becomes greater than a certain user-defined threshold. This rehashing mechanism allows the creation of hash tables that automatically adapt themselves to the number of entries (i.e. of keys) so that linear searches through the entries in one bin are kept under a certain limit.

Data Structures

NDHashInfo

Structure defining the members of a hash table.

Identifier	Description
CompareProc	Comparison procedure
HashProc	Hashing procedure
CloneDataProc	Entry data cloning procedure. If set to NULL, no cloning is carried out
DisposeDataProc	Data disposing procedure. If set to NULL, no disposing is carried out. Used in conjunction with HashCloneDataProc

<code>CloneKeyProc</code>	Entry key cloning procedure. If set to NULL, no cloning is carried out
<code>DisposeKeyProc</code>	Key disposing procedure. If set to NULL, no disposing is carried out. Used in conjunction with <code>HashCloneKeyProc</code>
<code>NumBins</code>	Initial number of bins. The number of bins can perfectly grow while the hash table is being used.
<code>MaxEntriesPerBin</code>	Maximum average number of entries per bin allowed before requesting an increase in the number of bins.
<code>GrowPercBins</code>	Percentage of <code>NumBins</code> that are allocated when <code>MaxEntriesPerBin</code> is reached.

Constructors and Destructor

Constructors

`NDHash::NDHash(void);`

Default hash table construction

`NDHash::NDHash(HashInfoCPtr info);`

Constructs the hash table with the information obtained from 'info'.

Destructor

`void NDHash::~~NDHash(void);`

Default hash table destruction.

Convenience Functions

Resetting a Hash Table

Reset

`void NDHash::Reset(void);`

Resets the contents of the hash table to the defaults as when created.

Creating and Disposing Hash Tables

Defining a Hash Table

The following parameters should be specified when creating an instance of a hash table:

CompareProc

`BoolEnum NDHash::CompareProc(HashKeyVal key1, HashKeyVal key2);`

Procedure that compares two entries. The programmer can install one of the comparison procedures provided by Open Interface.

```
CmpEnum result = (*HashCompareProc)(key1, key2);
```

HashProc**HashLenVal NDHash::HashProc(HashKeyVal arg1, HashLenVal arg2);**

Procedure that returns the result of the hashing operation on its argument. The programmer can install one of the hash procedures provided by Open Interface. Of course, the hash procedure should in general be related to the comparison proc.

```
HashLenVal val = (*HashProc)(key, numBins);
```

DataCloneProc**HashDataVal NDHash::DataCloneProc(HashDataVal data);**

Procedure returning a clone of the data of entry. The purpose is to allow to use the hash table to store information and not only refer to information stored elsewhere.

DataDisposeProc**void NDHash::DataDisposeProc(HashDataVal data);**

Procedure used to dispose data stored in the hash table for an entry (created through cloning).

KeyCloneProc**HashKeyVal NDHash::KeyCloneProc(HashKeyVal key);**

Procedure returning a clone of the data of entry. The purpose is to allow to use the hash table to store information and not only refer to information stored elsewhere.

KeyDisposeProc**void NDHash::KeyDisposeProc(HashKeyVal key);**

Procedure used to dispose data stored in the hash table for an entry (created through cloning).

Querying the Hash Table Information

QueryDefInfo**static void NDHash::QueryDefInfo(HashInfoPtr hashInfo);**

Fills 'hashInfo' with the default settings for a hash table.

GetDefIntInfo**static HashInfoCPtr NDHash::GetDefIntInfo(void);**

Returns the default settings for a hash table with integer keys.

GetDefPtrInfo**static HashInfoCPtr NDHash::GetDefPtrInfo(void);**

Returns the default settings for a hash table with pointer keys.

GetDefStrInfo

```
static HashInfoCPtr NDHash::GetDefStrInfo(void);  
static HashInfoCPtr NDHash::GetDefIStrInfo(void);
```

Returns the default settings for a hash table with string keys.

GetDefStrKeyClonedInfo

```
static HashInfoCPtr NDHash::GetDefStrKeyClonedInfo(void);
```

Returns the default settings for a hash table with cloned string keys.

QueryInfo

```
void NDHash::QueryInfo(HashInfoPtr hashInfo);
```

Fills 'hashInfo' with the values that were used to define the hash table.

Using Hash Tables

Using a created hash table is extremely simple. The program will start by inserting entries into the hash table, then getting the value associated to given keys in the hash table, eventually extracting them.

The following API provides support for these three operations.

Insert

```
void NDHash::Insert(HashKeyVal key, HashDataVal value);
```

Looks for an entry corresponding to 'key'. If there is one, it updates its contents so that it holds 'value', otherwise it adds one.

Extract

```
BoolEnum NDHash::Extract(HashKeyVal key, HashDataValPtr valPtr);
```

Looks for an entry corresponding to 'key'. If it finds it, it extracts the entry and updates valPtr with the value that was extracted.

If not, it sets *valPtr to NULL.

It returns `BOOL_TRUE` if an entry was extracted, `BOOL_FALSE` otherwise.

Note: If valPtr is NULL, it won't attempt to use it.

Lookup

```
BoolEnum NDHash::Lookup(HashKeyVal key, HashDataValPtr valPtr);
```

Looks for an entry corresponding to 'key'. If it finds it, it updates valPtr with the corresponding value. Otherwise, it sets *valPtr to NULL. It returns `BOOL_TRUE` if an entry was found, `BOOL_FALSE` otherwise.

Note: If valPtr is NULL, it won't attempt to use it.

Perform An Action On All The Entries

Type of the procedure that will be iterated on all entries in the hash table. It should return `PERF_CONTINUE` if the iteration should go on or `PERF_STOP` if it should stop.

```
PerfEnum HashPerfProc (HashCPtr hash, HashKeyVal key, HashDataVal keyValue,  
ClientPtr clientData);
```

```
Perf
```

```
PerfEnum NDHash::Perf(HashPerfProc perfProc, ClientPtr data);
```

Triggers the iteration of 'perfProc' for all the entries in the table. 'data' is a ClientPtr passed as last argument for each invocation of 'perfProc'.

Default Methods

Default Hashing

```
static HashLenVal NDHash::DefHashInt(HashKeyVal key, HashLenVal mod);
```

```
static HashLenVal NDHash::DefHashPtr(HashKeyVal key, HashLenVal mod);
```

```
static HashLenVal NDHash::DefHashStr(HashKeyVal key, HashLenVal mod);
```

```
static HashLenVal NDHash::DefHashIStr(HashKeyVal key, HashLenVal mod);
```

Returns the bin index computed by the default hashing procedure provided by Open Interface.

Default Comparison

```
static BoolEnum NDHash::DefCompareInt(HashKeyVal key1, HashKeyVal key2);
```

```
static BoolEnum NDHash::DefComparePtr(HashKeyVal key1, HashKeyVal key2);
```

```
static BoolEnum NDHash::DefCompareStr(HashKeyVal key1, HashKeyVal key2);
```

```
static BoolEnum NDHash::DefCompareIStr(HashKeyVal key1, HashKeyVal key2);
```

Returns the result of the default Open Interface comparison of key1 and key2.

Default String Cloning

```
static HashKeyVal NDHash::DefStrKeyClone(HashKeyVal key);
```

```
static void NDHash::DefStrKeyDispose(HashKeyVal key);
```

Hash Table Entries

The following API is provided to users who want to improve performances in some special cases.

AddGetEntry

```
HashEntryPtr NDHash::AddGetEntry(HashKeyVal key, HashDataVal value);
```

Adds an entry corresponding to 'key' without checking its previous existence. Useful just after creating a hash table and filling it up, or when using multiple level keys. It returns a pointer to the entry that was created.

InsertGetEntry

HashEntryPtr NDHash::InsertGetEntry(HashKeyVal key, HashDataVal value);

Same as above but tests whether there was an entry there before or not.

GetEntry

HashEntryPtr NDHash::GetEntry(HashKeyVal key);

Returns the pointer to the actual entry corresponding to 'key'. It will return NULL if not found. The entry can be used with the following calls:

EntryGetKey

static HashKeyVal NDHash::EntryGetKey(HashEntryCPtr entry);

Returns the key stored in the entry.

EntryGetValue

static HashDataVal NDHash::EntryGetValue(HashEntryCPtr entry);

Returns the value stored in the entry.

EntrySetValue

static void NDHash::EntrySetValue(HashEntryPtr entry, HashDataVal value);

Changes the value stored in the entry. (Of course, it is not possible to change the key directly in an entry.) It can be removed with the following call:

RemoveEntry

void NDHash::RemoveEntry(HashEntryPtr entry);

Removes the entry from the hash table. This mechanism is faster than extracting because there is no need to look for the entry corresponding to a key.

Statistics**QueryStats**

void NDHash::QueryStats(HashStatsInfoPtr stats);

Fills 'stats' with the statistical information corresponding to hash. The EntriesPerBin array has to be allocated by the caller, but it is filled by the call.

NDHashStatsInfo

Identifiers	Description
NumBins;	Number of currently allocated bins
NumEntries;	Number of entries globally
EntriesPerBin;	of Ints: entries per bin

Array of HashLenVal which contains the number of entries in each one of the bins. If the array is NULL, HASH_QueryStats won't attempt to use it.

24 *Heap Class*

This class implements a very simple heap structure in Open Interface, oriented towards its use as priority queue.

Overview

A heap is used to keep track of objects ordered according to a priority scheme. Operations are limited to inserting objects in the heap, removing objects from the heap, and retrieving the highest priority object from the heap. Inserting objects into the heap, as well as retrieving the object with the highest priority from the heap are $O(\ln n)$ operations, where n is the number of objects in the heap.

Heap Class

Heap is the base class for heaps of objects ordered to a given priority.

Constructor and Destructor

Constructor

```
NDHeap::NDHeap(void);
```

Default heap construction.

Destructor

```
NDHeap::~NDHeap(void);
```

Default heap destruction.

Heap Size

GetSize

```
HeapIndexVal NDHeap::GetSize(void);
```

Returns the number of entries in the heap.

Heap Manipulation

Add

void NDHeap::Add(HeapKeyVal key, ClientPtr client);

Insertion with no reorder: the heap structure is temporarily incorrect, a call to NDHeap::Correct will be necessary before the heap can actually be used.

Correct

void NDHeap::Correct(void);

Heap correction: the structure of the key is corrected. This call will be made in general after a series of calls to NDHeap::Add.

Insert

void NDHeap::Insert(HeapKeyVal key, ClientPtr client);

Insertion with reorder: the entry will be inserted in the heap structure according to its key (the highest key will always remain first).

QueryFirst

BoolEnum NDHeap::QueryFirst(BoolEnum extract, HeapKeyValPtr keyPtr, ClientPtrPtr dataPtr);

Extraction of the top-most entry: if it can find one, it returns `BOOL_TRUE` and sets ``keyPtr'` and ``dataPtr'`; if not it returns `BOOL_FALSE`. If `extract` is `BOOL_TRUE`, the top-most entry is extracted from the structure.

typedef PerfEnum (C_FAR * HeapPerfProc) (HeapPtr, HeapEltPtr, ClientPtr);

Perf

PerfEnum NDHeap::Perf(HeapPerfProc proc, ClientPtr clientData);

Performs ``proc'` on each entry of the heap. ``proc'` gets called with ``clientData'` as last argument.

25 *ISet Class*

This class implements a data structure to represent sets of numeric intervals.

Overview

This class implements a data structure to represent sets of numeric intervals. This class provides the same functionality as the Set class but the implementation is specialized for "interval sets", i.e. numeric sets which mostly contain clusters of contiguous values, and so which are much better represented as sets of disjoint intervals instead of sets of atomic values.

For such interval sets, it is technically possible to use the Set class to represent them but this would be very inefficient.

The reverse is also true: it is technically possible to use the ISet class to represent normal sets (each object could be stored as an interval which contains one element) but the storage and the set operations would be inefficient.

Each interval is specified with two values: Begin and End. Intervals are "closed-open" (i.e. Begin values are included, End values are not). So the interval [Begin, End[contains all values between (Begin) and (End-1).

Two special values do not follow the same rule and are reserved to represent the infimum (lower bound) and the supremum (upper bound) of the value space. These values are used to specify unbounded or halfbounded intervals.

Data Structures

NDISetInterval

Type of an interval. Intervals must be closed-open (i.e. they contain all points between Begin and End-1). Thus we must have: End > Begin.

Identifier	Description
ISetEltVal	Begin
ISetEltVal	End

Constructors and Destructor Interval Sets

Constructor

```
NDISet::NDISet(void);
```

Default iset construction.

Destructor

NDISet::~~NDISet(void);

Default iset destruction.

Special Intervals

UniversalSet

static ISetPtr NDISet::UniversalSet(void);

Returns a pointer to a shared "universal" set (i.e. a set which contains all possible values).

Adding and Removing Intervals

AddIntervals

void NDISet::AddIntervals(ISetLenVal n, ISetIntervalPtr intervals);

Adds n intervals to an ISet. It builds the union.

RemoveIntervals

void NDISet::RemoveIntervals(ISetLenVal numIntervals, ISetIntervalPtr intervals);

Removes 'n' intervals from an ISet. It builds the difference.

QueryIntervals

void NDISet::QueryIntervals(ISetLenVal numIntervals, ISetIntervalPtr intervals);

Fills intervals with the intervals in the iset.

SetIntervals

void NDISet::SetIntervals(ISetLenVal numIntervals, ISetIntervalPtr intervals);

Defines the intervals in the iset to be those specified by intervals.

GetNumIntervals

ISetLenVal NDISet::GetNumIntervals(void);

Returns the number of intervals in the set.

IsAll

BoolEnum NDISet::IsAll(void);

Returns `BOOL_TRUE` if the set contains all possible elements, i.e. is the interval `[ISET_ELTVALLINF, ISET_ELTVALLSUP]`.

GetMinElt

ISetEltVal NDISet::GetMinElt(void);

Returns the smallest element in the set.

GetMaxElt

ISetEltVal NDISet::GetMaxElt(void);

Returns the biggest element in the set.

ContainsElt

BoolEnum NDISet::ContainsElt(ISetEltVal elt);

Returns BOOL_TRUE if the set contains 'elt'.

ContainsIntervals

BoolEnum NDISet::ContainsIntervals(ISetLenVal numIntervals, ISetIntervalPtr intervals);

Returns BOOL_TRUE if the set includes 'interval'.

QueryComplement

void NDISet::QueryComplement(ISetPtr compl);

Computes the complement of 'iset' and puts the result into 'compl'.

Comparing and Combining Two Sets

The functions below are the same as in the Set class (see Setpub.h) except that they expect ISet objects.

MixGetPartSet

MixQueryParts

static ISetMixPartSet NDISet::MixGetPartSet(ISetPtr A, ISetPtr B);

static void NDISet::MixQueryParts(ISetPtr A, ISetPtr B, ISetMixPartSet part, ISetPtr C); Same as the equivalent calls in the Set package, but for ISet objects.

26 *MCH Class*

The MCH class implements the Open Interface machine specific definitions and macros.

Technical Summary

Some of the platforms that Open Interface supports require special attention in the areas of compiler specific keywords, operating system and windowing system specifics. The constants, flags, and macros necessary to accommodate these needs are defined in this class.

The MCH class API is divided into the following categories.

- Compiler Information.
- Microsoft Windows Utilities.
- Operating System Information.
- Special compiler keywords.
- Windowing System Information.

See also:

App, Base, Dsply, ErrW classes.

C_CONST

Defines a portable version of the the C “const” keyword.

C_CONST is a macro that defines a portable version of the the C “const” keyword.

See also

C_FAR, C_NEAR, C_HUGE, C_NOSHARE, C_READONLY

C_EXPORT

Declares function prototypes for export.

C_EXPORT is a macro used to declare function prototypes for exported functions of a library. Your code does not need to use this macro, you are free to use standard exporting conventions.

C_EXPORT is defined as:

```
#define C_EXPORT(lib, vers, ret, proc, args) ret proc L(args)
```

See also

C_CONST, C_FAR, C_NEAR, C_NOSHARE, C_READONLY,
C_VOLATILE

C_FAR

Defines Microsoft Windows “far” keyword for all compilers.

C_FAR defines the Microsoft Windows “far” keyword so that it is accepted by all compilers. The “far” keyword only applies to segmented architectures such as Microsoft Windows and OS/2.

C_FAR is defined as:

```
#ifdef DOSMSWIN
#define C_FAR    far
#endif
#ifdef OS2PM
#define C_FAR    far
#endif
#ifndef C_FAR
#define C_FAR
```

See also

C_CONST, C_EXPORT, C_NEAR, C_NOSHARE, C_READONLY, C_VOLATILE

C_NEAR

Defines Microsoft Windows “near” keyword for all compilers.

C_NEAR defines the Microsoft Windows “near” keyword so that it is accepted by all compilers. The “near” keyword only applies to segmented architectures such as Microsoft Windows and OS/2.

C_NEAR is defined as:

```
#ifdef DOSMSWIN
#define C_NEAR    near
#endif
#ifdef OS2PM
#define C_NEAR    near
#endif
#ifndef C_NEAR
#define C_NEAR
```

See also

C_CONST, C_EXPORT, C_FAR, C_NOSHARE, C_READONLY, C_VOLATILE

C_NOSHARE

Defines VMS specific “noshare” keyword for all compilers.

C_NOSHARE defines the VMS specific “noshare” keyword so that it is accepted by all compilers. The “noshare” keyword only applies to the VMS operating system.

C_NOSHARE is defined as:

```
#if MCH_OS == MCH_OSVMS
#define C_NOSHARE    noshare
#endif
#ifndef C_NOSHARE
#define C_NOSHARE
```

See also

C_CONST, C_EXPORT, C_FAR, C_NEAR, C_READONLY, C_VOLATILE

C_READONLY

Defines VMS specific “readonly” keyword for all compilers.

C_READONLY defines the VMS specific “readonly” keyword so that it is accepted by all compilers. The “readonly” keyword only applies to the VMS operating system.

C_READONLY is defined as:

```
#if MCH_OS == MCH_OSVMS
#define C_READONLY    readonly
#endif
#ifndef C_READONLY
#define C_READONLY
#endif
```

See also

C_CONST, C_EXPORT, C_FAR, C_NEAR, C_NOSHARE, C_VOLATILE

C_REG...

Register variables.

These are described below.

Field	Description
C_REG1	Defined as ‘register’ if machine has ≥ 1 registers
C_REG2	Defined as ‘register’ if machine has ≥ 2 registers
C_REG3	Defined as ‘register’ if machine has ≥ 3 registers
C_REG4	Defined as ‘register’ if machine has ≥ 4 registers
C_REG5	Defined as ‘register’ if machine has ≥ 5 registers
C_REG6	Defined as ‘register’ if machine has ≥ 6 registers
C_REG7	Defined as ‘register’ if machine has ≥ 7 registers
C_REG8	Defined as ‘register’ if machine has ≥ 8 registers

It is up to you to prioritize and use registers in your routines.

You may need to undefine some registers starting from the end if the microprocessor on your machine does not support all eight of these registers.

C_SIGNED

Data type for signed integers on ANSI-C compilers.

C_SIGNED is signed on compilers which support the C signed keyword for signed integers (all ANSI-C compilers fit this criterion). C_SIGNED is "" (nothing) on other compilers (most others define char as a signed char).

Before porting to a machine which does not have an ANSI-C compiler, you should check the sign of chars, for example you can execute the following:

```
printf(((int)(char)-1) == -1 ? "char is signed" : "char is unsigned");
```

to check the sign of chars.

See also C_CONST, C_EXPORT, C_FAR, C_HUGE, C_NEAR, C_NOSHARE, C_READONLY, C_REG..., C_VOLATILE

C_VOLATILE

Defines compiler specific “volatile” keyword for all compilers.

C_VOLATILE defines the compiler specific “volatile” keyword so that it is accepted by all compilers. The “volatile” keyword is only supported by some compilers.

C_VOLATILE is defined as:

```
#if MCH_OS == MCH_OSVMS
#define C_VOLATILE    volatile
#endif
#ifndef C_VOLATILE
#define C_VOLATILE
#endif
```

See also C_CONST, C_EXPORT, C_FAR, C_NEAR, C_NOSHARE, C_READONLY

MCH_CHAR... MCH_WCHAR...

Defines the primary character set.

It can take the values described below.

Identifier	Description
MCH_CHARASCII	Primary character set is ASCII
MCH_CHAREBCDIC	Primary character set is EBCDIC (IBM).

MCH_WCHAR defines the method used for encoding multi-byte characters. It can take the values described below.

Identifier	Description
MCH_WCHARSINGLE	Strings use a single byte only.
MCH_WCHARMIXED	Strings can contain mixture of single and multi-byte characters.
MCH_WCHARDOUBLE	Strings use a uniform 2-byte coding scheme.

In addition, you should define the MCH_CHARLIB... flag to specify which support library is used for 2-byte characters. See mchpub.h.

See also

Char, WChar, UChar, CStr

MCH_CHIP...

Specifies the chip architecture of a machine.

MCH_CHIP defines the microprocessor chip architecture of a machine. It can take any of the MCH_CHIP... values listed above.

For each environment supported by Open Interface, there should be one entry describing its chip architecture, operating system, and windowing system. For example:

```
#ifndef MYENVIR
#define MCH_CHIPMCH_CHIPXXXX
#define MCH_OSMCH_OSBXXXX
#define MCH_WINMCH_WINXXXX
#define ... // optional flags
#endif
```

where MYENVIR represents a given environment and XXXX represents a flag suffix appropriate to a given environment.

See also

MCH_OS..., MCH_WIN...

MCH_MSWDLLCODE

Implements Microsoft Windows specific

MCH_MSWDLLCODE is a macro to be used to implement Microsoft Windows specific code. It checks to see if Microsoft Windows is the current platform and includes the code if it is.

MCH_MSWDLLCODE is defined as:

```
# define MCH_MSWDLLCODE(c) c
#else
# define MCH_MSWDLLCODE(c)
#endif
```

MCH_OS...

Defines the running operating system.

These constants are the possible values for MCH_OS. MCH_OS is defined to be one of the below values to indicate the operating system running.

Identifier	Description
MCH_OSUNIX	Unix
MCH_OSVMS	VMS
MCH_OSOS2	OS/2
MCH_OSMAC	Macintosh
MCH_OSCMS	VM/CMS
MCH_OSWIN16	Targets Microsoft Win16-based code. See note below.
MCH_OSWIN32	Targets Microsoft Win32-based code. See note below.

Note that on the MS Windows platform, the operating system cannot be determined at compile time. The specific OS (whether Windows NT, WIndows95, Windows 3.1, or DOS) must be queried at runtime using NDMswOs::GetOsInfo.

See also MCH_WINMAC, MCH_WINMSWIN, MCH_WINPM, MCH_WINUIS, MCH_WINX11

MCH_WCHAR...

Defines the method used for encoding multi-byte characters.

See MCH_CHAR.

MCH_WIN...

Defines windowing system running.

These constants are the possible values for MCH_WIN. MCH_WIN is defined to be one of the above values to indicate the windowing system running.

Identifier	Description
MCH_WINX11	X Windows, version 11.
MCH_WINPM	Presentation Manager.
MCH_WINMS	Microsoft Windows.
MCH_WINMAC	Macintosh.
MCH_WINCHR	Character based.

See also

MCH_OSCMS, MCH_OSWIN16, MCH_OSWIN32, MCH_OSMAC,
MCH_OSOS2, MCH_OSUNIX, MCH_OSVMS

Compiler Information

MCH_Cc

This symbol defines the compiler that is being used. It can take any of the MCH_CC_XXX values below. MCH_CC identifies a particular compiler "product line". So, for example, there is only one value for GNU's GCC on all platforms.

On UNIX, it is not always possible to identify a particular compiler product based on predefined compiler flags. So, we introduced the MCH_CC_UNIX value to represent the "generic" UNIX compilers.

This header is set up so that you get a compilation error when your compiler is not officially supported. This scheme is not entirely reliable, especially on UNIX where we do not have enough information to identify the compilers precisely. If your compilation fails because the compiler has not been identified as one of the officially supported compiler, you can try to compile by defining MCH_CC as 0 (MCH_CC_UNSUPPORTED) on the command line, at your own risk.

MCH_CC_UNSUPPORTED	Not officially supported, try at your own risks
MCH_CC_MICROSOFT	Microsoft Visual C++ compiler
MCH_CC_BORLAND	Borland C/C++ compiler
MCH_CC_WATCOM	Watcom's wcc compiler
MCH_CC_MPW	Apple's MPW compiler
MCH_CC_THINKC	Think-C compiler
MCH_CC_MWERKS	Metrowerks' CodeWarrior compiler
MCH_CC_IBMC2	IBM's C compiler for OS/2

MCH_CC_VAXc	Digital's VAX C compiler
MCH_CC_DECc	Digital's ANSI C compiler
MCH_CC_DECCXX	Digital's C++ compiler
MCH_CC_GNU	GNU's gcc compiler
MCH_CC_UNIX	Generic UNIX compiler
MCH_CC_HP	HP's C compiler for HPUX.

This class implements a generic 'notifier'.

Overview

Notifications involve two entities:

- The 'notifier': this is the class which notifies other classes when certain events occur.
- The 'clients': these are the classes which are interested in receiving notifications from the 'notifier'.

To implement a 'notifier', you must do the following things:

- Make that notifier accessible to client classes by defining an exported function in your class API (i.e. `MYNFY_GetNotifier`).
- Define the argument which will be passed to your clients when you notify them. The notifier will call the procedures that the clients have registered, passing them as second argument a 'notification information' (defined generically as a `ClientPtr`) which is specific to the notifier. In simple cases, you will pass a notification code. In more complex cases, you might want to pass a pointer to a structure containing the notification information.
- Call `NFIER_Broadcast` in places where you need to notify your clients.

To implement a 'client' (to use the services of a 'notifier'), you must do the following things:

- Get the 'notifier' by calling the appropriate routine in the API of the 'notifier' class (i.e. `MYNFY_GetNotifier`).
- Register a notification procedure to that notifier, for example by calling `NFIER_ClientNewRegister`.

This call will return a 'notifier client' pointer which will be passed as first argument to your callback procedure. You can cancel your registration to the notifier, for example by calling `NFIER_UnregisterDisposeClient`.

- You may want to associate some client data with the 'notifier client' pointer by calling `NFIER_ClientSetClientData`. Then, your notification procedure can retrieve that client data information by calling `NFIER_ClientGetClientData`.
- Your notification procedure will be called with two arguments: the first one is the 'notifier client' pointer, the second one is the notification information set up by the notifier at the time he called 'Broadcast'. In case you had associated some client data with the 'notifier client' pointer, you can retrieve it by calling `NFIER_ClientGetClientData` on the first argument received by your notification procedure.

Creating and Disposing

Constructors

NDNotifier::NDNotifier(void);

Default notifier construction.

NDNotifierClient::NDNotifierClient(void);

Default notifier client construction.

NDNotifierClient::NDNotifierClient(NotifierClientProc proc);

Constructs the notifier client with `proc` as call-back.

Destructors

NDNotifier::~NDNotifier(void);

Default notifier destruction.

NDNotifierClient::~NDNotifierClient(void);

Default notifier client destruction.

Broadcasting a Notification

Broadcast

void NDNotifier::Broadcast(ClientPtr clientData);

Broadcasts a notification (`clientData` contains the notification information) to all the clients which have registered.

Notifier Client Creation and Destruction

Client Construction

NDNotifierClient::NDNotifierClient(void);

Default notifier client construction.

NDNotifierClient::NDNotifierClient(NotifierClientProc proc);

Constructs the notifier client with `proc` as call-back.

Client Destruction

NDNotifierClient::~NDNotifierClient(void);

Default notifier client destruction.

Associating Client Data with the Notifier Client Pointer

ClientSetClientData

```
void NDNfier::ClientSetClientData(ClientPtr clientData);  
    Associates `clientData' with `nfierClient'.
```

ClientGetClientData

```
ClientPtr NDNfier::ClientGetClientData(void);  
    Retrieves the client data previously associated with `nfierClient'.
```

Notifier Client Registration and Unregistration

RegisterNfierClient

```
void NDNfier::RegisterNfierClient(NfierClientPtr client);  
    Adds a notifier client to the list of clients of a given notifier. The notifier  
    client will be called through its notification call-back whenever  
    NFIER_Broadcast on the notifier is done.
```

Convenience: Unregistration, destruction and deallocation

UnregisterNfierClient

```
void NDNfier::UnregisterNfierClient(NfierClientPtr client);  
    Removes a notifier client from the list of clients of a notifier.
```


This class implements several popular compression algorithms.

Overview

Short Description of the Compression Algorithms:

All the following algorithms are fully reversible (i.e. no information is lost during compression). These algorithms are:

- RLE (Run Length Byte Encoding). Sequences of more than 3 identical bytes are encoded as 3 bytes: [128, nb_bytes, byte]. 128 is encoded as: [128, 0]. This algorithm is very popular because it is very simple and works well on simple images. It should not be used to pack text or random data. Used in: Windows BMP file format, MacPaint file format.
- PackBits. Similar to RLE but sequences of N identical bytes are encoded as 2 bytes: [- N + 1, byte], while sequences of N different bytes are encoded as N+1 bytes: [N - 1, byte1, ..., byteN]. In both cases, N <= 128. A little more difficult to compress than RLE. Decompression is easy. This implementation is byte-oriented like the one on the Mac or in TIFF. Used in: TIFF file format, Mac memory Pixmap.
- CCITT Group3 and Group4. This is a modified version of the Huffman method. Sequences of 1s and 0s are coded by their respective length, each length is then encoded into codes according to a fixed table. It has a terrible worst-case performance (with gray patterns). It has been purposely designed and optimized to compress fax transmission or any other kind of simple monochrome bitmaps where most pixels are white. It should not be used for an ASCII text file or to compress color images (use LZW instead). Used in: TIFF file format.
- LZW (Lempel-Zif & Welch). When a string of bytes is repeated at different locations, it is stored in a string table and each occurrence of the string is encoded by its index in the string table. This string table is different for every strip, and, remarkably, does not need to be kept around for the decompressor. The trick is to make the decompressor build the same table as it built when compressing the data. LZW has a very good compression ratio (5 for image files, 3 for English text), even for random binary data (factor of 2). Decompression is slower than RLE or PackBits but faster than CCITT. It is a little bit tricky to implement efficiently and requires to allocate dynamically a big array (~40k) for the string table. Used in: compress utility on Unix, Stuffit on Mac, ARC on PC, GIF files, TIFF file format, ...

Choice of a Compression Algorithm:

It is important to understand that the choice of a compression algorithm should depend of what you want to compress. Some algorithms perform

well on a certain type of data while they don't perform as well on other type of data. Here is a short comparison:

	RLE	PackBits	CCITT	LZW
Source stream unit	byte	byte	bit	byte
Compression code unit	byte	byte	bit	bit
Compression speed	fast	fast	slow	slow
Decompression speed	very fast	very fast	very slow	slow
C. ratio for text	bad	bad	negative	very good
C. ratio for fax	good	good	very good	very good
C. ratio for simple image	good	good	negative	very good
C. ratio for complex image	poor	poor	negative	good

So the most appropriate choice should be:

- If data is very small, do not compress it at all.
- Else if data is text, use LZW.
- Else if data is fax, use CCITT.
- Else if data is simple image, use RLE or PackBits.
- Else use LZW.

That's the approach used in TIFF file format.

If speed is not critical, the choice can be simple: use LZW. That's the approach used in GIF file format. Another method is to try several algorithms and choose the one with the best compression ratio. Or, a little bit smarter, you can first take some samples of the data at different locations and try to infer which kind of data it is and which algorithm would yield the best result. That's the approach used by most file compression utilities.

Constructors and Destructor

Constructors

NDPack::NDPack(void);

Default pack object construction.

Destructor

NDPack::~NDPack(void);

Default pack object destruction.

API Usage

Compression

Compression can be achieved by calling one of the ``.Encode'` routines.

Before calling an encoding routine, you must fill a `PackRec` structure. `<Bytes>` should point to a buffer which contains the data that you want to compress. `<BytesSize>` should be the size of this buffer.

You must also allocate the `<Codes>` buffer. Since you probably do not know yet what the size of the compressed data will be, you should allocate this buffer to accommodate the worst possible case. For instance, if you want to compress `N` bytes with the PKB method, you should allocate `<Codes>` for `PACK_WORSTCASEPKB(N)` bytes. Set `CodesSize` to the allocation size of `<Codes>`.

Once the `PackRec` structure is filled, `PACK_XXXEncode` will compress exactly `BytesSize` bytes from `<Bytes>`. The resulting codes are stored into `<Codes>`.

At the end, `<CodesSize>` is modified and set to the number of bytes actually written into `<Codes>`. So the actual compression ratio can be computed with:

```
compression_ratio = <CodesSize> / <BytesSize>.
```

If you write the resulting codes into a file, you should also save the value of `<BytesSize>`. You will probably need it later for the decoding (unless you can recompute it from other parameters, for instance from the width and height of a bitmap).

Decompression

Decompression can be achieved by calling one of the `PACK_XXXDecode` routines.

Before calling a decoding routine, you must fill a `PackRec` structure. `Codes` should point to a buffer which contains the data that you want to decompress. `<CodesSize>` should be the size of this buffer.

You must also allocate the `<Bytes>` buffer. The size of this buffer is usually stored with the compression data or can be computed from other parameters (like the width and height of a bitmap). Set `BytesSize` to that size, and allocate `<Bytes>` for at least that size.

Once the `PackRec` structure is filled, `PACK_..Decode` will decompress the compressed data from `<Codes>` and put the result into `<Bytes>`. The decompression process stops as soon as `<BytesSize>` bytes have been obtained in the `<Bytes>` buffer (note that `<CodesSize>` is not used to stop the decompression process). At the end, `<CodesSize>` is modified and set to the number of bytes actually read from `<Codes>`.

Worst Case Performances

Item	Description
<code>PACK_WORSTCASERLE(n) (n + n / 2)</code>	Happens with an alternance of 128 (encoded as [128, 0]) and of a different byte.
<code>PACK_WORSTCASEPKB(n) (n + 1 + (n - 1) / 128)</code>	Happens when there is no repetition of characters. Each strip is encoded as [N-1, byte1, ..., byteN], with $N \leq 128$.
<code>PACK_WORSTCASECCITT(n) (n * 9 / 2)</code>	Happens when 0s and 1s alternate ('01' encoded as '000111010').
<code>PACK_WORSTCASELZW(n) (n * 3 / 2 + 4)</code>	Happens with true random data.
<code>PACK_WORSTCASE</code> <code>PACK_WORSTCASECCITT</code>	Worst case of all worst cases: worst case for the CCITT encoding.

RLE (Run Length Encoding)

RleEncode

void NDPack::RleEncode(void);

Encodes with RLE algorithm.

RleDecode

void NDPack::RleDecode(void);

Decodes with RLE algorithm.

PackBits

PkbEncode

void NDPack::PkbEncode(void);

Encodes with PackBits.

PkbDecode

void NDPack::PkbDecode(void);

Decodes with PackBits.

The decoding should use the same depth and endianness parameters as those used during encoding. The recommended values are:

- Normal compression should use 8 for `byteDepth` and 0 for `lzwFlags`.
- GIF format sets `lzwFlags` to `PACK_LZWLITTLEENDIAN`.
- TIFF format sets `lzwFlags` to `PACK_LZWTIFFGLITCH`, except for files generated by a few PC softwares which sets it to `PACK_LZWLITTLEENDIAN`.

In addition, many commercial softwares (Hijaak for instance) follow an old version of TIFF specifications and do not start each image strip with a `ClearCode`, as they should do according to TIFF 5.1 or TIFF 6.0. This module supports this particular case as well.

LzwEncode

```
void NDPack::LzwEncode(PackDepthVal depth, PackLzwFlags flags);
```

Encodes with LZW.

LzwDecode

```
void NDPack::LzwDecode(PackDepthVal depth, PackLzwFlags flags);
```

Decodes with LZW.

CCITT Fax Compression

Overview

This section implements Group3 (also known as T4 encoding) and Group4 (also known as T6 encoding) compressions. These compressions have been designed solely for the compression of fax documents. Special flags and parameters are fully described in CCITT reference documents, in TIFF 6.0 specification, and in a separate TIFF Class F standard document. The TIFF documents can be retrieved by anonymous ftp from CompuServe or sgi.com in the `graphics/tiff` directory.

CcittEncode**CcittDecode**

```
void NDPack::CcittEncode(PackSizeVal width, PackCcittFlags ccittFlags);
```

```
void NDPack::CcittDecode(PackSizeVal width, PackCcittFlags ccittFlags);
```

Respectively, encodes and decodes using the CCITT compression mechanisms.

Width: number of pixels per row. Row data is assumed to be byte-aligned, so the number of bytes per row is actually $(width+7)/8$. **BytesSize** must be an exact multiple of the number of bytes per row. If there is only one row, **width** can be set to 0 or to $8*BytesSize$. **ccittFlags**: see above for **PackCcittFlags**. The codes are considered to be in big-endian format (as recommended by TIFF standard). For TIFF Class-F files which use little-endian format (or reverse bit order), you can simply swap all the bytes before calling `CcittDecode` or after calling `CcittEncode`.

Examples:

Standard values for **ccittFlags** are:

- TIFF compression scheme 2 (also known as CCITT Group3 1D or CCITT RLE): **CCITTALIGN** flag is always set.
- TIFF compression scheme 32771 (a.k.a. CCITT RLE with word alignment): Same as scheme 2 but **CCITTALIGNWORD** is also defined.
- TIFF compression scheme 3 (a.k.a. CCITT Group3 or CCITT T4 encoding): **CCITTEOL** and **CCITTRTC** are always set, **CCITTALIGN** depends on bit 2 of **t4Options**, **CCITT2D** depends on bit 0 of **t4Options**, **CCITTK4** depends on the resolution of the image (as specified by CCITT).
- TIFF compression scheme 3 in Class F documents: Same as standard

scheme 3 except that CCITTRTC is never set, while CCITTALIGN is always set.

- TIFF compression scheme 4 (a.k.a. CCITT Group 4 or CCITT T6 encoding): CCITTGROUP4 is always set.

Note: The 'Uncompressed Mode' defined in TIFF is not supported yet.

General Case

Encode

void NDPack::Encode(PackMethodEnum *method*);

Encodes with the method specified by 'method'.

Decode

void NDPack::Decode(PackMethodEnum *method*);

Decodes with the method specified by 'method'.

PackMethodEnum

Index one of the above methods.

Item	Description
K_METHODLZW	Additional parameters are set to 8 for byteDepth and 0 for lzwFlags.
PACK_METHODCCITT	Additional parameters are set to G3-1D encoding with one single row.

29 *PfId Class*

The `PfId` module contains the API to describe and manage the persistent fields of a resource.

Overview

Scope of Documented API

For now, only a small part of that API is documented and officially supported.

Permanent Field Data Types

`PfIdTypeEnum`

Enumerated type describing the possible types for persistent fields

Identifier	Description
<code>PFLD_TYPEBAD,</code>	invalid type.
<code>PFLD_TYPEINT16,</code>	16 bit signed integer.
<code>PFLD_TYPEUINT16,</code>	16 bit unsigned integer.
<code>PFLD_TYPEINT32,</code>	32 bit signed integer.
<code>PFLD_TYPEUINT32,</code>	32 bit unsigned integer.
<code>PFLD_TYPESTR,</code>	string stored as a <code>Str</code> .
<code>PFLD_TYPESTRARRAY,</code>	array of strings (not implemented yet).
<code>PFLD_TYPEVSTR,</code>	string stored as a <code>VStrPtr</code> .
<code>PFLD_TYPEVSTRARRAY,</code>	array of <code>VStrPtrs</code> .
<code>PFLD_TYPERES,</code>	pointer to resource. <code>NULL</code> is illegal.
<code>PFLD_TYPERES0,</code>	pointer to resource, <code>NULL</code> is legal.
<code>PFLD_TYPERESARRAY,</code>	array of <code>ResPtr</code> .
<code>PFLD_TYPEDATA32,</code>	arbitrary length binary data (undoc.).
<code>PFLD_TYPEDATA8,</code>	internal: do not use.
<code>PFLD_TYPEDATA16,</code>	internal: do not use.

Field Categories

PfIdCatEnum

Enumerated type for field categories. The main use of field categories is to support the filtering of resource attributes when we expand resources to the right in the resource browser.

Identifier	Description
PFLD_CATOBSOLETE	obsolete field.
PFLD_CATNONE	invalid category.
PFLD_CATTEXT	field contains text.
PFLD_CATOTHER	field is none of the others.
PFLD_CATSIZE	field contains size, position information.
PFLD_CATCOLOR	field contains a color resource.
PFLD_CATFONT	field contains a font resource.
PFLD_CATPEN	field contains a pen resource.
PFLD_CATPATTERN	field contains a pattern resource.
PFLD_CATCURSOR	field contains a cursor resource.
PFLD_CATICON	field contains an icon resource.
PFLD_CATKEYS	field contains a KyElt or KyLst resource.
PFLD_CATWGT	field contains a widget resource.

Note: Fields with CATOBSOLETE are considered only when compiling the rc format if their Offset is NULL, they are ignored, otherwise they are taken into account. This allows us to smoothly change names of fields, just leave the old entry in the fields array but mark it as obsolete.

Data Structures

NDPFId

Description of a persistent field to the resource manager.

Persistent Feild	Description
Name	Name of the field (which will prefix the value of the field in the.rc file)
Offset	Offset of the field in the resource data structure. See the RCLAS_OFFSET macro in rclasp.h.
EnumType	Data type of the field. One of the PFLD_TYPEXXX constants defined above.
EnumCat	Category for the field. One of the PFLD_CAT constants above.
ClientData	32 bits of client data information

WARNING:

The same PFIdRec structure is shared by a class and all its subclasses. Thus, the offset should be the same for all subclasses. (see comment in RES_RegisterClass and RES_RegisterSubClass).

30 *Point Class*

This class implements the point object.

Overview

A 'point' object is specified by two coordinates called 'x' and 'y' (which can be 16-bit or 32-bit values).

It is normally used to localize a point on a 2-D plane (in which case the coordinates are relative to some origin) or to measure the extent of a 2-D object (in which case both coordinates should be positive).

Genericity Issue

In C, the API is implemented as macros and so is common to both 16-bit and 32-bit implementations. In the macros declarations, 'CoordGenVal' and 'PointGenPtr' are generic types which map to Int16/Point16Ptr or to Int32/Point32Ptr.

In C++, the API is implemented as two separate classes: NdPoint16 and NdPoint32.

Constructors / Destructor

Construct

```
inline NDPoint16::NDPoint16(void);
```

```
inline NDPoint32::NDPoint32(void);
```

Default construction.

ConstructWithValues

```
inline NDPoint16::NDPoint16(Int16 xVal, Int16 yVal);
```

```
inline NDPoint32::NDPoint32(Int32 xVal, Int32 yVal);
```

Construction with values.

Destruct

```
inline NDPoint16::~~NDPoint16(void);
```

```
inline NDPoint32::~~NDPoint32(void);
```

Default destruction.

Sets and Queries

GetX

SetX

```
inline Int16 NDPoint16::GetX(void) const;
```

```
inline void NDPoint16::SetX(Int16 val);
```

```
inline Int32 NDPoint32::GetX(void) const;
```

```
inline void NDPoint32::SetX(Int32 val);
```

Gets/sets the X coordinates of point `p`.

GetY

SetY

```
inline Int16 NDPoint16::GetY(void) const;
```

```
inline void NDPoint16::SetY(Int16 val);
```

```
inline Int32 NDPoint32::GetY(void) const;
```

```
inline void NDPoint32::SetY(Int32 val);
```

Gets/sets the Y coordinates of point `p`.

SetXY

```
inline void NDPoint16::SetXY(Int16 xVal, Int16 yVal);
```

```
inline void NDPoint32::SetXY(Int32 xVal, Int32 yVal);
```

Sets the X and Y coordinates of point `p` to `xVal` and `yVal` respectively.

IncXY

```
inline void NDPoint16::IncXY(Int16 dx, Int16 dy);
```

```
inline void NDPoint32::IncXY(Int32 dx, Int32 dy);
```

Increments the X and Y coordinates by `dx` and `dy` respectively.

SetSameXY

```
inline void NDPoint16::SetSameXY(Int16 val);
```

```
inline void NDPoint32::SetSameXY(Int32 val);
```

Sets both X and Y coordinates to `val`.

Reset

```
inline void NDPoint16::Reset(void);
```

```
inline void NDPoint32::Reset(void);
```

Resets both X and Y coordinates to 0.

IsNull

```
inline BoolEnum NDPoint16::IsNull(void);
```

```
inline BoolEnum NDPoint32::IsNull(void);
```

Returns `BOOL_TRUE` if both coordinates are 0.

Equals**inline BoolEnum NDPoint16::Equals(Point16Ptr p2);****inline BoolEnum NDPoint32::Equals(Point32Ptr p2);**Returns `BOOL_TRUE` if ``p1'` and ``p2'` have same coordinates.**AbsDist****inline Int16 NDPoint16::AbsDist(Point16Ptr p2);****inline Int32 NDPoint32::AbsDist(Point32Ptr p2);**Returns the distance between `p1` and `p2` (using the "L1" distance, which is defined as the max between the distance in X and the distance in Y).**IsInRectExt****inline BoolEnum NDPoint16::IsInRectExt(Point16Ptr ext);****inline BoolEnum NDPoint32::IsInRectExt(Point32Ptr ext);**Returns `BOOL_TRUE` if ``p'` is in the rectangle with origin (0, 0) and extension ``ext'`.

31 *Pool Class*

In some cases, memory management can be significantly sped up by the use of memory pools which are tuned up to better handle allocation & deallocation of structures of a given size.

Overview

Pool oriented memory management

The following code allows the application to:

- Create a pool of memory for a pre-allocated number of cells of a given size.
- Allocate a cell in the pool, eventually increasing its capacity by a user-defined number of cells if necessary.
- Deallocate cells in the pool.

It is oriented towards cases in which the application has a pretty good idea on its needs.

By playing around with `NumPreAllocCells` and `NumGrowCells`, applications should be able to get to a situation in which allocation is very fast and not much is lost when going over the `NumPreAllocCells`.

As with the `Ptr` module which handles the general heap, the memory pool manager has built-in mechanisms to detect memory overwrite and incorrect free operations.

When creating memory pools, the programmer has whole control on the way the memory pool is initialized and will behave as allocations and deallocations will take place.

Example:

```
C_DEFSTRUCT(S_MyStruct) {
    Str    Field1;
    WinPtr Field2;
    ..../..
};
```

The application usually allocates between 50 and 100 such structures, occasionally going over. It may temporarily need up to 5 such structures for intermediate computations that need to be fast. The application can handle them through a pool in which 75 are pre-allocated and that grows by 10 cells when more is needed. It will also need to ask the memory pool manager to preserve at least 5 empty cells before trying to deallocate unneeded fragments.

```
< INIT >
PoolInfoRec    info;
info.CellSize = sizeof(S_MyStructRec);
info.NumPreAllocCells = 75;
info.NumGrowCells = 10;
info.FreeFragThreshNumCells = 5;
PoolPtr new NdPool(&info);
```

```

    ../..
myStruct = (S_MyStructPtr)pool->NewPtr();

    ../..
pool->DisposePtr(myStruct);

    ../..
delete pool;
< END >

```

In terms of internal implementation, memory pools work with fragments which are guaranteed to contain at least the number of cells specified by the programmer.

Even if memory pools are oriented towards fast allocation, they do not keep everything allocated all the time. When a pool is created, the programmer can specify a threshold number of free cells over which unnecessary blocks or fragments will automatically be deallocated. By setting this number to a reasonable value, he/she can make sure that future allocations will result in fragment allocations only if the allocations exceed that number.

Note: By its very nature, the use of memory pools is limited to the cases in which what is allocated in the pool is never resized. Memory pools work exactly the same way on all platforms, including MS-Windows 16 bits.

Pool Definition

Identifier	Description
CellSize;	Size in bytes of the objects allocated in the pool. Open Interface handles all the problems related to alignment on the different platforms.
NumPreAllocCells;	Number of such cells created in the first main fragment. This fragment is created when the pool is initialized.
NumGrowCells;	Number of cells created in each subsequent fragment. These fragments are created when there is no more free cell in any of the other fragments of the pool.
FreeFragThreshNumCells;	When a fragment becomes empty, Open Interface first makes sure that the rest of the pool contains at least this number of free cells before getting rid of this fragment. If it is set to -1, then Open Interface will never deallocate any fragment (it makes it faster but more greedy).

Constructors and Destructor

Constructors

NDPool::NDPool(void);

Default memory pool construction.

NDPool::NDPool(PoolInfoCPtr info);

Memory pool construction. The memory pool gets constructed according to the information contained in *info*. In particular, all the cells that need to be preallocated are.

Destructor

NDPool::~NDPool(void);

Default memory pool destruction.

Setting/Querying the Information on a Memory Pool

SetInfo

void NDPool::SetInfo(PoolInfoCPtr poolInfo);

Updates the memory pool with the information from *poolInfo*.

QueryInfo

void NDPool::QueryInfo(PoolInfoPtr poolInfo);

Fills *poolInfo* with the information with which the memory pool was created.

Allocating and Deallocating

NewPtr

HugePtr NDPool::NewPtr(void);

Returns a pointer to a cell allocated in the pool. The allocation mechanism is geared toward performance and an allocation operation is in general limited to an arithmetic operation on pointers.

DisposePtr

void NDPool::DisposePtr(HugePtr ptr);

Deallocates *ptr*. *ptr* must have been allocated in the pool. The deallocation mechanism is also geared toward performance. A deallocation is in general limited to an arithmetic operation on pointers and a linking of pointers.

Statistics

NDPoolFragStatsInfo

<code>NumCells</code>	Number of cells in fragment .
<code>NumAllocCells</code>	Number of allocated cells .

NDPoolStatsInfo

<code>NumCells</code>	Total number of cells .
<code>NumAllocCells</code>	Number of allocated cells .
<code>Frag</code>	Array of <code>FragStatsInfoPtr</code> .

QueryStats

void `NDPool::QueryStats(PoolStatsInfoPtr stats);`

Fills `stats` with the statistics information on the pool. The `Frag` array in `stats` should be created by the caller, but it will get filled by the call.

ResetStats

static void `NDPool::ResetStats(PoolStatsInfoPtr stats);`

Resets `stats`.

32 *Ptr Class*

The Ptr class implements the Open Interface memory manager.

Technical Overview

NDPtr offers several advantages over the standard malloc and free:

- Ptr works on MS/Windows exactly the same way it works on the other platforms. In particular, Ptr avoids using handles for heap allocation of small blocks of memory,
- Ptr offers better error handling.
- Any memory allocated is zero-blanked.
- Ptr writes marks at the beginning and the end of buffers and check that the marks are still there when you reallocate or free the buffer,
- When the DBG_ON flag is set, Ptr writes a special pattern in the buffer when the buffer is freed, so that you are sure to find corrupted data in a buffer after having freed it,
- Ptr maintains statistics on the number of pointers and the number of bytes allocated.

See also:

Base, Res, Err classes

Data Types

PtrStats

Data structure to describe memory management statistics.

PtrStatsPtr is a pointer to PtrStatsRec, a data structure that holds memory management statistics. The Size field is the total size in bytes currently allocated. The Num field holds the number of pointers allocated. The PeakSize field holds the highest size allocated during the current execution. The PeakNum field stores the highest number of pointers allocated during the current execution.

The fields of this structure are described below:

Identifier	Description
Size	Number of bytes currently allocated Size represents the total number of bytes requested by all PTR_New or PTR_HugeNew calls, this is not exactly the number of bytes actually allocated by the system on your machine. First of all Open Interface keeps a few extra bytes as "debug marks" with each pointers (Debug Libraries only) and also stores the pointer's size, and secondly the memory allocation system call (malloc, ...) may allocate memory in bigger blocks. The correct way to get the total memory allocated by your application is to use another system call, like FreeMem on Mac. On the other hand, if you find that Size doesn't return to the same value after an operation that should not use memory it is a sure indication that your application has a memory leak due to missing PTR_Dispose calls. In that case the difference in Size can help you figure out what structures are not released.
Num	Number of pointers currently allocated.
PeakSize	Peak value for Size since application was launched.
PeakNum	Peak value for Num since application was launched.

See also

QueryStats, NDPtr::StatsOutput.

Enumerated Types

PtrFailEnum

Failure from the Ptr class signalling a problem with memory allocation or deallocation.

Identifier	Description
PTR_FAILNOMEM	Cannot allocate memory because either the size requested is too big, memory ran out, or the memory manager data is corrupted.
PTR_FAILSIZE	Size passed to NDPtr::NEW or NDPtr::SetSize is invalid (i.e., negative size). You must use the NDPtr::Huge... routines to allocate buffers larger than HUGELIMIT (65520), otherwise your code won't be portable to MS Windows.
PTR_FAILNULL	NDPtr::Dispose or NDPtr::SetSize was passed a NULL pointer.
PTR_FAILMARKBEGIN	Beginning debugging marker is corrupted. You may be writing in areas of memory that you did not allocate.
PTR_FAILMARKEND	Ending debugging marker is corrupted.
PTR_FAILMARKFREE	You are trying to free a pointer that is already been freed by NDPtr::Dispose.

These constants represent the different errors signalled by the PTR class. The failures are signalled through a callback procedure that your program can override, using NDPtr::SetFailProc.

See also

NDPtr::SetFailProc, NDPtr::DefFailProc, NDPtr::GetFailProc

Alignment

Most RISC machines can store certain data types only on addresses which have certain alignment constraints.

Some functions are provided to help with data alignment (for now, they assume that you want your data aligned for the worst case, more specific alignment support may be provided later).

GetAlignedSize

Returns the smallest aligned size for the size passed.

static PtrSizeVal NDPtr::GetAlignedSize (PtrSizeVal size);

NDPtr::GetAlignedSize returns the smallest size that is both aligned and that can hold the requested size.

AlignCheck

Checks whether the pointer is aligned for the worst case scenario.

static void NDPtr::AlignCheck (HugeCPtr ptr);

Different data types require different amounts of space and alignment constraints. This macro checks to see if the ptr passed is aligned to the worst case needs of the current platform.

MCH_ALIGN is defined as:

```
#if MCH_ALIGN == 1
#define PTR_ALIGNCHECK(a) a
#else
#define PTR_ALIGNCHECK(a)  DBG_CHECK((((long)(a)) % MCH_ALIGN)
== 0)
#endif
```

Alloc, Free, and Realloc

The following routines allow you to allocate, reallocate, and free pointers.

The distinction between 'normal' and 'huge' pointers is introduced so that we can write code which will be portable to the large (but not huge) memory models on DOS/MS-Windows and OS/2-PM. When you compile in large model, normal pointers can not access memory blocks larger than 64k bytes because of the segmented architecture. Huge memory blocks can be referenced only by 'Huge' pointers.

On MS-Windows and OS/2-PM, you can not use NDPtr::New or NDPtr::SetSize for a size which is greater than HUGELIMIT. If ever this happens, NDPtr::New or NDPtr::SetSize will fail (PTR_FAILSIZE).

On platforms which don't have a segmented architecture (UNIX, VMS, MAC, and MS/Windows when compiling in 32-bit mode with WatcomC compiler), this limit is not enforced by default but can be enforced if you set the environment variable OIT_ENFORCE64KLIMIT to TRUE. Enforcing the same limit allows to detect on those machines problems which would otherwise appear only at the time you port your code to the PC.

Recommendations

If you are developing on PC, you absolutely need to use the correct memory attribute (i.e. C_NEAR/C_FAR/C_HUGE) when declaring all your pointers. We recommend that you use huge pointers only when you really need them because the dereferencing of a huge pointer is significantly slower than the dereferencing of a normal pointer. So you should use 'New' most of the time, and 'HugeNew' only in the cases where you really need memory blocks larger than 64K.

If your code will never be ported to PC, you can unset this variable and use indifferently NDPtr::New or NDPtr::HugeNew. However, experience shows that most users decide to port their application to PC sooner or later, even if they did not consider it initially. So, even if you are not developing for the PC, we recommend that you write code which remains "PC-clean".

Note: You are not allowed to assign a 'huge' pointer to a 'normal' pointer variable, or to pass a 'huge' pointer to a routine expecting a 'normal' pointer. Unfortunately, only the PC compilers will give you warnings in such cases.

GetSize

Returns the size of which the buffer 'ptr' is allocated.

```
static PtrSizeVal NDPtr::GetSize(VoidPtr ptr);
```

SetSize

Reallocates the 'ptr' buffer for a different size 'size'. If the new size is larger than the old size, the difference is initialized with zeros. The pointer returned might be different from the original pointer, even if it is reallocated for a smaller size.

The new size must be less than HUGELIMIT, otherwise you should allocate a huge pointer and free the non huge pointer. If 'ptr' is NULL, NDPtr::SetSize calls NDPtr::New(size) and returns the result.

```
static VoidPtr NDPtr::SetSize(VoidPtr ptr, PtrSizeVal size);
```

New

Allocates a new pointer of the size specified or for named structure.

```
static VoidPtr NDPtr::New (PtrSizeVal size);
```

NDPtr::New allocates a new pointer of size specified and returns that pointer.

The buffer that the new pointer points to is initialized with zeros before the new pointer is returned.

Dispose

Frees memory allocated for a given pointer.

```
static void NDPtr::Dispose (VoidPtr ptr);
```

When DBG_ON is not defined, NDPtr::Dispose is the same as using standard C free routine — it frees the memory allocated for the ptr passed. When DBG_ON is defined, NDPtr::Dispose overwrites the buffer with a

special pattern so that contents of that buffer are completely destroyed. This will aid in your pointer debugging.

HugeNew
 HugeDispose
 HugeGetSize
 HugeSetSize

The following functions are analogous to the above, but they operate on huge pointers:

```
static HugePtr NDPtr::HugeNew (PtrHugeSizeVal size);
static void NDPtr::HugeDispose (HugePtr hugeptr);
static PtrHugeSizeVal NDPtr::HugeGetSize (HugePtr ptr);
static HugePtr NDPtr::HugeSetSize (HugePtr ptr, PtrHugeSizeVal size);
```

Same as before except that the sizes can be larger than HUGELIMIT. You must use the NDPtr::Huge... routines to allocate buffers larger than HUGELIMIT (65520), otherwise your code won't be portable to MS/Windows.

Macintosh Note: MPW has a limit of 8MB for allocating memory. If you try to allocate more than this, 0 bytes are allocated.

Functions for Memory Copy, Move, Set

Clear

Sets the set of bytes specified in a buffer to zero.

```
static void NDPtr::Clear (VoidPtr ptr, PtrSizeVal number);
```

NDPtr::Clear sets the number of bytes starting at ptr to zero.

See also

NDPtr::Copy, NDPtr::Move, NDPtr::Set

Set

Sets the specified number of bytes of a buffer to a value specified.

```
static void NDPtr::Set (VoidPtr dest, Byte byte, PtrSizeVal number);
```

NDPtr::Set sets each of the first number of bytes beginning at dest to c. For example, if c is "a" and number is 4, the first four bytes starting at dest will all be "aaaa".

SetSize

Changes the size of the pointer specified.

```
static VoidPtr NDPtr::SetSize (VoidPtr ptr, PtrSizeVal size);
```

NDPtr::SetSize changes the buffer size that ptr points to. Essentially, it reallocates the pointer with the new size and copies the data from the old buffer to the new buffer, in case the pointer changed. If the new buffer

increased in size, then the end of the buffer (between old end and new end) is initialized with zeros.

Copy

Copies the number of bytes specified to a disjoint location in a buffer.

```
static void NDPtr::Copy (VoidPtr dest, VoidCPtr src, PtrSizeVal number);
```

NDPtr::Copy copies number bytes starting at src to destination. This routine assumes disjoint source and destination areas.

Move

Copies `size' bytes from `src' to `dst'. `src' and `dst' may overlap.

```
static void NDPtr::Move (VoidCPtr p1, VoidCPtr p2, PtrHugeSizeVal size);
```

Swap

Copies `size' first bytes of `p1' to `p2' and vice versa. `p1' and `p2' must NOT overlap.

```
static void NDPtr::Swap (VoidCPtr p1, VoidCPtr p2, PtrHugeSizeVal size);
```

Cmp

Compares the `size' first bytes of `p1' and `p2'. The bytes are compared as their unsigned values bytes.

```
static CmpEnum NDPtr::Cmp (VoidCPtr p1, VoidCPtr p2, PtrHugeSizeVal size);
```

Matches

Returns whether `p1' and `p2' match exactly on their first `size'.

```
static BoolEnum NDPtr::Matches (VoidCPtr p1, VoidCPtr p2, PtrHugeSizeVal size);
```

Move

Copies the number of bytes specified to an overlapping location.

```
static void NDPtr::Move (VoidPtr dest, VoidCPtr src, PtrSizeVal size);
```

NDPtr::Move copies number bytes starting at src to destination. This routine can handle both disjoint and overlapping source and destination areas.

HugeClear
HugeSet
HugeSetSize
HugeCopy
HugeMove
HugeSwap
HugeCmp
HugeMatches
HugeMove

The following functions are analogous to the above, but they operate on huge pointers:

```

static void NDPtr::HugeClear (HugePtr hugeptr, PtrHugeSizeVal size);
static void NDPtr::HugeSet (HugePtr dest, Byte byte, PtrHugeSizeVal size);
static HugePtr NDPtr::HugeSetSize (HugePtr hugeptr, PtrHugeSizeVal size);
static void NDPtr::HugeCopy (HugePtr dest, HugeCPtr src, PtrHugeSizeVal size);
static void NDPtr::HugeMove (HugeCPtr p1, HugeCPtr p2, PtrHugeSizeVal size);
static void NDPtr::HugeSwap (HugeCPtr p1, HugeCPtr p2, PtrHugeSizeVal size);
static CmpEnum NDPtr::HugeCmp (HugeCPtr p1, HugeCPtr p2, PtrHugeSizeVal size);
static BoolEnum NDPtr::HugeMatches(HugeCPtr p1, HugeCPtr p2, PtrHugeSizeVal size);
static void NDPtr::HugeMove (HugePtr dest, HugeCPtr src, PtrHugeSizeVal size);

```

You must use the NDPtr::Huge... routines to allocate buffers larger than HUGELIMIT (65520), otherwise your code won't be portable to MS/Windows.

NDPtr::HugeSet is similar to NDPtr::Set, but fills with an integer *i* instead of a character.

NDPtr::HugeSetSize reallocates *ptr* to the size specified. Similar to NDPtr::SetSize, but NDPtr::HugeSetSize you can set a pointer for sizes much larger than the defined HUGELIMIT (65520).

Statistics

QueryStats

Determines the current memory manager statistics.

```
static void NDPtr::QueryStats (PtrStatsPtr ptr);
```

NDPtr::QueryStats retrieves the current statistics on the memory manager and places the results in the record that *ptr* points to.

StatsOutput

Outputs memory manager statistics to standard output.

```
static void NDPtr::StatsOutput (void);
```

NDPtr::StatsOutput outputs the current memory manager statistics to standard output.

Low-level Byte Copies

Operations performed on values stored at address *ptr+offset*.

GetByte

Reads value stored at address *ptr+'offset'*.

```
static Byte NDPtr::GetByte (VoidCPtr ptr, PtrSizeVal offset);
```

SetByte

Sets new value at address *ptr+'offset'* to *'byte'*.

```
static void NDPtr::SetByte(VoidPtr ptr, PtrSizeVal offset, Byte byte);
```

CopyByte

Copies value stored at 'src'+srcOffset' to dst+'dstOffset'.

```
static void NDPtr::CopyByte(VoidPtr dst, PtrSizeVal dstOffset, VoidCPtr src,
    PtrSizeVal srcOffset);
```

SwapByte

Exchanges values at ptr1+'offset1' and 'ptr2'+offset2'.

```
static void NDPtr::SwapByte (VoidPtr p1, PtrSizeVal offset1, VoidPtr p2, PtrSizeVal offset2);
```

See also NDPtr::WriteInt8, NDPtr::WriteInt16, NDPtr::WriteInt32

HugeCopyByte

HugeGetByte

HugeSetByte

HugeSwapByte

The following functions are analogous to the above, but they operate on huge pointers:

```
static Byte NDPtr::HugeGetByte(HugeCPtr ptr, PtrHugeSizeVal offset);
```

```
static void NDPtr::HugeSetByte(HugePtr ptr, PtrHugeSizeVal offset, Byte byte);
```

```
static void NDPtr::HugeCopyByte(HugePtr dst, PtrHugeSizeVal dstOffset, HugeCPtr src,
    PtrHugeSizeVal srcOffset);
```

```
static void NDPtr::HugeSwapByte(HugePtr p1, PtrHugeSizeVal offset1, HugePtr p2,
    PtrHugeSizeVal offset2);
```

Machine-Independent Memory Representations for Integers

Integers are not stored in memory the same way on all machines. Memory representations may differ by the order of bytes within the integer. On Big-Endian machines, most significant byte is stored in lowest address. On Little-Endian machines, most significant byte is in highest address. We define as "machine-dependent format" the format available on the local machine. We define as "standard format" the Big-Endian representation.

Note: The same functions can be applied for signed or unsigned integers.

Int8ToStd

Int16ToStd

Int32ToStd

Converts in place an integer from the machine-dependent format to the standard format. These functions have no requirement concerning alignment.

```
static void NDPtr::Int8ToStd(Int8Ptr valp);
```

```
static void NDPtr::Int16ToStd(Int16Ptr valp);
```

```
static void NDPtr::Int32ToStd(Int32Ptr valp);
```


Int8ToMch
Int16ToMch
Int32ToMch

Converts in place an integer from standard format into the machine-dependent format. These functions have no requirement concerning alignment.

```
static void NDPtr::Int8ToMch(Int8Ptr valp);  
static void NDPtr::Int16ToMch(Int16Ptr valp);  
static void NDPtr::Int32ToMch(Int32Ptr valp);
```

ReadInt8
ReadInt16
ReadInt32

Reads a machine-dependent integer from a memory buffer 'ptr' where integers are in standard format. The destination pointer must be aligned as a pointer to an Int8/16/32.

```
static void NDPtr::ReadInt8(VoidCPtr ptr, Int8Ptr valp);  
static void NDPtr::ReadInt16(VoidCPtr ptr, Int16Ptr valp);  
static void NDPtr::ReadInt32(VoidCPtr ptr, Int32Ptr valp);
```

WriteInt8
WriteInt16
WriteInt32

Writes a machine-dependent integer into a memory buffer 'ptr' where integers are in standard format. The source must be aligned as a pointer to an Int8/16/32.

```
static void NDPtr::WriteInt8(VoidPtr ptr, Int8CPtr valp);  
static void NDPtr::WriteInt16(VoidPtr ptr, Int16CPtr valp);  
static void NDPtr::WriteInt32(VoidPtr ptr, Int32CPtr valp);
```

See also

NDPtr::SwapByte.

Memory Representations for Strings

Strings are not stored in memory the same way on all platforms. We define as "standard format" the case where '\n' is mapped to 0x0A and '\r' to 0x0D. The only exception is MPW-C where these characters are inverted.

Note: These macros do not perform any special translation for Kanji characters. Therefore these macros are portable only within one standard:

UJS (or EUC): Sun, NEc
SJIS (shift-JIS): HP, Sony, PC, Mac

StrToStd

Converts strings to and from standard format.

static void NDPtr::StrToStd (Str str, StrIVal len);

StrToMch

Converts in place a string from machine-dependent format to standard format.

static void NDPtr::StrToMch (Str str, StrIVal len);

Converts in place a string from standard format to machine-dependent format.

See also NDPtr::ReadStr, NDPtr::WriteStr.

ReadStr

Read and write machine-dependent strings to and from standard format.

static void NDPtr::ReadStr (HugeCPtr ptr, Str str, StrIVal len);

WriteStr

Reads a machine-dependent string from a memory buffer where strings are stored in standard format.

static void NDPtr::WriteStr (HugePtr ptr, CStr cstr, StrIVal len);

Writes a machine-dependent string into a memory buffer where strings are stored in standard format.

Strings are not stored in memory the same way on all platforms. We define as “standard format” the case where ‘\n’ is mapped to 0x0A and ‘\r’ to 0x0D. The only exception is MPW-C where these characters are inverted.

See also NDPtr::StrToStd, NDPtr::StrToMch.

Errors Signalled by Ptr Class

These failure are signalled through a call-back that the program can override. The default call-back calls ERR_Fail and displays an error window with the corresponding error message.

If the program does not want that behaviour, it can install its own call-back there and do whatever it pleases. The value returned by the call-back is in turn returned by the PTR call if needed.

For instance, if the program wants to avoid triggering an error when trying to allocate memory, and get NULL as a return value instead, it could do the following:

```
if (failCode == PTR_FAILNOMEM) {
    return NULL;
} else {
    return NdPtr::DefFailProc(failCode, size);
}
}
< somewhere before >
NDPtr::SetFailProc(S_MyFailProc);
< in the caller code>
ptr = NDPtr::New(1000000);
if (ptr == NULL) {
    ..../..
}
```

Of course, the change is system-wide. This means that the calls made by Open Interface will have the same behaviour.

Trapping failures is a good programming principle, except in situations when memory allocation is made tentatively and the caller has alternative solutions that are less time-consuming. Thus, we recommend that the programmer use the failing principle, except in those precise places where memory allocation is done tentatively.

GetFailProc

Returns the custom failure callback procedure installed.

static PtrFailProc NDPtr::GetFailProc (void);

NDPtr::GetFailProc retrieves the previously installed custom failure procedure.

See also NDPtr::SetFailProc, NDPtr::DefFailProc, PtrFailEnum...

SetFailProc

Sets a custom failure callback procedure.

static void NDPtr::SetFailProc (PtrFailProc failProc);

NDPtr::SetFailProc overrides the default failure procedure (to call ERR_Fail) and installs failProc as the failure routine for memory allocation. The client application typically performs this task immediately after loading and initializing the resource.

See also NDPtr::GetFailProc, NDPtr::DefFailProc, PtrFailEnum...

DefFailProc

Default method to trap memory management failures.

static HugePtr NDPtr::DefFailProc(PtrFailEnum fail, PtrHugeSizeVal size);

33 *RClas Class*

The RClas class defines what “resource classes” are and provides the API to access them.

Note; The normal procedure to create a new class is to select Add SubClass or Add Class in the Resource Browser. Refer to the User’s Guide for details. The new class that you create in the Resource Browser will automatically be generated with the code necessary to register the class and create new instances. Refer to the Programmer’s Guide for details about the code generated.

Persistent Data

The RClas class is the base class for resource class meta-information holders. The fields of the RClasRec class are described below:

Field	Description
SizeOfRes	Size of resource instance in bytes.
Name	Name of the class.
Fields	Pointer to an array describing persistent fields.
Flags	Class flags. See RCLAS_FLAG...
DefNotify	Default notification handler. Will be installed by default in all instances of the class.
New	Creates an instance (for C++ classes).
Delete	Deletes an instance (for C++ classes).
Construct	Constructs an instance (for C++ classes).
Destruct	Destructs an instance (for C++ classes).
ParentClass	Pointer to the parent class. By tracing the ParentClass pointers of any class, the Res class is eventually reached (see NDRes::Class()) and its parent class is NULL.
ClientData	Field for storing 32 bits of data to be associated with the class.
Version	Version number for the class.
ClassId	Class id for the class.

See also

Res, RLib, Wgt, EdRes classes.

RClasFlagsSetEnum

These flags are described below.

Identifier	Description
RCLAS_FLAGESO	Resource is esoteric (for advanced programmers only).
RCLAS_FLAGWGT	Resource is a widget, including Panel and Win.

RCLAS_FLAGPANEL	Resource is a container, including panel.
RCLAS_FLAGMENU	Resource is a menu, a menu bar, a menu item or a menu separator.
RCLAS_FLAGEXPORTED	Resource will be exported by default.
RCLAS_FLAGCOMPOSITE	Resource has widgets in its children field.
RCLAS_FLAGUNKNOWN	Resource loaded from a .dat file and belongs to a class that is not registered (unknown).
RCLAS_FLAGNOTINSTANTIABLE	No instance of this resource class can be created in OpenEdit.
RCLAS_FLAGDONTDISPLAY	Resource class should not be displayed in resource browser.
RCLAS_FLAGNOTINPALETTE	Class icon should not appear in the Window Editor widget palette.
RCLAS_FLAGNOTSUBCLASSABLE	Cannot be subclassed

See also

RClasRec.

CPlusRegister

Registers a new resource class with the resource manager.

static RClasPtr NDRClas::CPlusRegister (CStr name, RClasNewProc nProc, RClasDeleteProc dProc, ResNfyProc nfy, RClasPtr pClass, PFldPtr oiFields);

Informs the Resource Manager that a new resource class rclas is available. You only need to use this routine if you are creating a new class. If so, you should include this procedure in your C template file and declare it in your class' public header (Classpub.h) file.

If the flag RCLAS_FLAGSUBOFFSET is set, the attribute 'SubOffset' in the RClasRec structure will be set to the size of the parent class resource structure, and the offsets in the list of fields will be shifted by the same SubOffset value.

See also

RClasRec, NDRClas::Add.

Allocation/Deallocation

Default allocation method for all Open Interface classes. The default 'new' method for Open Interface classes calls this method in the C++ version. This member is only used through the RCLAS_CPP... macros.

OperatorNew

VoidPtr NDRClas::OperatorNew (PtrSizeVal size);

Default deallocation method for all Open Interface classes. The default 'delete' method for Open Interface classes calls this method in the C++ version. This member is only used through the RCLAS_CPP... macros.

OperatorDelete

void NDRClas::OperatorDelete (VoidPtr *obj*);

Member Functions

Get...

Functions that get the various fields of an RClas structure.

PtrSizeVal NDRClas::GetSizeOfRes (void);

CStr NDRClas::GetName (void);

PFldPtr NDRClas::GetFields (void);

RClasFlagsSet NDRClas::GetFlags(void);

RClasPtr NDRClas::GetParentClass (void);

ResPtr NDRClas::GetTemplate (void);

CStr NDRClas::GetModName (void);

RClasVersionVal NDRClas::GetVersion (void);

The NDRClas::Get... functions return the various fields of the given rclas structure. See class overview for a list of all fields of the structure.

See also

NDRClas::Set...

GetDefNfy

Get the default notification handler of an RClas.

ResNfyProc NDRClas::GetDefNfy (void);

The NDRClas::GetDefNfy procedure returns the default notification handler (the DefNotify field) of the given rclas structure.

See also

NDRClas::SetDefNfy, NDRClas::Get...

Querying Database of Resource Classes

FindByName

Returns a class by name.

static RClasPtr NDRClas::FindByName (CStr *name*);

NDRClas::FindByName returns the class whose name is classname, NULL if there is no such class. The set of resource classes already loaded in the program can be queried through the following calls:

GetFirst

Returns the first alphabetical resource class.

RClasPtr NDRClas::GetFirst (void);

The NDRClas::GetFirst function returns the first alphabetical resource class.

See also

NDRClas::GetNext, NDRClas::FindByName

GetNext

Returns the class that is alphabetically after the class specified.

RClasPtr NDRClas::GetNext (void);

NDRClas::GetNext returns the class that is alphabetically after class.

See also

NDRClas::GetFirst, NDRClas::FindByName.

Testing Inheritance

IsSubClassOf

Determines whether one class is a subclass of another.

BoolEnum NDRClas::IsSubClassOf (RClasCPtr parentclass);

NDRClas::IsSubClassOf returns BOOL_TRUE if childclass inherits (directly or indirectly) from parentclass.

See also

RClasRec

Set...

Functions that set the various fields of an RClas structure.

void NDRClas::SetSizeOfRes (PtrSizeVal size);**void NDRClas::SetName (Str name);****void NDRClas::SetFields (PFldPtr fields);****void NDRClas::SetFlags (RClasFlagsSet flags);****void NDRClas::SetParentClass (RClasPtr parent);****void NDRClas::SetVersion (RClasVersionVal version);****void NDRClas::SetModName (CStr modname);**

The NDRClas::Set... functions set the various fields of the given rclas structure. See RClasRec for a list of all the fields of the structure.

See also

NDRClas::Get...

SetDefNfy

Set the default notification handler for an RClas.

void NDRClas::SetDefNfy (ResNfyProc *defnfy*);

The NDRClas::SetDefNfy procedure sets the DefNotify field of the given rclas structure to the given defnfy handler.

See also

NDRClas::GetDefNfy, NDRClas::Set...

ProcessDefNfy

Trigger the default notification procedure on an instance.

void NDRClas::ProcessDefNfy (ResPtr *res*, ResNfyEnum *code*);

NDRClas::ProcessDefNfy calls the default notification procedure installed in rclas, with *res* and *code* as arguments. *res* must be an instance of rclas or of a subclass of rclas.

See also NDRClas::ProcessParentDefNfy

ProcessParentDefNfy

Trigger the parent default notification procedure on an instance.

void NDRClas::ProcessParentDefNfy (ResPtr *res*, ResNfyEnum *code*);

NDRClas::ProcessParentDefNfy first determines the parent class of rclas, and then calls the default notification procedure installed in the parent class, with *res* and *code* as arguments. *res* must be an instance of the parent class, of rclas, or of a subclass of rclas.

34 *Rect Class*

The Rect class implements basic definitions for points and rectangles used in various drawing operations.

Technical Summary

Point and rectangle structures are available that use 16 or 32 bit storage.

Rectangles defined by Origin and Extent

A 'rectangle' object is normally specified as an origin point 'Ori' and an extent 'Ext', and identifies a rectangular area on a 2D plane. A rectangle is valid only if Ext.x and Ext.y are positive or null.

The rectangle contains all the points (x, y) so that:

$$\text{Ori.x} \leq x < \text{Ori.x} + \text{Ext.x}$$

$$\text{Ori.y} \leq y < \text{Ori.y} + \text{Ext.y}$$

The rectangle contains exactly (Ext.x * Ext.y) points.

The line (x == Ori.x + Ext.x) is outside of the rectangle (off right).

The line (y == Ori.y + Ext.y) is outside of the rectangle (off bottom).

Rectangles defined by 'Beg' and 'End' coordinates

Instead of defining a rectangle by Origin (Ori.x, Ori.y) and Extent (Ext.x, Ext.y), we can also define a rectangle by its left, right, top, and bottom coordinates. The 'left' coordinate is easy to use (same as Ori.x), but the 'right' coordinate is naturally ambiguous: is it the right-most possible value (Ori.x + Ext.x - 1) or the first value outside of the rectangle (Ori.x + Ext.x)? Same problem for the 'bottom' coordinate.

So, instead of left/right and top/bottom coordinates, we will use BegX/EndX and BegY/EndY defined as:

$$\text{BegX} = \text{Ori.x} \quad \text{EndX} = \text{Ori.x} + \text{Ext.x}$$

$$\text{BegY} = \text{Ori.y} \quad \text{EndY} = \text{Ori.y} + \text{Ext.y}$$

There is an API to query and set each of these coordinates. The 'Set' calls are written so that each coordinate is independent from the other ones. For instance, NDRect::SetBegX will modify both 'Ori.x' and 'Ext.x' so that BegX becomes some given value while EndX remains unchanged.

Invalid rectangles

During some computations, a rectangle can become temporarily invalid, i.e. Ext.x or Ext.y becomes temporarily negative. There is an API to detect when a rectangle is invalid and an API to correct the Origin and Extent to make the rectangle valid again.

Native rectangle representation

This module also provides some API to convert between an OI rectangle and a native rectangle. You should also include the platform-specific header file (`xpub.h`, `mswpub.h`, or `pmpub.h`) to get the appropriate type declaration. In this module, the native rectangle type is represented by `'RectNat'`.

See also: `Draw` and `Rgn` class.

Point Functions

AbsDist

Returns the absolute distance between two points.

Int16 `NDPoint16::AbsDist(Point16Ptr p2);`

Int32 `NDPoint32::AbsDist(Point32Ptr p2);`

`POINT_ABSDIST` returns the absolute distance between “this” point and point `p2` (using the “L1” distance, which is defined as the max between the distance in X and the distance in Y).

ContainsPoint

ContainsPoint32

Determines whether a rectangle contains the point specified.

BoolEnum `NDRect16::ContainsPoint(Point16CPtr p);`

BoolEnum `NDRect32::ContainsPoint(Point32CPtr p);`

`NDRect::ContainsPoint` returns `BOOL_TRUE` if “this” rectangle contains point `p`. (There are 16-bit and 32-bit versions of this function.)

See also `NDRect::16IsEmpty`, `NDRect::32IsEmpty`.

SetOriExtXY

Sets all Origin/Extent coordinates.

void `NDRect16::SetOriExtXY(Int16 orix, Int16 oriy, Int16 extx, Int16 exty);`

void `NDRect32::SetOriExtXY(Int32 orix, Int32 oriy, Int32 extx, Int32 exty);`

`NDRect::SetOriExtXY` sets the x, y coordinates and extent.

IncOriExtXY

Increments all Origin/Extent coordinates.

void `NDRect16::IncOriExtXY(Int16 orix, Int16 oriy, Int16 extx, Int16 exty);`

void `NDRect32::IncOriExtXY(Int32 orix, Int32 oriy, Int32 extx, Int32 exty);`

IsEmpty

Determines whether a point is empty.

`EXT_IsEmpty` returns `BOOL_TRUE` if point is empty; otherwise, `BOOL_FALSE`.

SetXY

Sets point coordinates.

```
void NDPoint16::SetXY(Int16 xVal, Int16 yVal);
```

```
void NDPoint32::SetXY(Int32 xVal, Int32 yVal);
```

Sets the X and Y coordinates of point `p` to `xVal` and `yVal` respectively.

IncXY

Specifies a new point location.

```
void NDPoint16::IncXY(Int16 dx, Int16 dy);
```

```
void NDPoint32::IncXY(Int32 dx, Int32 dy);
```

Increments the X and Y coordinates by `dx` and `dy` respectively.

SetByPoints

Sets the coordinates of a rectangle to the points specified.

```
void NDRect16::SetByPoints(Point16CPtr beg, Point16CPtr end);
```

```
void NDRect32::SetByPoints(Point32CPtr beg, Point32CPtr end);
```

NDRect::SetByPoints sets the coordinates of rect with `beg` and `end` as opposite corners.

IsEmpty

Determines whether a rectangle is empty.

```
BoolEnum NDRect16::IsEmpty(void);
```

```
BoolEnum NDRect32::IsEmpty(void);
```

Returns `BOOL_TRUE` if rectangle `r` is empty (i.e., its extent is (0,0); otherwise, `BOOL_FALSE`.

Reset

Resets the coordinates of a rectangle to 0.

```
void NDRect16::Reset(void);
```

```
void NDRect32::Reset(void);
```

Makes `r` empty by changing its extent to (0,0).

Rect Functions

Equals

Returns `BOOL_TRUE` if `r1` and `r2` are identical.

```
BoolEnum NDRect16::Equals (Rect16CPtr r2);
```

```
BoolEnum NDRect32::Equals (Rect32CPtr r2);
```

IncludesNonEmptyRect

Returns `BOOL_TRUE` if `r1` is included in `r2` (assuming `r1` is not empty).

BoolEnum NDRect16::IncludesNonEmptyRect (Rect16CPtr r1);

BoolEnum NDRect32::IncludesNonEmptyRect (Rect32CPtr r1);

Copy

Copies a rectangle.

void NDRect16::Copy (Rect16CPtr src);

void NDRect32::Copy (Rect32CPtr src);

NDRect::Copy copies the src rectangle to the dest rectangle.

CopyResetOri

Copies 'src' into 'dst', but then sets dst->Ori to (0,0).

void NDRect16::CopyResetOri (Rect16CPtr src);

void NDRect32::CopyResetOri (Rect32CPtr src);

Intersects

Determines whether two rectangles intersect.

BoolEnum NDRect16::Intersects (Rect16CPtr r2);

BoolEnum NDRect32::Intersects (Rect32CPtr r2);

Returns `BOOL_TRUE` if 'r1' and 'r2' intersect; otherwise, `BOOL_FALSE`.

See also

NDRect::IsEmpty, NDRect::IsEmpty32, NDRect::IncludesRect.

IncludesRect

Determines whether a rectangle contains the rectangle specified.

BoolEnum NDRect16::IncludesRect (Rect16CPtr r1);

BoolEnum NDRect32::IncludesRect (Rect32CPtr r1);

Returns `BOOL_TRUE` if 'r1' is included in 'r2' ('r1' can be empty).

Union

Determines the union of two rectangles.

void NDRect16::Union (Rect16CPtr src);

void NDRect32::Union (Rect32CPtr src);

Sets 'dst' to the union of 'dst' and 'src'.

Intersection

void NDRect16::Intersection (Rect16CPtr src);

void NDRect32::Intersection (Rect32CPtr src);

Sets 'dst' to the intersection of 'dst' and 'src'.

MakeFit

Repositions a rectangle so that it is contained within the rectangle specified.

void NDRect16::MakeFit (Rect16CPtr out);

void NDRect32::MakeFit (Rect32CPtr out);

MakeFit repositions in so that it is contained within out.

MoveInside

Moves one rectangle inside another.

void NDRect16::MoveInside (Rect16CPtr out);

void NDRect32::MoveInside (Rect32CPtr out);

MoveInside moves rect so that it is inside outrect.

See also

NDRect::ContainsPoint, NDRect::IncludesRect, NDRect::MakeFit.

IsValid

Determines whether a rectangle has valid coordinates.

BoolEnum NDRect16::IsValid(void);

BoolEnum NDRect32::IsValid(void);

IsValid determines whether a rectangle r has valid coordinates.

MakeValid

Changes the coordinates of the rectangle specified to make them valid.

void NDRect16::MakeValid(void);

void NDRect32::MakeValid(void);

This function makes the rect passed a valid rectangle. This means that if either ext coordinate is zero, it changed to 1 and if either ext coordinate is negative it is subtracted from the origin and then readjusted.

See also

NDRect::IsValid

Rectangles Defined by Origin and Extent

Get...

Int16 NDRect16::GetOriX(void);

Int16 NDRect16::GetOriY(void);

Int16 NDRect16::GetExtX(void);

Int16 NDRect16::GetExtY(void);

Int32 NDRect32::GetOriX(void);

Int32 NDRect32::GetOriY(void);

Int32 NDRect32::GetExtX(void);

Int32 NDRect32::GetExtY(void);

Gets one Origin/Extent coordinate.

Set...

```
void NDRect16::SetOriX(Int16 val);  
void NDRect16::SetOriY(Int16 val);  
void NDRect16::SetExtX(Int16 val);  
void NDRect16::SetExtY(Int16 val);  
void NDRect32::SetOriX(Int32 val);  
void NDRect32::SetOriY(Int32 val);  
void NDRect32::SetExtX(Int32 val);  
void NDRect32::SetExtY(Int32 val);
```

Sets one Origin/Extent coordinate.

Rectangles Defined by Beginning and End

Get...

```
Int16 NDRect16::GetOriX(void);  
Int16 NDRect16::GetOriY(void);  
Int16 NDRect16::GetExtX(void);  
Int16 NDRect16::GetExtY(void);  
Int32 NDRect32::GetOriX(void);  
Int32 NDRect32::GetOriY(void);  
Int32 NDRect32::GetExtX(void);  
Int32 NDRect32::GetExtY(void);
```

Gets one Begin/End coordinate.

Set...

```
void NDRect16::SetOriX(Int16 val);  
void NDRect16::SetOriY(Int16 val);  
void NDRect16::SetExtX(Int16 val);  
void NDRect16::SetExtY(Int16 val);  
void NDRect32::SetOriX(Int32 val);  
void NDRect32::SetOriY(Int32 val);  
void NDRect32::SetExtX(Int32 val);  
void NDRect32::SetExtY(Int32 val);
```

Sets one Begin/End coordinate.

35 *Res Class*

The Res class implements the Open Interface resource manager. It defines the core resource class from which all resource classes will be derived.

Technical Summary

Persistent objects

A resource is a potentially 'persistent object', which means an object which can be loaded from and saved to an external file and thus 'persist' after an application has terminated, in contrast with 'volatile objects' which exist only (usually in RAM) during the execution of an application.

Some object oriented environments (Objective-C, Eiffel) provide an 'automatic filer' which can load and save whole graphs of objects. Our resource manager is different from such filers for several reasons:

- only resources can be saved and loaded, non-resource objects cannot be saved because the resource manager does not have the meta-information (description of fields) needed to save these objects.-resources are named (except resources which are created through calls to RES_Create instead of being loaded from a resource database).
- only certain fields (persistent fields) of resources are saved and loaded. The other fields (volatile fields) only exist at run-time.
- the resource manager can only handle Direct Acyclic Graphs (DAG) of objects. This is a limitation (we cannot save objects which refer to each other) but it allows us to implement reference counting on resources so that we can automatically free resources once they are not in used by anyone.

Resources can be represented in three forms:

- in RAM, at run time. Then we have a 'resource object'.
- on disk, in a text format. This is the 'rc' format.
- on disk, in a binary format. This is the 'dat' format.

The 'rc' format is provided so that you can edit your resource definitions directly with a text editor, keep them under a source control system and port them to a different system (the rc files are portable, the dat files are not). A resource can be 'compiled' from rc format (RC --> RAM). A resource can be 'output' to an rc file (RAM --> RC).

The 'dat' format is the format which will be used to load and save resources at run time. It is efficiently indexed so that resources can be loaded faster than they would be from an rc file. A resource can be 'loaded' from a dat file (DAT --> RAM). A resource can be 'saved' to a dat file (RAM --> DAT).

Responder objects

A resource is also an object to which notifications (or messages) can be sent and which provides ways for programs to customize the response to these

notifications. Whenever a program needs that kind of mechanism, it will need to create instances of the Res class to have somebody to notify to.

Hierarchical Organization

Resources are organized hierarchically:

libraries --> modules --> resources --> subresources --> subsubresources ..

The organization of software in modules and libraries of modules is rather conventional. Our resource organization parallels this software organization. The idea is that a code module may need some resources. It is thus natural to group the resources by module. Every module will have its own rc file (which plays the same role as the C source file for the module, except that a module sometimes has several C source files associated with it whereas we are limited to one rc file per module). The compilation of the rc files of all the modules belonging to the same library produces a single dat file (which plays the same role as the object library or shared library produced after compilation of the C source files).

To summarize:

- rc file: one per module.
- dat file: one per library.

A given module can contain several resources. Some resources are 'flat objects' and do not have subresources (String, Font, Cursor, ...), others have subresources, i.e. windows have their widgets as subresources, and with panels, we can get subsub...resources of any depth.

Resource naming

The naming of resources is based on the hierarchical organization. Every module must be assigned a unique name. If you want to follow strictly our naming conventions, the name should be less than 5 characters so that we can build derived filenames (with pub.h suffix) of less than 8+3 characters (the DOS limit).

Then, the name of top level resources (direct children of a module) has the following format:

"Module.ResName"

where "Module" is the module name and "ResName" is the name of the resource.

The naming of subresources is very straightforward:

"Module.ResName.SubResName.SubSubResName"

A priori, names can consist of any characters excluding '.'. We nevertheless recommend that you use only letters, numbers and underscores and that you start your names by a letter because resource names are sometimes used to generate procedure names and thus must conform with the syntactic constraints on procedure names.

Object Oriented Organization

Resources are also organized in classes (see any good book on Object Oriented Programming, Smalltalk Vol1 being probably the best

introduction to the subject). The classes themselves are organized hierarchically, subclasses inheriting from their parent class. Our object oriented scheme makes it simple to implement single inheritance but would also allow multiple inheritance (actually used internally in one place). This scheme will be fully documented later.

At the top level of this class hierarchy is the "Res" class from which all resource classes will inherit. Quite a few classes (StrL, StrR, Font, Color, Curs, ..., Wgt) inherit directly from Res. Most toolkit classes (TBut, TArea, Sb, Panel, ...) inherit from Wgt. Complex widgets which scroll a document (TEd, LBox, Brows, ...) inherit from the SArea class which itself inherits from Panel.

It is very important to distinguish the hierarchy of resources (module --> window --> panel --> widget) from the hierarchy of resource classes (res --> wgt --> panel --> sarea --> lbox).

Attached Versus Detached Resources

Open Interface distinguishes two types of resources: - resources which should be shared by all the objects referencing them.

For example, it is very interesting to have font or color resources which are shared by all the widgets using them. Everytime we load a font to initialize the Font field of a widget, we should load a shared font resource.

- Resources which should not be shared. For example, windows should not be shared. An application which allows to bring several instances of the same window (i.e. a text processor in which you can edit several files, each file in its own window) needs to load several instances of the same window resource instead of sharing a single window.

If you load a resource as 'attached', the resource manager will keep track of that resource and subsequent 'load' calls on the same resource name will return the same resource pointer and increment the reference count on the resource. The resource will be shared.

If you load a resource as 'detached', the resource manager will 'detach' the resource and a new resource will be allocated on the next 'load' call. The next 'load' call on the same resource name will return a different resource pointer. The resource will not be shared.

Reference Counting

The reference count of a resource keeps track of how many times the resource is shared (how many time it has been 'loaded'). It is always 1 for detached resources (except if they were attached before being detached with is a rather dangerous operation) but will be ≥ 1 for shared attached resources.

Most of the RES calls which return resource pointers might load resources. All these calls set the reference count of the resource to 1 if the resource was not loaded previously, otherwise they increment the reference count by one. Once you do not need the resource pointer any more, you should call NDRes::Release which will decrease the reference count and free the resource once the reference count reaches 0.

This is very similar to what you have to do with the memory manager. Every time you allocate a new pointer, you should free it when you do not need it any more.

With the resource manager, every resource pointer that you get through a call to the Res API should be released with a call to `NDRes::Release` when it is not needed any more. If every client of the resource manager follows that policy, shared resources will be freed automatically when they are not used by any client (because the resource manager maintains reference counts).

Constructing and Destructing a Resource

In C++, we want to provide our users with the following constructors:

```
myres = new MyRes; copy of the class template
myres = new MyRes(fullname) loading by name from the resource
myres = new MyRes(mod, res) file.
myres = new MyRes(fullname, rrtca) same as above with
myres = new MyRes(mod, res, rrtca) runtime info for the sub
widgets.
```

We also need to set things up so that the resource manager can properly create and delete objects in the Res subclasses. To achieve this, we record a "New" and a "Delete" proc in the RClas structure. In the body of these procs, we create and delete the objects with the C++ new and delete operators, so that they are always properly constructed. But we need an additional constructor because we need to pass some information that was prepared by the resource manager through the New proc (the default constructor would clone the template which is not what we want). The default constructor needs to be implemented in a special way. It must pass the current RClas info to the Res level so that we know what template to use to initialize the persistent fields.

We have two different cases:

When we pass the name(s) explicitly, the Res level does the lookup and finds the persistent description. When we do not pass the name(s) explicitly, we have to pass some information from which we can find the persistent description.

Also, we already have "Construct" and "Destruct" methods defined for C and attached to the RClas object.

The new strategy is the following:

Each class defines two constructors:

```
MyRes::MyRes(CStr mod, CStr res = NULL,
             ResRunTimeClassArrayPtr rrtca = NULL)
MyRes::MyRes(RClasPtr, RClasCreateCPtr = NULL)
```

In C, we only allow heap allocation of resources and the resources will be allocated by calls such as `RES_Create`, `RES_Load`, etc which are implemented at the Res level.

In C++, we allow all types of allocation (heap, stack, member). Programmers should use the C++ "new" operator to allocate on the heap but they may also use routines such as `RES_Create` or `NDRes::Load` (use is discouraged, though).

The OI resource classes are registered through `NDRClas::Register`. They MUST provide a `New` and a `Delete` method (CHANGE). The `Construct` and `Destruct` `RClas` methods will not be used any more for OI classes.

Customers may still register C classes with `Construct` and `Destruct` methods but in this case their classes will be "C" only and their C constructors and destructors will be called by the default `New` a `Delete` `RClas` methods.

When we compile the OI classes in C++, the constructors are set up so that everything is constructed through the standard C++ constructor cascade.

When we compile the OI classes in C, the C version of the `New` procedure takes care of the construction. It first calls the parent class' `New` method and then performs its class specific construction.

We also need a special hack to handle stack and member allocation. The resource manager automatically deletes resources when they are not referenced any more (windows being terminated and their widgets). This did not raise any special problem as long as the resources were always allocated on the heap. But now that we allow stack and member allocation in C++, we have to let the resource manager know that certain resources should be destructed but not deleted.

To achieve this, we override the C++ `new` and `delete` operators so that we can easily find out whether an object has been allocated on the stack or on the heap. The overridden `delete` operator does not free the pointer if the object was not allocated on the heap.

We should not assume that resources are filled with zeros after allocation. We cannot either zero them out with `NDPtr::Clear` because we would erase virtual function pointers. So, we must explicitly initialize all the fields in every class' constructor.

Scope of the documented API

For now, the supported and documented API has been limited to routines which load resources and query the resource manager. We have voluntarily excluded routines which modify the organization of resources (renaming resources, deleting resources, listing resources...). We are very interested to know what your needs are (if any) before disclosing other aspects of this API. Also, you will be able to absorb rather quickly this relatively simple API instead of being lost in a much larger API, most of which would not be relevant to most applications.

Since `Res` is the parent of all resources, all resources possess the following fields.

Field	Description
<code>ResClass</code>	Pointer to the <code>RClasRec</code> structure, which describes the class to which the resource belongs.
<code>ResInitialized</code>	Indicates whether volatile fields have been initialized.
<code>ClientData</code>	32 bit storage for any data or pointer you wish to associate with the resource.
<code>NfyData</code>	32 bit storage for information used by some notifications.
<code>Notify</code>	Pointer to the class-specific notification procedure.

See Also:

RLib, Wgt classes.

Creating and Disposing

Clone

ResPtr NDRes::Clone(BoolEnum deep);

Creates a resource with all the persistent fields copied from 'sourceRes'. The new resource is created detached. To load multiple instances of the same resource from the resource file, you should use NDRes::LoadDetach instead of NDRes::Clone on an existing resource.

The second argument is a boolean which indicates whether we want a deep copy (cloning all descendant resources too) or not.

Release

Deallocates persistent resource fields.

void NDRes::Release (void);

NDRes::Release deallocates the persistent fields of resource, being careful about shared fields. It also decrements the reference count of the resource. If the reference count reaches 0, the volatile fields are freed (by sending a NDRes::NfyEnd notification) if the ResInitialized flag was set. Then the persistent fields and the resource structure itself are freed. NDRes::Release will also be called on all children of the resource and on all resources referenced by persistent fields of the resource.

This call frees the resource in RAM. It does not affect the .dat file.

See also

NDRes::Create, NDRes::Clone

Class

Returns a pointer to the Res class.

static RClasPtr NDRes::Class (void);

NDRes::Class() returns the pointer to the Res class, the root of the hierarchy of resource classes. This call is useful to initialize the ParentClass field of a new resource class.

See also

RClasRec

Use

Increments the reference count of a resource.

void NDRes::Use (void);

If you are storing pointers to shared resources in objects which have a relatively long life-span, you should increase the reference count of the

shared resources to ensure that these resources will not be released before the objects which reference them. Then, you should call `NDRes::Release` on these resources when you release the objects which reference them.

Note: The calls which "load" resources increment the reference count, so, usually, you will not need to call `NDRes::Use` after such calls.

Saving To a Resource Database

SaveDat

Saves attached resources to library database (.DAT) file.

void NDRes::SaveDat (void);

`NDRes::SaveDat` saves resource to a library database (.DAT) file.

Output to a Text Resource File

FilenameOutputRc

Output to text resource file.

void NDRes::FilenameOutputRc (CStr filename);

`NDRes::FilenameOutputRc` outputs resource to the text file specified by filename.

Resource Library Initialization

You can use Open Interface to build non window based applications (terminal oriented or batch mode).

You will get the resource manager (to load text messages), the error handling mechanism and all the 'Core' level utilities (i.e. Str, VStr, Array).

If your application does not use any windows you can initialize it with `NDRes::LibInit` instead of the usual `NDRes::LibInit` from the GW module. This will give you a faster startup time and will also greatly reduce the size of the executable if you are statically linked with the Open Interface libraries (you only need to link with `ndres` and `ndcore`).

Note: You should not use the `APP_Xxx` calls in case you initialize at the Res level instead of the Gw level. Also, applications which only use the Res level are not portable to environments such as Macintosh and MS/Windows where applications must be window based.

LibInstall

Installs the Res library, making all Open Interface libraries available, except Genwin.

static void NDRes::LibInstall (void);

`NDRes::LibInstall` installs the Res library.

See also

`NDRes::LibInit`, `NDRes::LibLoadInit`, `GW_LibInstall`, `TKIT_LibInstall`

LibLoadInit

Loads and initializes the Res library, making all Open Interface libraries available, except Genwin.

static void NDRes::LibLoadInit (void);

`NDRes::LibLoadInit` loads and initializes the Res library.

See also

`NDRes::LibInstall`, `NDRes::LibInit`, `GW_LibLoadInit`, `TKIT_LibLoadInit`

LibExit

Exits the Res library, making all Open Interface libraries unavailable.

static void NDRes::LibExit (void);

`NDRes::LibExit` exits the RES library.

See also `NDRes::LibInit`, `NDRes::LibInstall`, `NDRes::LibLoadInit`, `GW_LibExit`, `TKIT_LibExit`

LibInit

Installs, loads, and initializes the Res library, making all Open Interface libraries available, except Genwin.

static void NDRes::LibInit (void);

`NDRes::LibInit` initializes the Res library.

If your application does not use any windows, you can initialize it with `NDRes::LibInit` instead of the usual `GW_LibInit`. This will give you a faster startup time and will also greatly reduce the size of the executable if you are statically linked with the Open Interface libraries (you only need to link with `ndres` and `ndcore`).

You should not use the `APP_xxx` calls in case you initialize at the Res level instead of the GW level. Also, applications which only use the Res level are not portable to environments such as Macintosh and MS/Windows where applications must be window based.

See also

`NDRes::LibInstall`, `NDRes::LibLoadInit`, `NDRes::LibExit`, `GW_LibInit`, `TKIT_LibInit`

Loading and Finding Resources

Resources may be stored persistently. Standard applications will use resources stored in resource databases, by loading them, and then customizing their behavior through the responding mechanism.

Normally, you will load all resources except windows and menus through `NDRes::LoadInit`. This way, resources such as fonts, colors, cursors, ... will be shared by all the clients which use them.

Windows should be loaded through the `NDWin::LoadInit` and menus through `NDMnu::LoadInit`.

Usually, you do not load the widgets of a window yourself, they are loaded automatically when you load the window through the corresponding `NDWin::LoadInit` or `NDWin::Load` call.

All these routines signal a failure in case the resource cannot be loaded or does not exist. They NEVER return `NULL`. These routines increment the reference count of the resource being loaded.

LoadByFullName

Loads a resource using a single parameter.

static ResPtr NDRes::LoadByFullName (CStr modres);

`NDRes::LoadByFullName` loads and returns the resource identified by `modres`, which is a string name in the format:

```
ClassName.ResName
```

where `ResName` is the name of a resource and `ClassName` is the name of the class that contains `ResName`.

`NDRes::LoadByFullName` loads the resource without initializing it. Therefore, you should ordinarily use `NDRes::LoadInit` or `NDRes::LoadInitDetach` rather than `NDRes::LoadByFullName` to load your applications.

`NDRes::LoadByFullName` never returns `NULL`; instead, it signals a failure if resource does not exist or cannot be loaded

See also `NDRes::Load`, `NDRes::LoadInit`, `NDRes::LoadInitDetach`, `NDRes::FindByFullName`

Load

Loads a resource using two parameters.

static ResPtr NDRes::Load (CStr module, CStr resource);

`NDRes::Load` loads and returns resource, which is resource contained in class.

`NDRes::Load` loads an attached resource given its module name `'moduleName'` and its resource name `'resName'` (relative to the module name).

Calling `NDRes::Load` twice will return the SAME resource pointer.

`NDRes::LoadByFullName` and `RES_Load` do not initialize the volatile fields of the resource. These calls can be used instead of the `LoadInit` calls in the following situations:

- If you want the `NDRes::NfyInit` notification to be processed by a notification procedure other than the default class notification procedure. The problem with `NDRes::LoadInit` is that it does not give you a chance to install your own notification procedure in the resource or one of its children before sending the `RES_NfyInit` notification. If you really want to process `NDRes::NfyInit` in a special way (i.e. for a custom

widget), you should first load the resource, then install the notification procedure(s) and then initialize it with `NDRes::SendNfyInit`. Actually, if you want to perform this type of initialization on a window, you should use the corresponding `NDWin::Load` and `NDWin::Init` (see `winpub.h`).

- If your application does not need to initialize the volatile fields (this is the case of `rescomp` but won't be the case of many other applications).

See also

`NDRes::Load`, `NDRes::LoadInit`, `NDRes::LoadInitDetach`

LoadDetach

Loads a detached resource.

static ResPtr NDRes::LoadDetach (CStr module, CStr resource);

The `NDRes::LoadDetach` function loads and returns a detached resource. If the resource has already been loaded attached, the new resource being loaded in `Detach` mode is cloned from the attached resource in memory rather and is not reloaded from the `.dat` file.

This function never returns `NULL`; it signals a failure or a non-existent resource or a resource that cannot be loaded.

See also

`NDRes::LoadInitDetach`, `NDRes::Load`, `NDRes::LoadInit`, `NDRes::Load`, `NDRes::FindByFullName`, `NDWin::LoadInit`, `NDMnu::LoadInit`

LoadInit

Loads and initializes an attached resource.

static ResPtr NDRes::LoadInit (CStr module, CStr resource);

The `NDRes::LoadInit` function loads and initializes the attached resource specified by class and resource. It returns the resource pointer. If the resource does not exist, a failure will be signalled without terminating the application.

Calling `NDRes::LoadInit` twice returns the same resource pointer, so use this function for resources that will need to be accessed several times and shared among client applications, for example: fonts, colors, cursors. (`NDWin::LoadInit`, by contrast, returns a separately allocated pointer to resource, so is less suitable for shared resources.)

This routine does not return `NULL`. It signals a failure if the resource does not exist or cannot be loaded.

See also

`NDRes::LoadInitDetach`, `NDRes::FindByFullName`, `NDRes::Load`, `NDRes::LoadDetach`, `NDWin::LoadInit`, `NDMnu::LoadInit`.

LoadInitDetach

Loads and initializes a detached resource.

static ResPtr NDRes::LoadInitDetach (CStr module, CStr resource);

The `NDRes::LoadInitDetach` function loads, initializes, and returns a detached resource.

Each time you call `NDRes::LoadInitDetach`, a new resource pointer is allocated. Therefore, the function is appropriate for resources that are not shared, for example: windows and popups. Windows and popups are usually loaded through `NDWin::LoadInit` and `NDMnu::LoadInit`, which both call `NDRes::LoadInitDetach`.

This function never returns `NULL`; it signals a failure for a non-existent resource or a resource that cannot be loaded.

See also

`NDRes::LoadDetach`, `NDRes::Load`, `NDRes::LoadInit`, `NDRes::Load`, `NDRes::FindByFullName`, `NDWin::LoadInit`, `NDMnu::LoadInit`

LoadChildren

Loads all of the children resources for a resource.

void NDRes::LoadChildren (void);

The `NDRes::LoadChildren` loads all of the children resources of the specified resource.

See also

`NDRes::GetNumChildren`, `NDRes::GetNthChild`

FindByFullName

Loads the attached resource specified, returning `NULL` if the resource does not exist.

static ResPtr NDRes::FindByFullName (CStr Mod.Res);

`NDRes::FindByFullName` loads and returns the resource identified by `Mod.Res`, which is a string name in the format:

`ClassName.ResName`

where `ResName` is the name of a resource and `ClassName` is the name of the class that contains `ResName`.

Like `NDRes::Load`, this function loads a resource without initializing it. Unlike `NDRes::Load`, it returns `NULL` if the resource does not exist.

Use this function to check whether a resource exists, not as a standard means of loading resources.

See also

`NDRes::Load`

Find

Loads a resource by class name and resource name, returning `NULL` if the resource does not exist.

static ResPtr NDRes::Find (CStr modname, CStr resname);

NDRes::Find loads the resource specified by modname and resname, returning NULL if the resource specified does not exist.

See also

NDRes::Load

Accessing the Name of a Resource

IsNamed

Determines whether a resource has a name or not.

BoolEnum NDRes::IsNamed (void);

NDRes::IsNamed returns `BOOL_TRUE` if the resource has a name; otherwise, it returns `BOOL_FALSE`. If a resource was created dynamically with `NDRes::Create`, it will not have a name.

See also

NDRes::GetName, NDRes::QueryFullName

GetName

Returns the name of a resource as a NULL-terminated string.

CStr NDRes::GetName (void);

NDRes::GetName returns the name of res as a NULL-terminated string. If the resource is unnamed, "" is returned.

The name of resource is the name you assigned to it in Open Editor or, if you use `NDRes::Create` to create the widget, a default name.

See also

NDRes::QueryFullName, NDRes::Create

QueryFullName

Determines the full name of a resource.

void NDRes::QueryFullName (Str name, StrIVal length);

NDRes::QueryFullName queries for the full name of res, and the result is received in name, which is a buffer whose size is specified by length.

The full name of resource is a Str that follows the format:

```
"ClassName.ParentRes.Res"
```

where `ClassName` is the name of the class in which the resource is used, `ParentRes` is the name of the resource containing the widget (for example, a window resource), and `Res` is the name you have given to the resource. For example: `Writer.Win.TbutOk`.

The fullname appears to the right of `Name:` in the resource's entry in a RC file.

Accessing Client Data of a Resource

A user defined ClientData can be attached to each resource. This can be used to store application related information with each resource. Open Interface does not use the ClientData for internal purposes.

SetClientData

Sets the specified client data of a resource.

void NRes::SetClientData (ClientPtr data);

NRes::SetClientData sets the data passed into the resource.

GetClientData

Returns the client data of a resource.

ClientPtr NRes::GetClientData (void);

NRes::GetClientData retrieves the client data for the resource specified.

Accessing Children of a Resource

GetNumChildren

Returns the total number of child resources belonging to a resource.

ArrayIVal NRes::GetNumChildren (void);

The NRes::GetNumChildren function returns the total number of children belonging to res as an integer.

See also

NRes::GetNthChild, NRes::LoadChildren

GetNthChild

Returns the child resource specified for a resource.

ResPtr NRes::GetNthChild (ArrayIVal child);

The NRes::GetNthChild returns a pointer to the child resource specified by its number.

Note that the children of the resource must already be loaded.

NRes::GetNthChild does not increment the reference count of the child resource returned.

See also

NRes::GetNumChildren, NRes::LoadChildren

Accessing the Class of A Resource

GetClass

Returns the class of the resource specified.

RClasCPtr NDRes::GetClass (void);

NDRes::GetClass returns the class of the resource specified. The same pointer is returned if called on different instances of the same class.

See also

NDRClas::GetFirst, NDRClas::GetNext, NDRClas::FindByName

InheritsFrom

Determines whether one class inherits from another class.

BoolEnum NDRes::InheritsFrom (RClasCPtr parentclass);

NDRes::InheritsFrom returns `BOOL_TRUE` if childclass inherits (directly or indirectly) from parentclass.

See also

NDRes::CheckClass, NDRClas::IsSubClassOf

Resource States

IsInitialized

Determines whether a resource has been initialized already.

BoolEnum NDRes::IsInitialized (void);

NDRes::IsInitialized returns `BOOL_TRUE` if the resource has been initialized; otherwise, it returns `BOOL_FALSE`.

Resource Notifications

Nfy...

Defines class notification codes.

Notification codes for classes. Similar enumerated types exist for all classes in the form `SubResNfyEnum`, where `SubRes` is the short name of a particular class: for example, `TButNfyEnum` for the `TBut` class.

These notifications are described below:

Identifier	When Sent	Action
<code>NDRes::NfyInit</code>	Persistent fields have been initialized.	Initialize volatile fields of resource.

<code>NDRes::NfyEnd</code>	Resource is no longer needed.	Deallocate volatile fields of resource that you allocated with <code>NfyInit</code> , and, if the value is not overwritten by an new <code>NFYINIT</code> , you should <code>RESET</code> the volatile fields to <code>NULL</code> .
<code>NDRes::NfyReset</code>	Persistent fields have been modified.	Reinitialize volatile fields, that is, deallocate and reallocate them according to current state of the persistent fields.
<code>NDRes::NfyReMmgr</code>	Not documented yet.	
<code>NDRes::NfyCtrlData</code>	Not documented yet.	
<code>NDRes::NfySetData</code>	Not documented yet.	
<code>NDRes::NfyGetData</code>	Not documented yet.	
<code>NDRes::NfyDestroyed</code>		Sent just before the resource is destructed by Open Interface
<code>NDRes::NfyDeallocate</code>		Sent when the resource is about to get deallocated. You should remove any existing reference to the resource. By default, the resource is deallocated.

Whenever a resource is initialized, reset, or freed, the resource manager calls the notification procedure associated with the resource, passing the resource as the first argument and the appropriate `ResNfyEnum` code as the second argument. Usually, the notification procedure responds by calling the default notification procedure of the class to which the resource belongs (`ClassName::DefNfy`) with the same two arguments.

The default notification procedure initializes and allocates the volatile fields of the resource when called with `NDRes::NfyInit` and frees the allocated fields on receipt of a `NDRes::NfyEnd`. `NDRes::NfyReset` is usually sent when the persistent fields have been modified, in which case the volatile fields need to be deallocated and reallocated according to the current state of the persistent fields.

`NDRes::NfyEnd` deallocates the volatile fields of a resource that you allocated with `NDRes::NfyInit`, and if the value is not overwritten by an new `NDRes::NfyInit`, you should `RESET` the volatile fields to `NULL`.

The notification codes of a subclass are always a superset of the codes defined by its parent class, as part of the class's inheritance mechanism. Thus, `PanelNfyEnum` inherits codes from `WgtNfyEnum`, which in turn inherits codes from `ResNfyEnum`. Since `Res` is the parent of all resource classes, all resource classes inherit the `ResNfyEnum` notification codes.

See also `NDRes::NfyInherit`, `NDRes::SendNfyInit`, `PANEL_SetSubNfyProc`

ResNfyProc

Pointer to a resource notification procedure.

Pointer to a resource notification procedure. The notification procedure receives a subclass of `ResPtr` and a subclass of `ResNfyEnum` as arguments

See also `NDRes::SetNfyProc`, `NDRes::GetNfyProc`, `RClasRec`

DefNfy

Default notification handler for a resource class.

void NDRes::DefNfy (ResNfyEnum *notif*);

The NDRes::DefNfy procedure is the default notification handler for all resource classes. All resource classes, as subclasses of Res, use the NDRes::DefNfy default handler as part of their default response to their notification codes. Class-specific default notification handlers are in the form SubRes::DefNfy, where SubRes is the short name of the resource: for example, NDLBox::DefNfy.

All resource notification routines should include a call to NDRes::DefNfy to initialize notification codes the handler does not process itself.

See also NDRes::SendNfyInit

SetNfyProc

Sets a client notification routine for a resource.

void NDRes::SetNfyProc (ResNfyProc *nfyproc*);

The NDRes::SetNfyProc macro overrides the default notification procedure and installs *nfyproc* as the client notification routine for resource. The client application typically performs this task immediately after loading and initializing the resource.

See also NDRes::GetNfyProc, ResNfyProc

GetNfyProc

Returns the currently installed notification routine for a resource.

ResNfyProc NDRes::GetNfyProc (void);

NDRes::GetNfyProc retrieves the previously installed default notification procedure.

SetNfyHandler

Installs "proc" to process the "nfy" notification messages sent to "res". "proc" will be called with the resource, the notification code "nfy" and the *nfyData* corresponding to the notification 'nfy' as arguments.

"proc" will usually process the notification message. It can at any point invoke the default behaviour for the class.

void NDRes::SetNfyHandler(ResNfyEnum *nfy*, ResNfyHandlerProc *proc*);**GetNfyHandlerProc**

Returns the handler procedure installed for 'res' to process the 'nfy' message.

If no handler has previously been installed using NDRes::SetNfyHandlerProc(), NULL is returned.

ResNfyHandlerProc NDRes::GetNfyHandlerProc(ResNfyEnum *nfy*);**RemoveNfyHandler**

Remove the handler installed to process the 'nfy' message to 'res'.

Removing a handler at the instance level has for effect of letting the class handler process the message.

void NDRes::RemoveNfyHandler (ResNfyEnum nfy);

SetNfyHandlerClientData

Associates 'data' with the call-back defined for the resource and the notification 'code'. 'data' can later be retrieved using NDRes::GetNfyHandlerClientData if needed.

void NDRes::SetNfyHandlerClientData (ResNfyEnum nfy, ClientPtr data);

GetNfyHandlerClientData

Returns the ClientData installed for 'res' to process the 'nfy' message.

ClientPtr NDRes::GetNfyHandlerClientData (ResNfyEnum nfy);

See also NDRes::SetNfyProc, ResNfyProc

Sending Notifications

Sending versus Posting

Object Oriented system generally define two different ways of communicating messages to objects. They either:

- send the message synchronously, meaning that the responder code of the object is activated and terminates the handling of the message before the senders code actually returns from the 'send' call (Send),
- post the message asynchronously, meaning that the message is only put at the receivers' disposal before the senders code returns from the 'send' call, with no guarantee that the receiver actually processed the message.

The first method is far simpler to code and more efficient since it directly translates into the activation of methods in the receivers object (function calls).

At the Res level, Open Interface only provides synchronous sends. The Wgt class (see wgtpub.h) offers a more sophisticated Post mechanism (Recal notifications) which may be combined with other event mechanisms as described in eventpub.h giving the program a lot of flexibility.

Synchronous notifications work the following way:

- the sender makes a call to send a notification with or without data
- the call will activate the notification procedure installed for the receiver resource
- the corresponding virtual member function gets activated (for instance, sending NDRes::NfyInit to a Res instance will activate the corresponding NfyInit virtual method)
- the function is responsible for handling the notification, it may for instance choose to activate the virtual member function installed at the parent class's level.

Sending A Notification With Data

In many cases, a single notification code cannot convey enough information from the sender to the receiver. Open Interface provides another higher

level call to send a notification to a resource with a program defined `ClientPtr` that the receiver can query, and modify.

Typically, the sender will use a fragment of code like:

```

        receiver->SendNfyData(nfy, data);
and the receiver, in its notification procedure will use:
.../...
case nfy: {
    ClientPtr data;
    data = res->GetNfyData();
    < do something with it >
}
break;
.../...

```

SendNfy

Ends a notification to the notification procedure of a resource.

void NRes::SendNfy (ResNfyEnum *notif*);

`NRes::SendNfy` notifies the clients of resource. This macro is usually called indirectly through the individual `NDClass::SendNfy` routines, where `Class` is the name of a resource class, for example, `NDWin::SendNfy`.

Ordinarily you will not use `NRes::SendNfy` directly, unless you are creating a custom widget. If so, you may call this routine to trap notifications.

LockedSendNfyData

Notifies resource clients and sends data specified.

void NRes::LockedSendNfyData (ResNfyEnum *notif*, ClientPtr *data*);

`NRes::LockedSendNfyData` notifies the clients of resource. This macro is usually called indirectly through the individual `CLASS::SendNfyData` routines, where `CLASS` is the name of a resource class, for example, `NDWin::SendNfy`.

Ordinarily you will not use `NRes::LockedSendNfyData` directly, unless you are creating a custom widget. If so, you may call this routine to trap notifications.

See also

`NRes::SendNfyData`, `NRes::GetNfyData`, `NRes::SetNfyData`

SendNfyData

Notifies resource clients and sends data specified.

void NRes::SendNfyData (ResNfyEnum *notif*, ClientPtr *data*);

`NRes::SendNfyData` notifies the clients of resource. This macro is usually called indirectly through the individual `CLASS::SendNfyData` routines, where `CLASS` is the name of a resource class, for example, `NDWin::SendNfy`.

Ordinarily you will not use `NRes::SendNfyData` directly, unless you are creating a custom widget. If so, you may call this routine to trap notifications.

See also

`NDRes::LockedSendNfyData`, `NDRes::SetNfyData`, `NDRes::GetNfyData`

`GetNfyData`

Returns the notify data of a resource.

ClientPtr NDRes::GetNfyData (void);

`NDRes::GetNfyData` retrieves the notify data for the resource specified.

`SendNfyInit`

Sends a `NDRes::NfyInit` to the resource specified.

void NDRes::SendNfyInit (void);

`NDRes::SendNfyInit` sends a `NDRes::NfyInit` notification to the resource if it has not already been initialized (the `ResInitialized` flag was not already set). This macro is usually called indirectly through the individual `CLASS::SendNfyInit` routines, where `Class` is the name of a resource class, for example, `NDWin::SendNfyInit`.

Ordinarily you will not use `NDRes::SendNfyInit` directly, unless you are creating a custom widget. If so, you may call this macro to trap the `NDRes::NfyInit` notification to initialize the volatile fields after the persistent fields have been initialized.

See also `NDRes::SendNfyEnd`, `NDRes::SendNfyReset`, `NDRes::SendNfy`

`SendNfyEnd`

Sends a `NDRes::NfyEnd` to the resource specified.

void NDRes::SendNfyEnd (void);

`NDRes::SendNfyEnd` sends a `NDRes::NfyEnd` notification to the resource if it has already been initialized (the `ResInitialized` flag was previously set). This macro is usually called indirectly through the individual `CLASS::SendNfyEnd` routines, where `CLASS` is the name of a resource class, for example, `NDWin::SendNfyEnd`.

Ordinarily you will not use `NDRes::SendNfyEnd` directly, unless you are creating a custom widget. If so, you may call this macro to trap the `NDRes::NfyEnd` notification to deallocate the volatile fields before destroying or resetting the widget.

See also `NDRes::SendNfyInit`, `NDRes::SendNfyReset`, `NDRes::SendNfy`

`SendNfyReset`

Sends a `NDRes::NfyReset` to the resource specified.

void NDRes::SendNfyReset (void);

`NDRes::SendNfyReset` sends a `NDRes::NfyReset` notification to the resource. In case the resource has already been initialized (the `ResInitialized` flag was already set), it sends a `NDRes::NfyEnd` notification followed by a `NDRes::NfyInit` notification. This macro is usually called indirectly through the individual `CLASS::SendNfyReset` routines, where `Class` is the name of a resource class, for example, `NDWin::SendNfyReset`.

Ordinarily you will not use `NDRes::SendNfyReset` directly, unless you are creating a custom widget. If so, you may call this macro to trap the `NDRes::NfyReset` notification to update the volatile fields after the widget received changes to its persistent fields.

See also

`NDRes::SendNfyEnd`, `NDRes::SendNfyInit`, `NDRes::SendNfy`

Responding to a Notification

`ClassDefNfy`

Trigger the default notification procedure on an instance.

`void NDRes::ClassDefNfy (ResNfyEnum code);`

`NDRes::ClassDefNfy` activates the resource's class default response for `'code'`. In general, an instance of a given class is going, in its notification procedure, to directly customize the behaviour on some of the notifications but will in general delegate the rest of the processing to the default notification procedure defined for its class.

See also

`NDRes::ParentClassDefNfy`

`ParentClassDefNfy`

Trigger the parent default notification procedure on an instance.

`void NDRes::ParentClassDefNfy (ResNfyEnum code);`

`NDRes::ParentClassDefNfy` first determines the parent class of `res`, and then calls the default notification procedure installed in the parent class, with `res` and `code` as arguments.

See also

`NDRes::ClassDefNfy`

Control Data

`SendCtrlNfyData`

`void NDRes::SendCtrlNfyData(ResNfyEnum code, ResCtrlNfyPtr resCtrlNfy);`

Send a `CtrlNfyData` notification.

```
#define RES_CTRLNFYINHERIT(type)\
    ResCtrlDataPtr CtrlData;\
    C_SYMCAT2(type,Ptr)CtrlCaller;\
    C_SYMCAT2(type,NfyEnum)CtrlMsg;\
    RES_CTRLNFYINHERIT(Res) };
```

Command Management

This section describes the default routing mechanism for commands. We refer the reader to `cmdpub.h` for details on commands. The default command routing mechanism used in Open Interface propagates command objects through the GUI, and also to non-GUI objects (any kind of resource) by means of notifications. The mechanisms described here concern (1) the default command routing, and (2) the updating of command sources.

Command Routing

Issuing a command is done as follows: (1) the command source sends self an "command issue" notification with a `CmdPtr` as `NfyData`. The answer to this notification consists in choosing a suitable starting point "start" for the command routing, and send a "command route" notification.

When receiving a "route" notification, most objects will submit first the notification to their active component if any in the form of another "route" notification; if the command is still not handled, they try to handle it by sending self a "command" notification, then return.

GetNfyCmd

CmdPtr NRes::GetNfyCmd(void);

Returns the `CmdPtr` associated to a command notification. This call can only be invoked while processing one of the command related notifications.

General Purpose

IsCmdSource

BoolEnum NRes::IsCmdSource(void);

Returns whether the resource has the `RES_FLAGISCOMMANDSOURCE` flag set.

Command Sources

CmdSend

The following calls apply to resources that have the `RES_FLAGISCOMMANDSOURCE` flag set.

void NRes::CmdSend(ResPtr , CmdCtlEnum *ctl*);

Start routing of the resource's command at given 'start' point. (start is sent a the `NRes::NfyCommandRoute` notification with 'ctl' as argument).

CmdIssue

void NRes::CmdIssue(void);

Issue the resource's command for execution.

CmdUpdate

void NRes::CmdUpdate(void);

Issue the resource's command as command query for self updating. This call is the default answer to the `NRes::NfyUpdateView` notification when the resource has the `RES_FLAGISCOMMANDSOURCE` flag set.

Handling Command Notifications

CmdTableHandle

void NRes::CmdTableHandle(CmdTablePtr table);

To be used upon `NRes::NfyCommand` notifications fetches the command object and searches the given command table, using the default `NRes::TableHandle` method.

Resource Scripting

In the current Open Interface implementation, installing a script in a resource has as a side effect the fact that the installed notification procedure is not activated.

ExecuteScript

BoolEnum NRes::ExecuteScript(ResNfyEnum code);

This function causes the script for the event 'code' to be executed, if such a script is attached to the resource. It returns a boolean value indicating whether or not such a script was executed. The purpose of this call is to allow finer degree of control when mixing C and scripts for a resource than is available by calling `NDScript::DefNfy` (documented in the file `scrtpub.h`) This is because this `NDScript::DefNfy` will automatically call the resource's class notification procedure if there is no script to execute for the resource.

Error Handling Utilities

CheckClass

Provides a way for recovering from the specification of an invalid class.

void NRes::CheckClass (RClasCPtr class);

`NRes::CheckClass` is a macro that verifies whether `res` belongs to class or subclass. If not, a failure is signalled.

VERIFY

Provides a handler for recovering from the specification of an invalid class.

void NRes::VERIFY (RClasCPtr class);

`NRes::VERIFY` is a macro that verifies whether resource belongs to class or subclass. If not, a failure is signalled.

This macro is the same as `NRes::CheckClass` if `DBG_ON` is defined. This macro is disabled if `DBG_ON` is not defined.

36 *Region Class*

The Region class implements Open Interface regions.

Technical Summary

A 'region' can represent any arbitrary bounded set of points on a 2-D plane. This module is optimized for regions which can be represented as a small number of disjoint rectangles.

This class implements regions and mathematical manipulations of regions. A region can represent any arbitrary bounded set of points in a plane. This module is optimized for regions which can be represented as a small number of disjoint rectangles.

Inheritance Path

NDRRegion is a subclass of the NDRRes (resource) class; the region class inherits the same characteristics defined for resources.

NDRRes->NDRRegion

Related Classes

Related important include files are isetpub.h, rectpub.h, errpub.h.

See Also: Rect and Draw classes.

Enumerated Types

RgnPosEnum

Defines codes for the relative position of two regions after a comparison operation.

RgnPosEnum indicates the position of two regions that an operation is performed on. If the second region is completely inside the first region, RGN_POSINSIDE is indicated. If the intersection of the two regions is completely empty, RGN_POSOUTSIDE is indicated. Finally, if neither region is completely inside the other, RGN_POSCROSS is indicated.

See also

NDRRegion::RectPos

Empty Region

IsEmpty

Determines whether a region is empty.

BoolEnum NDRRegion::IsEmpty (void);

NDRRegion::IsEmpty returns `BOOL_TRUE` if the region is empty; otherwise, it returns `BOOL_FALSE`.

See also

NDRRegion::IsEqual, NDRRegion::IsPointInside

Reset

Resets a region to empty.

void NDRRegion::Reset (void);

NDRRegion::Reset resets region to an empty region.

See also

NDRRegion::Clone, NDRRegion::Translate, NDRRegion::QueryBounds

Region Rectangular Bounds

QueryBounds

Determines the boundaries of a region.

void NDRRegion::QueryBounds (Rect16Ptr *rectangle*);

NDRRegion::QueryBounds determines the geometry of the smallest rectangle that encloses region. It then places the result in *rectangle*.

See also

NDRRegion::Clone, NDRRegion::Reset, NDRRegion::Translate

Region Translation

Translate

Translates a region by the offset specified.

void NDRRegion::Translate (Point16CPtr *point*);

NDRRegion::Translate performs a mathematical translation of a region by adding an offset. The translation adds the offset specified by *point* to the origin of each rectangle in region.

See also NDRRegion::Clone, NDRRegion::Reset, NDRRegion::QueryBounds

Comparisons with other Regions

IsEqual

Determines whether two regions are equal.

BoolEnum NDRegion::IsEqual (RegionCPtr region2);

NDRegion::IsEqual returns BOOL_TRUE if region1 and region2 are equal; otherwise, it returns BOOL_FALSE.

See also

NDRegion::IsEmpty, NDRegion::IsPointInside

RectPos

Returns a code indicating the position of a rectangle relative to a region.

RgnPosEnum NDRegion::RectPos (Rect16CPtr rectangle);

NDRegion::RectPos returns the position of the region.

See also

RgnPosEnum

IsPointInside

Determines whether a point is inside a region.

BoolEnum NDRegion::IsPointInside (Point16CPtr point);

NDRegion::IsPointInside returns BOOL_TRUE if the point is inside the region; otherwise, it returns BOOL_FALSE.

See also

NDRegion::IsEqual, NDRegion::IsEmpty

Operations between Two Regions

RgnSet

Sets the coordinates of one region as specified by another.

void NDRegion::RgnSet (RegionCPtr src);

NDRegion::RgnSet sets the coordinates of dest as specified by src. In other words, it copies the coordinates of src to dest.

See also NDRegion::RgnIntersect, NDRegion::RgnUnion, NDRegion::RgnSubtract, NDRegion::RgnXOR

RgnIntersect

Creates an intersection of two regions.

void NDRegion::RgnIntersect (RegionCPtr region2);

NDRegion::RgnIntersect creates the region representing the intersection of region1 and region2 and places the result in region1.

See also

NDRegion::RgnSet, NDRegion::RgnUnion, NDRegion::RgnSubtract, NDRegion::RgnXOR

RgnUnion

Creates a union of two regions.

void NDRRegion::RgnUnion (RegionCPtr region2);

NDRRegion::RgnUnion creates a union between region1 and region2 and places the result in region1.

See also

NDRRegion::RgnIntersect, NDRRegion::RgnSet, NDRRegion::RgnSubtract, NDRRegion::RgnXOr

RgnSubtract

;Subtracts one region from another.

void NDRRegion::RgnSubtract (RegionCPtr region2);

NDRRegion::RgnSubtract subtracts region2 from region1 and places the result in region1.

See also

NDRRegion::RgnIntersect, NDRRegion::RgnUnion, NDRRegion::RgnSet, NDRRegion::RgnXOr

RgnXOr

Performs an exclusive Or on two regions.

void NDRRegion::RgnXOr (RegionCPtr region2);

NDRRegion::RgnXOr performs an exclusive Or on region1 and region2 and places the result in region1.

See also

NDRRegion::RgnIntersect, NDRRegion::RgnUnion, NDRRegion::RgnSubtract, NDRRegion::RgnSet

Operations between a Region and a Rectangle

RectSet

Associates a region with a rectangle.

void NDRRegion::RectSet (Rect16CPtr rectangle);

NDRRegion::RectSet sets region to the coordinates in rectangle.

See also

NDRRegion::RectIntersect, NDRRegion::RectUnion, NDRRegion::RectSubtract, NDRRegion::RectXOr

RectIntersect

Creates an intersection of a region and a rectangle.

void NDRRegion::RectIntersect (Rect16CPtr *rectangle*);

NDRRegion::RectIntersect creates a new region that is the intersection of region and rectangle. It places the result in region.

See also

NDRRegion::RectSet, NDRRegion::RectUnion, NDRRegion::RectSubtract, NDRRegion::RectXOr

RectUnion

Creates a union of a rectangle and a region.

void NDRRegion::RectUnion (Rect16CPtr *rectangle*);

NDRRegion::RectUnion creates a union between region and rectangle and places the result in region.

See also

NDRRegion::RectIntersect, NDRRegion::RectSet, NDRRegion::RectSubtract, NDRRegion::RectXOr

RectSubtract

Subtracts a rectangle from a region.

void NDRRegion::RectSubtract (Rect16CPtr *rectangle*);

NDRRegion::RectSubtract subtracts rectangle from region and places the result in region.

See also

NDRRegion::RectIntersect, NDRRegion::RectUnion, NDRRegion::RectSet, NDRRegion::RectXOr

RectXOr

Performs an exclusive Or between a region and a rectangle.

void NDRRegion::RectXOr (Rect16CPtr *rectangle*);

NDRRegion::RectXOr performs an exclusive Or on region and rectangle and places the result in region.

See also

NDRRegion::RectIntersect, NDRRegion::RectUnion, NDRRegion::RectSubtract, NDRRegion::RectSet

Regions Specified by a Polygon

Constructor

inline NDRRegion::NDRgn(Point16CPtr *points*, Int *num*, BoolEnum *winding*);

Constructs the region as a polygonal region.

'Points' is an array of points describing the vertices of the polygon(relative to a common origin, not to each other).

num' is the number of vertices in the array (it must be > 0).

'Winding' is a boolean indicating whether we should use the windingrule (BOOL_TRUE) or the odd-even rule (BOOL_FALSE) to determinewhether a given point is inside or outside the polygon.

The resulting region contains exactly:

- all the points in the edges, including the vertices themselves.
- all the points in the subregions delimited by the edges if the subregion is 'inside' the polygon according to the filling rule.

Performing an Action on Each Rectangle Component of a Region

A region can be decomposed into an union of disjoint rectangles. The following API invokes a callback method for each of the rectangle components of a region.

```
typedef PerfEnum (C_FAR * RgnPerfProc) (Rect16Ptr rect, ClientPtr data);
```

Method called for each rectangle component. 'Rect' is the current rectangle coordinates. 'Data' is some client data passed to RGN_PropagateAction.

PropagateAction

```
PerfEnum NDRegion::PropagateAction (RgnPerfProc proc, ClientPtr clientdata);
```

Calls 'proc' for each rectangle component of the region 'rgn'. 'data' is some client data which will be passed to 'proc' as an additional parameter. The region is scanned top to bottom, left to right.

37 *RLib Class*

The RLib class implements Open Interface resource library object.

Technical Summary

A resource library object keeps track of the library name, the dat file name and other attributes edited through the library editor in open editor.

It also keeps track of the file pointer and other I/O information needed to access the dat file.

Scope of the documented API

For now, the supported and documented API has been limited to the routines which open a dat file.

The RLib object currently maintains some information related to makefile generation. The makefile generations scheme will be replaced in the future by a more flexible and powerful scheme. At that time, we will document more of the RLib API.

We are also interested in knowing what your needs are on that API.

Inheritance Path

NDRLib is a subclass of the NDRes (resource) class; the resource library class inherits the same characteristics defined for resources.

NDRes->NDRLib

See Also:

Res, Wgt classes.

Accessing Libraries

Find

Returns a pointer to a library.

static RLibPtr NDRLib::Find(CStr lib);

Returns a pointer to the resource library with that name if it's already loaded, or NULL if it can't find it. NDRLib::GetLibName is the opposite API

GetLibName

Returns the name of a library.

CStr NDRLib::GetLibName(void);

Returns the name of the library lib. This is the opposite of NDRLib::Find.

GetFirst

Returns the first library in the list.

static RLibPtr NDRLib::GetFirst(void);

Returns the first library in the list of resource libraries already loaded.

GetNext

Returns the next library in the list.

RLibPtr NDRLib::GetNext(void);

Returns the next library after *lib* in the list of resource libraries already loaded. Returns NULL if '*lib*' was the last one.

Loading, Unloading, and Closing

These first 3 routines use the ND_PATH search path to locate the file in case the file cannot be found in the current directory. They return a RLibPtr pointer to the private RLib data structure that you can keep for later use with other APIs. They return immediately if the library is already loaded.

RLIB_LoadLibFile is safer than RLIB_LoadFile because it will check that libname matches the name stored in the dat file and you will get a failure if the dat file is not the one you expected.

LoadEdit

Loads a library database file by full name in read-write mode and returns a pointer to the library.

static RLibPtr NDRLib::LoadEdit (CStr libname, CStr filename);

NDRLib::LoadEdit loads a libname in read-write mode and returns its pointer. You should open your library with NDRLib::LoadEdit (instead of NDRLib::LoadLibFile) if you intend to use NDRes::SaveDat.

See also

NDRes::SaveDat

LoadFile

Loads a library database file by full name and returns it.

static RLibPtr NDRLib::LoadFile (CStr fullname);

NDRLib::LoadFile loads the library database file identified by fullname, then returns a pointer to it.

See also

NDRLib::LoadLibFile

LoadLibFile

Loads a library.

static RLibPtr NDRLib::LoadLibFile (CStr *libname*, CStr *filename*);

NDRLib::LoadLibFile loads the library database file identified by fullname, then returns a pointer to it.

Unload

Unloads library 'lib' from memory and closes the file.

static void NDRLib::Unload(void);

This call is the "safe" version of NDRLib::Dispose because all resources from lib are first detached before the lib structure is deleted from memory. (You must explicitly release each resource with NDRes::Release to free the memory that it occupies)

Dispose

Unloads library 'lib' and all the resources it contains and closes the file.

void NDRLib::Dispose(void);

WARNING: THIS IS THE UNSAFE VERSION OF NDRLib::Unload. You must be sure that no resources from lib are being referenced by other resources in memory (for instance if your library only contains windows and widgets, and no icons or color resources used else where). NDRLib::Dispose doesn't perform any checking before deleting the content of lib!

Open

Opens a file.

void NDRLib::Open(void);

Opens the file associated with 'lib'.

Close

Closes a file.

void NDRLib::Close(void);

Closes the file associated with 'lib'.

38 *SBuf Class*

The SBuf class is the base class for large string buffers.

Technical Summary

The SBuf class implements a string buffer which supports insertions and deletions inside large strings with a reasonable performance level. The SBuf class is implemented as a gap buffer which keeps track of a gap inside the string. This allows insertions to be handled efficiently, especially at the beginning of the string.

It is possible to handle multi-byte strings using SBuf::GetFwrd, SBuf::GetBwrd, similar to their strpub.h equivalents.

Simple Queries

GetLen

Returns the length of the string buffer.

StrIVal NDSBuf::GetLen(void);

Returns the length in bytes of the string contained in sbuf (not including the final zero).

See also

NDSBuf::GetStr, NDSBuf::GetSubStr

GetStr

Returns the contents of the string buffer.

CStr NDSBuf::GetStr(void);

Returns the string contained in the sbuf.

See also

NDSBuf::GetLen, NDSBuf::GetSubStr

GetSubStr

Return the string specified by index range.

CStr NDSBuf::GetSubStr(StrIVal index1, StrIVal index2);

Returns the substring between index1 and index2 (index1 included, index2 not included), properly terminated by a zero.

See also

NDSBuf::GetStr, NDSBuf::GetLen

Iteration

GetBwrd
GetFwrd

Return character code before of after index specified.

CharCode NDSBuf::GetFwrd(StrIVal index, StrIValPtr wp);

CharCode NDSBuf::GetBwrd(StrIVal index, StrIValPtr wp);

Returns the code of the character after or before index. If wp is not null, it will be set to the width of the character which has been returned. When the end of string is reached, 0 is returned and *wp is set to 0.

See also

NDSBuf::GetByte

GetByte

Byte NDSBuf::GetByte (StrIVal index);

Returns the byte at index specified. This is a low level call and it is preferable to iterate with SBuf::GetFwrd and SBuf::GetBwrd rather than with SBuf::GetByte.

See also

NDSBuf::GetFwrd, NDSBuf::GetBwrd

Miscellaneous Queries

CountToIndex

Convert character count to index.

StrIVal NDSBuf::CountToIndex (StrIVal charCount);

Converts between a character count and the corresponding offset in bytes in the sbuf.

See also

NDSBuf::IndexToCount

IndextoCount

Convert byte offset to character count.

StrIVal NDSBuf::IndexToCount(StrIVal index);

Converts between a character count and the corresponding offset in bytes index in the sbuf.

See also

NDSBuf::CountToIndex

Changing Contents

Set...

Replace the contents of the string buffer.

```
void NDSBuf::SetStr(CStr str);
void NDSBuf::SetStrSub(CStr str, StrIVal slen);
void NDSBuf::SetVStr(VStrCPtr vstr);
void NDSBuf::SetSBuf(SBufCPtr sbuf2);
```

Changes the contents of the sbuf with a copy of str, vstr or sbuf2.

See also

NDSBuf::Clear, NDSBuf::Insert

Clear

Clear context of string buffer specified.

```
void NDSBuf::Clear(void);
```

Resets the contents of the sbuf.

See also

NDSBuf::Set, NDSBuf::Insert

Insert...

Insert character, string, variable string or string buffer at index specified.

```
StrIVal NDSBuf::InsertChar(StrIVal index1, ChCode chcode);
StrIVal NDSBuf::InsertStr(StrIVal index1, CStr str);
StrIVal NDSBuf::InsertStrSub(StrIVal index1, CStr str, StrIVal slen);
StrIVal NDSBuf::InsertVStr(StrIVal index1, VStrCPtr vstr);
StrIVal NDSBuf::InsertSBuf(StrIVal index1, SBufCPtr sbuf2);
```

Inserts ch, str, vstr or sbuf into the sbuf at index1. returns the index at the end of the inserted string.

See also

NDSBuf::Clear, NDSBuf::Set, NDSBuf::Append

Append...

Append string, variable string or string buffer.

```
void NDSBuf::AppendChar(ChCode chcode);
void NDSBuf::AppendStr(CStr str);
void NDSBuf::AppendStrSub(CStr str, StrIVal slen);
void NDSBuf::AppendVStr(VStrCPtr vstr);
void NDSBuf::AppendSBuf(SBufCPtr sbuf2);
```

Appends str, vstr or sbuf2 to the end of the sbuf.

RemoveRange

Remove range of characters specified.

void NDSBuf::RemoveRange(StrIVal *index1*, StrIVal *index2*);

Removes the range of characters between *i1* and *i2* (*i1* included, *i2* not included). *i1* and *i2* should verify: $i1 < i2$.

See also

NDSBuf::RemoveChar

ReplaceChar

Replace string character.

StrIVal NDSBuf::ReplaceChar (StrIVal *index1*, ChCode *ch*);

Replaces character at *index1* by *ch*. Returns the index after the inserted character.

See also

NDSBuf::RemoveChar, NDSBuf::RemoveRange

Truncate

Truncate string at index specified.

void NDSBuf::Truncate(StrIVal *index*);

Truncates sbuf at index.

See also

NDSBuf::Clear, NDSBuf::Set, NDSBuf::Append

RemoveChar

Remove character at index specified.

void NDSBuf::RemoveChar (StrIVal *index*);

Removes the character at index *i*.

See also

NDSBuf::RemoveRange

Case Conversion

UpCase...

Convert string to upper case.

void NDSBuf::UpCase (void);

void NDSBuf::UpCaseSub (StrIVal *index1*, StrIVal *index2*);

These routines perform upper case conversion on the specified range of characters.

See also

NDSBuf::DownCase...

DownCase...

Convert string to lower case.

void NDSBuf::DownCase (void);

void NDSBuf::DownCaseSub (StrIVal *index1*, StrIVal *index2*);

These routines perform lower case conversion on the specified range of characters.

See also

NDSBuf::UpCase...

Matching

MatchesI...

Returns whether a string match is found with case specified.

BoolEnum NDSBuf::MatchesIChar (StrIVal *index*, ChCode *ch*, BoolEnum *icase*, StrIValPtr *endp*);

BoolEnum NDSBuf::MatchesIStr (StrIVal *index*, CStr *str*, BoolEnum *icase*, StrIValPtr *endp*);

BoolEnum NDSBuf::MatchesIStrSub (StrIVal *index*, CStr *str*, StrIVal *slen*, BoolEnum *icase*, StrIValPtr *endp*);

BoolEnum NDSBuf::MatchesISBuf (StrIVal *index*, SBufPtr *sbuf2*, BoolEnum *icase*, StrIValPtr *endp*);

Returns a boolean indicating whether the substring starting at *index* matches the *ch*, *str*, or *sbuf2* argument. If *icase* is `BOOL_TRUE`, the matching is case independent. If *endp* is not `NULL`, it will be set to the index of the end of the match (even if the match is incomplete).

See also

NDSBuf::Matches..., NDSBuf::IMatches...

Matches...

Returns whether a case sensitive string match is found.

BoolEnum NDSBuf::MatchesChar (StrIVal *index*, ChCode *ch*, StrIValPtr *endp*);

BoolEnum NDSBuf::MatchesStr (StrIVal *index*, CStr *str*, StrIValPtr *endc*);

BoolEnum NDSBuf::MatchesStrSub (StrIVal *index*, CStr *str*, StrIVal *slen*, StrIValPtr *end*);

BoolEnum NDSBuf::MatchesSBuf (StrIVal *index*, SBufPtr *sbuf2*, StrIValPtr *endp*);

Returns a boolean indicating whether the substring starting at *index* matches the *ch*, *str*, or *sbuf2* argument. The string comparison is case sensitive. If *endp* is not `NULL`, it will be set to the index of the end of the match (even if the match is incomplete).

These functions are the same as `NDSBuf::IMatches`, with *icase* set to `BOOL_FALSE`.

See also

`NDSBuf::MatchesI...`, `NDSBuf::IMatches...`

`IMatches...`

Returns whether a case insensitive string match is found.

BoolEnum `NDSBuf::IMatchesChar(StrIVal index, ChCode ch, StrIValPtr endp);`

BoolEnum `NDSBuf::IMatchesStr(StrIVal index, CStr str, StrIValPtr endp);`

BoolEnum `NDSBuf::IMatchesStrSub(StrIVal index, CStr str, StrIVal slen, StrIValPtr endp);`

BoolEnum `NDSBuf::IMatchesSBuf(StrIVal index, SBufPtr sbuf2, StrIValPtr endp);`

Returns a boolean indicating whether the substring starting at `index` matches the `ch`, `str`, or `sbuf2` argument. The string comparison is case sensitive. If `endp` is not `NULL`, it will be set to the index of the end of the match (even if the match is incomplete).

These functions are the same as `NDSBuf::IMatches`, with `icase` set to `BOOL_TRUE`.

See also

`NDSBuf::MatchesI...`, `NDSBuf::Matches...`

39 *Script Class*

This class implements the Open Interface script language.

Technical Summary

This class implements the Open Interface script language. There are two types of script which can be developed with Elements Environment 2.0.

The first type of script can be attached to any Open Interface resource. These scripts are usually attached to widget resources in an Open Interface graphical user interface, and hence are commonly referred to as widget scripts, but there is nothing in the architecture which restricts them to only being usable with widget resources. The execution and management of widget scripts is handled entirely by the Open Interface libraries - once a script has been associated with a widget or resource no user interaction is required to cause it to be compiled or to be executed. This type of script is documented in more detail in the following sections.

The second type of script is intended for advanced use, and it allows scripting functionality to be embedded in an existing C or C++ application. These scripts are not attached to resources, and hence are referred to as bare scripts, in that they exist by themselves. An API is provided so that the application developer has full control over the compilation, execution and disposal of these scripts. The API to control bare scripts is defined at the end of this header file.

Widget Scripts

Scripts can be attached to any Open Interface resource and have been implemented as a new permanent field in the resource structure. As such they are saved along with the rest of a resources permanent fields into the ascii and binary resource files, and are compiled into an executable form on the fly during resource initialization.

A script is divided up into a series of handlers, where each handler is responsible for handling a particular event sent to the widget/resource. The format of a handler is given below:

```
on event INITIALIZE
    statement 1
    statement 2
    ...
end event
```

In essence a handler is delimited by a pair of "on event XXX" and "end event" statements, where XXX is the name of the event which will cause this handler to be executed when the the event in question is sent to the widget/resource. The statements within the handler follow a grammar which is a subset of the C programming language, with a few minor extensions. Hence statements can be entered in free form (they do not have to fit on a single line) and must be terminated with a semicolon. A brief

overview of the features of the language are given below; further details are provided in the script language reference manual.

Variables

Three levels of scoping are provided for variables - local, script and global. Variable declarations are similar to those in C in that a declaration is composed of a data type keyword (described below) followed by a list of variable names. The scope of the variable name is determined in part by where the variable declaration occurs. Local variable declarations are made within a handler, before any of the executable statements, and their scope is limited to the handler in which they are declared. Script variables are declared outside a handler, and their scope extends from the point at which they are declared to the end of that particular script. Global variables are declared in a similar way to script variables, the difference being that their definitions are preceded by the "global" keyword. The value of a global variable can be accessed from any script which contains a declaration of that variable, and in any particular script the scope of the global variable declaration extends from the point at which the declaration occurs to the end of the script.

Script Data Types

Item	Description
<code>integer</code>	A signed integer number, whose size is the natural size for the machine on which the script is running. Currently this is 32 bits everywhere except for OSF/1 executing on an Alpha AXP machine, where the size is 64 bits.
<code>float</code>	A single precision floating point number.
<code>double</code>	A double precision floating point number.
<code>pointer</code>	This is a special read only data type. The only way to modify the value of a pointer variable is to assign to it the return value of a verb registered as returning a pointer value (see below for the definition of a verb), or to assign to it the value zero. Any other attempt to modify the value of a pointer variable will result in a script compilation error when the widget/resource is initialized. The main use of pointer variables in the script language is to cache widget and resource pointers, and to pass those pointers in as arguments to other verbs. There is currently no pointer dereferencing in the script language.
<code>string</code>	This is a nul-terminated array of ascii characters.

Statements

There are four basic kinds of statements in the script language:

1. Variable assignment statements - a variable can be assigned the value of an arbitrarily complex expression. Full details on the operators available within expressions and their precedence can be found in the script language reference manual.
2. Conditional statements - this is the if construct found in C, and behaves in exactly the same way.
3. Loop statements - this is the while loop construct found in C, and behaves in exactly the same way.

4. Calls to verbs - a verb is a routine which is external to the script and is written in a traditional programming language (usually C), and has been registered with the script compiler so that scripts can call out to such external routines. Several of the Open Interface API calls are registered as verbs with the scripting environment when the ScVrb library is initialized.

SELF

The special variable SELF designates the target of the notification inside an event handler. So in a script attached to a widget you can use the SELF variable to access the widget itself.

Note: Variable names are case sensitive (as in C), so SELF must be written in upper case.

Using the Scripting Environment

If you just wish to develop an application using the verbs which are packaged with Open Interface (ie. you will not be providing any custom verbs of your own) then you only have to add two statements to the main routine of your program, after all the other Open Interface libraries have been initialized:

```
SCRPT_LibInit();
SCVRB_LibInit();
```

The first statement initializes the scripting environment, the second statement performs some further initialization for using scripts in a graphical environment and registers all of the verbs which are packaged with Open Interface. Since SCVRB_LibInstall and SCVRB_LibLoadInit make calls to their counterparts in the ScVrb library, there is no need to make an explicit call to SCRPT_LibInit if the ScVrb library is also being initialized in an application.

LibInstall

```
static void NDScript::LibInstall(void);
```

Installing the script library.

LibLoadInit

```
static void NDScript::LibLoadInit(void);
```

Initialization and loading the script library.

LibInit

```
static void NDScript::LibInit(void);
```

Installing and initializing the script library.

LibExit

```
static void NDScript::LibExit(void);
```

Unloading and uninstalling the script library.

Extending the Script Language

The Open Interface script language can be extended in three ways:

1. By registering new symbolic constants for use within the scripts.
2. By registering new events for which scripts can be executed.
3. By registering new verbs which can be called from scripts.

Registering Constants

In the script language constants are named integer values. A certain number of these are defined when the ScVrb library is initialized, but additional constants can also be registered. To register a set of constants an array of RegisterConstRec structures is constructed, where each element of the array is used to define a single constant.

Since the script language does not copy the strings which are passed in this array, but instead just copies the pointers to those strings, then it is important to make sure that the storage for those strings will not be deallocated during the course of the applications execution. The last element in the array of RegisterConstRec structures must have all of its fields set to zero so that the routine which registers the constants knows when to stop processing the array.

The array so constructed is then passed by reference to SCRPT_RegisterConstants, which also takes a second boolean parameter which indicates if a warning should be issued if a constant is encountered in the array whose name has already been registered. If this parameter is BOOL_FALSE then no warning is issued, otherwise a warning is issued.

RegisterConstants

```
static void NDScript::RegisterConstants(ScrtRegisterConstPtr constants,
    BoolEnum checkDup);
```

Registers the constant identified by constants. If checkDup is BOOL_TRUE, a warning will be issued if a constant with the same name has already been registered.

NDScriptRegisterConst

Item	Description
Name	Name of the constant, referred to .
Constant	Value.

Registering Events

Events are registered in a similar fashion to constants, but this time the array passed to the registering function is an array of RegisterEventRec structures.

As before, the array should be terminated by a structure in which all of the above fields are set to zero, and the strings referenced inside the structure should not be placed in storage which is deallocated during the course of the programs execution.

The array so constructed is then passed by reference to `SCRPT_RegisterEvents`, which also takes a second boolean parameter which indicates if a warning should be issued if an event is encountered in the array whose name has already been registered. If this parameter is `BOOL_FALSE` then no warning is issued, otherwise a warning is issued.

RegisterEvents

static void NDScript::RegisterEvents(ScrptRegisterEventPtr event, BoolEnum checkDup);

Registers the event identified by event. If checkDup is `BOOL_TRUE`, a warning will be issued if an event with the same name has already been registered.

NDScriptRegisterEvent

Item	Description
Name	Name of the event, referred to .
Code	Notification code (ex: <code>TBUT_NFYHIT</code>) .
Class	Resource class for which the event is registered. If <code>NULL</code> , then the event is registered for all responder classes.

Registering Verbs

Verbs are registered in a similar fashion to constants and events, except that a little more information is required. The array passed to the registering function is an array of `RegisterVerbRec` structures. As before, the array should be terminated by a structure in which all of the above fields are set to zero, and the strings referenced inside the structure should not be placed in storage which is deallocated during the course of the programs execution.

The array so constructed is then passed by reference to `SCRPT_RegisterVerbs`, which also takes a second boolean parameter which indicates if a warning should be issued if a verb is encountered in the array whose name has already been registered. If this parameter is `BOOL_FALSE` then no warning is issued, otherwise a warning is issued.

The mapping between the return type of a verb in the script language and the corresponding return type of the C routine which implements that verb is as follows:

Script	C
void	void
integer	Long
float	float
double	double
pointer	ClientPtr
string	Str

The mapping between the type of an argument to a verb and the corresponding type of the argument in the C implementation of that verb is as follows:

Script	C
integer	Long
float	float*
double	double*
pointer	ClientPtr
string	VStrPtr

In essence all of the parameters passed into the C routine can be typecast into a ClientPtr.

Constants describing the return data type of a verb or script

Item

SCRPT_VALUEINTEGER
 SCRPT_VALUEFLOAT
 SCRPT_VALUEDOUBLE
 SCRPT_VALUEPOINTER
 SCRPT_VALUESTRING

Compatibility with Open Interface 3.0

Item

Item	Description
SCRPT_VERBVOID	SCRPT_VALUEVOID
SCRPT_VERBINTEGER	SCRPT_VALUEINTEGER
SCRPT_VERBFLOAT	SCRPT_VALUEFLOAT
SCRPT_VERBDOUBLE	SCRPT_VALUEDOUBLE
SCRPT_VERBPOINTER	SCRPT_VALUEPOINTER
SCRPT_VERBSTRING	SCRPT_VALUESTRING

Constants describing the data type of an argument

Item

SCRPT_ARGFLOAT
 SCRPT_ARGDOUBLE
 SCRPT_ARGPOINTER
 SCRPT_ARGSTRING

Maximum number of arguments that can be supplied to a verb

Item

SCRPT_MAXVERBARGS

NDScriptRegisterVerb

Item	Description
Name	A pointer to a nul terminated string which contains the name of the verb.
Proc	The address of the entry point of the routine which implements the verb.
NumArgs	Count of the number of arguments which the verb takes. A verb can be passed up to 16 arguments, and the script compiler will use this information to ensure that a verb is being called with the correct number of arguments.
Type	Return type of the verb. This is an integer which specifies the return type, and valid values for this field are given by the SCRPT_VERBXXX constants defined below.
ArgTypes[SCRPT_MAXVERBARGS]	A 16 character array, each element of which specifies the type of the corresponding argument in the verbs argument list. Valid values for these constants are given by the SCRPT_ARGXXX constants defined below. The script compiler uses this information to check that the correct type of value is being passed as an argument to a verb. If a particular element of this array is zero, then the compilers type checking mechanism is disabled for the corresponding argument.
CName	Ignored for now, but reserved for future expansion.

RegisterVerbs

static void NDScript::RegisterVerbs(ScrptRegisterVerbPtr verb, BoolEnum checkDup);

Registers the verb identified by `verb'. If `checkDup' is `BOOL_TRUE`, a warning will be issued if an event with the same name has already been registered.

SetStringReturnValue

static void NDScript::SetStringReturnValue(Str val);

As the above table shows, verbs which are registered to return a string value are expected to return a Str value to the script engine. When the verb has finished executing the script engine will make a copy of the string which has been returned to it, which means that the string must remain in scope after the verb has finished executing. Because of this it is not possible to return a string which is contained within a local array declared within the body of the verb's code, and so if the string which is returned to the script engine is held in a buffer local to the verb then that verb must be declared as static, eg.:

This is wrong:

```
Str MyStringVerb L0()
{
    char    buffer[256];
           Code which fills the buffer
    return buffer;    buffer is not in scope
                       after verb exits
}
```

This is correct:

```
Str MyStringVerb L0()
{
    static char    buffer[256];
    Code which fills the buffer
    return buffer;    buffer is in scope after
                    verb exits
}
```

One problem with this is that it is difficult to return strings which have been allocated dynamically by the verb, because at the point at which the verb stops executing the string has to be valid, and hence the verb cannot deallocate the string before returning control back to the script engine, resulting in a memory leak. To get around this problem the verb should make a call to the function `NDScript::SetStringReturnValue` immediately before deallocating the string and returning control to the script engine, eg.:

This is correct:

```
Str MyDynamicStringVerb L0()
{
    Str    buf;

    buf = PTR_New(someSize);
    code which fills the dynamically allocated
    buffer

    SCRPT_SetStringReturnValue(buf);
    PTR_Dispose(buf);
    return NULL; return value is ignored if
                SCRPT_SetStringReturnValue has
                been called
}
```

Running a Script in Standalone Applications

Run
ExecuteApp

static void NDScript::RunApp(CStr fileName);

static void NDScript::ExecuteApp(void);

Loads the script from the file identified by `fileName`, compiles it and executes it.

This routine is used in the standalone application for running applications built entirely using the script language. The routine takes one parameter, which is the name of an ascii text file containing an application startup script. The script is loaded from the file and compiled, and then executed by being sent the `APPSTARTUP` event, and so the script should include a handler for this event. A typical example of such a script would be:

```
on event APPSTARTUP
    RLIB_LoadFile("myfile.dat");
    WIN_OpenByName("mymod.win");
end event
```

Bare Scripts

As mentioned at the beginning of this header file, bare scripts are intended to allow scripting functionality to be embedded within an existing or new C/C++ application. The application developer is given full control over when the scripts are compiled, when and how often they are executed, and when a script's compiled form is disposed. Since these scripts are not attached to resources, there is no requirement that the application have a graphical interface.

The syntax of a bare script is very similar to that of a widget script, with the exception that a bare script does not contain event handlers. Instead it contains a set of variable definitions followed by a series of executable script statements - these statements are executed in order when the compiled form of the script is executed. Further details on the syntax of the script language can be found in the reference manuals.

Compile

static ScriptPtr NDScript::Compile(Str sourceCode);

This function compiles the bare script passed to it as a string, and returns a pointer to the compiled form of that script. If a NULL value is returned this signifies that compilation was unsuccessful due to compilation errors.

CompileFile

static ScriptPtr NDScript::CompileFile(Str fileName);

This is a convenience function. It compiles the bare script contained in the file specified by 'fileName' and returns a pointer to the compiled form of the script. If a NULL value is returned this signifies that compilation was unsuccessful due to compilation errors.

CompileResource

static ScriptPtr NDScript::CompileResource(Str resName);

This is a convenience function. It compiles the bare script contained in the string resource 'resName' and returns a pointer to the compiled form of the script. If a NULL value is returned this signifies that compilation was unsuccessful due to compilation errors.

Execute

static BoolEnum NDScript::Execute(ScriptPtr script);

This function executes the compiled script 'script'. The function returns control to the caller when the script has finished executing. This can happen for one of three reasons, summarised below:

GetReturnType

static Int32 NDScript::GetReturnType(ScriptPtr script);

Obtains the type of the value which was returned by execution of the script 'script'. The value returned is one of the predefined SCRIPT_VALUEXXX constants.

QueryReturnValue

```
static void NDScript::QueryReturnValue(ScriptPtr script, ClientPtr value);
```

Copies the value which was returned by execution of the script ``script'` into the buffer pointed to by ``value'`. For all return types except strings ``value'` should point to a buffer large enough to hold the corresponding type. For strings, ``value'` should point to a location in memory into which will be written the address of the beginning of the string value.

Dispose

```
static void NDScript::Dispose(ScriptPtr script);
```

This function disposes the compiled bare script ``script'`, freeing all of the memory used by the compiled form.

Return type	Reason
SCRIPT_VALVOID	The script executed a <code>`return;'</code> statement, or the last line of the script was reached and that line was not a return statement.
Anything else	Anything elseThe script executed a return statement which passes a value back to the caller e.g. <code>`return 1;'</code> .
N/A	The script was terminated prematurely due to a runtime error.

40 *Set Class*

This class implements a data structure to represent sets of objects.

Overview

This class implements a data structure to represent sets of objects. It is a generic data structure in the sense that each element of the array is big enough to contain either a basic type (short or long int) or a pointer to an object that you have allocated separately.

When you add elements to the set, the buffer holding the elements of the array will be reallocated automatically and the pointer to the set object (SetPtr) will NOT change. A set differs from an array in that you can not access an element by index. Instead, you can add or remove an element and perform usual set operations (like union and intersection).

Constructors and Destructor

NDSer

```
inline void NDSer::NDSer(void);
```

Default set construction.

NDSer

```
inline NDSer::~NDSer(void);
```

Default set destruction.

Special Shared Sets

EmptySet

```
static SetPtr NDSer::EmptySet(void);
```

Returns a pointer to a shared empty set.
This set should not be modified.

Adding, Removing, Accessing Elements

AddElt

```
void NDSer::AddElt(SetEltVal elt);
```

Adds 'elt' to the set (unless it is already in).

RemoveElt**void NDSer::RemoveElt(SetEltVal elt);**

Removes 'elt' from the set (unless it is not in).

AddElts**void NDSer::AddElts(SetLenVal n, SetEltValPtr elts);**

Adds elts[0], elts[1], ...,elts[n-1] to the set (unless they are already in).

RemoveElts**void NDSer::RemoveElts(SetLenVal n, SetEltValPtr elts);**

Removes elts[0], elts[1], ...,elts[n-1] from the set (unless they are not in the set).

GetNumElts**SetLenVal NDSer::GetNumElts(void);**

Returns the number of elements in set.

Reset**void NDSer::Reset(void);**

Empties the set.

Copy**void NDSer::Copy(SetCPtr src);**

Empties 'dst', then copies the contents of 'src' into 'dst'.

QueryElts**void NDSer::QueryElts(SetLenVal n, SetEltValPtr elts);**

Queries the first 'n' elements of the set and put them into 'elts'. Use NDSer::GetNumElts to get the number of elements in the set.

SetElts**void NDSer::SetElts(SetLenVal n, SetEltValPtr elts);**

Sets the first 'n' elements of the set to the values taken from 'elts'. Use NDSer::GetNumElts to get the number of elements in the set.

ContainsElt**BoolEnum NDSer::ContainsElt(SetEltVal elt);**

Returns BOOL_TRUE if 'elt' is in the set.

Comparing and Combining Two Sets

When comparing or combining two sets: A and B. Their elements can be divided into three regions or parts:

1. Elements which are only in A.
2. Elements which are only in B.
3. Elements which are in both A and B.

Useful combinations of these parts are:

Set	Description
Union of A and B (parts 1+2+3)	(logically equivalent to the OR operator)
Intersection of A and B (part 3)	(logically equivalent to the AND operator)
Difference of A and B (part1)	(logically equivalent to: A AND NOT B)

Symmetric difference of A and B ([1]+[2]), i.e. the difference between the union of A and B and the intersection of A and B. (logically equivalent to the XOR operator).

SetMixPartSetEnum

Bit set representing the different parts involved in set operations.

```
SET_MIXPART1 = (1 << (1)), SET_MIXPART1BIT = 1,    Refers to part [1] above
SET_MIXPART2 = (1 << (2)), SET_MIXPART2BIT = 2,    Refers to part [2] above
SET_MIXPART3 = (1 << (3)), SET_MIXPART3BIT = 3     Refers to part [3] above
```

MixGetPartSet

```
static SetMixPartSet NDSets::MixGetPartSet(SetCPtr A, SetCPtr B);
```

Compares two sets A and B and returns the set of parts which are not empty.

AreEqual

```
static BoolEnum NDSets::AreEqual(SetCPtr A, SetCPtr B);
```

Returns BOOL_TRUE if sets A and B are equal.

MixQueryParts

```
static void NDSets::MixQueryParts(SetCPtr A, SetCPtr B, SetMixPartSet parts, SetPtr C);
```

Combines A and B and extracts the specified parts. The result is stored in C (which must be a distinct Set object, C must not point to the same object as A or B).

41 *Str Class*

The Str class implements the Open Interface string data structures and utilities. The functions in this class support English and languages other than English by operating on both single-byte and multibyte characters.

Technical Summary

The Str class is functionally very similar to the C RTL string package (e.g., `strlen`, `strcat`) but offers the following advantages:

- consistent naming
- better error handling.
- support for multibyte character sets

Use the STR class instead of the standard C RTL routines if you need a portable set of library routines that support the multibyte character sets required by applications intended for Asian for European markets. Rather than relying on RTL, whose standard varies among vendors, the STR class takes advantage of the major industry standards for character encoding.

Basic string types

The Open Interface Str class defines Native and UNICODE string types.

A NatStr string is a pointer to an array of NatChar and/or NatCode characters. There are also pointers to a native string pointer. Types accommodate cases where the native string is constant, where the pointer to the string is constant, or where both are constant.

A Str string is an array of Char and/or ChCode characters. There are also types to accommodate the cases where the string is constant, where the pointer to the string is constant, or where both are constant.

A UniStr string contains UNICODE characters only. There are also pointers to UniStr pointers. The UniStr can be constant, the pointer to the UniStr can be constant, or they can both be constant.

For more information about Native and UNICODE character types, see the Char class.

Strings Vs Binary Data

The Str module deals with strings (text intended for humans), not with arbitrary binary data. So you should not use the Char or Str type when dealing with binary data, you should use the VoidPtr, Byte or BytePtr types.

Indexing Strings

The various string types are defined as “huge” pointers. This type qualifier is only relevant in segmented architectures such as DOS or OS/2. By considering all strings as “huge” we avoid many complications with strings which are larger than 32 KBytes.

- StrVal integer type to index strings. A 32 bit integer is necessary to support “huge” strings. This type is defined in charpub.h.

Characters

The character types are described in detail in charpub.h Here is a summary of the main character types:

```
NatChar  "native" string byte.
NatCode  "native" character code (encodes multi-byte characters)
Char     "internal" string byte.
ChCode   "internal" character code (encodes multi-byte characters)
UniCode  UNICODE character code.
CharInfoVal      domain + level + lexical cat + case info + ascii-ness + ...
```

Code Types And Code Sets

The charpub.h header files gives detailed information about these topics.

Basic Strings and Substrings

A basic string is a null-terminated array of bytes. In the simple case of an ASCII or ISO LATIN1 string, each byte encodes a character. In UNICODE or one of the Japanese encodings, a string might contain a mixture of single byte and double byte characters.

The calls contained in this class are provided in two versions:

A Str version in which strings are passed as simple pointers

A SubStr version a pointer to the beginning of a substring and its length are passed as separate.

In the substrings calls, the a substring is specified by a pointer to the beginning of the substring and a length. The substring is not necessarily terminated by a null at the specified length. Also, in all the substring calls, a length of -1 is interpreted as an unknown length, in which case the terminating null is used as an end-of-string indicator. The substring calls stop if a NULL is encountered before the specified length.

Operations which write into string buffers receive a pointer to the buffer and the size of the buffer. These operations never overflow the destination buffer and always terminate their output with a NULL byte, except when specified otherwise.

Higher-Level String Objects and APIs

Open Interface contains the following high level string objects to support the buffer reallocation and complex string manipulations that Str objects are not intended to support.

String Object	Description
VStr	General purpose, compact string object which handles buffer reallocation automatically.
SBuf	String object larger than VStr. Keeps track of the gap inside the string, so that successive insertions can be performed efficiently, even at the beginning of a string. This object is designed to support complex string manipulations (insertions, deletions, formatting) in an efficient way.

Use the VStr object for storing strings and the SBuf object for manipulating strings. Only a limited set of operations such as append and format are provided on the VStr object. The SBuf API is much more complete and also provides a simple API to temporarily attach an SBuf object to a VStr object or to a stack buffer and to detach the SBuf object afterwards.

See the VStr class for more information.

Str Class Operations

The Str class functions enable you to perform the following operations:

- Create strings.
- Dispose of strings.
- Set strings.
- Append to strings.
- Find string length.
- Iterate through strings.
- Write into string buffers.
- Compare strings.
- Match strings.
- Search through strings.
- Scan for numeric values.
- Format numeric values
- Convert between cases.
- Load strings from resources.

The following functions are also supported:

- String formatted print (printf).
- String formatting (sprintf).
- String scan (sscanf).

Note: Users should understand that the C runtime library routines such as strcpy, strcat or sprintf are unsafe because the caller cannot specify the size for which the buffer has been allocated and thus, in general, there is a risk of overflow. See the Writing into String Buffers section for details. The Open Interface ctrlpub.h header file defines the C runtime library routines for use with Open Interface but is not documented in the API Reference Manuals.

Data Types

NatStr

Defines a native string type.

```
typedef NatChar C_HUGE* NatStr;
```

```
typedef C_INVAR NatChar C_HUGE* NatCStr;
```

A native string is a zero-terminated string in the native encoding defined by the ND_CHARNAIVE environment variable. A native string can include any combination of single-byte and multibyte characters.

Use `NatStr` types for human-readable text, not binary data. Use `void*`, `Byte`, or `BytePtr` for manipulating binary data.

See also

`Str`, `UniStr`, `ND_CHARNATIVE`

NatStrPtr

Defines a pointer to a native string type.

Data type defining a pointer to a native string.

A native string is a zero-terminated string in the native encoding defined by the `ND_CHARNATIVE` environment variable. A native string can include any combination of single-byte and multibyte characters.

Use `NatStr` types for human-readable text, not binary data. Use `void*`, `Byte`, or `BytePtr` for manipulating binary data.

See also

`NatStr`

Str

Defines a string type.

`Str` is a data type defining a string type. A string is a zero-terminated string, represented in the encoding as defined by the `ND_CHARNATIVE` environment variable. A string can include any combination of single-byte and multibyte characters.

Use `Str` types for human-readable text, not binary data. Use `void*`, `Byte`, or `BytePtr` for manipulating binary data.

See also

`NatStr`, `UniStr`, `ND_CHARNATIVE`

StrIVal

A 32-bit integer used for indexing strings and characters.

StrIValPtr

Data type for a pointer to a `StrIVal` value.

StrPtr

Data type for a string pointer.

Data type for a string pointer.

See also

`Str`

UniStr

Defines a UNICODE string type.

Data type defining a UNICODE string type. A UniStr string is an array of UNICODE characters.

See also

Unicode

UniStrPtr

Defines a pointer to a UNICODE string.

Data type defining a pointer to a UNICODE string which is an array of UNICODE characters.

Use UniStr types for human-readable text, not binary data. Use void*, Byte, or BytePtr for manipulating binary data.

See also

UniStr, Unicode

Cloning and Disposing

We recommend that you use VStr objects rather than simple Str pointers for dynamically allocated strings. The VStr object encapsulates the length of the string and thus avoids useless length recomputations.

NewSet

Returns a new copy of a string.

static Str NDStr::NewSet(CStr string);

Returns a new copy of a string.

For dynamically allocated strings, use VStr objects rather than simple Str pointers. The VStr object encapsulates the length of the string, so it avoids length recomputations.

See also

NDStr::Clone

Clone

Returns a new copy of a string.

static Str NDStr::Clone(CStr string);

Definition

NDStr::Clone returns a new copy of a string. This function is an alias for NDStr::NewSet.

For dynamically allocated strings, use VStr objects rather than simple Str pointers. The VStr object encapsulates the length of the string, so it avoids length recomputations.

See also

`NDStr::NewSet`

NewSetSub

Returns a new substring.

static Str NDStr::NewSetSub(CStr string, StrIVal len);

`NDStr::NewSetSub` returns a new string containing a substring of the given length.

For dynamically allocated strings, use `VStr` objects rather than simple `Str` pointers. The `VStr` object encapsulates the length of the string, so it avoids length recomputations.

See also

`NDStr::SetNew`

Dispose

Disposes of a string buffer.

static void NDStr::Dispose(Str string);

`NDStr::Dispose` disposes of a string buffer.

For dynamically allocated strings, use `VStr` objects rather than simple `Str` pointers. The `VStr` object encapsulates the length of the string, so it avoids length recomputations.

See also

`NDStr::Dispose0`

Dispose0

Disposes of a string buffer if the buffer is not `NULL`.

static void NDStr::Dispose0(Str string);

Disposes of a string buffer if the buffer is not `NULL`.

For dynamically allocated strings, use `VStr` objects rather than simple `Str` pointers. The `VStr` object encapsulates the length of the string, so it avoids length recomputations.

See also

`NDStr::Dispose`

Set and Append

The following routines allow you to change the contents of a string or to append a string to an existing string. They take the address of a string as first argument so that they can reallocate the string if necessary.

You are encouraged to use the VStr or SBuf modules when performing complex string manipulations. These calls should be reserved for simple cases only.

Set

Setting a new string to contain the contents of an existing string.

static void NDStr::Set(StrPtr stringptr, CStr string) ;

Setting a new string to contain the contents of an existing string. The stringptr parameter is a pointer to an address of an existing string so that memory for the string can be reallocated if necessary.

Use this call for simple cases only. Use VStr or SBuf objects when performing complex string manipulations.

See also

NDStr::SetSub

SetSub

Assigns a substring as the contents of an existing string.

static void NDStr::SetSub(StrPtr stringptr, CStr subptr, StrIVal length);

NDStr::SetSub assigns a substring as the contents of an existing string. The stringptr parameter is a pointer to an address of an existing string so that memory for the string can be reallocated if necessary.

Use this call for simple cases only. Use VStr or SBuf objects when performing complex string manipulations.

See also

NDStr::Set

Append

Appends a string.

static void NDStr::Append(StrPtr stringptr, CStr string);

NDStr::Append appends a string to an existing string. The stringptr parameter is a pointer to an address of an existing string so that memory for the string can be reallocated if necessary.

Use this call for simple cases only. Use VStr or SBuf objects when performing complex string manipulations.

See also

NDStr::AppendSub

AppendSub

Appends a substring.

static void NDStr::AppendSub(StrPtr stringptr, CStr subptr, StrIVal length);

NDStr::AppendSub appends a substring to an existing string. The stringptr parameter is a pointer to an address of an existing string so that memory for the string can be reallocated if necessary.

Use this call for simple cases only. Use VStr or SBuf objects when performing complex string manipulations.

See also

NDStr::Append

String Length

GetLen

Returns the length of a string.

static StrIVal NDStr::GetLen(CStr string);

NDStr::GetLen returns the length of string in bytes. The length does not include any null terminators. Replaces NDStr::Len.

See also

NDStr::GetTruncLen

GetTruncLen

Returns the number of bytes in a string that can be copied into a buffer of a given size.

static StrIVal NDStr::GetTruncLen(CStr string, StrIVal size);

NDStr::GetTruncLen returns the number of bytes in string which can be copied into buffer of a given size. A multibyte character might be truncated if the length of the string is larger than the buffer, the length of the string in the buffer is not necessarily equal to size - 1.

See also

NDStr::GetLen

Iterating through Strings

Iterating through strings requires some special care because strings may contain multi-byte characters. One way is to access the string byte by byte (Char by Char) and to get the length of characters by calling NDStr::GetLen.

Another (safer) way is to use the following API calls which return character codes and advance the index in the string.

To iterate forwards in a string, you should use NDStr::GetFwrdd instead of reading byte by byte, except when you are only interested in ASCII characters and performance is critical (you can test the ASCII-ness first and call NDStr::GetLen only on non ASCII characters).

To iterate backwards, you have no other choice than using `NDStr::GetBwrld` because iterating backwards is not a straightforward operation in general. For example, the SJIS code type allows ASCII letters as second byte of multi-byte characters, so the length of a character cannot be derived simply from the value of its last byte.

GetCode

Returns the character code located at the beginning of a string.

static ChCode NDStr::GetCode(CStr string);

`NDStr::GetCode` returns the character code located at the beginning of string. The length of the character is not set.

See also

`NDStr::NatGetCode`

GetFwrld

Returns the character code at the beginning of a string and sets its length.

static ChCode NDStr::GetFwrld(CStr string, StrValPtr lengthptr);

`NDStr::GetFwrld` returns the character code at the beginning of string and sets `lengthptr` to the length of the character. `NDStr::GetFwrld` always sets `lengthptr`. `NDStr::GetFwrld` does not test whether `lengthptr` is NULL. When `NDStr::GetFwrld` encounters a NULL character, it sets the length pointer to one. `NDStr::GetFwrld` returns zero when the end of the string is reached.

See also

`NDStr::NatGetFwrld`, `NDStr::GetBwrld`

GetBwrld

Returns the code found in front of the specified location.

static ChCode NDStr::GetBwrld(CStr string, StrVal index, StrValPtr lengthptr);

`NDStr::GetBwrld` returns the character code found in front of the location in string given by `index` and sets `lengthptr` to the length of the character. `NDStr::GetBwrld` does not test whether `lengthptr` is zero and always sets the length pointer at the end of the operation.

See also

`NDStr::GetFwrld`

CtGetCode

Returns the character code found at the beginning of a string.

static NatCode NDStr::CtGetCode(NatCStr natstring, CtCPtr codetype);

`NDStr::CtGetCode` returns the native character code located at the beginning of a native string. The length of the character is not set. See `NDStr::CtGetFwrld`.

See also

`NDStr::GetCode`, `NDStr::NatGetCode`, `NDStr::CtGetFwrd`

`CtGetFwrd`

Returns the character code found at the beginning of a string and sets the length.

static `NatCode NDStr::CtGetFwrd(NatCStr natstring, CtCPtr codetype, StrIVaIPtr lengthptr)`

`NDStr::CtGetFwrd` returns the character code found at the beginning of the string and sets `lengthptr` to the length of the character. `NDStr::CtGetFwrd` does not test whether `lengthptr` is zero and always sets `lengthptr` at the end of the operation.

See also

`NDStr::GetFwrd`, `NDStr::NatGetFwrd`

`CtGetBwrd`

Returns the code found in front of a location in a string.

static `NatCode NDStr::CtGetBwrd(NatCStr natstring, CtCPtr codetype, StrIVaI index, StrIVaIPtr lengthptr)`;

`NDStr::CtGetBwrd` returns the native character code found in front of the location given by `index` in an encoded string. Sets `lengthptr` to the length of the character. `NDStr::CtGetBwrd` does not test whether `lengthptr` is zero and always sets `lengthptr` at the end of the operation. If the `index` is zero, `NDStr::CtGetBwrd` returns zero and sets `lengthptr` to zero.

See also

`NDStr::GetBwrd`

Writing into String Buffers

The main problem when we are writing into string buffers is that what we write may be too big for the buffer and may cause an overflow. The C RTL routines such as `strcpy`, `strcat` or `sprintf` are unsafe because the caller cannot specify the size for which the buffer has been allocated and thus, in general, there is a risk of overflow.

Most of the `Str` API implements operations which access strings in read-only mode but we also provide routines which write into string buffers, even if the preferred API for string manipulations is the `SBuf` API.

The main routines are safe and return all the information necessary to find out if the operations resulted in a truncation or not. We also provide some routines which are unsafe or which do not indicate whether a truncation occurred or not. These routines are provided mostly for compatibility purposes.

The “safe” routines all use the same general API principle for the first two arguments and their return value. The principle can be illustrated taking the example of the `NDStr::Put` call.

```
len = NdStr::Put(buf, size, str, endp)
```

The convention used for the returned value may sound somewhat awkward but is actually very practical when we have to concatenate various values in a string. For example, we can write:

```
Char    buf[MAXSIZE];
S       s    = buf;
StrIVal size= MAXSIZE;
StrIVal len;
len = NdSTR::PutDecInt(s, size, i);s += len; size -= len;
len = NdStr::Put(s, size, " ", NULL);s += len; size -= len;
len = NdStr::PutDecInt(s, size, j);s += len; size -= len;
```

If an overflow occurs, the string will be properly truncated, size will become 0 and subsequent NDStr::Put calls will return immediately.

If you want to check the overflow condition, you can compare len and size after every NDStr::Put call (or compare size with 0 after the increment/decrement operations). Then, you may reallocate the buffer and retry the NDStr::Put operation.

The 'endp' argument is useful if you want to reallocate the buffer in case of overflow and continue the Put operation with the remaining string. If truncation is harmless, you do not need to worry about reallocation and you can simply pass NULL as 'endp'. In some calls, the 'endp' argument is a little more complex (for example in formatting routines, it describes a synchronization point between the format and what has been written). In other calls, such as calls which format numeric values, there is no need for an 'endp' argument.

Put

Writes a string into a buffer.

```
static StrIVal NDStr::Put (Str buffer, StrIVal size, CStr string, StrIValPtr endptr);
```

NDStr::Put writes a string into a buffer and truncates the string if it is too large. NDStr::Put always terminates the buffer with a null byte and never writes more than size bytes into the buffer (including the terminating null).

If the operation can be done without truncation, the value returned will be the number of characters written to the buffer not including the terminating null. In this case, the value returned is strictly less than size. If the operation resulted in a truncation, size is returned. If the endpoint is not NULL, it is set to the number of characters which have been copied. The endpoint is the same as the returned value if no truncation occurs. If an overflow occurs, the endpoint is set to size-x, where x is the width of the character which caused the overflow.

See also

NDStr::PutAscii, NDStr::PutCode, NDStr::PutSub

PutSub

Writes a substring into a string buffer.

```
static StrIVal NDStr::PutSub(Str buffer, StrIVal size, CStr subptr, StrIVal length, StrIValPtr endptr);
```

Writes a substring into a string buffer. After the writing the character code, NDStr::PutSub terminates the buffer with NULL.

See also

`NDStr::Put`

PutAscii

Writes an ASCII character into a string.

static StrIVal NDStr::PutAscii(Str *buffer*, StrIVal *size*, Char *char*);

`NDStr::PutAscii` writes the ASCII character into the string buffer. After the writing the character, `NDStr::PutAscii` terminates the string buffer with `NULL`. In debugging mode, `NDStr::PutAscii` signals a failure if the character is not an ASCII character.

Use `NDStr::PutAscii` to append a single character to a string. To write many characters sequentially, use `NDStr::WriteAscii` instead.

See also

`NDStr::WriteAscii`, `NDStr::NatPutAscii`

PutCode

Writes a character code into a string.

static StrIVal NDStr::PutCode(Str *buffer*, StrIVal *size*, ChCode *chcode*);

`NDStr::PutCode` writes a character code into the string buffer. After the writing the character code, `NDStr::PutCode` terminates the buffer with `NULL`.

Use `NDStr::PutCode` to append a single character code to a string. To write many character codes sequentially, use `NDStr::WriteCode`.

See also

`NDStr::Put`, `NDStr::WriteCode`, `NDStr::NatPutCode`

WriteAscii

Writes an ASCII character into a string without terminating the string with `NULL`.

static StrIVal NDStr::WriteAscii(Str *buffer*, StrIVal *size*, Char *char*);

`NDStr::WriteAscii` writes an ASCII character into a string without terminating the string with `NULL`. If writing the character would overflow the buffer, `NDStr::WriteAscii` writes a `NULL` and returns `size`.

`NDStr::WriteAscii` is used for writing a sequence of character codes into a string. To write a single character, use `NDStr::PutAscii`.

See also

`NDStr::PutAscii`, `NDStr::NatWriteAscii`

WriteCode

Writes a character code into a string without terminating the string with `NULL`.

static StrIVal NDStr::WriteCode(Str buffer, StrIVal size, ChCode chcode);

Writes a character code into a string buffer without terminating the string with NULL. If writing the character code would overflow the buffer, NDStr::WriteCode writes a NULL and returns size.

NDStr::WriteCode is used for writing a sequence of character codes into a string. To write a single character, use NDStr::PutCode.

See also

NDStr::PutCode , NDStr::NatWriteAscii

NatPutAscii

Writes an ASCII character into a native string.

static StrIVal NDStr::NatPutAscii(NatStr natstring, StrIVal size, NatChar natcode);

Writes an ASCII character into a native string. After the writing the character, NDStr::NatPutAscii terminates the native string with NULL. In debugging mode, NDStr::NatPutAscii signals a failure if the character is not an ASCII character.

NDStr::NatPutAscii is useful for appending a single character to a native string. To write many characters sequentially, use NDStr::NatWriteAscii.

See also

NDStr::PutAscii, NDStr::NatWriteAscii

NatWriteAscii

Writes an ASCII character into a native string without terminating the string with NULL.

static StrIVal NDStr::NatWriteAscii(NatStr natstring, StrIVal size, NatChar natcode);

Writes an ASCII character into a native string without terminating the string with NULL. If writing the character would overflow the native buffer, NDStr::NatWriteAscii writes a NULL and returns size.

NDStr::NatWriteAscii is used for writing a sequence of characters into a native string. For writing single characters, use NDStr::NatPutAscii.

See also

NDStr::WriteAscii, NDStr::NatPutAscii

NatPutCode

Writes a native character code into a native string.

static StrIVal NDStr::NatPutCode (NatStr natstring, StrIVal size, NatCode natcode);

Writes a native character code into a native string. After the writing the character code, NDStr::NatPutCode terminates the string buffer with NULL. NDStr::NatPutCode is useful for appending a single native character code to a native string. To write many native character codes sequentially, use NDStr::NatWriteCode.

See also

`NDStr::PutCode`, `NDStr::NatWriteCode`

`NatWriteCode`

Writes a native character code into a native string without terminating the string with `NULL`.

static `StrIVal` `NDStr::NatWriteCode`(`NatStr` *buffer*, `StrIVal` *size*, `NatCode` *natcode*);

`NDStr::NatWriteCode` writes a native character code into a native string without terminating the string with `NULL`. If writing the native character code would overflow the buffer, `NDStr::NatWriteCode` writes a `NULL` and returns `size`.

`NDStr::NatWriteCode` is used for writing a sequence of character codes into a string. For writing single characters, use `NDStr::NatPutCode`.

See also

`NDStr::WriteCode`, `NDStr::PutCode`

Basic String Comparisons

`Cmp`
`ICmp`

Compares two strings.

static `CmpEnum` `NDStr::Cmp`(`CStr` *string1*, `CStr` *string2*);

static `CmpEnum` `NDStr::ICmp`(`CStr` *string1*, `CStr` *string2*);

`NDStr::Cmp` compares two strings and returns a `CmpEnum` as a result. Bytes are compared one by one. Lower case words are sorted after all upper case words. In the ASCII range, characters are sorted in ASCII order, even on an EBCDIC platform.

If *string1* is alphabetically lesser than *string2*, `NDStr::Cmp` returns `CMP_UNDER`; if they are equal, it returns `CMP_EQUAL`; and if *string1* is greater than *string2*, `NDStr::Cmp` returns `CMP_OVER`.

`NDStr::ICmp` is exactly the same, but it ignores case differences in the ASCII range.

See also

`NDStr::CmpSub`

CmpSub
ICmpSub

Compares two substrings.

```
static CmpEnum NDStr::CmpSub(CStr subptr1, StrIVal length1, CStr subptr2,  
StrIVal length2);
```

```
static CmpEnum NDStr::ICmpSub(CStr subptr1, StrIVal length1, CStr subptr2,  
StrIVal length2);
```

NDStr::CmpSub compares two substrings and returns a CmpEnum as a result. See NDStr::Cmp.

NDStr::ICmpSub is exactly the same, but it ignores case differences in the ASCII range.

See also

NDStr::Cmp

Equals
IEquals

Compares two strings for equality.

```
static BoolEnum NDStr::Equals(CStr string1, CStr string2);
```

```
static BoolEnum NDStr::IEquals(CStr string1, CStr string2);
```

NDStr::Equals returns a boolean indicating whether or not the two strings are equal.

NDStr::IEquals performs the same function but ignores case differences in the ASCII range.

See also

NDStr::Cmp, NDStr::EqualsSub

EqualsSub
IEqualsSub

Compares two strings for equality.

```
static BoolEnum NDStr::EqualsSub(CStr subptr1, StrIVal length1, CStr subptr2,  
StrIVal length2);
```

```
static BoolEnum NDStr::IEqualsSub(CStr subptr1, StrIVal length1, CStr subptr2,  
StrIVal length2);
```

NDStr::EqualsSub returns a Boolean value indicating whether the two substrings are equal.

NDStr::IEqualsSub performs the same function but ignores case differences in the ASCII range.

See also

NDStr::Cmp, NDStr::Equals

Testing Matches

MatchesChar

Tests whether a string matches a character.

```
static BoolEnum NDStr::MatchesChar(CStr string, ChCode chcode, StrIVAlPtr lengthptr);
```

NDStr::MatchesChar tests whether string matches chcode. The Boolean return value indicates whether the match was successful. If lengthptr is not NULL, it is set to the end of the match.

Matches IMatches

Tests whether one string matches another string.

```
static BoolEnum NDStr::Matches(CStr string1, CStr string2, StrIVAlPtr lengthptr);
```

```
static BoolEnum NDStr::IMatches(CStr string1, CStr string2, StrIVAlPtr lengthptr);
```

NDStr::Matches tests whether string1 matches string2. The Boolean return value indicates whether the match was successful. If lengthptr is not NULL, it is set to the end of the match.

NDStr::IMatches is the same as NDStr::Matches but ignores case differences in the ASCII range only.

MatchesPat IMatchesPat

Tests whether one string matches another string containing a pattern.

```
static BoolEnum NDStr::MatchesPat(CStr string, CStr pattern, StrIVAlPtr lengthptr);
```

```
static BoolEnum NDStr::IMatchesPat(CStr string, CStr pattern, StrIVAlPtr lengthptr);
```

NDStr::MatchesPat tests whether pattern (which is a string containing a pattern) matches string. The Boolean return value indicates whether the match was successful. If lengthptr is not NULL, it is set to the end of the match.

pattern contains a very simple pattern which can accept a question mark (?) to indicate an optional character and an asterisk (*) to indicate any substring. Regular expressions are not supported.

NDStr::IMatchesPat is the same as NDStr::MatchesPat but it ignores case differences in the ASCII range only.

See also

NDStr::MatchesPatSub

MatchesPatSub IMatchesPatSub

Tests whether one substring matches another substring which contains a pattern.

```
static BoolEnum NDStr::MatchesPatSub(CStr subptr1, StrIVal length1, CStr subpattern ,
StrIVal length2, StrIValPtr lengthptr);
```

```
static BoolEnum NDStr::IMatchesPatSub(CStr subptr1, StrIVal length1, CStr subpattern ,
StrIVal length2, StrIValPtr lengthptr);
```

NDStr::MatchesPatSub tests whether a substring given by subpattern matches another substring given by subptr1. The Boolean return value indicates whether the match was successful. If lengthptr is not NULL, it is be set to the end of the match.

The second substring contains a very simple pattern which can accept a question mark (?) to indicate an optional character and an asterisk (*) to indicate any substring. Regular expressions are not supported.

NDStr::IMatchesPatSub is the same as NDStr::MatchesPatSub but it ignores case differences in the ASCII range only.

See also

NDStr::MatchesPat

MatchesSub

IMatchesSub

Tests whether two substrings match.

```
static BoolEnum NDStr::MatchesSub(CStr subptr1, StrIVal length1, CStr subptr2,
StrIVal length2, StrIValPtr lengthptr);
```

```
static BoolEnum NDStr::IMatchesSub(CStr subptr1, StrIVal length1, CStr subptr2,
StrIVal length2, StrIValPtr lengthptr);
```

NDStr::MatchesSub tests whether two substrings match. The Boolean return value indicates whether the match was successful. If lengthptr is not NULL, it is be set to the end of the match.

NDStr::IMatchesSub ignores case differences in the ASCII range only.

See also

NDStr::Matches

Searching

FindFirst

IFindFirst

Finds the first occurrence of a string.

```
static StrIVal NDStr::FindFirst(CStr string1, CStr string2);
```

```
static StrIVal NDStr::IFindFirst(CStr string1, CStr string2);
```

NDStr::FindFirst finds the first occurrence of string2 within string1 and returns its index. Returns -1 if the search fails.

NDStr::IFindFirst performs the same function but ignores case differences in the ASCII range only.

See also

`NDStr::FindLast`

FindFirstChar

Finds the first occurrence of a character.

static StrIVal NDStr::FindFirstChar(CStr string, ChCode chcode);

`NDStr::FindFirstChar` finds the first occurrence of `chcode` within `string` and returns the index. Returns -1 if the search fails.

See also

`NDStr::FindLastChar`

FindFirstCharSub

Find the first occurrence of a character in a substring.

static StrIVal NDStr::FindFirstCharSub(CStr subptr, StrIVal length, ChCode chcode);

`NDStr::FindFirstCharSub` finds the first occurrence of `chcode` in a substring and returns its index. Returns -1 if the search fails.

See also

`NDStr::FindFirstChar`, `NDStr::FindLastCharSub`

FindFirstSub

IFindFirstSub

Switchable case-independent search for the first occurrence of a substring.

static StrIVal NDStr::FindFirstSub(CStr subptr1, StrIVal length1, CStr subptr2, StrIVal length2);

static StrIVal NDStr::IFindFirstSub(CStr subptr1, StrIVal length1, CStr subptr2, StrIVal length2);

`NDStr::FindFirstSub` finds the first occurrence of a substring within a substring and returns the index. Returns -1 if the search fails.

`NDStr::IFindFirstSub` performs the same function but ignores case differences in the ASCII range only.

See also

`NDStr::FindFirst`, `NDStr::FindLastSub`

FindIFirst

Switchable case-independent search for the first occurrence of a string.

static StrIVal NDStr::FindIFirst(CStr string1, CStr string2, BoolEnum casef);

`NDStr::FindIFirst` finds the first occurrence of `string2` within `string1`, with or without taking the case into account. The Boolean argument set to true indicates that the search should ignore case in the ASCII range.

See also

`NDStr::FindILast`

FindFirstSub

Switchable case-independent search for the first occurrence of a substring.

```
static StrIVal NDStr::FindIFirstSub(CStr subptr1, StrIVal length1, CStr subptr2,  
StrIVal length2, BoolEnum casei);
```

NDStr::FindIFirstSub finds the first occurrence of a substring given by subptr2 within a substring, given by subptr1, with or without taking the case into account. When set to true, the casei argument indicates that the search should ignore case in the ASCII range.

See also

NDStr::FindIFirst, NDStr::FindILastSub

FindLast

Switchable case-independent search for the last occurrence of a string.

```
static StrIVal NDStr::FindILast(CStr string1, CStr string2, BoolEnum casei);
```

NDStr::FindILast finds the last occurrence of a string within another string, with or without taking the case into account. The Boolean argument set to true indicates that the search should ignore case in the ASCII range.

See also

NDStr::FindIFirst

FindLast IFindLast

Finds the last occurrence of a string.

```
static StrIVal NDStr::FindLast(CStr string1, CStr string2);
```

```
static StrIVal NDStr::IFindLast(CStr string1, CStr string2);
```

NDStr::FindLast finds the last occurrence of string2 within string1 and returns the index. Returns -1 if the search fails.

NDStr::IFindLast performs the same function but ignores case differences in the ASCII range only.

See also

NDStr::FindFirst

FindLastChar

Finds the last occurrence of a character within a string

```
static StrIVal NDStr::FindLastChar(CStr string, ChCode chcode);
```

NDStr::FindLastChar finds the last occurrence of a character in string and returns its index. Returns -1 if the search fails.

See also

NDStr::FindFirstChart

FindLastCharSub

Finds the last occurrence of a character in a substring.

```
static StrIVal NDStr::FindLastCharSub(CStr subptr, StrIVal length, ChCode chcode);
```

NDStr::FindLastCharSub finds the last occurrence of chcode in a substring and returns its index. Returns -1 if the search fails.

See also

NDStr::FindLastChar, NDStr::FindFirstCharSub

FindLastSub**FindILastSub**

Switchable case-independent search for the last occurrence of a substring.

```
static StrIVal NDStr::FindLastSub(CStr sbuptr1, StrIVal length1, CStr subptr2, StrIVal length2);
```

```
static StrIVal NDStr::FindILastSub(CStr sbuptr1, StrIVal length1, CStr subptr2, StrIVal length2, BoolEnum casei);
```

NDStr::FindILastSub finds the last occurrence of a substring given by subptr2 within a substring given by sbuptr1, with or without taking the case into account. The Boolean argument casei set to true indicates that the search should ignore case in the ASCII range.

NDStr::FindLastSub finds the last occurrence of a substring given by subptr2 within a substring, given by sbuptr1.

NDStr::IFindLastSub ignores case differences in the ASCII range only.

See also

NDStr::FindLast, NDStr::FindFirstSub

Scanning of Numeric Values

GetDec...

Returns the integer value found at the beginning of a decimal integer string.

```
static Int NDStr::GetDecInt(CStr string, StrIValPtr endptr);
```

```
static Int16 NDStr::GetDecInt16(CStr string, StrIValPtr endptr);
```

```
static Int32 NDStr::GetDecInt32(CStr string, StrIValPtr endptr);
```

```
static UInt NDStr::GetDecUInt(CStr string, StrIValPtr endptr);
```

```
static UInt16 NDStr::GetDecUInt16(CStr string, StrIValPtr endptr);
```

```
static UInt32 NDStr::GetDecUInt32(CStr string, StrIValPtr endptr);
```

These functions return the integer value found at the beginning of a decimal integer string if endptr is not NULL. endptr is set to the end of the numeric substring. The integer can be signed or unsigned and is assumed to be expressed in decimal notation. If string does not contain a numeric value, these calls return zero and set endptr to zero .

Calls are provided for `Int`, `Int32`, `Int16`, `UInt`, `UInt32` and `UInt16`. These versions cast the result of the corresponding `Int32` or `UInt32` calls and do not signal over/underflows.

See also

`NDStr::SubGetDec...`, `NDStr::GetHex...`, `NDStr::GetRadix...`,
`NDStr::GetDouble`

GetHex...

Returns the integer value found at the beginning of a hexadecimal integer string.

```
static Int NDStr::GetHexInt(CStr string, StrIVAlPtr endptr);
static Int16 NDStr::GetHexInt16(CStr string, StrIVAlPtr endptr);
static Int32 NDStr::GetHexInt32(CStr string, StrIVAlPtr endptr);
static UInt16 NDStr::GetHexUInt16(CStr string, StrIVAlPtr endptr);
static UInt32 NDStr::GetHexUInt32(CStr string, StrIVAlPtr endptr);
static UInt NDStr::GetHexUInt(CStr string, StrIVAlPtr endptr);
```

These functions return the integer value found at the beginning of a hexadecimal integer string if `endptr` is not `NULL`. `endptr` is set to the end of the numeric substring. The integer can be signed or unsigned and is assumed to be expressed in hexadecimal notation. If the string does not contain a numeric value, these calls return zero and set `endptr` to zero .

Calls are provided for `Int`, `Int32`, `Int16`, `UInt`, `UInt32` and `UInt16`. These versions cast the result of the corresponding `Int32` or `UInt32` calls and do not signal over/underflows.

See also

`NDStr::SubGetHex...`, `NDStr::GetDec...`, `NDStr::GetRadix...`,
`NDStr::GetDouble`

GetRadix...

Returns the integer value found at the beginning of an integer string.

```
static Int NDStr::GetRadixInt(CStr string, Int radix, StrIVAlPtr endptr);
static Int16 NDStr::GetRadixInt16(CStr string, Int radix, StrIVAlPtr endptr);
static Int32 NDStr::GetRadixInt32(CStr string, Int radix, StrIVAlPtr endptr);
static UInt NDStr::GetRadixUInt(CStr string, Int radix, StrIVAlPtr endptr);
static UInt16 NDStr::GetRadixUInt16(CStr string, Int radix, StrIVAlPtr endptr);
static UInt32 NDStr::GetRadixUInt32(CStr string, Int radix, StrIVAlPtr endptr);
```

These calls return the integer value found at the beginning of an integer string if `endptr` is not `NULL`. The radix of the integer can be a number between 2 and 36. The integer can be signed or unsigned. `endptr` is set to the end of the numeric substring. If the string does not contain a numeric value, these calls return zero and set `endptr` to zero .

Calls are provided for `Int`, `Int32`, `Int16`, `UInt`, `UInt32` and `UInt16`. These versions cast the result of the corresponding `Int32` or `UInt32` calls and do not signal over/underflows.

See also

`NDStr::GetDec...`, `NDStr::GetHex...`, `NDStr::GetDouble`

GetDouble

Returns the double real numeric value found at the beginning of a double real string.

static Double NDStr::GetDouble(CStr string, StrIVAlPtr endptr);

These functions return the double real value found at the beginning of a double real integer string if `endptr` is not NULL. If string does not contain a numeric value, this call returns zero and sets `endptr` to zero.

See also

`NDStr::SubGetDouble...`, `NDStr::GetDec...`, `NDStr::GetHex...`,
`NDStr::GetRadix...`

SubGetDec...

Returns the integer value found at the beginning of a decimal integer substring.

static Int NDStr::SubGetDecInt (CStr subptr, StrIVAl length, StrIVAlPtr endptr);

static Int16 NDStr::SubGetDecInt16 (CStr subptr, StrIVAl length, StrIVAlPtr endptr);

static Int32 NDStr::SubGetDecInt32 (CStr subptr, StrIVAl length, StrIVAlPtr endptr);

static UInt NDStr::SubGetDecUInt (CStr subptr, StrIVAl length, StrIVAlPtr endptr);

static UInt16 NDStr::SubGetDecUInt16 (CStr subptr, StrIVAl length, StrIVAlPtr endptr);

static UInt32 NDStr::SubGetDecUInt32 (CStr subptr, StrIVAl length, StrIVAlPtr endptr);

These functions return the integer value found at the beginning of a decimal integer substring of length if the endpoint pointer is not NULL. The endpoint is set to the end of the numeric substring. The integer can be signed or unsigned and is assumed to be expressed in decimal notation. If the substring does not contain a numeric value, these calls return zero and set the endpoint to zero .

Calls are provided for `Int`, `Int32`, `Int16`, `UInt`, `UInt32` and `UInt16`. These versions cast the result of the corresponding `Int32` or `UInt32` calls and do not signal over/underflows.

See also

`NDStr::GetDec...`, `NDStr::SubGetHex...`, `NDStr::SubGetRadix...`,
`NDStr::SubGetDouble`

SubGetHex...

Returns the integer value found at the beginning of a hexadecimal integer substring.

```

static Int NDStr::SubGetHexInt(CStr subptr, StrIVal length, StrIValPtr endptr);
static Int16 NDStr::SubGetHexInt16(CStr subptr, StrIVal length, StrIValPtr endptr);
static Int32 NDStr::SubGetHexInt32(CStr subptr, StrIVal length, StrIValPtr endptr);
static UInt NDStr::SubGetHexUInt(CStr subptr, StrIVal length, StrIValPtr endptr);
static UInt16 NDStr::SubGetHexUInt16(CStr subptr, StrIVal length, StrIValPtr endptr);
static UInt32 NDStr::SubGetHexUInt32(CStr subptr, StrIVal length, StrIValPtr endptr);

```

These functions return the integer value found at the beginning of a hexadecimal integer substring of length length if the endpoint pointer is not NULL. The endpoint is set to the end of the numeric substring. The integer can be signed or unsigned and is assumed to be expressed in hexadecimal notation. If the substring does not contain a numeric value, these calls return zero and set the endpoint to zero .

Calls are provided for Int, Int32, Int16, UInt, UInt32 and UInt16. These versions cast the result of the corresponding Int32 or UInt32 calls and do not signal over/underflows.

See also

NDStr::GetHex..., NDStr::SubGetDec..., NDStr::SubGetRadix...,
NDStr::SubGetDouble

SubGetRadix...

Returns the integer value found at the beginning of an integer substring.

```

static Int NDStr::SubGetRadixInt (CStr subptr, StrIVal length, Int radix, StrIValPtr endptr);
static Int16 NDStr::SubGetRadixInt16(CStr subptr, StrIVal length, Int radix,
    StrIValPtr endptr);
static Int32 NDStr::SubGetRadixInt32(CStr subptr, StrIVal length, Int radix,
    StrIValPtr endptr);
static UInt NDStr::SubGetRadixUInt(CStr subptr, StrIVal length, Int radix,
    StrIValPtr endptr);
static UInt16 NDStr::SubGetRadixUInt16(CStr subptr, StrIVal length, Int radix,
    StrIValPtr endptr);
static UInt32 NDStr::SubGetRadixUInt32(CStr subptr, StrIVal length, Int radix,
    StrIValPtr endptr);

```

These calls return the integer value found at the beginning of an integer substring if the endpoint is not NULL. The radix of the integer can be a number between 2 and 36. The integer can be signed or unsigned. The endpoint is set to the end of the numeric substring. If the substring does not contain a numeric value, these calls return zero and set the endpoint to zero .

Calls are provided for Int, Int32, Int16, UInt, UInt32 and UInt16. These versions cast the result of the corresponding Int32 or UInt32 calls and do not signal over/underflows.

See also

NDStr::GetRadix..., NDStr::SubGetDec..., NDStr::SubGetHex...,
NDStr::SubGetDouble

SubGetDouble

Returns the double real numeric value found at the beginning of a double real substring.

static Double NDStr::SubGetDouble(CStr subptr, StrIVal length, StrIValPtr endptr);

Returns the double real numeric value found at the beginning of a double real substring of length if the endpoint is not NULL. The endptr is set to the end of the numeric substring. If the substring does not contain a numeric value, this call return zero and sets the endpoint to zero.

See also

NDStr::GetDouble, NDStr::SubGetHex..., NDStr::SubGetRadix...,
NDStr::SubGetDec

Formating the Numeric Values**PutDec...**

Converts a decimal integer into its textual representation in a string buffer.

static StrIVal NDStr::PutDecInt(Str buffer, StrIVal size, Int int);

static StrIVal NDStr::PutDecInt16 (Str buffer, StrIVal size, Int16 int);

static StrIVal NDStr::PutDecInt32(Str buffer, StrIVal size, Int32 int);

static StrIVal NDStr::PutDecUInt(Str buffer, StrIVal size, UInt int);

static StrIVal NDStr::PutDecUInt16(Str buffer, StrIVal size, UInt16 int);

static StrIVal NDStr::PutDecUInt32(Str buffer, StrIVal size, UInt32 int);

The NDStr::PutDec... functions convert a decimal integer into its textual representation in a string buffer. These functions convert 8-bit, 16-bit, and 32-bit decimal integers. The NDStr::PutDecU... functions convert unsigned decimal integers.

See also

NDStr::Put, NDStr::PutHex..., NDStr::PutRadix..., NDStr::PutDouble

PutHex...

Converts a hexadecimal integer into its textual representation in a string buffer.

static StrIVal NDStr::PutHexInt(Str buffer, StrIVal size, Int int);

static StrIVal NDStr::PutHexInt16(Str buffer, StrIVal size, Int16 int);

static StrIVal NDStr::PutHexInt32(Str buffer, StrIVal size, Int32 int);

static StrIVal NDStr::PutHexUInt(Str buffer, StrIVal size, UInt int);

static StrIVal NDStr::PutHexUInt16(Str buffer, StrIVal size, UInt16 int);

static StrIVal NDStr::PutHexUInt32(Str buffer, StrIVal size, UInt32 int);

Converts a hexadecimal integer into its textual representation in a string buffer. The NDStr::PutHexU... functions convert unsigned hexadecimal

integers. These functions convert 8-bit, 16-bit, and 32-bit hexadecimal integers.

See also

NDStr::Put, NDStr::PutDec..., NDStr::PutRadix..., NDStr::PutDouble

PutRadix...

Using the radix, converts an integer into its textual representation in the string buffer.

static StrIVal NDStr::PutRadixInt(Str buffer, StrIVal size, Int radix, Int int);

static StrIVal NDStr::PutRadixInt16(Str buffer, StrIVal size, Int radix, Int16 int);

static StrIVal NDStr::PutRadixInt32(Str buffer, StrIVal size, Int radix, Int32 int);

static StrIVal NDStr::PutRadixUInt(Str buffer, StrIVal size, Int radix, UInt int);

static StrIVal NDStr::PutRadixUInt16(Str buffer, StrIVal size, Int radix, UInt16 int);

static StrIVal NDStr::PutRadixUInt32(Str buffer, StrIVal size, Int radix, UInt32 int);

Using the radix, converts an integer into its textual representation in the string buffer. For example, NDStr::PutRadixInt converts 100 to "64" if the radix is 16.

The NDStr::PutRadixU... functions convert unsigned hexadecimal integers. These functions convert 8-bit, 16-bit, and 32-bit integers.

See also

NDStr::Put, NDStr::PutDec..., NDStr::PutHex..., NDStr::PutDouble

PutDouble

Converts a double precision value into its textual representation in the string buffer.

static StrIVal NDStr::PutDouble(Str buffer, StrIVal size, Double double);

NDStr::PutDouble converts a double precision value into its textual representation in the string buffer.

See also

NDStr::Put, NDStr::PutDec..., NDStr::PutHex...

Basic Conversions

AsciiUpCase

Converts a string to upper case.

static void NDStr::AsciiUpCase(Str string);

NDStr::AsciiUpCase converts the ASCII characters in the string to upper case. Non-ASCII characters are ignored. string is converted in place.

See also

NDStr::AsciiUpCaseSub, NDStr::AsciiDownCase,
NDStr::AsciiDownCaseSub

AsciiUpCaseSub

Converts a substring to upper case.

static void NDStr::AsciiUpCaseSub(Str subptr, StrIVal length);

NDStr::AsciiUpCaseSub converts the ASCII characters in the substring to upper case. Non-Ascii characters are ignored. The substring is converted in place.

See also

NDStr::AsciiUpCase, NDStr::AsciiDownCase, NDStr::AsciiDownCaseSub

AsciiDownCase

Converts a string to lower case.

static void NDStr::AsciiDownCase(Str string);

NDStr::AsciiDownCase converts the ASCII characters in the string to lower case. Non-ASCII characters are ignored. string is converted in place.

See also

NDStr::AsciiUpCase, NDStr::AsciiDownCaseSub

AsciiDownCaseSub

Converts a substring to lower case.

static void NDStr::AsciiDownCaseSub(Str subptr, StrIVal length);

NDStr::AsciiDownCaseSub converts the ASCII characters in the substring to lower case. Non-ASCII characters are ignored. The substring is converted in place.

See also

NDStr::AsciiUpCase, NDStr::AsciiDownCase, NDStr::AsciiDownCaseSub

PutAsciiUpper

Same as NDStr::Put, but also converts the ASCII characters in the string to upper case.

**static StrIVal NDStr::PutAsciiUpper(Str buffer, StrIVal size, CStr string,
StrIValPtr lengthptr);**

Same as NDStr::Put, but also converts the ASCII characters in the string to upper case. The string is converted in place. NDStr::PutAsciiUpper permits ASCII conversion.

See also

NDStr::PutAsciiLower, NDStr::PutAsciiUpperSub

PutAsciiLower

Same as `NDStr::Put`, but also converts the ASCII characters in the string to lower case.

```
static StrIVal NDStr::PutAsciiLower(Str buffer, StrIVal size, CStr string,
StrIValPtr lengthptr);
```

Same as `NDStr::Put`, but also converts the ASCII characters in the string to lower case. The `string` is converted in place. `NDStr::PutAsciiLower` permits ASCII conversion or simple code conversion only.

See also

`NDStr::Put`, `NDStr::PutAsciiUpper`, `NDStr::PutAsciiLowerSub`

PutAsciiUpperSub

Same as `NDStr::PutSub`, but also converts the ASCII characters in the substring to upper case.

```
static StrIVal NDStr::PutAsciiUpperSub(Str buffer, StrIVal size, CStr subptr, StrIVal length,
StrIValPtr endptr);
```

Same as `NDStr::PutSub`, but also converts the ASCII characters in the substring to upper case. The `substring` is converted in place. `NDStr::PutAsciiUpperSub` permits ASCII conversion.

See also

`NDStr::PutSub`, `NDStr::PutAsciiUpperSub`

PutAsciiLowerSub

Same as `NDStr::PutSub`, but also converts a substring to lower case.

```
static StrIVal NDStr::PutAsciiLowerSub(Str buffer, StrIVal size, CStr subptr, StrIVal length,
StrIValPtr endptr);
```

Same as `NDStr::PutSub`, but also converts the ASCII characters in the substring to lower case. The `substring` is converted in place. `NDStr::PutAsciiLowerSub` permits ASCII conversion

See also

`NDStr::PutSub`, `NDStr::PutAsciiUpperSub`

Loading from Resources

ResLoad**ResLoadNth**

Returns a string from a `StrR` or `StrL` resource.

```
static CStr NDStr::ResLoad(CStr mod, CStr resource, StrIValPtr lengthptr);
```

```
static CStr NDStr::ResLoadNth(CStr mod, CStr resource, ArrayIVal n, StrIValPtr lengthptr);
```

Returns a string from a `StrR` or `StrL` resource. If the length pointer is not `NULL`, it is set to the length of the string. If the resource does not exist, the function signals an error.

`NDStr::ResLoadNth` returns the *n*th string in the resource.

See also

`NDStr::ResFind`

ResFind
ResFindNth

Finds and returns a string from a `StrR` or `StrL` resource.

static CStr NDStr::ResFind(CStr mod, CStr resource, StrIVaIPtr lengthptr);

static CStr NDStr::ResFindNth(CStr mod, CStr resource, ArrayIVaI n, StrIVaIPtr lengthptr);

Finds and returns a string from a `StrR` or `StrL` resource. If the length pointer is not `NULL`, it is set to the length of the string. If the resource does not exist, the function returns `NULL`.

`NDStr::ResLoadNth` finds and returns the *n*th string in the resource.

See also

`NDStr::ResLoad`.

Conversions Between Code Types

FromCt

static StrIVaI NDStr::FromCt(Str buf, StrIVaI size, NatCStr ctstr, CtCPtr ct, StrCvtCtxPtr ctx);

static StrIVaI NDStr::FromCtSub(Str buf, StrIVaI size, NatCStr ctbuf, StrIVaI ctslen, CtCPtr ct, StrCvtCtxPtr ctx);

ToCt

static StrIVaI NDStr::ToCt(NatStr ctbuf, StrIVaI size, CStr str, CtCPtr ct, StrCvtCtxPtr ctx);

static StrIVaI NDStr::ToCtSub(NatStr ctbuf, StrIVaI size, CStr str, StrIVaI slen, CtCPtr ct, StrCvtCtxPtr ctx);

Converts ``ctstr'`, a ``ct'` encoded string to ``buf'`, a `Str`, or ``str'`, a `Str` to ``ctbuf'`, a ``ct'` encoded buffer. The conventions for ``buf/ctbuf'`, ``size'` and ``len'` are the standard ``Put'` conventions (see above). If ``ct'` is `NULL`, the native code type is assumed. ``ctx'` may be `NULL` or a pointer to an `StrCvtCtx` structure which will be filled with the ``buf'` and ``str'` positions where the conversion can be resumed after the destination buffer has been reallocated.

`NDStr::ToCt` may stop converting if ``str'` contains characters which do not belong to the ``ct'` code set. In this case ``ctx->FmtPos'` will be less than the length of ``str'`. This will only happen if your application mixes strings encoded in various code sets. When this happens, you should inquire the code set of the offending character and resume conversion with a code type which covers this code set. An alternative is to ignore offending characters and eventually replace them with a “missing” character (i.e. ?).

FromUni

```
static StrIVal NDStr::FromUni (Str buf, StrIVal size, UniCStr unistr, CharCvtSet flags,  
    StrCvtCtxPtr ctx);
```

```
static StrIVal NDStr::FromUniSub(Str buf, StrIVal size, UniCStr unistr, StrIVal unilen,  
    CharCvtSet flags, StrCvtCtxPtr ctx);
```

ToUni

```
static StrIVal NDStr::ToUni (UniStr unistr, StrIVal unisize, CStr str, CharCvtSet flags,  
    StrCvtCtxPtr ctx);
```

```
static StrIVal NDStr::ToUniSub (UniStr unistr, StrIVal unisize, CStr str, StrIVal slen,  
    CharCvtSet flags, StrCvtCtxPtr ctx);
```

Routines to convert between internal string (Str) and UNICODE (UniStr). The `unysize` and `unilen` are sizes and length in number of 16 bit integers, not in number of bytes. The `flags` allow to specify conversion flags to control UNICODE specific conversion options such as precomposed vs decomposed form, compatibility area mapping, ...

Only the UNICODE specific conversion flags are supported here. Other types of conversions (i.e. case conversions) are only supported on the Str type.

42 *StrL Class*

The StrL class implements the Open Interface string data structures and utilities.

Technical Summary

Open Interface supports a two types of strings: standard C strings (Str) and variable length (VStr) strings. The Str class provides the tools for standard C strings that may or may not allow for two byte characters for use in languages other than English.

The API is divided into tools for conversions, comparisons, concatenations, formatting, and string queries. Many of the tools are similar to standard C libraries, but are implemented to support C strings.

The Str class API is divided into the following categories:

- String formatting (sprintf).
- String length.
- String scan (sscanf).

The string list class inherits its fields along the following path:

As a subclass of the Res (resource) class, the string list resource class inherits the same fields defined for the resource class. The string list resource class has no class-specific fields.

See also:

StrR and Str classes.

Class

Class

Returns a pointer to the string list class.

static RClasPtr NDStrL::Class (void);

NDStrL::Class returns a pointer to a string list class data structure.

Accessing the Strings

GetLen

Returns the number of strings stored in a string list resource.

ArrayIVal NDStrL::GetLen (void);

NDStrL::GetLen returns the number of strings stored in the string list resource specified by strlist.

GetNthStr

Returns a string specified by index from a string list resource.

CStr NDStrL::GetNthStr (ArrayIVal *index*);

NDStrL::GetNthStr returns a string specified by index from the string list resource specified by *strlist*. The first string should be retrieved with the index zero. If the string list resource contains *N* strings, the last one is therefore specified at index *N*-1. It returns NULL if the resource does not exist or the index is out of range.

See also

NDStrL::SetNthStr

SetNthStr

Replaces an existing string in a string list resource with a new string.

void NDStrL::SetNthStr (ArrayIVal *index*, CStr *string*);

NDStrL::SetNthStr substitutes a string in the string list resource specified by *strlist* and *index* with a new string. The first string should be set with the index zero. If the string list resource contains *N* strings, the last one is therefore specified at index *N*-1. It returns NULL if the resource does not exist or the string index is out of range.

See also

NDStrL::GetNthStr, NDStrL::AddStr, NDStrL::AddStrAtIndex

AddStr

Adds a new string to a string list resource.

void NDStrL::AddStr (CStr *string*);

NDStrL::AddStr adds *string* to the string list resource specified by *strlist*. The new string appears last in the list of strings.

See also

NDStrL::AddStrAtIndex, NDStrL::RemoveIndex, NDStrL::SetNthStr

AddStrAtIndex

Inserts a new string in a string list resource at the position specified.

void NDStrL::AddStrAtIndex (ArrayIVal *index*, CStr *string*);

NDStrL::AddStrAtIndex adds *string* to the string list resource specified by *strlist*. The new string's insertion point is determined by *index* which specifies the number of the string to insert the new string before.

See also

NDStrL::AddStr, NDStrL::RemoveIndex, NDStrL::SetNthStr

RemoveIndex

Removes a string from a string list resource at the position specified.

void NDCtrl::RemoveIndex (ArrayIVal index);

NDCtrl::RemoveIndex removes a string from the string list resource specified by strlist. Index specifies which string to remove from the list.

See also

NDCtrl::AddStr, NDCtrl::AddStrAtIndex, NDCtrl::SetNthStr

LoadNthStr

Returns a string from a string list resource by resource name and index.

static CStr NDCtrl::LoadNthStr (CStr modname, CStr stringresname, ArrayIVal index);

NDCtrl::LoadNthStr returns the string resource specified by the modname and stringresname. Index specifies which string to load from the list. It fails if the resource does not exist. It returns NULL if the index is out of the string list range.

See also

NDCtrl::LOADNTHSTR, NDCtrl::FindNthStr

FindNthStr

Returns the string of a string list resource.

static CStr NDCtrl::FindNthStr (CStr modname, CStr stringresname, ArrayIVal index);

NDCtrl::FindNthStr returns the string resource specified by the modname and stringresname. Index specifies which string to return from the list. It returns NULL if the index is out of the string list range or the resource does not exist.

See also

NDCtrl::FindNthStr, NDCtrl::LoadNthStr

43 *StrR Class*

The StrR class implements the Open Interface string data structures and utilities.

Technical Summary

Open Interface supports a two types of strings: standard C strings (Str) and variable length (VStr) strings. The Str class provides the tools for standard C strings that may or may not allow for two byte characters for use in languages other than English.

The API is divided into tools for conversions, comparisons, concatenations, formatting, and string queries. Many of the tools are similar to standard C libraries, but are implemented to support C strings.

The Str class API is divided into the following categories:

- String formatting (sprintf).
- String length.
- String scan (sscanf).

The string resource class inherits its fields along the following path:

Res -> StrR

As a subclass of the Res (resource) class, the string resource class inherits the same fields defined for the resource class. The string resource class has no class-specific fields.

See also:

StrR and Str classes.

Class

Class

Returns a pointer to the string resource class.

static RClasPtr NDStrR::Class (void);

NDStrR::Class returns a pointer to a string resource class data structure.

Loading a String Resource

LoadStr

Loads the string resource by resource name.

static CStr NDCStrR::LoadStr (CStr *modname*, CStr *stringresname*);

NDCStrR::LoadStr loads the string resource specified by the *modname* and *stringresname*. It fails if the resource does not exist.

FindStr

Returns the string contained in a string resource.

static CStr NDCStrR::FindStr (CStr *modname*, CStr *stringresname*);

NDCStrR::FindStr returns the string resource specified by the *modname* and *stringresname*. It returns NULL if the resource does not exist.

Accessing Text

GetStr

Returns the string contained in a string resource.

CStr NDCStrR::GetStr (void);

NDCStrR::GetStr returns the string from the string resource specified by *stres*.

SetStr

Replaces the string in a string resource with a new string.

void NDCStrR::SetStr (CStr *string*);

NDCStrR::SetStr substitutes the string in the string resource specified by *stres* with a new string.

Accessing the Id

GetId

Returns the id of a string resource.

StrIdVal NDCStrR::GetId (void);

NDCStrR::GetStr returns the Id contained in the string resource.

SetId

Changes the id value of a string resource.

void NDCStrR::SetId (StrIdVal *Id*);

NDCStrR::SetId changes the Id contained in the string resource.

44 *Var Class*

This file defines the variant data type.

Type System

VarTypeEnum

Methods	Description
<code>VAR_TYPE_NONE</code>	No type. Used for initializing type tags. Equivalent to <code>void</code> .
<code>VAR_TYPE_UNSUPPORTED</code>	Unsupported type. Used when there is no OA type which matches a particular native type.
<code>VAR_TYPE_VOID_PTR</code>	Type for a pointer to void

Basic Types

Basic types have a direct C mapping, so their type descriptors contain all the type information that is needed to interact with them.

Methods

```

VAR_TYPE_BASE_INT
VAR_TYPE_BASE_UINT
VAR_TYPE_BASE_LONG
VAR_TYPE_BASE_ULONG
VAR_TYPE_BASE_FLOAT
VAR_TYPE_BASE_DOUBLE
VAR_TYPE_BASE_CHAR
VAR_TYPE_BASE_WCHAR
VAR_TYPE_BASE_BOOLEAN
VAR_TYPE_BASE_BYTE

```

C, C++ and Corba Basic Types**Methods**

```

VAR_TYPE_BASE_INT8
VAR_TYPE_BASE_INT16
VAR_TYPE_BASE_INT32
VAR_TYPE_BASE_INT64
VAR_TYPE_BASE_UINT8
VAR_TYPE_BASE_UINT16
VAR_TYPE_BASE_UINT32
VAR_TYPE_BASE_UINT64

```

ND-Specific Basic Types (Implementation Types)**Methods**

```

VAR_TYPE_BASE_DATE
VAR_TYPE_BASE_TIME
VAR_TYPE_BASE_CURRENCY

```

ND-Specific Character-Related Types

The difference between a character code and a wide char is that in general, a string will not be an array of character codes, whereas a wide string `_is_` an array of wide chars. Moreover, depending on the encoding system, a character code may exceed 2^{16} , while a `wchar_t` value is a 16 bits value.

Methods

```

VAR_TYPE_BASE_CHARCODE

```

Native Constructed Types**Methods**

```

VAR_TYPE_NAT_STR
VAR_TYPE_NAT_WSTR

```

Description

An array of `char` (regardless of code set & of the encoding system)
 An array of `wchar_t` (to be defined)

“Power” Types**Methods**

```

VAR_TYPE_VAR
VAR_TYPE_OAOBJ
VAR_TYPE_NULL
VAR_TYPE_NULLOBJ

```

Description

Type for an “any”, i.e. a type that can hold a value of any other type. Called `VARIANT` in a system such as OLE2. Does not have an immediate form.
 Reference to an Object Access object
 The variant contains a `NULL` value
 Reference to a `NULL` object

“Variant” Management

“Value” Substructure

Substructure of the “any” structure described below Forward declarations of types needed by VAR. The following types are not defined in core yet. They should be defined at one point.

```
typedef Char VARWChar;
```

Class

The nested classes are a hack to allow overloading based on our derived type definitions (this hack is documented in Taligent's book, page. 105)

InitClass

```
static void NDVar::InitClass(void);
```

Called by OA library initialization to initialize the NDVar class. Must be called before any instances of NDVar are created.

UnloadClass

```
static void NDVar::UnloadClass(void);
```

Called by OA library exit to unload any static initialization of this class.

Constructors

```
NDVar::NDVar(void);
```

Default constructor. Constructs an empty OA variant initialized to the type VAR_TYPE_NONE.

```
NDVar::NDVar(NDVarCRef anyToCopy);
```

Copy constructor. Does a shallow copy, i.e. references are aliased.

```
NDVar::NDVar(const NDVar::C_<type> C_FAR& v)
```

Constructor from value. C++ does not let us define overloaded constructors based on derived types that might come from the same base type, so we work around the problem by defining nested classes that allow us to do the thing. For example, we can write:

```
NDVar var(NDVar::C_Int32(23));
NDVar(NDVar::C_Int32(32));
```

Destructor

```
~NDVar(void);
```

```
NDVar::~~NDVar(void);
```

Destroys the OA variant and deletes any string, object or variant referenced by this variant.

Assignment

NDVarRef NDVar::operator=(NDVarCRef anyToCopy);

Assignment operator. Does a shallow copy of `anyToCopy` into this object. i.e., References are aliased.

Conversion Methods

The caller does not care about the actual type contained within the variant, and simply asks for a conversion (in place or not) into a given type. For each conversion, two possibilities are given: make a call that will fail if the conversion is not possible (standard call) or make a call that will return a success code.

Convert

TryConvert

void NDVar::Convert(VarTypeEnum destType);

BoolEnum NDVar::TryConvert(VarTypeEnum destType);

Converts the type of this any to the type specified by `destType`.

ConvertToValue

TryConvertToValue

void NDVar::ConvertToValue(void);

BoolEnum NDVar::TryConvertToValue(void);

If this any contains a reference the method converts this any to a value obtained by dereferencing the reference.

CopyToType

TryCopyToType

VarPtr NDVar::CopyToType(VarTypeEnum destType);

BoolEnum NDVar::TryCopyToType(VarTypeEnum destType, VarPtrPtr valuePtr);

Returns an NDVar which contains this any converted to the type specified by `destType`.

CopyToValue

TryCopyToValue

VarPtr NDVar::CopyToValue(void);

BoolEnum NDVar::TryCopyToValue(NDVarPtr destValue);

If this contains a reference, the method returns an NDVar containing a value obtained by dereferencing the reference. Otherwise returns a NDVar which contains a copy of the value in this any.

CopyToInt
TryCopyToInt

Int NDVar::CopyToInt(void);
BoolEnum NDVar::TryCopyToInt(IntPtr valuePtr);
Int8 NDVar::CopyToInt8(void);
BoolEnum NDVar::TryCopyToInt8(Int8Ptr valuePtr);
Int16 NDVar::CopyToInt16(void);
BoolEnum NDVar::TryCopyToInt16(Int16Ptr valuePtr);
Int32 NDVar::CopyToInt32(void);
BoolEnum NDVar::TryCopyToInt32(Int32Ptr valuePtr);
Int64 NDVar::CopyToInt64(void);
BoolEnum NDVar::TryCopyToInt64(Int64Ptr valuePtr);

If this object contains a reference this method writes the value of `ori` into the reference. If this object does not contain a reference an exception is generated.

CopyToUInt
TryCopyToUInt

UInt NDVar::CopyToUInt(void);
BoolEnum NDVar::TryCopyToUInt(UIntPtr valuePtr);
UInt8 NDVar::CopyToUInt8(void);
BoolEnum NDVar::TryCopyToUInt8(UInt8Ptr valuePtr);
UInt16 NDVar::CopyToUInt16(void);
BoolEnum NDVar::TryCopyToUInt16(UInt16Ptr valuePtr);
UInt32 NDVar::CopyToUInt32(void);
BoolEnum NDVar::TryCopyToUInt32(UInt32Ptr valuePtr);
UInt64 NDVar::CopyToUInt64(void);
BoolEnum NDVar::TryCopyToUInt64(UInt64Ptr valuePtr);

If this object contains a reference this method writes the value of `ori` into the reference. If this object does not contain a reference an exception is generated.

CopyToLong
TryCopyToLong

Long NDVar::CopyToLong(void);
BoolEnum NDVar::TryCopyToLong(LongPtr valuePtr);
ULong NDVar::CopyToULong(void);
BoolEnum NDVar::TryCopyToULong(ULongPtr valuePtr);

CopyToFloat
TryCopyToFloat

Float NDVar::CopyToFloat(void);
BoolEnum NDVar::TryCopyToFloat(FloatPtr valuePtr);

CopyToDouble
TryCopyToDouble

Double NDVar::CopyToDouble(void);
BoolEnum NDVar::TryCopyToDouble(DoublePtr valuePtr);

CopyToChar
TryCopyToChar

Char NDVar::CopyToChar(void);
BoolEnum NDVar::TryCopyToChar(CharPtr valuePtr);

CopyToVARWChar
TryCopyToVARWChar

VARWChar NDVar::CopyToVARWChar(void);
BoolEnum NDVar::TryCopyToVARWChar(VARWCharPtr valuePtr);

CopyToBoolean
TryCopyToBoolean

BoolEnum NDVar::CopyToBoolean(void);
BoolEnum NDVar::TryCopyToBoolean(BoolEnumPtr valuePtr);

CopyToByte
TryCopyToByte

Byte NDVar::CopyToByte(void);
BoolEnum NDVar::TryCopyToByte(BytePtr valuePtr);

CopyToCharCode
TryCopyToCharCode

CharCode NDVar::CopyToCharCode(void);
BoolEnum NDVar::TryCopyToCharCode(CharCodePtr valuePtr);

CopyToStr
TryCopyToStr

Str NDVar::CopyToStr(void);
BoolEnum NDVar::TryCopyToStr(StrPtr valuePtr);
 Caller must delete the returned Str.

CopyToVARWStr
TryCopyToVARWStr

VARWStr NDVar::CopyToVARWStr(void);
BoolEnum NDVar::TryCopyToVARWStr(VARWStrPtr valuePtr);
 Caller must delete the returned VARWStr.

CopyToClientPtr

TryCopyToClientPtr

ClientPtr NDVar::CopyToClientPtr(void);

BoolEnum NDVar::TryCopyToClientPtr(ClientPtrPtr valuePtr);

Caller must NOT delete the returned ClientPtr..

Update

TryUpdate

void NDVar::Update(NDVarCRef ori);

BoolEnum NDVar::TryUpdate(NDVarCRef ori);

Information Methods

Clear

void NDVar::Clear(void);

Empties this variant. After this call the variant will be of type VAR_TYPE_NONE.

GetType

VarTypeEnum NDVar::GetType(void);

Returns the type contained in this variant.

ContainsRef

BoolEnum NDVar::ContainsRef(void);

Returns BOOL_TRUE if this variant contains a reference. Returns BOOL_FALSE if this variant contains a value.

IsEmpty

BoolEnum NDVar::IsEmpty(void);

Returns BOOL_TRUE if the type of this variant is VAR_TYPE_NONE. Returns BOOL_FALSE otherwise.

IsNull

BoolEnum NDVar::IsNull(void);

Returns BOOL_TRUE if the type of this variant is VAR_TYPE_NULL. Returns BOOL_FALSE otherwise.

IsNullObj

BoolEnum NDVar::IsNullObj(void);

Returns BOOL_TRUE if the type of this variant is VAR_TYPE_NULLOBJ. Returns BOOL_FALSE otherwise.

45 *VarDs Class*

This module specifies the variant data source.

Variant Data Source Value

This module implements the variant data source. A variant data source is a data source that keeps track of a single value stored in a variant. The variant data source can contain a value of any of the types supported by the Variant class (see `varpub.h`).

The variant data source is a subclass of the pure virtual data source, and implements specific methods to get and set the associated value.

Class

static RClasPtr NDVarLs::Class(void);

Returns a pointer to the variant list data source resource class.

QueryValue

void NDVarDs::QueryValue(VarPtr var);

Returns the value associated to the variant data source in `var`.

GetValue

VarPtr NDVarDs::GetValue(void);

Returns the value associated to the variant data source. The value returned should be destructed and disposed, `VARDS_QueryValue` is a preferable call.

SetValue

BoolEnum NDVarDs::SetValue(VarPtr var);

Sets the value of the data source by internally creating an edition object and committing the changes created through it.

void NDVarDsEdit::SetValue(VarPtr var);

Sets the value of the data source edition object. Once the edition object is committed, the value will copied into the associate variant data source.

Notifications

DefNfy

void NDVarDs::DefNfy(VarDsNfyEnum code);

Default notification procedure for the VarDs class

Variant Data Source

```
RClasPtr VarDsGetClass(void);  
void VarDsConstruct(ResPtr res, RClasCPtr rclas, RClasCreateCPtr create);  
void VarDsDestruct(ResPtr res);
```

Design Overview

This module implements the graph of variant datasources. A *graph of variant datasources* is a datasource that keeps track of a collection of nodes and edges that have properties stored in variants. Each one of these variants can hold a value of any of the types supported by the Variant class (see `VARPUB.H`).

The graph of variant datasources contains a collection of nodes and edges. The nodes and edges are never accessed directly as objects. Instead, node and edge *accessors* are created as pointers to nodes and edges contained by the graph datasource. The accessors are then used in conjunction with the graph object to operate on nodes and edges, and to set and get properties on the nodes and edges.

The DataSource data structure is private. It is a subclass of Res.

Class

static RClasPtr NDVarGr::Class(void);

Returns a pointer to the resource class for VarGr.

Graph Properties

Methods of the graph object associated with the graph title and number of nodes in the graph.

Graph Title

The graph of variant datasources can have a title that views can use to identify it.

GetTitle

CStr NDVarGr::GetTitle(void);

SetTitle

BoolEnum NDVarGr::SetTitle(CStr str);

Gets and sets the title for the graph, if any. The Set routine sets the title of the table to "title" by internally creating and committing an edit, and returns `BOOL_TRUE` if the internal edit succeeded, `BOOL_FALSE` if it did not.

SetTitle

void NDVarGrEdit::SetTitle(CStr str);

Sets the title of the graph edit object to "title." When the edit object is committed, the title will be copied into the graph.

GetNumNodes

VarGrIndexVal NDVarGr::GetNumNodes(void);

Returns the number of nodes in the graph.

GetNumRootNodes

VarGrIndexVal NDVarGr::GetNumRootNodes(void);

Returns the number of root nodes in the graph. A *root node* is any node with no parent.

GetNumEdges

VarGrIndexVal NDVarGr::GetNumEdges(void);

Returns the number of edges in the graph.

Node and Edge Accessors

Use accessors to traverse the graph datasource for nodes and edges.

Node Accessor

The node accessor is created and manipulated completely independently of the graph. It is a pointer to a node.

NDVarGrNodeAccessor

NDVarGrNodeAccessor::NDVarGrNodeAccessor(void);

Default node-accessor construction.

NDVarGrNodeAccessor::NDVarGrNodeAccessor(VarGrNodeAccessorCPtr *cnode*);

Constructs the node accessor with the information obtained from “*cnode*.”

Edge Accessors

There is an *all* edge accessor that can navigate through all edges in the graph, and there is an individual class of edge accessor for each of the three types of edges that a node may have. From the perspective of a node, an *in* edge comes from a parent, an *out* edge leads to a child, and an *undirected* edge leads to a neighbor.

“All” Edge Accessor

NDVarGrAllEdgeAccessor

NDVarGrAllEdgeAccessor::NDVarGrAllEdgeAccessor(void);

**NDVarGrAllEdgeAccessor::NDVarGrAllEdgeAccessor
(VarGrAllEdgeAccessorCPtr *cedge*);**

Default edge-accessor construction.

**NDVarGrAllEdgeAccessor::NDVarGrAllEdgeAccessor
(VarGrAllEdgeAccessorCPtr *cedge*);**

Constructs the edge accessor with the information obtained from “*cedge*.”

“In” edge accessor

`NdVarGrInEdgeAccessor`

`NdVarGrInEdgeAccessor::NdVarGrInEdgeAccessor(VarGrNodeAccessorPtr node);`

Default “in” edge-accessor construction.

`NdVarGrInEdgeAccessor::NdVarGrInEdgeAccessor(VarGrInEdgeAccessorCPtr cedge);`

Constructs the “in” edge accessor with the information obtained from “*cedge*.”

“Out” Edge Accessor

`NdVarGrOutEdgeAccessor`

`NdVarGrOutEdgeAccessor::NdVarGrOutEdgeAccessor(VarGrNodeAccessorPtr node);`

Default “out” edge-accessor construction.

`NdVarGrOutEdgeAccessor`

`NdVarGrOutEdgeAccessor::NdVarGrOutEdgeAccessor(VarGrOutEdgeAccessorCPtr cedge);`

Constructs the “out” edge accessor with the information obtained from “*cedge*.”

Undirected Edge Accessor

`NdVarGrUndirEdgeAccessor`

`NdVarGrUndirEdgeAccessor::NdVarGrUndirEdgeAccessor(VarGrNodeAccessorPtr node);`

Default undirected-edge-accessor construction.

`NdVarGrUndirEdgeAccessor::NdVarGrUndirEdgeAccessor(VarGrUndirEdgeAccessorCPtr cedge);`

Constructs the undirected edge accessor with the information obtained from “*cedge*.”

Node Accessors Navigation

This information describes how to navigate in a graph datasource with a node accessor. The node accessor can be absolutely positioned using these methods:

- `GoFirstRoot`
- `GoNthRoot`
- `GoIndexed`
- `GoID`

The following methods are relative to the current node the accessor is already positioned on:

- `GoFirstParent`
- `GoNthParent`

- GoFirstChild
- GoNthChild
- GoFirstNeighbor
- GoNthNeighbor
- GoNext
- GoPrev

GoFirstRoot**void NDVarGrNodeAccessor::GoFirstRoot(void);**

Positions the node accessor on the first root node in the graph.

GoNthRoot**void NDVarGrNodeAccessor::GoNthRoot(VarGrIndexVal *index*);**

Positions the node accessor on the Nth root node identified by index “index” in the graph.

GoIndexed**void NDVarGrNodeAccessor::GoIndexed(VarGrIndexVal *index*);**

Positions the node accessor on the node identified by index “index.”

GoID**void NDVarGrNodeAccessor::GoID(VarPtr *id*);**

Positions the node accessor on the node identified by ID “id.”

GoFirstParent**void NDVarGrNodeAccessor::GoFirstParent(void);**

Positions the node accessor on the first parent node of the node where it is currently positioned.

GoNthParent**void NDVarGrNodeAccessor::GoNthParent(VarGrIndexVal *index*);**

Positions the node accessor on the Nth parent node identified by index “index” of the node where it is currently positioned.

GoFirstChild**void NDVarGrNodeAccessor::GoFirstChild(void);**

Positions the node accessor on the first child node of the node where it is currently positioned.

GoNthChild**void NDVarGrNodeAccessor::GoNthChild(VarGrIndexVal *index*);**

Positions the node accessor on the first Nth node identified by index “index” of the node where it is currently positioned.

GoFirstNeighbor

void NDVarGrNodeAccessor::GoFirstNeighbor(void);

Positions the node accessor on the first neighbor node of the node where it is currently positioned.

GoNthNeighbor

void NDVarGrNodeAccessor::GoNthNeighbor(VarGrIndexVal index);

Positions the node accessor on the Nth neighbor node identified by index “index” of the node where it is currently positioned.

GoNext

void NDVarGrNodeAccessor::GoNext(void);

Positions the node accessor on the next node. Should be used after a GoFirstRoot, GoFirstParent, GoFirstNeighbor, or GoFirstChild, or after a GoNthRoot, GoNthParent, GoNthNeighbor, or GoNthChild.

GoPrev

void NDVarGrNodeAccessor::GoPrev(void);

Positions the node accessor on the previous node. Should be used after a GoFirstRoot, GoFirstParent, GoFirstNeighbor, or GoFirstChild, or after a GoNthRoot, GoNthParent, GoNthNeighbor, or GoNthChild.

Edge-Accessor Navigation

This information describes how to traverse the edges in a graph datasource using there four varieties of edge accessors:

- “All” Edge Accessors
- “In” Edge Accessors
- “Out” Edge Accessors
- Undirected Edge Accessors

“All” Edge Accessors

The edge accessor is created from a specific node accessor and can only navigate through the edges for the node that the accessor was positioned on when it was created.

These methods perform navigation of the edge accessor through all the edges in a graph datasource.

GoFirst

void NDVarGrAllEdgeAccessor::GoFirst(void);

Positions the edge accessor on the first edge in the graph datasource.

GoNext

void NDVarGrAllEdgeAccessor::GoNext(void);

Positions the edge accessor on the next edge in the graph datasource.

GoPrev**void NDVarGrAllEdgeAccessor::GoPrev(void);**

Positions the edge accessor on the previous edge in the graph datasource.

GoIndexed**void NDVarGrAllEdgeAccessor::GoIndexed(VarGrIndexVal index);**

Positions the edge accessor on the edge identified by the index “index” in the graph datasource.

GoID**void NDVarGrAllEdgeAccessor::GoID(VarPtr id);**

Positions the edge accessor on the edge identified by the ID “id” in the graph datasource.

GoBetween**void NDVarGrAllEdgeAccessor::GoBetween(VarGrNodeAccessorPtr source,
VarGrNodeAccessorPtr target);**

Positions the edge accessor on the edge between the nodes identified by the accessors “source” and “target.”

“In” Edge Accessors

These methods perform navigation on the “in” edges of a node (the edges coming from parent nodes).

GoFirst**void NDVarGrInEdgeAccessor::GoFirst(void);**

Positions the edge accessor on the first edge of the node it was created for.

GoNext**void NDVarGrInEdgeAccessor::GoNext(void);**

Positions the edge accessor on the next edge of the node it was created for.

GoPrev**void NDVarGrInEdgeAccessor::GoPrev(void);**

Positions the edge accessor on the previous edge of the node it was created for.

GoIndexed**void NDVarGrInEdgeAccessor::GoIndexed(VarGrIndexVal index);**

Positions the edge accessor on the edge identified by the index “index” in the node the edge accessor was created for.

GoID**void NDVarGrInEdgeAccessor::GoID(VarPtr id);**

Positions the edge accessor on the edge identified by the ID “id” in the node the edge accessor was created for.

“Out” Edge Accessors

These methods perform navigation on the “out” edges of a node (the edges leading to child nodes).

GoFirst

void NDVarGrOutEdgeAccessor::GoFirst(void);

Positions the edge accessor on the first edge of the node it was created for.

GoNext

void NDVarGrOutEdgeAccessor::GoNext(void);

Positions the edge accessor on the next edge of the node it was created for.

GoPrev

void NDVarGrOutEdgeAccessor::GoPrev(void);

Positions the edge accessor on the previous edge of the node it was created for.

GoIndexed

void NDVarGrOutEdgeAccessor::GoIndexed(VarGrIndexVal index);

Positions the edge accessor on the edge identified by the index “index” in the node the edge accessor was created for.

GoID

void NDVarGrOutEdgeAccessor::GoID(VarPtr id);

Positions the edge accessor on the edge identified by the ID “id” in the node the edge accessor was created for.

Undirected Edge Accessors

These methods perform navigation on the *undirected* edges of a node (the edges leading to neighbor nodes).

GoFirst

void NDVarGrUndirEdgeAccessor::GoFirst(void);

Positions the edge accessor on the first edge of the node it was created for.

GoNext

void NDVarGrUndirEdgeAccessor::GoNext(void);

Positions the edge accessor on the next edge of the node it was created for.

GoPrev

void NDVarGrUndirEdgeAccessor::GoPrev(void);

Positions the edge accessor on the previous edge of the node it was created for.

GoIndexed

void NDVarGrUndirEdgeAccessor::GoIndexed(VarGrIndexVal *index*);

Positions the edge accessor on the edge identified by the index “index” in the node the edge accessor was created for.

GoID

void NDVarGrUndirEdgeAccessor::GoID(VarPtr *id*);

Positions the edge accessor on the edge identified by the ID “id” in the node the edge accessor was created for.

Adding and Removing Nodes

Adding and removing nodes in a graph datasource using a node accessor.

AddNode

BoolEnum NDVarGr::AddNode(VarGrNodeAccessorPtr *node*);

Adds a node at the position identified by the node accessor “node.” If there is a node already at this position, the new node will be inserted before the existing node. An edit object is internally used for the operation.

AddNode

void NDVarGrEdit::AddNode(VarGrNodeAccessorPtr *node*);

Adds a node at the position identified by the node accessor “node” to the edit object. If there is a node already at this position, the new node will be inserted before the existing node.

RemoveNode

Removes a node at the position identified by the node accessor “node.” An edit object is internally used for the operation. Returns `BOOL_TRUE` if the edit was successful.

RemoveNode

void NDVarGrEdit::RemoveNode(VarGrNodeAccessorPtr *node*);

Removes a node at the position identified by the node accessor “node” in the edit object. When the edit object is committed, the node will be removed from the graph.

Adding and Removing Edges

Adding and removing edges in a graph datasource using two node accessors.

AddDirEdge**BoolEnum NDVarGr::AddDirEdge(VarGrNodeAccessorPtr source,
VarGrNodeAccessorPtr target);**

Adds a directed edge from the node identified by accessor “source” to the node identified by accessor “target.” An edit object is internally used for the operation. Returns `BOOL_TRUE` if the edit was successful.

AddDirEdge**void NDVarGrEdit::AddDirEdge(VarGrNodeAccessorPtr source,
VarGrNodeAccessorPtr target);**

Adds a directed edge from the node identified by accessor “source” to the node identified by accessor “target” in the edit object. When the edit object is committed, the edge will be added to the graph.

AddUndirEdge**BoolEnum NDVarGr::AddUndirEdge(VarGrNodeAccessorPtr node1,
VarGrNodeAccessorPtr node2);**

Adds an undirected edge between the node identified by accessor “node1” and the node identified by accessor “node2.” An edit object is internally used for the operation. Returns `BOOL_TRUE` if the edit was successful.

AddUndirEdge**void NDVarGrEdit::AddUndirEdge(VarGrNodeAccessorPtr node1,
VarGrNodeAccessorPtr node2);**

Adds an undirected edge between the node identified by accessor “node1” and the node identified by accessor “node2” in the edit object. When the edit object is committed, the edge will be added to the graph.

RemoveEdge**BoolEnum NDVarGr::RemoveEdge(VarGrEdgeAccessorPtr edge);**

Removes the edge identified by accessor “edge.” An edit object is internally used for the operation. Returns `BOOL_TRUE` if the edit was successful.

RemoveEdge**void NDVarGrEdit::RemoveEdge(VarGrEdgeAccessorPtr edge);**

Removes the edge identified by accessor “edge” in the edit object. When the edit object is committed, the edge will be removed from the graph.

RemoveEdgeBetween**BoolEnum NDVarGr::RemoveEdgeBetween(VarGrNodeAccessorPtr source,
VarGrNodeAccessorPtr target);**

Removes the edge between the node identified by accessor “source” and the node identified by accessor “target.” An edit object is internally used for the operation. Returns `BOOL_TRUE` if the edit was successful.

RemoveEdgeBetween

```
void NDVarGrEdit::RemoveEdgeBetween(VarGrNodeAccessorPtr source,  
    VarGrNodeAccessorPtr target);
```

Removes the edge between the node identified by accessor “source” and the node identified by accessor “target” in the edit object. When the edit object is committed, the edge will be removed from the graph.

Graph-Node Properties

Using the graph and node accessors to get and set properties for nodes in the graph datasource.

Accessor Validity

Since the node accessor can be positioned on nodes that do not exist, use the following method to determine if the accessor is currently positioned on an existing node (valid) or not.

IsNodeValid

```
BoolEnum NDVarGr::IsNodeValid(VarGrNodeAccessorPtr node);
```

Returns `BOOL_TRUE` if the node accessor is currently positioned on an existing node.

AreNodesEqual

```
BoolEnum NDVarGr::AreNodesEqual(VarGrNodeAccessorPtr node1,  
    VarGrNodeAccessorPtr node2);
```

Returns `BOOL_TRUE` if both accessors refer to the same node in the graph.

Node Counts

Counts of parent, child, and neighbor nodes.

GetNodeNumParents

```
VarGrIndexVal NDVarGr::GetNodeNumParents(VarGrNodeAccessorPtr node);
```

Returns the number of parent nodes for the node identified by accessor “node.”

GetNodeNumChildren

```
VarGrIndexVal NDVarGr::GetNodeNumChildren(VarGrNodeAccessorPtr node);
```

Returns the number of child nodes for the node identified by accessor “node.”

GetNodeNumNeighbors

```
VarGrIndexVal NDVarGr::GetNodeNumNeighbors(VarGrNodeAccessorPtr node);
```

Returns the number of neighbor nodes for the node identified by accessor “node.”

Node ID

Each node in the graph has an ID that can be used to quickly access any given node in the graph. IDs are not required to be set for nodes in the graph.

QueryNodeID

void NDVarGr::QueryNodeID(VarGrNodeAccessorPtr node, VarPtr value);

Returns the ID of the node referenced by node accessor “node” into the variant “value.”

GetNodeID

VarPtr NDVarGr::GetNodeID(VarGrNodeAccessorPtr node);

SetNodeID

BoolEnum NDVarGr::SetNodeID(VarGrNodeAccessorPtr node, VarPtr value);

Gets and sets the ID for the node in the variant graph datasource. The caller is responsible for disposing of the variant returned from the get method. The set method sets the ID of the node to “value” by internally creating and committing an edit, and returns `BOOL_TRUE` if it succeeded.

SetNodeID

void NDVarGrEdit::SetNodeID(VarGrNodeAccessorPtr node, VarPtr value);

Sets the ID of the node referenced by node accessor “node” to “value” in the graph edit object. When the edit object is committed, the ID will be copied into the graph.

Node Value

Each node in the graph has a value.

QueryNodeValue

void NDVarGr::QueryNodeValue(VarGrNodeAccessorPtr node, VarPtr value);

Returns the value of the node referenced by node accessor “node” into the variant “value.”

GetNodeValue

VarPtr NDVarGr::GetNodeValue(VarGrNodeAccessorPtr node);

SetNodeValue

BoolEnum NDVarGr::SetNodeValue(VarGrNodeAccessorPtr node, VarPtr value);

Gets and sets the value for the node in the variant graph datasource. The caller is responsible for disposing of the variant returned from the get method. The set method sets the value of the node to “value” by internally creating and committing an edit, and returns `BOOL_TRUE` if it succeeded.

SetNodeValue**void NDVarGrEdit::SetNodeValue(VarGrNodeAccessorPtr node, VarPtr value);**

Sets the value of the node referenced by node accessor “node” to “value” in the graph edit object. When the edit object is committed, the value will be copied into the graph.

Node XOrigin

Each node in the graph has an x origin.

QueryNodeXOrigin**void NDVarGr::QueryNodeXOrigin(VarGrNodeAccessorPtr node, VarPtr value);**

Returns the x origin of the node referenced by node accessor “node” into the variant “value.”

GetNodeXOrigin**VarPtr NDVarGr::GetNodeXOrigin(VarGrNodeAccessorPtr node);****SetNodeXOrigin****BoolEnum NDVarGr::SetNodeXOrigin(VarGrNodeAccessorPtr node, VarPtr xorigin);**

Gets and sets the x origin for the node in the variant graph datasource. The caller is responsible for disposing of the variant returned from the get method. The set method sets the x origin of the node to “xorigin” by internally creating and committing an edit, and returns `BOOL_TRUE` if it succeeded.

SetNodeXOrigin**void NDVarGrEdit::SetNodeXOrigin(VarGrNodeAccessorPtr node, VarPtr xorigin);**

Sets the x origin of the node referenced by node accessor “node” to “xorigin” in the graph edit object. When the edit object is committed, the x origin will be copied into the graph.

Node YOrigin

Each node in the graph has a y origin.

QueryNodeYOrigin**void NDVarGr::QueryNodeYOrigin(VarGrNodeAccessorPtr node, VarPtr value);**

Returns the y origin of the node referenced by node accessor “node” into the variant “value.”

GetNodeYOrigin**VarPtr NDVarGr::GetNodeYOrigin(VarGrNodeAccessorPtr node);****SetNodeYOrigin****BoolEnum NDVarGr::SetNodeYOrigin(VarGrNodeAccessorPtr node, VarPtr yorigin);**

Gets and sets the y origin for the node in the variant graph datasource. The caller is responsible for disposing of the variant returned from the get

method. The set method sets the y origin of the node to “yorigin” by internally creating and committing an edit, and returns `BOOL_TRUE` if it succeeded.

SetNodeYOrigin

void NDVarGrEdit::SetNodeYOrigin(VarGrNodeAccessorPtr node, VarPtr yorigin);

Sets the y origin of the node referenced by node accessor “node” to “yorigin” in the graph edit object. When the edit object is committed, the y origin will be copied into the graph.

Node Height

Each node in the graph has a height.

QueryNodeHeight

void NDVarGr::QueryNodeHeight(VarGrNodeAccessorPtr node, VarPtr value);

Returns the height of the node referenced by node accessor “node” into the variant “value.”

GetNodeHeight

VarPtr NDVarGr::GetNodeHeight(VarGrNodeAccessorPtr node);

SetNodeHeight

BoolEnum NDVarGr::SetNodeHeight(VarGrNodeAccessorPtr node, VarPtr height);

Gets and sets the height for the node in the variant graph datasource. The caller is responsible for disposing of the variant returned from the get method. The set method sets the height of the node to “height” by internally creating and committing an edit, and returns `BOOL_TRUE` if it succeeded.

SetNodeHeight

void NDVarGrEdit::SetNodeHeight(VarGrNodeAccessorPtr node, VarPtr height);

Sets the height of the node referenced by node accessor “node” to “height” in the graph edit object. When the edit object is committed, the height will be copied into the graph.

Node Width

Each node in the graph has a width.

QueryNodeWidth

void NDVarGr::QueryNodeWidth(VarGrNodeAccessorPtr node, VarPtr value);

Returns the width of the node referenced by node accessor “node” into the variant “value.”

GetNodeWidth

VarPtr NDVarGr::GetNodeWidth(VarGrNodeAccessorPtr node);

SetNodeWidth

BoolEnum NDVarGr::SetNodeWidth(VarGrNodeAccessorPtr node, VarPtr width);

Gets and sets the width for the node in the variant graph datasource. The caller is responsible for disposing of the variant returned from the get method. The set method sets the width of the node to “width” by internally creating and committing an edit, and returns `BOOL_TRUE` if it succeeded.

SetNodeWidth

void NDVarGrEdit::SetNodeWidth(VarGrNodeAccessorPtr node, VarPtr width);

Sets the width of the node referenced by node accessor “node” to “width” in the graph edit object. When the edit object is committed, the width will be copied into the graph.

Additional Node Properties

Each node in the graph can have an arbitrary number of additional properties accessed by a key string.

QueryNodeProperty

void NDVarGr::QueryNodeProperty(VarGrNodeAccessorPtr node, CStr key, VarPtr value);

Returns the property of the node referenced by node accessor “node” with the string “key” into the variant “value.”

GetNodeProperty

VarPtr NDVarGr::GetNodeProperty(VarGrNodeAccessorPtr node, CStr key);

SetNodeProperty

BoolEnum NDVarGr::SetNodeProperty(VarGrNodeAccessorPtr node, CStr key, VarPtr value);

Gets and sets the properties accessed by “key” for the node in the variant graph datasource. The caller is responsible for disposing of the variant returned from the get method. The set method sets the property for “key” of the node to “value” by internally creating and committing an edit, and returns `BOOL_TRUE` if it succeeded.

SetNodeProperty

void NDVarGrEdit::SetNodeProperty(VarGrNodeAccessorPtr node, CStr key, VarPtr value);

Sets the property accessed by “key” of the node referenced by node accessor “node” to “value” in the graph edit object. When the edit object is committed, the property value will be copied into the graph.

RemoveNodeProperty

BoolEnum NDVarGr::RemoveNodeProperty(VarGrNodeAccessorPtr node, CStr key);

Removes the property accessed by “key” of the node referenced by the node accessor “node” in the variant graph datasource. The remove method

internally creates and commits an edit, and returns `BOOL_TRUE` if it succeeded.

RemoveNodeProperty

void NDVarGrEdit::RemoveNodeProperty(VarGrNodeAccessorPtr node, CStr key);

Removes the property accessed by “key” of the node referenced by node accessor “node” in the graph edit object. When the edit object is committed, the property will be removed from the graph.

Graph-Edge Properties

Using the graph and edge accessors to get and set properties for edges in the graph datasource.

Accessor Validity

Since the edge accessor can be positioned on nodes that do not exist, use the following method to determine if the accessor is currently positioned on an existing (valid) edge or not.

IsEdgeValid

BoolEnum NDVarGr::IsEdgeValid(VarGrEdgeAccessorPtr edge);

Returns `BOOL_TRUE` if the edge accessor is currently positioned on an existing edge.

AreEdgesEqual

**BoolEnum NDVarGr::AreEdgesEqual(VarGrEdgeAccessorPtr edge1,
VarGrEdgeAccessorPtr edge2);**

Returns `BOOL_TRUE` if both accessors refer to the same edge in the graph.

Edge Count

The count of edges for the type of the edge accessor.

GetEdgeNumEdges

VarGrIndexVal NDVarGr::GetEdgeNumEdges(VarGrEdgeAccessorPtr edge);

Returns the number of edges for the edge accessor “edge.”

Edge ID

Each edge in the graph has a ID that can be used to quickly access any given edge in the graph. IDs are not required to be set for edges in the graph.

QueryEdgeID

void NDVarGr::QueryEdgeID(VarGrEdgeAccessorPtr edge, VarPtr value);

Returns the ID of the edge referenced by edge accessor “edge” into the variant “value.”

GetEdgeID

VarPtr NDVarGr::GetEdgeID(VarGrEdgeAccessorPtr edge);

SetEdgeID

BoolEnum NDVarGr::SetEdgeID(VarGrEdgeAccessorPtr edge, VarPtr value);

Gets and sets the ID for the edge in the variant graph datasource. The caller is responsible for disposing of the variant returned from the get method. The set method sets the ID of the edge to “value” by internally creating and committing an edit, and returns `BOOL_TRUE` if it succeeded.

SetEdgeID

void NDVarGrEdit::SetEdgeID(VarGrEdgeAccessorPtr edge, VarPtr value);

Sets the ID of the edge referenced by edge accessor “edge” to “value” in the graph edit object. When the edit object is committed, the ID will be copied into the graph.

Edge Value

Each edge in the graph has a value.

QueryEdgeValue

void NDVarGr::QueryEdgeValue(VarGrEdgeAccessorPtr edge, VarPtr value);

Returns the value of the edge referenced by edge accessor “edge” into the variant “value.”

GetEdgeValue

VarPtr NDVarGr::GetEdgeValue(VarGrEdgeAccessorPtr edge);

SetEdgeValue

BoolEnum NDVarGr::SetEdgeValue(VarGrEdgeAccessorPtr edge, VarPtr value);

Gets and sets the value for the edge in the variant graph datasource. The caller is responsible for disposing of the variant returned from the get method. The set method sets the value of the edge to “value” by internally creating and committing an edit, and returns `BOOL_TRUE` if it succeeded.

SetEdgeValue

void NDVarGrEdit::SetEdgeValue(VarGrEdgeAccessorPtr edge, VarPtr value);

Sets the value of the edge referenced by edge accessor “edge” to “value” in the graph edit object. When the edit object is committed, the value will be copied into the graph.

Directed Edge

Each edge in the graph can be directed or not.

GetEdgelsDirected

BoolEnum NDVarGr::GetEdgeIsDirected(VarGrEdgeAccessorPtr edge);

SetEdgelsDirected

BoolEnum NDVarGr::SetEdgeIsDirected(VarGrEdgeAccessorPtr edge, BoolEnum directed);

Sets the “directedness” of the edge referenced by edge accessor “edge” to “directed” in the graph edit object. When the edit object is committed, the directed value will be copied into the graph.

SetEdgelsDirected

void NDVarGrEdit::SetEdgeIsDirected(VarGrEdgeAccessorPtr edge, BoolEnum directed);

Sets the “directedness” of the edge referenced by edge accessor “edge” to “directed” in the graph edit object. When the edit object is committed, the directed value will be copied into the graph.

Additional Edge Properties

Each edge in the graph can have an arbitrary number of additional properties accessed by a key string.

QueryEdgeProperty

void NDVarGr::QueryEdgeProperty(VarGrEdgeAccessorPtr edge, CStr key, VarPtr value);

Returns the property of the edge referenced by edge accessor “edge” with the string “key” into the variant “value.”

GetEdgeProperty

VarPtr NDVarGr::GetEdgeProperty(VarGrEdgeAccessorPtr edge, CStr key);

SetEdgeProperty

BoolEnum NDVarGr::SetEdgeProperty(VarGrEdgeAccessorPtr edge, CStr key, VarPtr value);

Gets and sets the properties accessed by “key” for the edge in the variant graph datasource. The caller is responsible for disposing of the variant returned from the get method. The set method sets the property for “key” of the edge to “value” by internally creating and committing an edit, and returns `BOOL_TRUE` if it succeeded.

SetEdgeProperty

void NDVarGrEdit::SetEdgeProperty(VarGrEdgeAccessorPtr edge, CStr key, VarPtr value);

Sets the property accessed by “key” of the edge referenced by edge accessor “edge” to “value” in the graph edit object. When the edit object is committed, the property value will be copied into the graph.

RemoveEdgeProperty

BoolEnum NDVarGr::RemoveEdgeProperty(VarGrEdgeAccessorPtr edge, CStr key);

Removes the property accessed by “key” of the edge referenced by the edge accessor “edge” in the variant graph datasource. The remove method

internally creates and commits an edit, and returns `BOOL_TRUE` if it succeeded.

RemoveEdgeProperty

void NDVarGrEdit::RemoveEdgeProperty(VarGrEdgeAccessorPtr edge, CStr key);

Removes the property accessed by “key” of the edge referenced by edge accessor “edge” in the graph edit object. When the edit object is committed, the property will be removed from the graph.

Node-Relationship Discovery

Using two node accessors to determine if the nodes they reference have a parent/child or neighbor relationship.

IsChildNode

BoolEnum NDVarGr::IsChildNode(VarGrNodeAccessorPtr source, VarGrNodeAccessorPtr target);

Returns `BOOL_TRUE` if “target” is a child of “source.”

IsParentNode

BoolEnum NDVarGr::IsParentNode(VarGrNodeAccessorPtr node, VarGrNodeAccessorPtr target);

Returns `BOOL_TRUE` if “target” is a parent of “source.”

IsNeighborNode

BoolEnum NDVarGr::IsNeighborNode(VarGrNodeAccessorPtr node, VarGrNodeAccessorPtr target);

Returns `BOOL_TRUE` if “target” is a neighbor of “source.”

Getting and Setting the Cursors

Methods of the graph object to get and set the node and edge cursors.

A graph of variant datasources keeps track of two cursors that can be on any node and edge. No special action is attached to the action of moving the cursor around in the datasource itself.

GetNodeCursor

VarGrNodeAccessorPtr NDVarGr::GetNodeCursor(void);

SetNodeCursor

BoolEnum NDVarGr::SetNodeCursor(VarGrNodeAccessorPtr node);

These routines get and set the current position of the node cursor. The caller is responsible for disposing of the accessor returned from the get method. The Set routine sets the graph node cursor to “node” by internally creating and committing an edit object. It returns `BOOL_TRUE` if the internal edit could take place, `BOOL_FALSE` if not.

SetNodeCursor

void NDVarGrEdit::SetNodeCursor(VarGrNodeAccessorPtr node);

Sets the graph node cursor in the edit object to “node.” When the edit object is committed, the node cursor of the graph will reflect the same value.

GetEdgeCursor

VarGrEdgeAccessorPtr NDVarGr::GetEdgeCursor(void);

SetEdgeCursor

BoolEnum NDVarGr::SetEdgeCursor(VarGrEdgeAccessorPtr edge);

These routines get and set the current object of the edge cursor. The caller is responsible for disposing of the accessor returned from the get method. The Set routine sets the graph edge cursor to “edge” by internally creating and committing an edit object. It returns `BOOL_TRUE` if the internal edit could take place, `BOOL_FALSE` if not.

SetEdgeCursor

void NDVarGrEdit::SetEdgeCursor(VarGrEdgeAccessorPtr edge);

Sets the graph edge cursor in the edit object to “edge.” When the edit object is committed, the edge cursor of the graph will reflect the same value.

Convenience Methods

Methods of the graph and graph edit objects to start an edit on the graph, start an edit on nodes or edges in the graph, and query the cyclic result of adding a edge.

StartNodeEdit

VarGrEditPtr NDVarGr::StartNodeEdit(VarGrNodeAccessorPtr node);

Opens an edit on the node identified by accessor “node,” and all operations are done through the edit object returned by this call.

StartEdgeEdit

VarGrEditPtr NDVarGr::StartEdgeEdit(VarGrEdgeAccessorPtr edge);

Opens an edit on the edge identified by accessor “edge,” and all operations are done through the edit object returned by this call.

QueryCyclicResult

**BoolEnum NDVarGr::QueryCyclicResult(VarGrNodeAccessorPtr source,
VarGrNodeAccessorPtr target);**

Returns `BOOL_TRUE` if a directed edge added from node “source” to node “target” would result in a cyclic graph. A cyclic graph contains at least one path that starts and ends at the same node.

Advanced Objects and Methods

Methods and objects that are not necessary for most operations on the graph object. Useful when subclassing the graph datasource.

Node and Edge Objects

Given an accessor for a node or edge, you can retrieve a node or edge object from the graph.

GetNode

VarGrNodePtr NDVarGr::GetNode(VarGrNodeAccessorPtr node);

Get a node from the graph. Nodes are retrieved from the graph by using a node accessor.

GetEdge

VarGrEdgePtr NDVarGr::GetEdge(VarGrEdgeAccessorPtr edge);

Get an edge from the graph. Edges are retrieved from the graph by using an edge accessor.

Node-Object Properties

GetNumChildren

VarGrIndexVal NDVarGrNode::GetNumChildren(void);

Returns the number of child nodes for the node.

GetChildNode

VarGrNodePtr NDVarGrNode::GetChildNode(VarGrIndexVal index);

Returns the child node object corresponding to the index “index.” The returned object should be destructed and disposed of by the caller.

GetNumParents

VarGrIndexVal NDVarGrNode::GetNumParents(void);

Returns the number of parent nodes for the node.

GetParentNode

VarGrNodePtr NDVarGrNode::GetParentNode(VarGrIndexVal index);

Returns the parent-node object corresponding to the index “index.” The returned object should be destructed and disposed of by the caller.

GetNumNeighbors

VarGrIndexVal NDVarGrNode::GetNumNeighbors(void);

Returns the number of neighbor nodes for the node.

GetNeighborNode**VarGrNodePtr NDVarGrNode::GetNeighborNode(VarGrIndexVal index);**

Returns the neighbor-node object corresponding to the index “index.” The returned object should be destructed and disposed of by the caller.

GetOutEdge**VarGrEdgePtr NDVarGrNode::GetOutEdge(VarGrIndexVal index);**

Returns the outgoing edge object corresponding to the index “index.” The returned object should be destructed and disposed of by the caller.

GetInEdge**VarGrEdgePtr NDVarGrNode::GetInEdge(VarGrIndexVal index);**

Returns the incoming edge object corresponding to the index “index.” The returned object should be destructed and disposed of by the caller.

GetUndirEdge**VarGrEdgePtr NDVarGrNode::GetUndirEdge(VarGrIndexVal index);**

Returns the undirected edge object corresponding to the index “index.” The returned object should be destructed and disposed of by the caller.

Node ID

Each node in the graph has an ID that can be used to quickly access any given node in the graph. IDs are not required to be set for nodes in the graph.

GetID**VarPtr NDVarGrNode::GetID(void);****SetID****BoolEnum NDVarGrNode::SetID(VarPtr value);**

Gets and sets the ID for the node. The caller is responsible for disposing of the variant returned from the get method. The set method sets the ID of the node to “value” by internally creating and committing an edit, and returns `BOOL_TRUE` if it succeeded.

SetID**void NDVarGrNodeEdit::SetID(VarPtr value);**

Sets the ID of the node to “value” in the graph-node edit object. When the edit object is committed, the ID will be copied into the node.

Node Value

Each node in the graph has a value.

QueryValue**void NDVarGrNode::QueryValue(VarPtr value);**

Returns the value of the node into the variant “value.”

GetValue

VarPtr NDVarGrNode::GetValue(void);

SetValue

BoolEnum NDVarGrNode::SetValue(VarPtr value);

Gets and sets the value for the node. The caller is responsible for disposing of the variant returned from the get method. The set method sets the value of the node to “value” by internally creating and committing an edit, and returns `BOOL_TRUE` if it succeeded.

SetValue

void NDVarGrNodeEdit::SetValue(VarPtr value);

Sets the value of the node to “value” in the graph-node edit object. When the edit object is committed, the value will be copied into the node object.

Node XOrigin

Each node in the graph has an *x* origin.

GetXOrigin

VarPtr NDVarGrNode::GetXOrigin(void);

SetXOrigin

BoolEnum NDVarGrNode::SetXOrigin(VarPtr xorigin);

Gets and sets the *x* origin for the node. The caller is responsible for disposing of the variant returned from the get method. The set method sets the *x* origin of the node to “*xorigin*” by internally creating and committing an edit, and returns `BOOL_TRUE` if it succeeded.

SetXOrigin

void NDVarGrNodeEdit::SetXOrigin(VarPtr xorigin);

Sets the *x* origin of the node in the graph-node edit object. When the edit object is committed, the *x* origin will be copied into the node.

Node YOrigin

Each node in the graph has a *y* origin.

GetYOrigin

VarPtr NDVarGrNode::GetYOrigin(void);

SetYOrigin

BoolEnum NDVarGrNode::SetYOrigin(VarPtr yorigin);

Gets and sets the *y* origin for the node. The caller is responsible for disposing of the variant returned from the get method. The set method sets the *y* origin of the node to “*yorigin*” by internally creating and committing an edit, and returns `BOOL_TRUE` if it succeeded.

SetYOrigin**void NDVarGrNodeEdit::SetYOrigin(VarPtr yorigin);**

Sets the *y* origin of the node in the graph-node edit object. When the edit object is committed, the *y* origin will be copied into the node.

Node Height

Each node in the graph has a height.

GetHeight**VarPtr NDVarGrNode::GetHeight(void);****SetHeight****BoolEnum NDVarGrNode::SetHeight(VarPtr height);**

Gets and sets the height for the node. The caller is responsible for disposing of the variant returned from the get method. The set method sets the height of the node to “height” by internally creating and committing an edit, and returns `BOOL_TRUE` if it succeeded.

SetHeight**void NDVarGrNodeEdit::SetHeight(VarPtr height);**

Sets the height of the node to “height” in the node edit object. When the edit object is committed, the height will be copied into the node.

Node Width

Each node in the graph has a width.

GetWidth**VarPtr NDVarGrNode::GetWidth(void);****SetWidth****BoolEnum NDVarGrNode::SetWidth(VarPtr width);**

Gets and sets the width for the node. The caller is responsible for disposing of the variant returned from the get method. The set method sets the width of the node to “width” by internally creating and committing an edit, and returns `BOOL_TRUE` if it succeeded.

SetWidth**void NDVarGrNodeEdit::SetWidth(VarPtr width);**

Sets the width of the node to “width” in the node edit object. When the edit object is committed, the width will be copied into the node.

Additional Node Properties

Each node in the graph can have an arbitrary number of additional properties accessed by a key string.

GetProperty**VarPtr NDVarGrNode::GetProperty(CStr key);****SetProperty****BoolEnum NDVarGrNode::SetProperty(CStr key, VarPtr value);**

Gets and sets the properties accessed by “key” for the node. The caller is responsible for disposing of the variant returned from the get method. The set method sets the property for “key” of the node to “value” by internally creating and committing an edit, and returns `BOOL_TRUE` if it succeeded.

SetProperty**void NDVarGrNodeEdit::SetProperty(CStr key, VarPtr value);**

Sets the property accessed by “key” of the node to “value” in the node edit object. When the edit object is committed, the property value will be copied into the node.

RemoveProperty**BoolEnum NDVarGrNode::RemoveProperty(CStr key);**

Removes the property accessed by “key” of the node. The remove method internally creates and commits an edit, and returns `BOOL_TRUE` if it succeeded.

RemoveProperty**void NDVarGrNodeEdit::RemoveProperty(CStr key);**

Removes the property accessed by “key” of the node in the edit object. When the edit object is committed, the property will be removed from the node object.

Edge-Object Properties

GetFromNode**VarGrNodePtr NDVarGrEdge::GetFromNode(void);**

Returns the “from” node where the edge originates. If the edge is undirected, this is the node that was given first when the edge was added.

GetToNode**VarGrNodePtr NDVarGrEdge::GetToNode(void);**

Returns the “to” node where the edge terminates. If the edge is undirected, this is the node that was given last when the edge was added.

Edge ID

Each edge in the graph has an ID that can be used to quickly access any given edge in the graph. IDs are not required to be set for edges in the graph.

GetID

VarPtr NDVarGrEdge::GetID(void);

SetID

BoolEnum NDVarGrEdge::SetID(VarPtr value);

Gets and sets the ID for the edge in the variant graph datasource. The caller is responsible for disposing of the variant returned from the get method. The set method sets the ID of the edge to “value” by internally creating and committing an edit, and returns `BOOL_TRUE` if it succeeded.

SetID

void NDVarGrEdgeEdit::SetID(VarPtr value);

Sets the ID of the edge to “value” in the edge edit object. When the edit object is committed, the ID will be copied into the edge.

Edge Value

Each edge in the graph has a value.

QueryValue

void NDVarGrEdge::QueryValue(VarPtr value);

Returns the value of the edge into the variant “value.”

GetValue

VarPtr NDVarGrEdge::GetValue(void);

SetValue

BoolEnum NDVarGrEdge::SetValue(VarPtr value);

Gets and sets the value for the edge. The caller is responsible for disposing of the variant returned from the get method. The set method sets the value of the edge to “value” by internally creating and committing an edit, and returns `BOOL_TRUE` if it succeeded.

SetValue

void NDVarGrEdgeEdit::SetValue(VarPtr value);

Sets the value of the edge to “value” in the edge edit object. When the edit object is committed, the value will be copied into the edge.

Directed Edge

Each edge in the graph can be directed or not.

GetIsDirected

BoolEnum NDVarGrEdge::GetIsDirected(void);

SetIsDirected

BoolEnum NDVarGrEdge::SetIsDirected(BoolEnum *directed*);

Sets the “directedness” of the edge to “directed” in the edge edit object. When the edit object is committed, the directed value will be copied into the edge object.

SetIsDirected

void NDVarGrEdgeEdit::SetIsDirected(BoolEnum *directed*);

Sets the “directedness” of the edge to “directed” in the edge edit object. When the edit object is committed, the directed value will be copied into the edge object.

Additional Edge Properties

Each edge in the graph can have an arbitrary number of additional properties accessed by a key string.

GetProperty

VarPtr NDVarGrEdge::GetProperty(CStr *key*);

SetProperty

BoolEnum NDVarGrEdge::SetProperty(CStr *key*, VarPtr *value*);

Gets and sets the properties accessed by “key” for the edge. The caller is responsible for disposing of the variant returned from the get method. The set method sets the property for “key” of the edge to “value” by internally creating and committing an edit, and returns `BOOL_TRUE` if it succeeded.

SetProperty

void NDVarGrEdgeEdit::SetProperty(CStr *key*, VarPtr *value*);

Sets the property accessed by “key” to “value” in the edge edit object. When the edit object is committed, the property value will be copied into the edge object.

RemoveProperty

BoolEnum NDVarGrEdge::RemoveProperty(CStr *key*);

Removes the property accessed by “key” of the edge. The remove method internally creates and commits an edit, and returns `BOOL_TRUE` if it succeeded.

RemoveProperty

void NDVarGrEdgeEdit::RemoveProperty(CStr *key*);

Removes the property accessed by “key” of the edge in the edit object. When the edit object is committed, the property will be removed from the edge object.

Edit Objects

The low-level code for updating a variant graph datasource needs to start an edit on the graph. Edits can be started either globally on the graph (when adding and removing nodes, for example), or locally on a given node or edge.

StartEdit

VarGrNodeEditPtr NDVarGrNode::StartEdit(void);

Open an edit for modifying the node. NULL will be returned if no edit could be opened; otherwise, a constructed edit object is returned.

StartEdit

VarGrEdgeEditPtr NDVarGrEdge::StartEdit(void);

Open an edit for modifying the edge. NULL will be returned if no edit could be opened; otherwise, a constructed edit object is returned.

Modification Descriptions

GetMods

VarGrModsCPtr NDVarGr::GetMods(void);

Get a description of the last modifications committed on the graph datasource.

Class Operations

Create

Creates and constructs a variant graph datasource.

47 *VarLs Class*

This class specifies the list of variants data source.

Design Overview

A list of variant data sources is a data source that keeps track of a list of values that could be considered to be stored in variants. Each one of these variants can hold a value of any of the types supported by the Variant class (see `varpub.h`).

A list of variant data sources can be manipulated at several levels. At the highest level, the list of variant data sources lets a user read values corresponding to list entries, open editions on them, modify them either directly or through editions.

The variant data source is a subclass of the pure virtual data source, and implements specific methods to get and set the associated value.

Class

Class

static RClasPtr NDVarLs::Class(void);

Returns a pointer to the variant list data source resource class.

Reading and Writing in the List

List Title

The list of variant data sources can have a title that views can use to identify it.

GetTitle

CStr NDVarLs::GetTitle(void);

Returns the title for the list, if any.

SetTitle

BoolEnum NDVarLs::SetTitle(CStr title);

Sets the title of the list to 'title' by internally creating and committing an edition. Returns `BOOL_TRUE` if the internal edit succeeded, `BOOL_FALSE` if not.

Row Titles

Each row in the list can have a title that can be used to identify them.

GetRowTitle

CStr NDVarLs::GetRowTitle(VarLsIndexVal *index*);

Returns the title for the row `index' in the list, if any.

SetRowTitle

BoolEnum NDVarLs::SetRowTitle(VarLsIndexVal *index*, CStr *title*);

Sets the title of the row identified by `index' to `title' by internally creating and committing an edition. Returns `BOOL_TRUE` if the edit succeeded, `BOOL_FALSE` if not.

GetMaxRowTitleStrLen

StrIVal NDVarLs::GetMaxRowTitleStrLen(void);

Returns the length of the longest string for a row title in the list, if the source can provide it. If it cannot provide it, returns 0.

Row Values

Each row “holds” a value. The value can be read and updated.

QueryRowValue

void NDVarLs::QueryRowValue(VarLsIndexVal *index*, VarPtr *value*);

Returns the value of the row `index' in the list.

GetMaxStrLen

StrIVal NDVarLs::GetMaxStrLen(void);

Returns the length of the longest string for any row in the list, if the source can provide it. If it cannot provide it, it will return 0.

GetRowValue

VarPtr NDVarLs::GetRowValue(VarLsIndexVal *index*);

Returns the value of the row `index' in the list. The caller is responsible for freeing the returned variant. `NDVarLs::QueryRowValue` is a preferable call.

SetRowValue

BoolEnum NDVarLs::SetRowValue(VarLsIndexVal *index*, VarPtr *value*);

Sets the row value by internally creating and committing an edition. It returns `BOOL_TRUE` if the internal edit succeeded. Returns `BOOL_FALSE` otherwise.

Modifying the List

GetNumRows

VarLsIndexVal NDVarLs::GetNumRows(void);

Returns the number of rows in the list data source.

SetNumRows

BoolEnum NDVarLs::SetNumRows(VarLsIndexVal numRows);

Sets the number of rows in the list internally creating and committing a edition object. If 'numRows' is greater than the current number of rows, rows are added without changing the contents of the existing rows. If it is smaller, then rows are removed. Returns `BOOL_TRUE` if the internal edit succeeded, `BOOL_FALSE` in any other case.

AddRow

BoolEnum NDVarLs::AddRow(VarLsIndexVal index);

Add a row at index 'index' in the list by internally creating and committing an edition object. Returns `BOOL_TRUE` if the internal edit succeeded, `BOOL_FALSE` in any other case.

RemoveRow

BoolEnum NDVarLs::RemoveRow(VarLsIndexVal index);

Removes a row from the list by internally creating and committing an edition. Returns `BOOL_TRUE` if the internal edit succeeded, `BOOL_FALSE` in any other case.

Reading and Setting the Cursor Row

A list of variant data sources keeps track of a cursor row. The cursor row can be read or set at any point. Internally, the cursor row does not correspond to any specific row, and no special action is attached to the action of moving the cursor.

GetCursorRow

VarLsIndexVal NDVarLs::GetCursorRow(void);

Returns the current position of the cursor of the list.

SetCursorRow

BoolEnum NDVarLs::SetCursorRow(VarLsIndexVal row);

Sets the current list cursor to 'row' by internally creating and committing an edition object. Returns `BOOL_TRUE` if the internal edit succeeded, `BOOL_FALSE` in any other case.

Edition Objects

The low level code for updating a variant list data source needs to start an edition on the list. Editions can be started either globally on the list (when adding and removing rows for example), or locally on a given row.

StartRowEdit

VarLsEditPtr NDVarLs::StartRowEdit(VarLsIndexVal index);

Open a edit (for modifying the `index` row). NULL will be returned if no edit could be opened. Otherwise a constructed edition is returned.

VarLsEdit AddRow

void NDVarLsEdit::AddRow(VarLsIndexVal index);

Adds a row at index `index` in the edition object. When the edition object is committed, the corresponding list will reflect the change.

VarLsEdit RemoveRow

void NDVarLsEdit::RemoveRow(VarLsIndexVal index);

Removes the row at index `index` in the edition object. When the edition object is committed, the corresponding list will reflect the change.

VarLsEdit SetCursorRow

void NDVarLsEdit::SetCursorRow(VarLsIndexVal index);

Sets the current edition object cursor to `row`. When the edition object is committed, the change will be reflected in the associated list.

VarLsEdit SetNumRows

void NDVarLsEdit::SetNumRows(VarLsIndexVal numRows);

Sets the number of rows in an edition object. If `numRows` is greater than the current number of rows, rows are added without changing the contents of the existing rows. If it is smaller, then rows are removed. When the edition object is committed, the changes will be propagated to the list.

VarLsEdit SetRowValue

void NDVarLsEdit::SetRowValue(VarLsIndexVal index, VarPtr value);

Sets the value corresponding to the row `index` of the edition object to `value`. When the edition object is committed, the value will be copied onto the row `index` of the list data source.

SetRowTitle

void NDVarLsEdit::SetRowTitle(VarLsIndexVal index, CStr title);

Sets the title of the row `index` of the edition object is committed, the title will be copied onto the row `index` in the list data source.

VarLSEdit SetTitle

void NDVarLsEdit::SetTitle(CStr str);

Sets the title of the list edition object to `title`. When the edition object is committed, the title will be copied into the list.

Modification Descriptions

GetMods

VarLsModsCPtr NDVarLs::GetMods(void);

Get a description of the last modifications made on the list through an edition object.

Notifications

DefNfy

void NDVarLs::DefNfy(VarLsNfyEnum code);

Default notification procedure for the VarLs class.

48 *VarTb Class*

This class specifies the table of variants data source.

Technical Overview

This class implements the table of variant data sources. A table of variant data sources is a data source that keeps track of a 2 dimensional table of values that could be considered to be stored in variants. Each one of these variants can hold a value of any of the types supported by the Variant class (see varpub.h).

A table of variant data sources can be manipulated at several levels. At the highest level, the table of variant data sources lets a user read values corresponding to table entries, open editions on them, modify them either directly or through editions.

The variant data source is a subclass of the pure virtual data source, and implements specific methods to get and set the associated value.

Class

The DataSource data structure is private. It is a subclass of Res.

Class

static RClasPtr NDVarTb::Class(void);

Returns a pointer to the resource class for VarTb.

Table Interaction

Read Support

The table of variant data sources can have a title that views can use to identify it.

GetNumRows

VarTbIndexVal NDVarTb::GetNumRows(void);

Returns the number of rows in the table.

GetNumColumns

VarTbIndexVal NDVarTb::GetNumColumns(void);

Returns the number of columns in the table.

QueryCellValue

```
void NDVarTb::QueryCellValue(VarTbIndexVal row,  
    VarTbIndexVal col, VarPtr value);
```

Returns the value of the cell at row `row` and column `col` the list.

GetCellValue

```
VarPtr NDVarTb::GetCellValue(VarTbIndexVal row,  
    VarTbIndexVal col);
```

Returns the value of the cell at row `row` and column `col` in the list. The caller is responsible for disposing the returned variant.

GetMaxColStrLen

```
StrIVal NDVarTb::GetMaxColStrLen(VarTbIndexVal col);
```

Returns the length of the longest string for a column in the table (including the title), if the source can provide it. If it cannot provide it, should return 0.

Row Title

Each row in the table can have a title that can be used to identify them.

GetRow Title

```
CStr NDVarTb::GetRowTitle(VarTbIndexVal index)
```

Returns the title for the row “index” in the table, if any. Each row in the table can have a title that can be used to identify them.

SetRowTitle

```
BoolEnum NDVarTb::SetRowTitle(VarTbIndexVal row, CStr title)
```

Sets the title of the row identified by “row” in the table to “title” by internally creating and committing a edition. Returns `BOOL_TRUE` if the edition succeeded, `BOOL_FALSE` if not.

GetColumnTitle

```
CStr NDVarTb::GetColumnTitle(VarTbIndexVal index);
```

Returns the title for the column “index” in the table, if any.

TableTitle

The table of variant data sources can have a title that views can use to identify it.

GetTitle

```
CStr NDVarTb::GetTitle(void);
```

Returns the title for the table, if any.

Reading and Setting the Cursor Row and Column

A table of variant data sources keeps track of a cursor row and a cursor column. Both can be read or set at any point. Internally, the cursor row and the cursor column do not correspond to any specific cell, and no special action is attached to the action of moving the cursor around.

GetCursorRow

VarTbIndexVal NDVarTb::GetCursorRow(void);

Returns the current position of the row cursor

GetCursorColumn

VarTbIndexVal NDVarTb::GetCursorColumn(void);

Returns the current position of the column cursor

Edition Support

StartRowEdit

VarTbEditPtr NDVarTb::StartRowEdit(VarTbIndexVal index);

Open a edition (for modifying the “index” row). NULL will be returned if no edition could be opened. Otherwise a constructed edition is returned.

StartCellEdit

VarTbEditPtr NDVarTb::StartCellEdit(VarTbIndexVal row, VarTbIndexVal col);

Open a edition (for modifying the cell at (“row, col”). NULL will be returned if no edition could be opened. Otherwise a constructed edition is returned.

SetNumRowColumns

BoolEnum NDVarTb::SetNumRowColumns(VarTbIndexVal numRows, VarTbIndexVal numCols);

Set up the number of rows and columns of the table by internally creating and committing a edition. This call may wipe out contents. It returns `BOOL_TRUE` if the transaction succeeded, `BOOL_FALSE` if not.

AddRow

BoolEnum NDVarTb::AddRow(VarTbIndexVal index);

Adds a row at index “index” to the table, by internally creating and committing a edition. Returns `BOOL_TRUE` if the internal edition could take place.

RemoveRow

BoolEnum NDVarTb::RemoveRow(VarTbIndexVal index);

Remove a row by internally creating and committing a edition. Returns `BOOL_TRUE` if the internal edition could take place.

AddColumn**BoolEnum NDVarTb::AddColumn(VarTbIndexVal *index*);**

Add a column by internally creating and committing a edition. Returns `BOOL_TRUE` if the internal edition could take place.

RemoveColumn**BoolEnum NDVarTb::RemoveColumn(VarTbIndexVal *index*);**

Removes the column at index “*index*” in the edition object. When edition object is committed, the column will be removed from the corresponding variant table.

SetColValue**BoolEnum NDVarTb::SetColValue(VarTbIndexVal *row*, VarTbIndexVal *col*, VarPtr *value*);**

Sets the value of the cell identified by “*row*, *col*” to “*value*” by internally creating and committing a edition. Returns `BOOL_TRUE` if the edition succeeded.

SetRowTitle**BoolEnum NDVarTb::SetRowTitle(VarTbIndexVal *row*, CStr *title*);**

Sets the title of the row identified by “*row*” to “*title*” by internally creating and committing a edition. Returns `BOOL_TRUE` if the edition succeeded.

SetColumnName**BoolEnum NDVarTb::SetColumnName(VarTbIndexVal *col*, CStr *title*);**

Sets the title of the column identified by “*col*” to “*title*” by internally creating and committing a edition. Returns `BOOL_TRUE` if the edition succeeded, `BOOL_FALSE` if not.

SetTitle**BoolEnum NDVarTb::SetTitle(CStr *str*);**

Sets the title of the table by internally creating and committing a edition. Returns `BOOL_TRUE` if the internal edition could take place.

SetCursorRow**BoolEnum NDVarTb::SetCursorRow(VarTbIndexVal *index*);**

Sets the current table row cursor by internally creating and committing a edition. Returns `BOOL_TRUE` if the internal edition could take place.

SetCursorColumn**BoolEnum NDVarTb::SetCursorColumn(VarTbIndexVal *index*);**

Sets the current table column cursor by internally creating and committing a edition. Returns `BOOL_TRUE` if the internal edition could take place.

Edition Objects

The low level code for updating a variant table data source needs to start an edition on the table. Editions can be started either globally on the list (when adding and removing rows for example), locally on a given row, a given column or a given cell.

StartEdit

VarTbEditPtr NDVarTb::StartEdit(void);

Opens an edition on the whole table data source. The operations are done through the edition object returned by this call.

SetNumRowColumns

**void NDVarTbEdit::SetNumRowColumns(
VarTbIndexVal numRows, VarTbIndexVal numCols);**

Sets up the number of rows and columns of an edition object. When the edition object is committed, the changes will be applied to the corresponding table. This call may wipe out contents.

AddRow

void NDVarTbEdit::AddRow(VarTbIndexVal index);

Adds a row at index “index” to the edition object. When the edition is committed, the row will be added at the corresponding index in the variant table.

RemoveRow

void NDVarTbEdit::RemoveRow(VarTbIndexVal index);

Removes the row at index “index” in the table. When the edition object is committed, the corresponding row will be removed in the corresponding variant table.

AddColumn

void NDVarTbEdit::AddColumn(VarTbIndexVal index);

Add a column at index “index” to the edition object. When the edition object is committed, the column will be added to the corresponding variant table.

RemoveColumn

void NDVarTbEdit::RemoveColumn(VarTbIndexVal index);

Remove a column through a edition.

SetCellValue

**void NDVarTbEdit::SetCellValue(
VarTbIndexVal row, VarTbIndexVal col, VarPtr value);**

Sets the value corresponding to the cell identified by “row” and “col” of the edition object to “value.” When the edition object is committed, the value will be copied onto the corresponding cell in the table through an edition.

SetRowTitle**void NDVarTbEdit::SetRowTitle(VarTbIndexVal *index*, CStr *title*);**

Sets the title corresponding to the row identified by “index” through an edition.

SetColumnName**void NDVarTbEdit::SetColumnName(VarTbIndexVal *index*, CStr *title*);**

Sets the title corresponding to the column identified by “index” through a edition. When the edition object is committed, the title will be copied onto the column “index” in the table data source.

SetTitle**void NDVarTbEdit::SetTitle(CStr *str*);**

Sets the title of the table through a edition object to “title.” When the edition object is committed, the title will be copied into the table.

SetCursorRow**void NDVarTbEdit::SetCursorRow(VarTbIndexVal *index*);**

Sets the table cursor row in the edition object to “index.” When the edition object is committed, the cursor row of the table will reflect the same value.

SetCursorColumn**void NDVarTbEdit::SetCursorColumn(VarTbIndexVal *index*);**

Sets the table cursor column of the edition object to “index.” When the edition object is committed, the cursor column of the table will reflect the same value.

Modifications Queries

GetMods**VarTbModsCPtr NDVarTb::GetMods(void);**

Get a description of the last modifications committed on the table data source.

Row Interaction

The largest part of the interface for this object is in fact in the NDDs interface. In particular, opening a edition for this object is done through NDDs::StartEdit.

Column Interaction

The largest part of the interface for this object is in fact in the NDDs interface. In particular, opening a edition for this object is done through NDDs::StartEdit.

Cell Interaction

The largest part of the interface for this object is in fact in the NDDs interface. In particular, opening a edition for this object is done through NDDs::StartEdit..

Virtual Interface Implementation

Variant List Implementation

```
extern "C" RClasPtr VarTbGetClass(void);  
extern "C" void VarTbConstruct(ResPtr res, RClasCPtr rclas, RClasCreateCPtr rCreate);  
extern "C" void VarTbDestruct(ResPtr res);
```

Variant List Row Implementation

```
extern "C" RClasPtr VarTbRowGetClass(void);  
extern "C" void VarTbRowConstruct(ResPtr res, RClasCPtr rclas,  
    RClasCreateCPtr rCreate);  
extern "C" void VarTbRowDestruct(ResPtr res);
```

Variant List Row Implementation

```
extern "C" RClasPtr VarTbColGetClass(void);  
extern "C" void VarTbColConstruct(ResPtr res, RClasCPtr rclas, RClasCreateCPtr rCreate);  
extern "C" void VarTbColDestruct(ResPtr res);
```

Variant List Cell Implementation

```
extern "C" RClasPtr VarTbCellGetClass(void);  
extern "C" void VarTbCellConstruct(ResPtr res, RClasCPtr rclas, RClasCreateCPtr rCreate);  
extern "C" void VarTbCellDestruct(ResPtr res);
```


Design Overview

This module implements the variant tree datasources. A *variant tree data source* is a datasource that keeps track of a hierarchical collection of nodes that have properties stored in variants. Each one of these variants can hold a value of any of the types supported by the Variant class (see VARPUB.H).

The variant tree data sources contain a collection of nodes. The nodes are never accessed directly as objects, instead node *accessors* are created to traverse the tree-datasource object. The accessors are then used in conjunction with the tree-datasource object to perform operations such as add nodes, remove nodes, and set values on the nodes.

The datasource data structure is private.

Class

static RClasPtr NDVarTr::Class(void);

Returns a pointer to the resource class for VarTr.

Class

static RClasPtr NDVarTrNodeAccessor::Class(void);

Returns a pointer to the resource class for VarTrNodeAccessor.

Tree-Datasource Properties

Methods of the tree-datasource object associated with the title and number of nodes in the tree datasource.

Tree Title

The variant tree datasources can have a title that views can use to identify it.

GetTitle

CStr NDVarTr::GetTitle(void);

BoolEnum NDVarTr::SetTitle(CStr str);

Gets the title for the tree datasource, if any.

SetTitle

void NDVarTrEdit::SetTitle(CStr str);

Sets the title of the tree edit object to "title." When the edit object is committed, the title will be copied into the tree.

Node-Accessor Navigation

How to traverse a tree datasource with a node accessor.

The node accessor can be positioned with the absolute methods:

- `GoFirstRoot`
- `GoNthRoot`

From any given position, the node accessor can also be positioned with these absolute methods:

- `GoFirstChild`
- `GoNthChild`
- `GoFirstSibling`
- `GoNthSibling`
- `GoParent`

Or with these relative methods:

- `GoNext`
- `GoPrev`

`GoFirstRoot`

`void NDVarTrNodeAccessor::GoFirstRoot(void);`

Positions the node accessor on the first root node.

`GoFirstChild`

`void NDVarTrNodeAccessor::GoFirstChild(void);`

Positions the node accessor on the first child node of the node where it is currently positioned.

`GoFirstSibling`

`void NDVarTrNodeAccessor::GoFirstSibling(void);`

Positions the node accessor on the first sibling node of the node where it is currently positioned.

`GoNext`

`void NDVarTrNodeAccessor::GoNext(void);`

Positions the node accessor on the next node.

`GoPrev`

`void NDVarTrNodeAccessor::GoPrev(void);`

Positions the node accessor on the previous node.

`GoParent`

`void NDVarTrNodeAccessor::GoParent(void);`

Positions the node accessor on the parent node of the node where it is currently positioned.

Convenient Navigation

GoNthRoot

void NDVarTrNodeAccessor::GoNthRoot(VarTrIndexVal *index*);

Positions the node accessor on the *n*th root node.

GoNthChild

void NDVarTrNodeAccessor::GoNthChild(VarTrIndexVal *index*);

Positions the node accessor on the *n*th child node of the node where it is currently positioned.

GoNthSibling

void NDVarTrNodeAccessor::GoNthSibling(VarTrIndexVal *index*);

Positions the node accessor on the *n*th sibling node of the node where it is currently positioned.

Adding and Removing Nodes

Adding and removing nodes in a tree datasource using a node accessor.

AddNode

BoolEnum NDVarTr::AddNode(VarTrNodeAccessorPtr *accessor*);

Adds a node at the position identified by the node accessor “*accessor*.” If there is a node already at this position, the new node will be inserted before the existing node. An edit object is internally used for the operation.

AddNode

void NDVarTrEdit::AddNode(VarTrNodeAccessorPtr *accessor*);

Adds a node at the position identified by the node accessor “*accessor*” to the edit object. If there is a node already at this position, the new node will be inserted before the existing node.

RemoveNode

BoolEnum NDVarTr::RemoveNode(VarTrNodeAccessorPtr *accessor*);

Removes a node at the position identified by the node accessor “*accessor*.” An edit object is internally used for the operation.

RemoveNode

void NDVarTrEdit::RemoveNode(VarTrNodeAccessorPtr *accessor*);

Removes a node at the position identified by the node accessor “*accessor*.”

RemoveTree

BoolEnum NDVarTr::RemoveTree(VarTrNodeAccessorPtr *accessor*);

Removes a tree starting from the position identified by the node accessor “*accessor*.” An edit object is used internally for the operation.

RemoveTree

void NDVarTrEdit::RemoveTree(VarTrNodeAccessorPtr accessor);

Removes a tree starting from the position identified by the node accessor “accessor.”

Class Operations

Tree-Node Properties

Using the node accessors to get and set properties for nodes in the tree datasource.

Tree-Node Discovery and Navigation

GetNumRoots

VarTrIndexVal NDVarTr::GetNumRoots(void);

Returns the number of root nodes in the tree datasource.

GetNumChildren

VarTrIndexVal NDVarTr::GetNumChildren(VarTrNodeAccessorCPtr accessor);

Returns the number of child nodes relative to the current “accessor” location.

GetNumSiblings

VarTrIndexVal NDVarTr::GetNumSiblings(VarTrNodeAccessorCPtr accessor);

Returns the number of sibling nodes relative to the current “accessor” location.

IsValidNode

BoolEnum NDVarTr::IsValidNode(VarTrNodeAccessorCPtr accessor);

Returns `BOOL_TRUE` if a node exists at the current “accessor” location.

AreNodesEqual

**BoolEnum NDVarTr::AreNodesEqual(VarTrNodeAccessorPtr node1,
VarTrNodeAccessorPtr node2);**

Returns `BOOL_TRUE` if both accessors refer to the same node in the tree.

Reading and Setting the Cursor

Methods of the tree-datasource object to read and set the node and edge cursors.

A tree variant datasource keeps track of one cursor that can be on any node. This cursor is a user-allocated node accessor. No special action is attached to the action of moving the cursor around in the datasource itself. The user can obtain the cursor by `VARTR_GetCursor` call, and use

`VARTRNODEACCESSOR_Go*` calls to manipulate the cursor to different locations.

GetCursor

VarTrNodeAccessorPtr NDVarTr::GetCursor(void);

SetCursor

BoolEnum NDVarTr::SetCursor(VarTrNodeAccessorPtr accessor);

These routines get and set the current position of the node cursor. The caller is responsible for disposing of the accessor returned from the get method. The Set routine sets the tree-node cursor to “node” by internally creating and committing an edit object. It returns `BOOL_TRUE` if the internal edit could take place, `BOOL_FALSE` if not.

DisposeCursor

void NDVarTr::DisposeCursor(void);

Destructs and deallocates a variant-tree-datasource cursor. The internal reference in the tree datasource is also reset to `NULL`. Use this call after the `VARTR_SetCursor` call to reset the cursor for later use.

SetCursor

void NDVarTrEdit::SetCursor(VarTrNodeAccessorPtr accessor);

Modifying the Tree Datasource

Tree-Node Values

QueryNodeValue

void NDVarTr::QueryNodeValue(VarTrNodeAccessorCPtr accessor, VarPtr value);

Returns the value of the node referenced by node accessor “node” into the variant “value.”

GetNodeValue

VarPtr NDVarTr::GetNodeValue(VarTrNodeAccessorCPtr accessor);

SetNodeValue

BoolEnum NDVarTr::SetNodeValue(VarTrNodeAccessorCPtr accessor, VarCPtr value);

Gets and sets the value for the node in the variant tree datasource. The caller is responsible for disposing of the variant returned from the get method. The set method sets the value of the node to “value” by internally creating and committing an edit, and returns `BOOL_TRUE` if it succeeded.

Tree-Node IDs

QueryNodeID

void NDVarTr::QueryNodeID(VarTrNodeAccessorCPtr accessor, VarPtr id);

Returns the data stored in the ID field of the node referenced by node accessor “node” into the variant “value.”

GetNodeID

VarCPtr NDVarTr::GetNodeID(VarTrNodeAccessorCPtr accessor);

SetNodeID

BoolEnum NDVarTr::SetNodeID(VarTrNodeAccessorCPtr accessor, VarCPtr id);

Gets and sets the ID for the node in the variant tree datasource. The caller is responsible for disposing of the variant returned from the get method. The set method sets the ID of the node to “value” by internally creating and committing an edit, and returns `BOOL_TRUE` if it succeeded.

StartEdit

VarTrEditPtr NDVarTr::StartEdit(void);

Opens an edit globally for the variant tree datasource. `NULL` will be returned if no edit could be opened; otherwise, a constructed edit is returned.

StartNodeEdit

VarTrNodeEditPtr NDVarTr::StartNodeEdit(VarTrNodeAccessorCPtr accessor);

Opens an edit globally for the variant tree node pointed to by the node accessor. `NULL` will be returned if no edit could be opened; otherwise, a constructed edit is returned.

Modifying the Tree-Node Datasource

SetNodeValue

void NDVarTrEdit::SetNodeValue(VarTrNodeAccessorCPtr accessor, VarCPtr value);

Sets the data for Value field of the node at the current “accessor” location to “value” in the tree edit object. When the edit object is committed, the new data is stored in the datasource.

SetValue

void NDVarTrNodeEdit::SetValue(VarCPtr value);

Sets the data for Value field of the node at the current “accessor” location to “value” in the node edit object. When the edit object is committed, the new data is stored in the datasource.

SetNodeID

**void NDVarTrEdit::SetNodeID(
VarTrNodeAccessorCPtr accessor, VarCPtr id);**

Sets the data for ID field of the node at the current “accessor” location to “id” in the tree edit object. When the edit object is committed, the new data is stored in the datasource.

SetID

void NDVarTrNodeEdit::SetID(VarCPtr id);

Sets the data for ID field of the node at the current “accessor” location to “id” in the node edit object. When the edit object is committed, the new data is stored in the datasource.

Modification Descriptions

GetMods

VarTrModsCPtr NDVarTr::GetMods(void);

Get a description of the last modifications made on the tree datasource through an edit object.

50 *VStr* Class

The *VStr* class implements the Open Interface variable string data structures and utilities.

Technical Summary

Variable strings support multibyte characters. For more information about multibyte characters, see the *Char* class. For more information about multibyte strings, see the *Str* class.

A *VStr* object is a string object which owns its buffer and automatically handles buffer reallocation when the string changes or grows. Use a *VStr* to represent string fields of aggregates: names of resources and button labels, for example. Use an *SBuf* string instead when you need to perform complex editing operations on potentially long strings.

The APIs in the Open Interface *VStr* class enable you to manipulate variable strings and obtain information about them. You can allocate and deallocate memory for variable strings; initialize and destroy them; change their contents; obtain the string length and string contents; concatenate, insert, and delete strings and characters; compare variable strings; load resources into them; and copy, initialize, and dispose of arrays.

The *VStr* class API is divided into the following categories:

- Accessing C string inside.
- Allocation and Deallocation.
- Comparisons.
- Concatenation and Duplication.
- Data Structures.
- Lists of variable strings.
- Loading from resource file.
- Queries.

See also

Str, *Array*, *Char* classes.

Changing Contents

SetStr

Replaces the contents of a variable string with a copy of a string.

void NDVStr::SetStr (CStr source);

NDVStr::SetStr replaces the contents of a variable string with a copy of a source string. Unlike in *NDVStr::SetVStr*, *source* is a string rather than a variable string.

SetNatStr

Replaces the contents of a variable string with a copy of a native string.

void NDVStr::SetNatStr(NatCStr *natstring*);

NDVStr::SetNatStr replaces the contents of a variable string with a copy of a native string.

SetCtStr

Replaces the contents of a variable string with a copy of an encoded native string.

void NDVStr::SetCtStr(CtCPtr *codetype*, NatCStr *natstring*);

NDVStr::SetCtStr replaces the contents of a variable string with a copy of a native string encoded in the code type passed.

SetStrSub

Replaces the contents of a variable string with a copy of a substring.

void NDVStr::SetStrSub(CStr *subptr*, StrIVal *length*);

NDVStr::SetStrSub replaces the contents of a variable string with a copy of a substring.

SetNatStrSub

Replaces the contents of a variable string with a copy of a native substring.

void NDVStr::SetNatStrSub(NatCStr *natsubptr*, StrIVal *length*);

NDVStr::SetNatStrSub replaces the contents of a variable string with a copy of a native substring.

SetCtStrSub

Replaces the contents of the vstr by a copy of str.

void NDVStr::SetCtStrSub(CtCPtr *ct*, NatCStr *str*, StrIVal *slen*);

Set

Copies one variable string to another.

void NDVStr::Set(VStrCPtr *vstr2*);

NDVStr::Set copies the contents of vstr2 into a variable string.

Copy

Copies one variable string to another.

void NDVStr::Copy (VStrCPtr *vstr2*);

NDVStr::Copy copies the contents of vstr2 into a variable string.

Queries**GetLen**

Returns the length of a variable string.

StrVal NDVStr::GetLen (void);

NDVStr::GetLen determines the length of the variable string passed and returns an integer indicating the string length.

GetStr

Returns the string equivalent of a variable string.

CStr NDVStr::GetStr (void);

NDVStr::GetStr retrieves the string equivalent of the variable string passed and returns it.

QueryStrSub

Finds a substring within a variable string.

void NDVStr::QueryStrSub(CStrPtr stringptr, StrValPtr lengthptr);

NDVStr::QueryStr sets the string pointer to the substring found within a variable string. Sets the length pointer to the length of the substring.

Concatenation, Insertion, Deletion

AppendStr

Appends a string to a variable string.

void NDVStr::AppendStr(CStr str);

NDVStr::AppendStr appends a string to a variable string.

AppendStrSub

Appends a substring to a variable string.

void NDVStr::AppendStrSub(CStr subptr, StrVal length);

NDVStr::AppendStrSub appends a substring to a variable string.

Append

Appends one variable string to another.

void NDVStr::Append(VStrCPtr vstring2);

NDVStr::Append appends one variable string to another. The variable string passed as the second argument is appended to the variable string passed as the first argument.

AppendChar

Appends a character to a variable string.

void NDVStr::AppendChar(ChCode chcode);

NDVStr::AppendChar appends a character to a variable string.

TruncAt

Truncates a variable string exactly to the length specified.

void NDVStr::TruncAt(StrIVal pos);

NDVStr::TruncAt truncates exactly to length.

Truncate

Truncates a variable string at or before the length specified.

void NDVStr::Truncate (StrIVal length);

NDVStr::Truncate truncates at or before length.

Clear

Resets a variable string.

void NDVStr::Clear(void);

NDVStr::Clear resets the contents of a variable string.

Comparisons

CmpStr
ICmpStr

Compares a variable strings with another string by comparing the characters in each string by code value.

CmpEnum NDVStr::CmpStr(CStr string2);

CmpEnum NDVStr::ICmpStr(CStr string2);

NDVStr::CmpStr and NDVStr::ICmpStr compare the a variable string with a string. The characters in each string are compared by code value. No attempt is made to compare characters across code sets. The ASCII order is used for ASCII characters, so a is sorted after Z, but between A and B.

NDVStr::ICmpStr is the same at NDVStr::CmpStr but ignores case differences in the comparison.

Use these calls when you need a fast way to perform comparisons, but you do not need a high degree of accuracy.

Cmp
ICmp

Compares two variable strings, ignoring case differences in the ASCII range,

CmpEnum NDVStr::Cmp(VStrCPtr vstring2);

CmpEnum NDVStr::ICmp(VStrCPtr vstring2);

NDVStr::Cmp compares the strings by comparing the characters in each string by code value.

NDVStr::ICmp is the same as NDVStr::Cmp but it ignores case differences in the ASCII range only.

Use these calls when you need a fast way to perform comparisons, but you do not need a high degree of accuracy.

Loading Resources

SetRes

Sets the given string resource as the contents of a variable string.

void NDVStr::SetRes (CStr class, CStr resource);

NDVStr::SetRes sets the given string resource as the contents of a variable string.

Arrays Of Strings

Constructor

VStr array construction.

NDVStrArray::NDVStrArray (void);

Default VStr array construction.

NDVStrArray::NDVStrArray (VStrArrayCPtr va2);

Constructs the VStr array as a clone of `va2`. Performs a `deep` copy, the VStr array contains copies of the strings in `va2`.

Destructor

Default VStr array destruction.

NDVStrArray::~~NDVStrArray(void);

Index

Numerics

2-byte characters 334

A

abort operations 228

ABS 180

absolute file names 296

absolute values 180

access bit constants 279

access rights 274, 278

accessors *See* graph datasources

ADOBE code sets 208

ADOBE code type 217

alignment 361

allocation 131, 147, 157

 failing 360

 memory pool 357

ANSI C compiler *See* C language

application programming interface (API)

 Args calls 127–129

 ArNum calls 131–137

 ArObj calls 139–146

 ArPtr calls 147–153

 ARRay calls 155–156

 ArRec calls 157–162

 Avl calls 163–168

 Base calls 169–181

 BBuf calls 183–191

 Cell calls 193–194

 Char calls 195–205

 Cs calls 207–214

 Ct calls 215–222

 Ds calls 223–226

 Err calls 227–245

 File calls 247–??

 FMgr calls 273–293

 FName calls 295–317

 Hash calls 319–324

 Heap calls 325–326

 ISet calls 327–329

 Mch calls 331–337

 Nfier calls 339–341

 Pack calls 343–348

 PFld calls 349–350

 Point calls 351–353

 Pool calls 355–358

 Ptr calls 359–??

 RClas calls 371–375

 Rect calls 377–382

 Res calls 383–404

 Rgn calls 405–410

 RLib calls 411–413

 SBuf calls 415–420

 Scrip calls 421–430

 Set calls 431–433

 Str calls 435–463

 StrL calls 465–467

 StrR calls 469–470

 Var calls 471–473

 VarDs calls 479–480

 VarGr calls 481–507

 VarLs calls 509–513

 VarTb calls 515–521

 VarTr calls 523–529

 VStr calls 531–??

applications 1

 exiting 243

 nonwindow-based 389

 running 428

APPSTARTUP event 428

argc/argv 127

arguments *See* command-line arguments

ARNUM_DECLARECLASS 131

ARNUM_DEFCLASS 132

ARNUM_DEFSTRUCT 132

ARNUM_ExtractElt 136

ARNUM_IMPLEMENTCLASS 132

ARPTR_DECLARECLASS 147

ARPTR_DEFCLASS 148

ARPTR_IMPLEMENTCLASS 147

array number

 append number 135

 append unique element 135

 construct allocated ARNUM 132

 construct ARNUM 132

 construct with 'len' 132

 default constructor 132

 default destructor 132

 detect element 134

 extract element 136

 extract Nth element 136

 find element 134

 find sorted element 135

 get length 133

 get Nth element 133

 get unbounded Nth element 133

 insert Nth element 135

 insert sorted element 135

 insert sorted unique element 135

 look up element 134

 look up sorted element 134

 reallocate ARNUM 133

 remove duplicates 136

 remove element 136

 remove Nth element 135

 remove sorted duplicates 137

 reset contents 132

 return empty ARNUM 133

 return range 133

 return sorted 136

 set length 133

 set Nth element 134

 set unbounded Nth element 134

 sort ARNUM 136

- sort extracted element 136
- array objects 139
 - append 144
 - append unique element 145
 - construct array 142
 - construct array with len 142
 - default constructor 142
 - destroy array 142
 - detect elt object 144
 - extract element 146
 - extract sorted element 146
 - find elt 144
 - find sorted element 144
 - get length 143
 - get Nth element 143
 - insert Nth element 145
 - insert sorted element 145
 - insert sorted unique element 145
 - look up element 144
 - look up sorted element 144
 - reallocate capacity 142
 - remove duplicate elements 146
 - remove element 145
 - remove Nth element 145
 - remove sorted duplicate elements 146
 - reset array 142
 - return array 146
 - return empty array 143
 - set length 142
 - set Nth element 143
 - sorting 146
 - testing index validity 143
- array pointer
 - append element 151
 - append unique element 151
 - construct ARPTR 148
 - construct with 0 elements 148
 - construct with len 148
 - default constructor 148
 - default destructor 148
 - detect elements 150
 - detect empty ARPTR 149
 - detect range 149
 - extract element 152
 - extract Nth element 152
 - extract sorted element 152
 - find elements 150
 - get length 149
 - get Nth element 149
 - get Nth element address 149
 - get unbounded Nth element 150
 - graphs 60, 62, 63
 - insert Nth element 151
 - insert sorted element 151
 - insert sorted unique element 152
 - look up element 150
 - look up sorted element 151
 - reallocates contents 149
 - remove elements 152
 - remove Nth element 152
 - remove sorted duplicates 153
 - removing duplicates 153
 - reset contents 148
 - returns sorted pointer 153
 - search matching key 151
 - set Nth element 150
 - set unbounded Nth element 150
 - sorting 152
- array record
 - append element 160
 - append unique element 160
 - construct with len 158
 - default constructor 158
 - default destructor 158
 - detect elements 159
 - detect empty record 159
 - extract element 161
 - extract Nth record 161
 - extracts sort element 162
 - find elements 160
 - find sorted element 160
 - get length 159
 - get Nth element 159
 - insert sorted element 161
 - insert sorted unique record 161
 - inserts Nth element 161
 - lookup elements 160
 - lookup sorted element 160
 - reallocates contents 158
 - remove elements 161
 - remove Nth element 161
 - remove sorted records 162
 - reset contents 158
 - return sorted record 162
 - returns range 159
 - set length 149, 158
 - set Nth element 159
 - sorting 162
- arrays element types 139
- ARREC_DECLARECLASS 157
- ARREC_DEFCLASS 158
- ARREC_IMPLEMENTCLASS 157
- ASCII character
 - writing to a native string 447
 - writing to a string 446, 447, 460, 461
- ASCII characters 201, 202
 - byte value mapping 215
 - converting to EBCDIC 204
 - converting to lower case 203
 - converting to native 205
 - converting to upper case 203
 - define primary set 334
 - get base value 203
 - get integer values 202
 - information definition 213
- ASCII code type 217
- assertion macros 178
- assertions 242
- assignment statements 422
- asynchronous notifications 399
- atomic data sources 12
- attached resources 385
- auto backup flag 256, 257
- autosizing graph nodes 81
- AvlNode 163

- change current node 167
- construct node, assign key 164
- default constructor 164
- default destructor 164
- find current key 166
- finding 168
- get current node 168
- get first/last node 166
- get key 164
- get left child 165
- get leftmost descendant node 165
- get nearest current node 168
- get next node 165
- get parent of current node 165
- get previous node 165
- get rightmost descendant node 165
- look up key 166
- set key 164
- AvlTree 163
 - callback function 166
 - constructors 165
 - default constructor 165
 - default destructor 165
 - extract node 168
 - extracting nodes 166
 - get number of nodes 166
 - go first or last node 167
 - inserting nodes 166, 168
 - position data structure 163
 - propagate action 167
 - set current node 167
- B**
- backup files 269
- bare scripts 421, 429
- Base class 169
- BASE_NOMINMAX 179
- binary data 183
- binary files 247
 - current absolute position 263
 - read byte 265
 - read N bytes 265
 - return current offset 263
 - set file position 263
 - write byte 265
 - write N bytes 266
- binary trees 163
 - See also* AvlNode; AvlTree
- BOOL_OF 173
- boolean conversion 173
- boolean values 173
- BoolEnum 173
- buffer *See* memory buffer; string buffer
- Byte 172
- BytePtr 172
- C**
- C language
 - export function prototype 331
 - freeing memory 362
 - portable const keyword 331
 - register variables 333
 - registering resources 387
 - scripting functionality 421
 - signed integers 333
 - volatile keyword 334
- C++ language
 - construct/destruct resources 386
 - registering resources 387
 - scripting functionality 421
- C_CONST 331
- C_INITOFFSET 180
- C_NOSHARE 332
- C_OFFSET 180
- C_READONLY 333
- C_REG... register variables 333
- calls to verbs 423
- CCITT fax compression 347
- CCITT Group3 algorithm 343, 347
- CCITT Group4 algorithm 343, 347
- ccittFlags 347
- CellPtr 193
- CellRec 193
- cells 193
 - range operations 194
- Char data type 196
- character
 - append to string buffer 417
 - ASCII base value 203
 - define default language 197
 - define native code type 197
 - finding 452, 453, 454
 - get 8-bit character length 199
 - get ASCII integer value 202
 - get code 443
 - get native character code 443
 - get native character length 199
 - global character pointer 197
 - information definition 213
 - lower case form 222
 - match string 450
 - multibyte character pointer 198
 - native character pointer 197
 - testing for ASCII 201, 202
 - UNICODE character pointer 198
 - upper case form 222
- character code 199
 - See also* code sets
 - getting from a string 443, 444
 - writing into a string 446, 447
- character conversion
 - ASCII character to EBCDIC 204
 - ASCII character to lower case 203
 - ASCII character to upper case 203
 - ASCII to native character 205
 - character to character code 220
 - character to control character 204
 - chcode to unicode 221
 - code set to code type 221
 - code type to code set 221

- control character to character 204
- cscode to unicode 213
- EBCDIC character to ASCII 205
- native character to ASCII 205
- unicode to chcode 221
- unicode to cscode 213
- character sets 195
 - define primary 334
- CHARINFO_UNKNOWN 213
- CHARINFO_UNKNOWN_FULLWIDTH 213
- CHARINFO_UNKNOWN_HALFWIDTH 213
- CharInfoVal 198
- CharPtr 197
- charts *See* graphs
- ChCode 198, 216
- ChCodePtr 197
- chip architecture 334
- CJK code sets 210
- CJK code type 218
- class
 - See also* resource class
 - default allocation method 372
 - get first alphabetically 373
 - get next alphabetically 374
 - verify for resource 404
 - verify inheritance 396
- client data
 - get resource 395
 - set resource 395
- ClientPtr 171
- closing files 260
- CmpEnum 174
- CmpProc 175
- CNS code sets 211
- CNS code type 219
- code mapping 215
- code pages 209
- code set objects
 - construct from csid 211
 - convert character 212
 - convert from unicode 213
 - convert to unicode 213
 - default destructor 211
 - default object constructor 211
 - get character length 211
 - get charinfo value 211
 - get global 213
 - get id 211
 - get native 213
 - get unicode 213
 - translate character 212
- code sets 207, 216
 - ADOBE 208
 - CNS 211
 - data type 208
 - defined 215
 - EBCDIC 210
 - HP 211
 - ISO 208
 - Japanese characters 210
 - Macintosh 208
 - MS/Windows 209
 - PC code pages 209
 - UNICODE 210
- code type ids 216, 219
- code type structures 215
- code types 215
 - code value 216
 - constructor 219
 - define data record 216
 - destructor 219
 - determine single-byte characters 222
 - get character code 220
 - get character value 220
 - get CharInfoVal 220
 - get maximum length 222
- coding scheme *See* code types
- collection classes 155
- collections
 - numeric values 131
 - object references 147
 - objects 139
 - records 157
- command objects 403
- command-line arguments 127
 - extract Nth argument 129
 - get application name 128
 - get first argument 129
 - get list 128
 - get next argument 129
 - get Nth argument 128
 - get number 128
 - initialization procedure 128
 - insert new argument 129
- commits 225
- comparison functions 175
- comparisons 174
 - file names 316
 - hash tables 320
 - sets 433
 - strings 534
- compilers
 - define in-use compiler 336
 - far keyword 332
 - near keyword 332
 - signed integers 333
 - VMS-specific keywords 332, 333
 - volatile keyword 334
- compression 344
- compression algorithms 343
 - See also* pack objects
- conditional statements 422
- conflicts, macros 179
- connected graphs 67
- constants 171, 424
 - access bits 279
 - file name components 303
 - file name error signals 302
 - file name syntax 300
 - Mac creator and type signature 279
 - registering 424

- system-specific file names 300
 - containers 225
 - context messages 232, 239
 - contiguous values 327
 - control characters 204
 - controlled access 6
 - conversions 196, 295
 - See also* character conversion
 - integer internal storage 366, 367
 - string internal storage 367, 368
 - coordinates 377
 - See also* rectangle
 - graph origins 61
 - set for region 407, 408
 - copying 173
 - core services 2
 - CpyEnum 173
 - CsIdEnum 208
 - Ct 216
 - CT_ID 216
 - CtIdEnum 216
 - current directory 313
 - current volume 313
 - cursor types (graphs) 73
 - view options 82
 - cyclic graphs 499
- D**
- DA data sources 10
 - dat format 383
 - data 319, 371
 - data alignment 184
 - data mapping 12
 - data source 223
 - See also* variant data source
 - abort edition 224
 - abort update 225
 - add contDs 225
 - add edition operation 224
 - commit edition 224
 - commit update 225
 - complete edition 224
 - constructor 226
 - controlled access 6
 - controlled access example 8
 - destructor 226
 - get associated view 224
 - get edition owner 225
 - get view option 223
 - open edition 224
 - open update 225
 - register view 223
 - remove contDs 225
 - set edition owner 224
 - set view option 223
 - unregister view 223
 - data source edition objects 479, 512, 519
 - Data Source Internals 9
 - DA data sources 10
 - IRE data sources 12
 - OI core 9
 - data source/view mechanism 2, 5
 - data access 6
 - examples 8, 13, 16
 - locking data 7
 - propagating events 5
 - data source/view types
 - CBOX_NFY 6
 - CBUT_NFY 6
 - LBOX_NFY 6
 - TED_NFY 6
 - data structures
 - AvlNode 163
 - Balanced Binary Tree 163
 - file I/O 251
 - hash tables 319
 - I/O buffering 185
 - Macintosh type and creator signatures 274
 - memory management statistics 359
 - store Macintosh signature 275
 - data types 169
 - allow macros overriding 179
 - file manager 273–276
 - file names 299
 - huge string 172
 - native string 437
 - native string pointer 438
 - scripts 422
 - signed integers 333
 - string 438
 - string pointer 438
 - UNICODE string 439
 - UNICODE string pointer 439
 - DataSource class 223
 - edition operations 226
 - modifications implementation 226
 - datasource edit objects 75
 - DBG_CHECK 176
 - DBG_CHECKSTR 176
 - DBG_ERROR 177
 - DBG_FILE 177
 - DBG_LINE 177
 - DBG_NIY 177
 - DBG_ON 178
 - DBG_REQUIRE 178
 - DBG_SCCS 178
 - DBG_SOURCE 178
 - deallocation 131, 147, 157
 - failing 360
 - debugging macros
 - activate source code 178
 - checks assertion truth 178
 - defining active 178
 - determine current file name 177
 - determine line number 177
 - expression failure check 176
 - hold SCCS info 178
 - debugging tools 169
 - decimal

- converting string to text 458
 - get string integer 454
 - decoding routines 345
 - decompression 345
 - detached resources 385
 - diagrammer *See* graph diagrammer
 - directories 299, 313
 - convert file to path 311
 - convert path to file 310
 - copy contents 287
 - create new 274, 286
 - deleting 289
 - get current 314
 - get parent 314
 - get top 312, 315
 - get wildcard expression 292
 - match files 291
 - move 288
 - purge files 291
 - query current 313
 - query current parent 314
 - query current top 312
 - query current volume 313
 - query home 315
 - query parent 315
 - query top 315
 - remove contents 289
 - rename 288
 - set current 313, 314
 - test for top level 312
 - test path 316
 - test specification 310
 - disconnected graphs 67, 68
 - disjoint rectangles 405
 - disposing
 - string 440
 - DLL code 335
 - DOS file I/O 247
 - Double data type 169
 - drawing operations 377
 - DS/V *See* data source/view mechanism
 - DsEditCompletionEnum 224
 - DsEditOpEnum 226
 - DsEditStateEnum 226
 - DsEditTypeEnum 226
 - DsModsSetEnum 226
- E**
- EAS 1
 - See also* EE applications
 - EBCDIC characters 204, 205
 - define primary set 334
 - EBCDIC code sets 210
 - EBCDIC code type 218
 - edge (defined) 62
 - edge accessors 71, 124, 482
 - creating 115–117
 - edge cursors 71, 73
 - view options 82
 - edge edit objects 76
 - edge ID values 64
 - edge pointer arrays 62, 63
 - edit objects 75–76, 507
 - adding titles 114
 - creating/destroying 113
 - edition interfaces 224
 - edition objects 479, 512, 519
 - EE applications 1
 - bi-directional linkage 2
 - services classes 2
 - elements
 - See also* array objects
 - accessing 143, 149, 159
 - adding 135, 144, 160
 - finding 143, 150
 - in arrays 139
 - numeric values 131
 - removing 135, 145, 161
 - removing duplicates 136, 146, 153, 162
 - Elements Environment Application Services 1
 - encoded native strings 532
 - encoding routines 344
 - endianness 184, 189
 - entry/exit macros 229
 - enumerated types 169
 - code set ids 208
 - code type ids 216
 - comparison results 174
 - convert integer to enum 174
 - copy results 173
 - data source edition 226
 - data source modification 226
 - describe file extension 285
 - describe file type 285
 - describe persistent field 350
 - edition interface 224
 - file compression 348
 - file manager 276–281, 283
 - file names 301
 - file open modes 254
 - horizontal direction 175
 - identify file types 283
 - interval sets 327
 - Macintosh creator and type signatures 279
 - memory manager 360
 - memory pool 358
 - perform action 175
 - persistent field categories 350
 - persistent fields 349
 - regions 405
 - register script event 425
 - register script verb 427
 - resource class flags 371
 - resource notification 396
 - set bit operations 433
 - specify file errors 252
 - specify file format 253
 - VarTypeEnum 471
 - version 176
 - vertical direction 175

- environment
 - chip architecture selection 334
 - operating system selection 335
 - windowing system selection 336
- environment variables 271
- ERR_ASSERT 242
- ERR_CHECK 242
- ERR_CHECKSTR 242
- ERR_DECLARE 234
- ERR_Exit 232
- ERR_EXTERN 234
- ERR_Fail 230
- ERR_FailSilent 231
- ERR_FailStr 230
- ERR_Fatal 231
- ERR_INMODULE 229
- ERR_ISLIBCREATED 234, 241
- ERR_LIB 234, 240
- ERR_LIBCREATEINIT 234, 241
- ERR_LIBDECLARE 241
- ERR_MAININIT 234, 241
- ERR_SETOPTINT 233, 240
- ERR_SETOPTSTR 233, 240
- ERR_SETOPTVSTR 233, 240
- ERR_TRACEALL 233, 239
- ERR_Warn 231
- ERR_WarnStr 231
- ERR_XIN 229
- ERR_XOUT 229
- error codes 302
- error handling global variables 233, 240
- error module 243
- errors 227
 - assertions 242
 - broadcast exit message 243
 - condition signals 243
 - context information 232, 239
 - default format procedure 240
 - default report procedures 240
 - fatal 231, 241
 - file I/O 252, 272
 - file manager 276
 - file name signals 302
 - format end user message 237
 - format traceback message 238
 - get global report 238
 - get pointer to reporting structure 237
 - get topmost frame 237
 - invoke default report 238
 - invoke global report 238
 - load message 240
 - mark error frame 238
 - memory 368
 - message numbers 232
 - output traceback 243
 - override global report 238
 - query state 242
 - recovering from 229
 - report status 238
 - reporting 235–237, 240, 243
 - retrying 229
 - signaling failures 230, 241
 - tracing 233, 239
- EVEN 180
- events 5
 - registering 424
- exceptions 227
 - Macintosh systems 245
 - UNIX systems 243–244
 - W16 API 244–245
- exit functions 243
- exit status 179
- EXIT_FAIL 179
- EXIT_OK 179
- export functions 331
- extent 377
 - See also* rectangle
- F**
- FailIfNotFound 257
- FailOnEOF 257, 258
- failures 177, 227
 - generate silent 241
 - signaling 230, 241
 - trapping 369
- fatal errors 231, 241
- file
 - close 260
 - create backup 269
 - create/open 260
 - detect end of file 262
 - determines node type 259
 - find file 258
 - flush output buffer 269
 - get auto backup flag 256
 - get client data 258
 - get default search path 270
 - get default search path name 271
 - get error 272
 - get FailIfNotFound flag 257
 - get FailOnEOF flag 258
 - get open format and mode 262
 - get real file name 255
 - get search path 256
 - get specified name 255
 - go to beginning 262
 - go to end 262
 - naming conventions 295
 - open 259, 260
 - overview 247
 - query current text position 263
 - query line position 264
 - query native reference 271
 - read line 269
 - read N text characters 267
 - read next character 266
 - read string 267
 - read text line 268

- read/write operations 264
- return current line number 264
- return current size 262
- return current text offset 263
- set auto backup flag 257
- set client data 258
- set default search path 270
- set default search path name 271
- set error 272
- set FailIfNotFound flag 257
- set FailOnEOF flag 257
- set line position 264
- set native reference 271
- set search path 256
- set specification name 255
- set text position 264
- specify format 253
- test for open file and mode 261
- testing read access 259
- testing write access 259
- truncate file 270
- write line 269
- write N text characters 267
- write next character 266
- write string 268
- write text line 268
- file attributes 247
- file compression *See* pack objects
- file handles 251
- file manager 247, 273
 - access rights 274, 278
 - add file type 285
 - check access permissions 281
 - check node type 283
 - copy directory 287
 - copy file 287
 - copy node 287
 - create file 274, 286
 - create new directory 274, 286
 - data types 273–276
 - delete directory 289
 - delete directory contents 289
 - delete file 289
 - delete node 289
 - detect concealed device 281
 - determine node existence 281
 - enumerated types 276–281, 283
 - error reporting 276
 - find file type ID 286
 - find file type information 286
 - get Macintosh signature 282
 - get Macintosh type 282
 - get node type 282
 - get Nth file type 285
 - get registered file types 285
 - modification and creation times 276
 - move directory 288
 - move file 288
 - move node 288
 - node owner information 275
 - node references 275
 - node types 278
 - non-asserting functions 290
 - perform directory action 291
 - perform volume action 293
 - purge file 291
 - remove file type 285
 - rename directory 288
 - rename file 288
 - rename node 288
 - return file wildcard pattern 292
 - return wildcard expression 292
- file name 247, 295
 - buffer size 302
 - check node type 283
 - compare 316
 - conversion status 301, 308
 - convert to absolute 316
 - convert to current syntax 307
 - convert to given syntax 307
 - data types 299
 - define components 300, 303
 - describe status value 309
 - determine convertibility 308
 - enumerated types 301
 - evaluate current expression 306
 - evaluate specified expression 306
 - extract and copy component 309
 - extract file component 309
 - find path name syntax 305
 - get and convert syntax 308
 - get component set 309
 - get current syntax 304
 - get native system syntax 304
 - get native temporary file 317
 - get real 255
 - get syntax 304, 307
 - get temporary file 317
 - identify syntax 301
 - make backup file 317
 - make name valid 306
 - merge path to full name 311, 312
 - parsing 311
 - query current syntax 304
 - query node information 282
 - reduce to component 310
 - reset conversion parameters 304
 - returning 255
 - set conversion flag 309
 - set conversion parameters 304
 - set current syntax 304
 - set temporary file 317
 - setting 255
 - signal errors 302
 - storing path name 299
 - syntax constants 300
 - test portability 308
 - test status 316
 - validate conversion 308
 - validate current syntax 306
 - validate system syntax 305
- file open modes 254
- file pointers 249, 251
- file type conversions 295
- FILE_FMTLINE mode 251
- FILE_FMTTEXT mode 252
- FileErrEnum 252

- FileFmtEnum 253
 - FileFmtEnum mode 249
 - FileIOEnum 254
 - FileIOEnum mode 249
 - FileLinePosPtr 251
 - FileLinePosRec 251
 - FileNatRefPtr 251
 - FileNatRefRec 251
 - FileOffsetVal 252
 - FilePtr 249, 251
 - FileTextPosPtr 252
 - FileTextPosRec 252
 - finding resources 390
 - floating point numbers 170
 - FMGR_ACCESS... constants 278
 - FMGR_MAC... constants 279
 - FMgrAccessSet 274
 - FMgrCreateDirPtr 274
 - FMgrCreateDirRec 274
 - FMgrCreateFilePtr 274
 - FMgrCreateFileRec 274
 - FmgrErrEnum 276
 - FMgrFileTypeEnum 277, 283
 - FMgrMacIdsPtr 274
 - FMgrMacIdsRec 274
 - FMgrMacIdVal 275
 - FMgrNodeEnum 278
 - FMgrNodePtr 273
 - FMgrNodeRec 273
 - FMgrOwnerPtr 275
 - FMgrOwnerRec 275
 - FMgrRefsVal 275
 - FMgrSizeVal 275
 - FMgrTimesPtr 275
 - FMgrTimesRec 275
 - FMgrTimeVal 276
 - FNAME_COMP... constants 303
 - FNAME_FAIL... constants 302
 - FNAME_MAXLEN 299, 302
 - FNAME_STXMASK... constants 300
 - FNameBuf 298, 299
 - FNameCompSet 300
 - FNameCompSetEnum 303
 - FNameParamsPtr 300
 - FNameParamsRec 300
 - FNameStatusEnum 301
 - FNameStxEnum 301
 - FNameStxMaskVal 300
 - FSS-UTF code type 218
- G**
- global string type 435
 - global variables 233, 240, 422
 - graph datasources 59
 - See also* graphs
 - accessors 61, 70, 114–117, 124
 - adding directed edge 62, 65, 124
 - example 124
 - adding edit objects 75–76, 113
 - adding nodes 60, 67, 117–124
 - adding undirected edge 63, 66, 124
 - example 125
 - autosizing nodes 81
 - basic components 60, 63
 - building 112–126
 - basic tasks 113
 - creating child nodes 119
 - creating edges 124
 - creating linked nodes 118
 - creating neighbor nodes 122
 - creating parent nodes 120
 - creating unlinked nodes 123
 - cursor types 73
 - default fields 60, 63
 - define cursor type options 82
 - define default link shape 110, 112
 - define default node shape 105
 - define edge connections 64
 - define frame color 102, 104
 - define label color 102, 104, 109, 111
 - define link color 108, 109, 110, 111
 - define node color 102, 104
 - display labels 103, 105, 110, 112
 - display nodes 101, 108
 - edge pointer arrays 62, 63
 - link unlinked nodes 124
 - navigating through nodes 62
 - origins 61
 - parent/child node relationships 69, 118
 - set default node shape 103
 - set edge ID 64
 - set edge values 64
 - set label font 103, 105, 109, 112
 - set link properties 98, 106
 - set node height 62, 84, 101
 - set node ID 61
 - set node properties 98, 99
 - set node values 61
 - set node width 62, 84, 101
 - set pen 103, 105, 109, 111
 - set view options 80
 - specify as readonly 83
 - updating 507
 - graph diagrammer 83–98
 - add bitmap file 87
 - define background color 88
 - define default link shape 98
 - define default node shape 96
 - define frame color 89, 94
 - define label color 89, 92, 95, 97
 - define link color 91, 92, 96, 97
 - define node color 89, 94
 - define orientation 86
 - define overview window 86
 - display node labels 90, 93, 95, 98
 - display node/links 88, 93
 - reference cyclic nodes 87

- set axes values 84–85
- set default link shape 93
- set default node shape 91
- set font 90, 92, 95, 97
- set grid alignment 85
- set node size 84
- set parameter values 81, 83
- set pen 90, 92, 95, 97
- set zoom level 86
- graph edit objects 499, 507
- graph properties 481
 - edges 495, 497, 504, 506
 - links 98, 106
 - nodes 98, 99, 490, 494, 503
- graphs 66
 - See also* graph datasources
 - accessor validity 490
 - adding and removing edges 488
 - adding and removing nodes 488
 - changing 499, 507
 - declare edge accessor 71
 - edge accessors 482
 - edge IDs 495, 504
 - edge-accessor navigation 485
 - edge-accessor validity 495
 - getting and setting cursors 498
 - neighboring nodes 70, 122
 - node accessors 482
 - node IDs 491, 501
 - node-accessor navigation 483
 - root nodes 68, 117
 - x origins 492, 502
 - y origins 492, 502

H

- hash table 319
 - add entry 323, 324
 - compare entries 320
 - construct with info 320
 - default comparison 323
 - default constructor 320
 - default destructor 320
 - default hashing procedure 323
 - default string cloning 323
 - defining members 319
 - dispose stored data 321
 - extract key entry 322
 - fill hashInfo 322
 - get bin index 323
 - get data clone 321
 - get default clone string info 322
 - get default integer info 321
 - get default pointer info 321
 - get default string info 322
 - get entry 323, 324
 - get entry key 324
 - get entry value 324
 - hashing procedures 321
 - insert key entry 322
 - lookup key entry 322
 - query default settings 321
 - query info 322

- query statistical info 324
- remove entry 324
- reset contents 320
- return data clone 321
- set entry value 324
- store information 321
- test entry existence 324
- trigger iteration 323
- heap 325
 - add with no reorder 326
 - corrected 326
 - default constructor 325
 - default destructor 325
 - extract top-most entry 326
 - get size 325
 - insert entry 326
 - perform proc on entry 326
- HorzEnum 175
- HP code sets 211
- HP code type 218
- huge pointer 172, 361, 362
 - allocate/deallocate 363
 - get byte 366
 - manipulation functions 364
 - set byte 366
 - swap byte 366
- HUGELIMIT 173
- HugePtr 172
- HugeStr 172

I

- I/O *See* input; output
- IBM mainframes file I/O 248
- InEdgePtrArr 60, 62
- inheritance 374
 - testing 396
- initialization 233, 241
- input 183, 247
- input tables 14
- Int 170
- INT_Compare 174
- INT_ToCmp 174
- Int16 170
- Int32 170
- Int64 170
- Int8 170
- integer constants 171
- integer data types 170
- integers
 - absolute value 180
 - convert to machine format 367
 - convert to standard format 366
 - evaluate odd 181
 - get decimal substring 456
 - get greater 180
 - get hex substring 457
 - get lesser 180
 - get radix 455

- get string 455
- get substrings 457
- getting decimal string 454
- memory allocation 366
- read machine-dependent 367
- signed 333
- write machine-dependent 367
- international applications 1
- interrupt handler 244
- interval sets 327
 - add intervals 328
 - check interval existence 329
 - comparing and combining 329
 - compute complement 329
 - count intervals 328
 - default constructor 327
 - default destructor 328
 - detect element 329
 - enumerated types 327
 - get all elements 328
 - get biggest element 329
 - get smallest element 328
 - get universal intervals 328
 - query intervals 328
 - remove intervals 328
 - set intervals 328
- IRE data sources 12
- ISO 8859-X code type 217
- ISO code sets 208
 - character information definition 213

J

- Japanese characters 210
- JIS0201 character information 214
- JIS0208 character information 214

K

- keys 319

L

- library 411
 - close 413
 - exit from resource 390
 - get first alphabetically 412
 - get name 411
 - get next alphabetically 412
 - get pointer 411
 - initialize and load script 423
 - initialize resource 390
 - install 389
 - install and initialize script 423
 - install script 423
 - load 412
 - load and edit 412
 - load and initialize 390
 - load and return 412
 - open 413
 - unload and close 413
 - unload and uninstall script 423

- Link Edition dialog 83

linking

- DA RecordSetData to ListBox 9
- IRE class with properties 13
- TextEdit to VariantDataSource 8
- TextEdit with IRE slot 16
- TextEdit with NXP slot 9

- linking units 234

- list data sources 7

- listbox views 15

- loading resources 390

- local variables 422

- locking data 7

- Long data type 169

- loops 422

- LZW algorithm 343

- Lzw algorithm 347

M

- MAC_HEADERS 240

- machine specific definitions 331

- Macintosh look

- code sets 208

- code type 217

- creator signatures 274, 275, 279

- enumerated file type 279

- exception handling 245

- file I/O 247

- file type 274

- operating system selection 335

- windowing system selection 336

- macros 169

- check node type 283

- code sets 211, 213

- compare two integers 174

- compute absolute integer value 180

- conflicts 179

- declare exported function 331

- define odd numbers 181

- determine even numbers 180

- entry/exit 229

- error context messages 233

- error handling global variable 234

- error reporting 240

- implement MSW code 335

- initialization 241

- numeric collections 131

- object collections 140

- object pointers 147

- offset in C structure 180

- portable const keyword 331

- provide offset in C structure 180

- record collections 157

- recovery and retry 239

- return empty value 181

- MAX 180

- MAXINT16 171

- MAXINT32 171

- MAXINT64 171

- MAXINT8 171
- MAXUINT16 171
- MAXUINT32 171
- MAXUINT64 171
- MAXUINT8 171
- MCH_Cc 336
- MCH_CHIP 334
- MCH_OS 335
- MCH_WCHAR 334
- MCH_WIN 336
- memory 355
 - allocation failures 360
 - get allocation failure callback 369
 - huge pointers 361, 362
 - integer size 131
 - manipulation tools 169
- memory buffer 183
 - clear 363
 - construct custom data bbuf 186
 - construct data bbuf 186
 - construct file bbuf 186
 - default constructor 186
 - destructor 187
 - detect modified page 190
 - flush changes to file 188
 - get beginning page position 190
 - get client data 189
 - get current pointer 190
 - get endianness 189
 - get paging data 189
 - get size 362
 - get total size 189
 - load current page 188
 - paging mechanism 183
 - paging methods 185, 191
 - query methods 191
 - read integers 187
 - read len bytes 187
 - resize 363
 - return current position 188
 - return pointer to current page 190
 - return pointer to first byte 190
 - seek operations 188
 - set absolute offset 188
 - set beginning page position 190
 - set client data 189
 - set current pointer 190
 - set endianness 189
 - set methods 191
 - set PageModified flag 190
 - set paging data 189
 - set pointer to current page 190
 - set pointer to first byte 190
 - set relative offset 188
 - set size 362, 363
 - set total size 189
 - skip bytes from current position 188
 - specialization flags 185
 - write integers 187
 - write len bytes 187
- memory manager 359
 - access blocks 361
 - alignment check 361
 - allocate huge pointer 363
 - allocate new pointer 362
 - clear pointer 363
 - compare pointers 364
 - convert integers 366
 - convert strings 367
 - copy pointer 364, 366
 - deallocate huge pointer 363
 - deallocate/dispose memory 362
 - describe statistics 359
 - error handling routines 368
 - get aligned size 361
 - get buffer size 362
 - get failure callback 369
 - get statistics 365
 - match pointer size 364
 - move pointer 364
 - output statistics 365
 - read integers 367
 - read pointer value 365
 - read strings 368
 - resize pointer 363
 - set buffer size 362, 363
 - set failure callback 369
 - set pointer values 365
 - store integers 366
 - store strings 367
 - swap pointer 364
 - swap pointer bytes 366
 - trap failures 369
 - write integers 367
 - write string 368
- memory pool 355
 - construct with info 357
 - deallocate pointer 357
 - default constructor 356
 - default destructor 357
 - definition 356
 - fill poolInfo 357
 - fill with statistical info 358
 - reset stats 358
 - return pointer to cell 357
 - update 357
- messages 232
 - send vs. post 399
- metadata 371
- microprocessor chip architecture 334
- Microsoft Windows *See* MSW look
- MIN 180
- MS/Windows code sets 209
- MS/Windows code type 217
- MSW look
 - huge strings 172
 - implement MSW code 335
 - operating system selection 335
 - portable far keyword 332
 - portable near keyword 332
 - windowing system selection 336
- multibyte character pointer 198
- multibyte characters 195, 215
 - See also* variable strings

- code mapping 215
- code types 216
- define encoding 334
- encoding 196, 199
- get byte-specific 200

N

- NatChar 196
- NatCharPtr 197
- NatCode 198
- NatCodePtr 197
- native character code
 - writing to a string 448
- native characters 195
 - converting to ASCII 205
 - pointer 197
- native file managers 273
- native interrupt handler 244
- native rectangle type 378
- native string
 - writing a native character code 448
 - writing an ASCII character 447
- native substrings 532
- NatPos 251, 252
- NatStr 437
- ND_CHARLANG 197
- ND_CHARNATIVE 196, 197
- NDAr::NDArObj 142
- NDArgs::GetAll 128
- NDArgs::GetExecName 128
- NDArgs::GetFirst 129
- NDArgs::GetNext 129
- NDArgs::GetNth 128
- NDArgs::GetNum 128
- NDArgs::Init 128
- NDArgs::InsertNth 129
- NDArgs::RemoveNth 129
- NDArNum::~~NDArNum 132
- NDArNum::AppendElt 135
- NDArNum::ContainsElt 134
- NDArNum::ExtractElt 136
- NDArNum::ExtractNthElt 136
- NDArNum::FindElt 134
- NDArNum::GetLen 133
- NDArNum::GetNthElt 133
- NDArNum::InsertNthElt 135
- NDArNum::IsEmpty 133
- NDArNum::IsInRange 133
- NDArNum::IsSorted 136
- NDArNum::LookupElt 134
- NDArNum::NDArNum 132
- NDArNum::RemoveDups 136
- NDArNum::RemoveElt 136
- NDArNum::RemoveNthElt 135
- NDArNum::Reset 132

- NDArNum::SetAlloc 133
- NDArNum::SetLen 133
- NDArNum::SetNthElt 134
- NDArNum::Sort 136
- NDArNum::SortedExtractElt 136
- NDArNum::SortedFindElt 135
- NDArNum::SortedInsertElt 135
- NDArNum::SortedLookupElt 134
- NDArNum::SortedRemoveDups 137
- NDArNum::SortedUniqInsertElt 135
- NDArNum::UnboundedGetNthElt 133
- NDArNum::UnboundedSetNthElt 134
- NDArNum::UniqAppendElt 135
- NDArObj::~~NDArObj 142
- NDArObj::AppendElt 144
- NDArObj::ContainsElt 144
- NDArObj::ExtractElt 146
- NDArObj::ExtractNthElt 145
- NDArObj::FindElt 144
- NDArObj::GetLen 143
- NDArObj::GetNthElt 143
- NDArObj::InsertNthElt 145
- NDArObj::IsEmpty 143
- NDArObj::IsInRange 143
- NDArObj::IsSorted 146
- NDArObj::LookupElt 144
- NDArObj::NDArObj 142
- NDArObj::RemoveDups 146
- NDArObj::RemoveElt 145
- NDArObj::RemoveNthElt 145
- NDArObj::SetLen 142
- NDArObj::SetNthElt 143
- NDArObj::Sort 146
- NDArObj::SortedExtractElt 146
- NDArObj::SortedFindElt 144
- NDArObj::SortedInsertElt 145
- NDArObj::SortedLookupElt 144
- NDArObj::SortedRemoveDups 146
- NDArObj::SortedUniqInsertElt 145
- NDArObj::UniqAppendElt 145
- NDArObjOfAROBJ_ELT::Reset 142
- NDArObjOfAROBJ_ELT::SetAlloc 142
- NDArPtr::~~NDArPtr 148
- NDArPtr::AppendElt 151
- NDArPtr::ConstructAlloc 148
- NDArPtr::ConstructArPtr 148
- NDArPtr::ContainsElt 150
- NDArPtr::ExtractElt 152
- NDArPtr::ExtractNthElt 152
- NDArPtr::FindElt 150
- NDArPtr::GetLen 149
- NDArPtr::GetNthElt 149
- NDArPtr::GetNthEltAddr 149
- NDArPtr::InsertNthElt 151

NDArPtr::IsEmpty 149
NDArPtr::IsInRange 149
NDArPtr::IsSorted 153
NDArPtr::LookupElt 150
NDArPtr::NDArPtr 148
NDArPtr::RemoveDups 153
NDArPtr::RemoveElt 152
NDArPtr::RemoveNthElt 152
NDArPtr::Reset 148
NDArPtr::SetAlloc 149
NDArPtr::SetLen 149
NDArPtr::SetNthElt 150
NDArPtr::Sort 152
NDArPtr::SortedExtractElt 152
NDArPtr::SortedFindElt 151
NDArPtr::SortedInsertElt 151
NDArPtr::SortedLookupElt 151
NDArPtr::SortedRemoveDups 153
NDArPtr::SortedUniqInsertElt 152
NDArPtr::UnboundedGetNthElt 150
NDArPtr::UnboundedSetNthElt 150
NDArPtr::UniqAppendElt 151
NDArRec::~~NDArRec 158
NDArRec::AppendElt 160
NDArRec::ContainsElt 159
NDArRec::ExtractElt 161
NDArRec::ExtractNthElt 161
NDArRec::FindElt 160
NDArRec::GetLen 159
NDArRec::GetNthElt 159
NDArRec::InsertNthElt 161
NDArRec::IsEmpty 159
NDArRec::IsInRange 159
NDArRec::IsSorted 162
NDArRec::LookupElt 160
NDArRec::NDArRec 158
NDArRec::RemoveDups 162
NDArRec::RemoveElt 161
NDArRec::RemoveNthElt 161
NDArRec::Reset 158
NDArRec::SetAlloc 158
NDArRec::SetLen 158
NDArRec::SetNthElt 159
NDArRec::Sort 162
NDArRec::SortedExtractElt 162
NDArRec::SortedFindElt 160
NDArRec::SortedInsertElt 161
NDArRec::SortedLookupElt 160
NDArRec::SortedRemoveDups 162
NDArRec::SortedUniqInsertElt 161
NDArRec::UniqAppendElt 160
NDAvlNode::~~NDAvlNode 164
NDAvlNode::GetFirstLeaf 165
NDAvlNode::GetKey 164
NDAvlNode::GetLastLeaf 165
NDAvlNode::GetLeftChild 165
NDAvlNode::GetNext 165
NDAvlNode::GetParent 165
NDAvlNode::GetPrev 165
NDAvlNode::GetRightChild 165
NDAvlNode::NDAvlNode 164
NDAvlNode::SetKey 164
NDAvlTree::~~NDAvlTree 165
NDAvlTree::CurExtractNode 168
NDAvlTree::CurFindKey 168
NDAvlTree::CurFindKeyKey 166
NDAvlTree::CurGetNearestNode 168
NDAvlTree::CurGetNode 168
NDAvlTree::CurInsertNode 168
NDAvlTree::ExtractNode 166
NDAvlTree::GetFirstNode 166
NDAvlTree::GetLastNode 166
NDAvlTree::GetLen 166
NDAvlTree::GoFirstNode 167
NDAvlTree::GoLastNode 167
NDAvlTree::GoNextNode 167
NDAvlTree::GoNode 167
NDAvlTree::GoPrevNode 167
NDAvlTree::InsertNode 166
NDAvlTree::LookupKey 166
NDAvlTree::NDAvlTree 165
NDAvlTree::PerfProc 166
NDAvlTree::PropagateAction 167
NDBBuf::~~NDBBuf 187
NDBBuf::CurPos 188
NDBBuf::Flush 188
NDBBuf::GetClientData 189
NDBBuf::GetCurPtr 190
NDBBuf::GetEndianity 189
NDBBuf::GetPageBeginPos 190
NDBBuf::GetPageBeginPtr 190
NDBBuf::GetPageEndPtr 190
NDBBuf::GetPagingData 189
NDBBuf::GetTotalSize 189
NDBBuf::IsPageModified 190
NDBBuf::LoadCurPage 188
NDBBuf::NDBBuf 186
NDBBuf::QueryMethods 191
NDBBuf::ReadNBytes 187
NDBBuf::SeekBy 188
NDBBuf::SeekTo 188
NDBBuf::SetClientData 189
NDBBuf::SetCurPtr 190
NDBBuf::SetEndianity 189
NDBBuf::SetMethods 191
NDBBuf::SetPageBeginPos 190
NDBBuf::SetPageBeginPtr 190
NDBBuf::SetPageEndPtr 190

-
- NDBBuf::SetPageModified 190
 - NDBBuf::SetPagingData 189
 - NDBBuf::SetTotalSize 189
 - NDBBuf::SkipRead 188
 - NDBBuf::SkipWrite 188
 - NDBBuf::WriteInt16 187
 - NDBBuf::WriteInt32 187
 - NDBBuf::WriteInt8 187
 - NDBBuf::WriteNBytes 187
 - NDBBuf::WriteUInt16 187
 - NDBBuf::WriteUInt32 187
 - NDBBuf::WriteUInt8 187
 - NDBinBuf::~~NDBinBuf 187
 - NDBinBuf::NDBinBuf 186
 - NDBinBuf::ReadInt16 187
 - NDBinBuf::ReadInt32 187
 - NDBinBuf::ReadInt8 187
 - NDBinBuf::ReadUInt16 187
 - NDBinBuf::ReadUInt32 187
 - NDBinBuf::ReadUInt8 187
 - NDChar::AsciiAlphaGetBase 203
 - NDChar::AsciiDigitGetInt 203
 - NDChar::AsciiGetControl 204
 - NDChar::AsciiGetEbedic 204
 - NDChar::AsciiGetGraph 204
 - NDChar::AsciiGetLower 203
 - NDChar::AsciiGetUpper 203
 - NDChar::AsciiHexDigitGetInt 203
 - NDChar::AsciiIsAlNum 202
 - NDChar::AsciiIsAlpha 202
 - NDChar::AsciiIsControl 202
 - NDChar::AsciiIsDigit 202
 - NDChar::AsciiIsGraph 202
 - NDChar::AsciiIsHexDigit 202
 - NDChar::AsciiIsLower 202
 - NDChar::AsciiIsOctDigit 202
 - NDChar::AsciiIsPrint 202
 - NDChar::AsciiIsPunct 202
 - NDChar::AsciiIsSpace 202
 - NDChar::AsciiIsUpper 202
 - NDChar::AsciiOctDigitGetInt 203
 - NDChar::CodeGetLen 199
 - NDChar::FromAscii 205
 - NDChar::GetByte 200
 - NDChar::GetByte1 200
 - NDChar::GetByte2 200
 - NDChar::GetByte3 200
 - NDChar::GetLen 199
 - NDChar::IsAscii 201
 - NDChar::IsAsciiAlNum 201
 - NDChar::IsAsciiAlpha 201
 - NDChar::IsAsciiControl 201
 - NDChar::IsAsciiDigit 201
 - NDChar::IsAsciiGraph 201
 - NDChar::IsAsciiHexDigit 201
 - NDChar::IsAsciiLower 201
 - NDChar::IsAsciiOctDigit 201
 - NDChar::IsAsciiPrint 201
 - NDChar::IsAsciiPunct 201
 - NDChar::IsAsciiSpace 201
 - NDChar::IsAsciiUpper 201
 - NDChar::NatGetByte 200
 - NDChar::NatGetByte1 200
 - NDChar::NatGetByte2 200
 - NDChar::NatGetByte3 200
 - NDChar::NatGetLen 199
 - NDChar::ToAscii 205
 - NDChar::EbedicGetAscii 205
 - NDCs::~~NDCs 211
 - NDCs::CvtChar 212
 - NDCs::FromUni 213
 - NDCs::GetCharInfo 211
 - NDCs::GetCharLen 211
 - NDCs::GetCsGlobal 213
 - NDCs::GetCsId 211
 - NDCs::GetCsNative 213
 - NDCs::GetCsUnicode 213
 - NDCs::NDCs 211
 - NDCs::ToUni 213
 - NDCs::TransChar 212
 - NDct::~~NDct 219
 - NDct::CvtChar 220
 - NDct::CvtCsToCt 221
 - NDct::CvtCtToCs 221
 - NDct::FromUni 221
 - NDct::GetBwrld 220
 - NDct::GetCtId 219
 - NDct::GetFwrld 220
 - NDct::GetInfo 220
 - NDct::GetLower 222
 - NDct::GetMaxCharLen 222
 - NDct::GetUpper 222
 - NDct::IsSingleOnly 222
 - NDct::NDct 219
 - NDct::ToUni 221
 - NDD interface 520
 - NDDGram view 59
 - options 80
 - origins 61
 - NDDs::~~NDDs 226
 - NDDs::AddContDs 225
 - NDDs::Class 223
 - NDDs::GetViewOption 223
 - NDDs::NDDs 226
 - NDDs::RegisterView 223
 - NDDs::RemoveContDs 225
 - NDDs::SetViewOption 223
 - NDDs::StartEdit 224

NDDs::StartUpdateEdit 225
NDDs::UnregisterView 223
NDDs::ViewGetDs 224
NDDsEdit::Abort 224
NDDsEdit::AddOperation 224
NDDsEdit::End 224
NDDsEdit::GetOwner 225
NDDsEdit::SetOwner 224
NDDsUpdateEdit::Abort 225
NDDsUpdateEdit::End 225
NDErr::Exit 241
NDErr::Fail 241
NDErr::FailAssert 241
NDErr::FailError 242
NDErr::FailSilent 241
NDErr::FailStr 241
NDErr::Fatal 241
NDErr::Format 240
NDErr::FrameDefReport 238
NDErr::FrameGetTop 237
NDErr::FrameIsReported 238
NDErr::FrameQueryFullTraceback 238
NDErr::FrameQueryMessage 237
NDErr::FrameQueryTraceback 238
NDErr::FrameReport 238
NDErr::FrameSetReported 238
NDErr::FrameTraceBack 243
NDErr::GetErrFuncCallPtr 237
NDErr::GetReportProc 238
NDErr::GetSysExceptionHandler 244
NDErr::InError 242
NDErr::LoadMsg 240
NDErr::ModExit 243
NDErr::MswIsInterruptRegistered 245
NDErr::MswRegisterInterrupt 245
NDErr::MswRegisterInterruptOnInit 245
NDErr::NoMacSignals 245
NDErr::Print 240
NDErr::SetReportPrint 240
NDErr::SetReportProc 238
NDErr::SetReportSilent 240
NDErr::SetSysExceptionHandler 244
NDErr::SysException 244
NDErr::SysExceptProc 244
NDErr::TraceBack 243
NDErr::Warn 242
NDErr::WarnStr 242
NDFile::Backup 269
NDFile::Close 260
NDFile::CreateOpen 260
NDFile::CurBinaryOffset 263
NDFile::CurLineNumber 264
NDFile::CurSize 262
NDFile::CurTextOffset 263
NDFile::Find 258
NDFile::Flush 269
NDFile::GetAutoBackup 256
NDFile::GetClientDat 258
NDFile::GetDefSearchPath 270
NDFile::GetDefSearchPathName 271
NDFile::GetError 272
NDFile::GetFailIfNotFound 257
NDFile::GetFailOnEof 258
NDFile::GetNodeType 259
NDFile::GetOpenFormat 262
NDFile::GetOpenMode 262
NDFile::GetRealName 255
NDFile::GetSearchPath 256
NDFile::GetSpecName 255
NDFile::GotoBeg 262
NDFile::GotoEnd 262
NDFile::IsAtEnd 262
NDFile::IsOpen 261
NDFile::IsOpenLine 261
NDFile::IsOpenRead 261
NDFile::IsOpenText 261
NDFile::IsOpenWrite 261
NDFile::IsReadable 259
NDFile::IsWritable 259
NDFile::Open 259
NDFile::QueryLinePos 264
NDFile::QueryNatRef 271
NDFile::QueryTextPos 263
NDFile::ReadByte 265
NDFile::ReadChar 266
NDFile::ReadLine 269
NDFile::ReadNBytes 265
NDFile::ReadNChars 267
NDFile::ReadStr 267
NDFile::ReadTextLine 268
NDFile::SeekBinaryBy 263
NDFile::SeekBinaryTo 263
NDFile::SetAutoBackup 257
NDFile::SetClientData 258
NDFile::SetDefSearchPath 270
NDFile::SetDefSearchPathName 271
NDFile::SetError 272
NDFile::SetFailIfNotFound 257
NDFile::SetFailOnEof 257
NDFile::SetLinePos 264
NDFile::SetNatRef 271
NDFile::SetSearchPat 256
NDFile::SetSpecName 255
NDFile::SetTextPos 264
NDFile::Truncate 270
NDFile::TryClose 260
NDFile::TryCreateOpen 260
NDFile::TryOpen 260

NDFFile::WriteByte 265
NDFFile::WriteChar 266
NDFFile::WriteLine 269
NDFFile::WriteNBytes 266
NDFFile::WriteNChars 267
NDFFile::WriteStr 268
NDFFile::WriteTextLine 268
NDFMgr::AddFileType 285
NDFMgr::AllFilesWildcard 292
NDFMgr::CheckDir 283
NDFMgr::CheckFile 283
NDFMgr::CopyDir 287
NDFMgr::CopyFile 287
NDFMgr::CopyNode 287
NDFMgr::CreateDir 286
NDFMgr::CreateFile 286
NDFMgr::DeleteDir 289
NDFMgr::DeleteDirContent 289
NDFMgr::DeleteFile 289
NDFMgr::DeleteNode 289
NDFMgr::DirWildcard 292
NDFMgr::Exists 281
NDFMgr::FindFileTypeId 286
NDFMgr::FindFileTypeInfo 286
NDFMgr::GetMacCreator 282
NDFMgr::GetMacType 282
NDFMgr::GetNodeType 282
NDFMgr::GetNthFileType 285
NDFMgr::GetNumFileTypes 285
NDFMgr::IsDevConcealed 281
NDFMgr::IsDir 283
NDFMgr::IsExecutable 281
NDFMgr::IsFile 283
NDFMgr::IsReadable 281
NDFMgr::IsVolume 283
NDFMgr::IsWritable 281
NDFMgr::MoveDir 288
NDFMgr::MoveFile 288
NDFMgr::MoveNode 288
NDFMgr::PerfDirFiles 291
NDFMgr::PerfVolumes 293
NDFMgr::PurgeDir 291
NDFMgr::QueryNodeInfo 282
NDFMgr::RemoveFileType 285
NDFMgr::TryCopyDir 290
NDFMgr::TryCopyFile 290
NDFMgr::TryCopyNode 290
NDFMgr::TryCreateDir 290
NDFMgr::TryCreateFile 290
NDFMgr::TryDeleteDir 290
NDFMgr::TryDeleteDirContent 290
NDFMgr::TryDeleteFile 290
NDFMgr::TryDeleteNode 290
NDFMgr::TryMoveDir 290
NDFMgr::TryMoveFile 290
NDFMgr::TryMoveNode 290
NDFMgrFileExt 285
NDFMgrFileType 285
NDFName::Cmp 316
NDFName::Convert 307
NDFName::ConvertFromTo 307
NDFName::ConvertInPlace 308
NDFName::CurDirStr 314
NDFName::CvtDirFileToPath 311
NDFName::CvtDirPathToFile 310
NDFName::CvtToAbsolute 316
NDFName::DirQueryParent 315
NDFName::Equal 317
NDFName::Evaluate 307
NDFName::EvaluateIn 306
NDFName::FindSyntax 305
NDFName::GetCompSet 309
NDFName::GetCurSyntax 304
NDFName::GetStatus 308
NDFName::GetSysSyntax 304
NDFName::GetTmpPath 317
NDFName::HomeDirStr 315
NDFName::IsAbsolute 316
NDFName::IsDirAsFile 310
NDFName::IsPortable 308
NDFName::IsTopDir 313, 316
NDFName::IsValid 306
NDFName::IsValidIn 306
NDFName::MakeBackupName 317
NDFName::MakeTmpFileName 317
NDFName::MakeValid 306
NDFName::MakeValidIn 306
NDFName::MergeFile 311
NDFName::MergePath 312
NDFName::ParentDirStr 314
NDFName::QueryComps 310
NDFName::QueryCurDir 313
NDFName::QueryCurParams 304
NDFName::QueryCurVolume 313
NDFName::QueryHomeDir 315
NDFName::QueryParentDir 314, 316
NDFName::QueryTopDir 312, 315
NDFName::ReduceComp 310
NDFName::Reset 305
NDFName::SetCurDir 313
NDFName::SetCurParams 304
NDFName::SetCurSyntax 304
NDFName::SetStatus 309
NDFName::SetTmpPath 317
NDFName::SplitFile 311
NDFName::SplitPath 311
NDFName::StatusGetMsg 309
NDFName::StxGetName 304

NDFName::SysTmpPath 317
 NDFName::TopDirStr 312
 NDFName::VolumeQueryCurDir 314
 NDFName::VolumeSetCurDir 314
 NDHash::~~NDHash 320
 NDHash::AddGetEntry 323
 NDHash::CompareProc 320
 NDHash::DataCloneProc 321
 NDHash::DataDisposeProc 321
 NDHash::DefCompareInt 323
 NDHash::DefCompareIStr 323
 NDHash::DefComparePtr 323
 NDHash::DefCompareStr 323
 NDHash::DefHashInt 323
 NDHash::DefHashIStr 323
 NDHash::DefHashPtr 323
 NDHash::DefHashStr 323
 NDHash::DefStrKeyClone 323
 NDHash::DefStrKeyDispose 323
 NDHash::EntryGetKey 324
 NDHash::EntryGetValue 324
 NDHash::EntrySetValue 324
 NDHash::Extract 322
 NDHash::GetDefIntInfo 321
 NDHash::GetDefIStrInfo 322
 NDHash::GetDefPtrInfo 321
 NDHash::GetDefStrInfo 322
 NDHash::GetDefStrKeyClonedInfo 322
 NDHash::GetEntry 324
 NDHash::HashProc 321
 NDHash::Insert 322
 NDHash::InsertGetEntry 324
 NDHash::KeyCloneProc 321
 NDHash::KeyDisposeProc 321
 NDHash::Lookup 322
 NDHash::NDHash 320
 NDHash::Perf 323
 NDHash::QueryDefInfo 321
 NDHash::QueryInfo 322
 NDHash::QueryStats 324
 NDHash::RemoveEntry 324
 NDHash::Reset 320
 NDHashInfo 319
 NDHashStatsInfo 324
 NDHeap::~~NDHeap 325
 NDHeap::Add 326
 NDHeap::Correct 326
 NDHeap::GetSize 325
 NDHeap::Insert 326
 NDHeap::NDHeap 325
 NDHeap::Perf 326
 NDHeap::QueryFirst 326
 NDISet::~~NDISet 328
 NDISet::AddIntervals 328
 NDISet::ContainsElt 329
 NDISet::ContainsIntervals 329
 NDISet::GetMaxElt 329
 NDISet::GetMinElt 328
 NDISet::GetNumIntervals 328
 NDISet::IsAll 328
 NDISet::MixGetPartSet 329
 NDISet::MixQueryParts 329
 NDISet::NDISet 327
 NDISet::QueryComplement 329
 NDISet::QueryIntervals 328
 NDISet::RemoveIntervals 328
 NDISet::SetIntervals 328
 NDISet::UniversalSet 328
 NDISetInterval 327
 NDNfier::~~NDNfier 340
 NDNfier::Broadcast 340
 NDNfier::ClientGetClientData 341
 NDNfier::ClientSetClientData 341
 NDNfier::NDNfier 340
 NDNfier::RegisterNfierClient 341
 NDNfier::UnregisterNfierClient 341
 NDNfierClient::~~NDNfierClient 340
 NDNfierClient::NDNfierClient 340
 NDPack::~~NDPack 344
 NDPack::CcittDecode 347
 NDPack::CcittEncode 347
 NDPack::Decode 348
 NDPack::Encode 348
 NDPack::LzwDecode 347
 NDPack::LzwEncode 347
 NDPack::NDPack 344
 NDPack::PkbDecode 346
 NDPack::PkbEncode 346
 NDPack::RleDecode 346
 NDPack::RleEncode 346
 NDPFIId 350
 NDPoint16::~~NDPoint16 351
 NDPoint16::AbsDist 353, 378
 NDPoint16::Equals 353
 NDPoint16::GetX 352
 NDPoint16::GetY 352
 NDPoint16::IncXY 352, 379
 NDPoint16::IsInRectExt 353
 NDPoint16::IsNull 352
 NDPoint16::NDPoint16 351
 NDPoint16::Reset 352
 NDPoint16::SetSameXY 352
 NDPoint16::SetX 352
 NDPoint16::SetXY 352, 379
 NDPoint16::SetY 352
 NDPoint32::~~NDPoint32 351
 NDPoint32::AbsDist 353, 378
 NDPoint32::Equals 353

NDPoint32::GetX 352
NDPoint32::GetY 352
NDPoint32::IncXY 352, 379
NDPoint32::IsInRectExt 353
NDPoint32::IsNull 352
NDPoint32::NDPoint32 351
NDPoint32::Reset 352
NDPoint32::SetSameXY 352
NDPoint32::SetX 352
NDPoint32::SetXY 352, 379
NDPoint32::SetY 352
NDPool::~~NDPool 357
NDPool::DisposePtr 357
NDPool::NDPool 356, 357
NDPool::NewPtr 357
NDPool::QueryInfo 357
NDPool::QueryStats 358
NDPool::ResetStats 358
NDPool::SetInfo 357
NDPoolFragStatsInfo 358
NDPoolStatsInfo 358
NDPtr::Copy 364
NDPtr::AlignCheck 361
NDPtr::Clear 363
NDPtr::Cmp 364
NDPtr::CopyByte 366
NDPtr::DefFailProc 369
NDPtr::Dispose 362
NDPtr::GetAlignedSize 361
NDPtr::GetByte 365
NDPtr::GetFailProc 369
NDPtr::GetSize 362
NDPtr::HugeClear 365
NDPtr::HugeCmp 365
NDPtr::HugeCopy 365
NDPtr::HugeCopyByte 366
NDPtr::HugeDispose 363
NDPtr::HugeGetByte 366
NDPtr::HugeGetSize 363
NDPtr::HugeMatches 365
NDPtr::HugeMove 365
NDPtr::HugeNew 363
NDPtr::HugeSet 365
NDPtr::HugeSetByte 366
NDPtr::HugeSetSize 363, 365
NDPtr::HugeSwap 365
NDPtr::HugeSwapByte 366
NDPtr::Int16ToMch 367
NDPtr::Int16ToStd 366
NDPtr::Int32ToMch 367
NDPtr::Int32ToStd 366
NDPtr::Int8ToMch 367
NDPtr::Int8ToStd 366
NDPtr::Matches 364
NDPtr::Move 364
NDPtr::New 362
NDPtr::QueryStats 365
NDPtr::ReadInt16 367
NDPtr::ReadInt32 367
NDPtr::ReadInt8 367
NDPtr::ReadStr 368
NDPtr::Set 363
NDPtr::SetByte 366
NDPtr::SetFailProc 369
NDPtr::SetSize 362, 363
NDPtr::StatsOutput 365
NDPtr::StrToMch 368
NDPtr::StrToStd 368
NDPtr::Swap 364
NDPtr::SwapByte 366
NDPtr::WriteInt16 367
NDPtr::WriteInt32 367
NDPtr::WriteInt8 367
NDPtr::WriteStr 368
NDRClas::CPlusRegister 372
NDRClas::FindByName 373
NDRClas::GetDefNfy 373
NDRClas::GetFields 373
NDRClas::GetFirst 374
NDRClas::GetFlags 373
NDRClas::GetModName 373
NDRClas::GetName 373
NDRClas::GetNext 374
NDRClas::GetParentClass 373
NDRClas::GetSizeOfRes 373
NDRClas::GetTemplate 373
NDRClas::GetVersion 373
NDRClas::IsSubClassOf 374
NDRClas::OperatorDelete 373
NDRClas::OperatorNew 372
NDRClas::ProcessDefNfy 375
NDRClas::ProcessParentDefNfy 375
NDRClas::SetDefNfy 375
NDRClas::SetFields 374
NDRClas::SetFlags 374
NDRClas::SetModName 374
NDRClas::SetName 374
NDRClas::SetParentClass 374
NDRClas::SetSizeOfRes 374
NDRClas::SetVersion 374
NDRect16::ContainsPoint 378
NDRect16::Copy 380
NDRect16::CopyResetOri 380
NDRect16::Equals 379
NDRect16::GetExtX 381, 382
NDRect16::GetExtY 381, 382
NDRect16::GetOriX 381, 382
NDRect16::GetOriY 381, 382

NDRect16::IncludesNonEmptyRect 380
NDRect16::IncludesRect 380
NDRect16::IncOriExtXY 378
NDRect16::Intersects 380
NDRect16::IsEmpty 379
NDRect16::IsValid 381
NDRect16::MakeFit 381
NDRect16::MakeValid 381
NDRect16::MoveInside 381
NDRect16::Reset 379
NDRect16::SetByPoints 379
NDRect16::SetExtX 382
NDRect16::SetExtY 382
NDRect16::SetOriExtXY 378
NDRect16::SetOriX 382
NDRect16::SetOriY 382
NDRect16::Union 380
NDRect32::ContainsPoint 378
NDRect32::Copy 380
NDRect32::CopyResetOri 380
NDRect32::Equals 379
NDRect32::GetExtX 381, 382
NDRect32::GetExtY 381, 382
NDRect32::GetOriX 381, 382
NDRect32::GetOriY 381, 382
NDRect32::IncludesNonEmptyRect 380
NDRect32::IncludesRect 380
NDRect32::IncOriExtXY 378
NDRect32::Intersection 380
NDRect32::Intersects 380
NDRect32::IsEmpty 379
NDRect32::IsValid 381
NDRect32::MakeFit 381
NDRect32::MakeValid 381
NDRect32::MoveInside 381
NDRect32::Reset 379
NDRect32::SetByPoints 379
NDRect32::SetExtX 382
NDRect32::SetExtY 382
NDRect32::SetOriExtXY 378
NDRect32::SetOriX 382
NDRect32::SetOriY 382
NDRect32::Union 380
NDRegion::IsEmpty 406
NDRegion::IsEqual 407
NDRegion::IsPointInside 407
NDRegion::NDRgn 409
NDRegion::PropagateAction 410
NDRegion::QueryBounds 406
NDRegion::RectIntersect 409
NDRegion::RectPos 407
NDRegion::RectSet 408
NDRegion::RectSubtrac 409
NDRegion::RectUnion 409
NDRegion::RectXOr 409
NDRegion::Reset 406
NDRegion::RgnIntersect 407
NDRegion::RgnSet 407
NDRegion::RgnSubtract 408
NDRegion::RgnUnion 408
NDRegion::RgnXOr 408
NDRes::CheckClass 404
NDRes::Class 388
NDRes::ClassDefNfy 402
NDRes::Clone 388
NDRes::CmdIssue 403
NDRes::CmdSend 403
NDRes::CmdTableHandle 404
NDRes::CmdUpdate 404
NDRes::DefNfy 398
NDRes::ExecuteScript 404
NDRes::FilenameOutputRc 389
NDRes::Find 394
NDRes::FindByFullName 393
NDRes::GetClass 396
NDRes::GetClientData 395
NDRes::GetName 394
NDRes::GetNfyCmd 403
NDRes::GetNfyData 401
NDRes::GetNfyHandlerClientData 399
NDRes::GetNfyHandlerProc 398
NDRes::GetNfyProc 398
NDRes::GetNthChild 395
NDRes::GetNumChildren 395
NDRes::InheritsFrom 396
NDRes::IsCmdSource 403
NDRes::IsInitialized 396
NDRes::IsNamed 394
NDRes::LibExit 390
NDRes::LibInit 390
NDRes::LibInstall 389
NDRes::LibLoadInit 390
NDRes::Load 391
NDRes::LoadByFullName 391
NDRes::LoadChildren 393
NDRes::LoadDetach 392
NDRes::LoadInit 392
NDRes::LoadInitDetach 393
NDRes::LockedSendNfyData 400
NDRes::NfyCtrlData 397
NDRes::NfyDeallocate 397
NDRes::NfyDestructed 397
NDRes::NfyEnd 397
NDRes::NfyGetData 397
NDRes::NfyInit 396
NDRes::NfyReset 397
NDRes::NfyResMgr 397
NDRes::NfySetData 397

NDRes::ParentClassDefNfy 402
NDRes::QueryFullName 394
NDRes::Release 388
NDRes::RemoveNfyHandler 399
NDRes::SaveDat 389
NDRes::SendCtrlNfyData 402
NDRes::SendNfy 400
NDRes::SendNfyData 400
NDRes::SendNfyEnd 401
NDRes::SendNfyInit 401
NDRes::SendNfyReset 401
NDRes::SetClientData 395
NDRes::SetNfyHandler 398
NDRes::SetNfyHandlerClientData 399
NDRes::SetNfyProc 398
NDRes::Use 388
NDRes::VERIFY 404
NDRgn::Translate 406
NDRLib::Close 413
NDRLib::Dispose 413
NDRLib::Find 411
NDRLib::GetFirst 412
NDRLib::GetLibName 411
NDRLib::GetNext 412
NDRLib::LoadEdit 412
NDRLib::LoadFile 412
NDRLib::LoadLibFile 413
NDRLib::Open 413
NDRLib::Unload 413
NDSBuf::AppendChar 417
NDSBuf::AppendSBuf 417
NDSBuf::AppendStr 417
NDSBuf::AppendStrSub 417
NDSBuf::AppendVStr 417
NDSBuf::Clear 417
NDSBuf::CountToIndex 416
NDSBuf::DownCase 419
NDSBuf::DownCaseSub 419
NDSBuf::GetBwrD 416
NDSBuf::GetByte 416
NDSBuf::GetFwrD 416
NDSBuf::GetLen 415
NDSBuf::GetStr 415
NDSBuf::GetSubStr 415
NDSBuf::IMatchesChar 420
NDSBuf::IMatchesSBuf 420
NDSBuf::IMatchesStr 420
NDSBuf::IMatchesStrSub 420
NDSBuf::IndexToCount 416
NDSBuf::InsertChar 417
NDSBuf::InsertSBuf 417
NDSBuf::InsertStr 417
NDSBuf::InsertStrSub 417
NDSBuf::InsertVStr 417
NDSBuf::MatchesChar 419
NDSBuf::MatchesIChar 419
NDSBuf::MatchesISBuf 419
NDSBuf::MatchesIStr 419
NDSBuf::MatchesIStrSub 419
NDSBuf::MatchesSBuf 419
NDSBuf::MatchesStr 419
NDSBuf::MatchesStrSub 419
NDSBuf::RemoveChar 418
NDSBuf::RemoveRange 418
NDSBuf::ReplaceChar 418
NDSBuf::SetSBuf 417
NDSBuf::SetStr 417
NDSBuf::SetStrSub 417
NDSBuf::SetVStr 417
NDSBuf::Truncate 418
NDSBuf::UpCase 418
NDSBuf::UpCaseSub 418
NDScripT::Compile 429
NDScripT::CompileFile 429
NDScripT::CompileResource 429
NDScripT::Dispose 430
NDScripT::Execute 429
NDScripT::ExecuteApp 428
NDScripT::GetReturnTpe 429
NDScripT::LibExit 423
NDScripT::LibInit 423
NDScripT::LibInstall 423
NDScripT::LibLoadInit 423
NDScripT::QueryReturnVlue 430
NDScripT::RegisterConstants 424
NDScripT::RegisterEvents 425
NDScripT::RegisterVerbs 427
NDScripT::RunApp 428
NDScripT::SetStringReturnVlue 427
NDScripTRegisterEvent 425
NDScripTRegisterVerb 427
NDSet::~~NDSet 431
NDSet::AddElt 431
NDSet::AddElts 432
NDSet::AreEqual 433
NDSet::ContainsElt 432
NDSet::Copy 432
NDSet::EmptySet 431
NDSet::GetNumElts 432
NDSet::MixGetPartSet 433
NDSet::MixQueryParts 433
NDSet::NDSet 431
NDSet::QueryElts 432
NDSet::RemoveElt 432
NDSet::RemoveElts 432
NDSet::Reset 432
NDSet::SetElts 432
NDStr::Append 441

NDStr::AppendSub 442
NDStr::AsciiDownCase 460
NDStr::AsciiDownCaseSub 460
NDStr::AsciiUpCase 459
NDStr::AsciiUpCaseSub 460
NDStr::Clone 439
NDStr::Cmp 448
NDStr::CmpSub 449
NDStr::CtGetBwrD 444
NDStr::CtGetCode 443
NDStr::CtGetFwrD 444
NDStr::Dispose 440
NDStr::Dispose0 440
NDStr::Equals 449
NDStr::EqualsSub 449
NDStr::FindFirst 451
NDStr::FindFirstCha 452
NDStr::FindFirstCharSub 452
NDStr::FindFirstSub(452
NDStr::FindIFirst 452
NDStr::FindIFirstSub 453
NDStr::FindILast 453
NDStr::FindILastSub 454
NDStr::FindLast 453
NDStr::FindLastChar 453
NDStr::FindLastCharSub 454
NDStr::FindLastSub 454
NDStr::FromCt 462
NDStr::FromCtSub 462
NDStr::FromUni 463
NDStr::FromUniSub 463
NDStr::GetBwrD 443
NDStr::GetCode 443
NDStr::GetDecInt 454
NDStr::GetDecInt16 454
NDStr::GetDecInt32 454
NDStr::GetDecUInt 454
NDStr::GetDecUInt16(454
NDStr::GetDecUInt32 454
NDStr::GetFwrD 443
NDStr::GetHexInt 455
NDStr::GetHexInt16 455
NDStr::GetHexInt32 455
NDStr::GetHexUInt 455
NDStr::GetHexUInt16 455
NDStr::GetHexUInt32 455
NDStr::GetLen 442
NDStr::GetRadixInt 455
NDStr::GetRadixInt16 455
NDStr::GetRadixInt32 455
NDStr::GetRadixUInt 455
NDStr::GetRadixUInt16 455
NDStr::GetRadixUInt32 455
NDStr::GetTruncLen(442
NDStr::ICmp 448
NDStr::ICmpSub 449
NDStr::IEquals 449
NDStr::IEqualsSub 449
NDStr::IFindFirst 451
NDStr::IFindFirstSub 452
NDStr::IFindLast 453
NDStr::IMatches 450
NDStr::IMatchesPa 450
NDStr::IMatchesPatSub 451
NDStr::IMatchesSub 451
NDStr::Matches 450
NDStr::MatchesChar 450
NDStr::MatchesPat 450
NDStr::MatchesPatSub 451
NDStr::MatchesSub 451
NDStr::NatPutAscii 447
NDStr::NatPutCode 447
NDStr::NatWriteAscii 447
NDStr::NatWriteCod 448
NDStr::NewSet 439
NDStr::NewSetSub 440
NDStr::Put 445
NDStr::PutAscii 446
NDStr::PutAsciiLower 461
NDStr::PutAsciiLowerSu 461
NDStr::PutAsciiUppE 460
NDStr::PutAsciiUpperSub 461
NDStr::PutCode 446
NDStr::PutDecInt 458
NDStr::PutDecInt16 458
NDStr::PutDecInt32 458
NDStr::PutDecUInt 458
NDStr::PutDecUInt16(458
NDStr::PutDecUInt32 458
NDStr::PutDouble 459
NDStr::PutHexInt 458
NDStr::PutHexInt16 458
NDStr::PutHexInt32 458
NDStr::PutHexUInt 458
NDStr::PutHexUInt16(458
NDStr::PutHexUInt32 458
NDStr::PutRadixIn 459
NDStr::PutRadixInt16 459
NDStr::PutRadixInt32 459
NDStr::PutRadixUInt 459
NDStr::PutRadixUInt16 459
NDStr::PutRadixUInt32 459
NDStr::PutSub 445
NDStr::ResFind 462
NDStr::ResFindNth 462
NDStr::ResLoad 461
NDStr::ResLoadNth 461
NDStr::Set 441

NDStr::SetSub 441
NDStr::SubGetDecInt 456
NDStr::SubGetDecInt16 456
NDStr::SubGetDecInt32 456
NDStr::SubGetDecUInt 456
NDStr::SubGetDecUInt16 456
NDStr::SubGetDecUInt32 456
NDStr::SubGetDouble 458
NDStr::SubGetHexInt 457
NDStr::SubGetHexInt16(457
NDStr::SubGetHexInt32 457
NDStr::SubGetHexUInt 457
NDStr::SubGetHexUInt16(457
NDStr::SubGetHexUInt32 457
NDStr::SubGetRadixInt 457
NDStr::SubGetRadixInt16 457
NDStr::SubGetRadixInt32 457
NDStr::SubGetRadixUInt 457
NDStr::SubGetRadixUInt16 457
NDStr::SubGetRadixUInt32 457
NDStr::ToCt 462
NDStr::ToCtSub 462
NDStr::ToUni 463
NDStr::ToUniSub 463
NDStr::WriteAscii 446
NDStr::WriteCode 447
NDStr::GetDouble 456
NDStrL::AddStr 466
NDStrL::AddStrAtIndex 466
NDStrL::Class 465
NDStrL::FindNthStr 467
NDStrL::GetLen 465
NDStrL::GetNthStr 466
NDStrL::LoadNthSt 467
NDStrL::RemoveIndex 467
NDStrL::SetNthStr 466
NDStrR::Class 469
NDStrR::FindStr 470
NDStrR::GetId 470
NDStrR::GetStr 470
NDStrR::LoadStr 470
NDStrR::SetId 470
NDStrR::SetStr 470
NDVar::~NDVar 473
NDVar::Clear 477
NDVar::ContainsRef 477
NDVar::Convert 474
NDVar::ConvertToValue 474
NDVar::CopyToBoolean 476
NDVar::CopyToByte 476
NDVar::CopyToChar 476
NDVar::CopyToCharCode 476
NDVar::CopyToClientPtr 477
NDVar::CopyToDouble 476
NDVar::CopyToFloat 476
NDVar::CopyToInt 475
NDVar::CopyToInt16 475
NDVar::CopyToInt32 475
NDVar::CopyToInt64 475
NDVar::CopyToInt8 475
NDVar::CopyToLong 475
NDVar::CopyToStr 476
NDVar::CopyToType 474
NDVar::CopyToUInt 475
NDVar::CopyToUInt16 475
NDVar::CopyToUInt32 475
NDVar::CopyToUInt64 475
NDVar::CopyToUInt8 475
NDVar::CopyToULong 475
NDVar::CopyToValue 474
NDVar::CopyToVARWChar 476
NDVar::CopyToVARWStr 476
NDVar::GetType 477
NDVar::InitClass 473
NDVar::IsEmpty 477
NDVar::IsNull 477
NDVar::IsNullObj 477
NDVar::NDVar 473
NDVar::operator= 474
NDVar::TryConvert 474
NDVar::TryConvertToValue 474
NDVar::TryCopyToBoolean 476
NDVar::TryCopyToByte 476
NDVar::TryCopyToChar 476
NDVar::TryCopyToCharCode 476
NDVar::TryCopyToClientPtr 477
NDVar::TryCopyToDouble 476
NDVar::TryCopyToFloat 476
NDVar::TryCopyToInt 475
NDVar::TryCopyToInt16 475
NDVar::TryCopyToInt32 475
NDVar::TryCopyToInt64 475
NDVar::TryCopyToInt8 475
NDVar::TryCopyToLong 475
NDVar::TryCopyToStr 476
NDVar::TryCopyToType 474
NDVar::TryCopyToUInt 475
NDVar::TryCopyToUInt16 475
NDVar::TryCopyToUInt32 475
NDVar::TryCopyToUInt64 475
NDVar::TryCopyToUInt8 475
NDVar::TryCopyToULong 475
NDVar::TryCopyToValue 474
NDVar::TryCopyToVARWChar 476
NDVar::TryCopyToVARWStr 476
NDVar::TryUpdate 477
NDVar::UnloadClass 473
NDVar::Update 477

NDVarDs::DefNfy 479
NDVarDs::GetValue 479
NDVarDs::QueryValue 479
NDVarDs::SetValue 479
NDVarDsEdit::SetValue 479
NDVarGr class 59
NDVarGr::AddDirectedEdge 489
NDVarGr::AddNode 488
NDVarGr::AddUndirectedEdge 489
NDVarGr::AreEdgesEqual 495
NDVarGr::AreNodesEqual 490
NDVarGr::Class 481
NDVarGr::GetEdge 500
NDVarGr::GetEdgeCursor 499
NDVarGr::GetEdgeId 496
NDVarGr::GetEdgeIsDirected 497
NDVarGr::GetEdgeNumEdges 495
NDVarGr::GetEdgeProperty 497
NDVarGr::GetEdgeValue 496
NDVarGr::GetMods 507
NDVarGr::GetNode 500
NDVarGr::GetNodeCursor 498
NDVarGr::GetNodeHeight 493
NDVarGr::GetNodeId 491
NDVarGr::GetNodeNumChildren 490
NDVarGr::GetNodeNumNeighbors 490
NDVarGr::GetNodeNumParents 490
NDVarGr::GetNodeProperty 494
NDVarGr::GetNodeValue 491
NDVarGr::GetNodeWidth 494
NDVarGr::GetNodeXOrigin 492
NDVarGr::GetNodeYOrigin 492
NDVarGr::GetNumEdges 482
NDVarGr::GetNumNodes 482
NDVarGr::GetNumRootNodes 482
NDVarGr::GetTitle 481
NDVarGr::IsChildNode 498
NDVarGr::IsEdgeValid 495
NDVarGr::IsNeighborNode 498
NDVarGr::IsNodeValid 490
NDVarGr::IsParentNode 498
NDVarGr::QueryCyclicResult 499
NDVarGr::QueryEdgeId 495
NDVarGr::QueryEdgeProperty 497
NDVarGr::QueryEdgeValue 496
NDVarGr::QueryNodeHeight 493
NDVarGr::QueryNodeId 491
NDVarGr::QueryNodeProperty 494
NDVarGr::QueryNodeValue 491
NDVarGr::QueryNodeWidth 493
NDVarGr::QueryNodeXOrigin 492
NDVarGr::QueryNodeYOrigin 492
NDVarGr::RemoveEdge 489
NDVarGr::RemoveEdgeBetween 489
NDVarGr::RemoveEdgeProperty 497
NDVarGr::RemoveNodeProperty 494
NDVarGr::SetEdgeCursor 499
NDVarGr::SetEdgeId 496
NDVarGr::SetEdgesDirected 497
NDVarGr::SetEdgeProperty 497
NDVarGr::SetEdgeValue 496
NDVarGr::SetNodeCursor 498
NDVarGr::SetNodeHeight 493
NDVarGr::SetNodeId 491
NDVarGr::SetNodeProperty 494
NDVarGr::SetNodeValue 491
NDVarGr::SetNodeWidth 494
NDVarGr::SetNodeXOrigin 492
NDVarGr::SetNodeYOrigin 492
NDVarGr::SetTitle 481
NDVarGr::StartEdgeEdit 499
NDVarGr::StartNodeEdit 499
NDVarGrAllEdgeAccessor::Go Between 486
NDVarGrAllEdgeAccessor::Go Previous 486
NDVarGrAllEdgeAccessor::GoFirst 485
NDVarGrAllEdgeAccessor::Gold 486
NDVarGrAllEdgeAccessor::GoIndexed 486
NDVarGrAllEdgeAccessor::GoNext 485
NDVarGrAllEdgeAccessor::NDVarGrAllEdge-
Accessor 482
NDVarGrEdge::GetFromNode 504
NDVarGrEdge::GetId 505
NDVarGrEdge::GetIsDirected 506
NDVarGrEdge::GetProperty 506
NDVarGrEdge::GetToNode 504
NDVarGrEdge::GetValue 505
NDVarGrEdge::QueryValue 505
NDVarGrEdge::RemoveProperty 506
NDVarGrEdge::SetId 505
NDVarGrEdge::SetIsDirected 506
NDVarGrEdge::SetProperty 506
NDVarGrEdge::SetValue 505
NDVarGrEdge::StartEdit 507
NDVarGrEdgeEdit::RemoveProperty 506
NDVarGrEdgeEdit::SetId 505
NDVarGrEdgeEdit::SetIsDirected 506
NDVarGrEdgeEdit::SetProperty 506
NDVarGrEdgeEdit::SetValue 505
NDVarGrEdit::AddDirectedEdge 489
NDVarGrEdit::AddNode 488
NDVarGrEdit::AddUndirectedEdge 489
NDVarGrEdit::RemoveEdge 489
NDVarGrEdit::RemoveEdgeBetween 490
NDVarGrEdit::RemoveEdgeProperty 498
NDVarGrEdit::RemoveNode 488
NDVarGrEdit::RemoveNodeProperty 495
NDVarGrEdit::SetEdgeCursor 499
NDVarGrEdit::SetEdgeId 496

- NDVarGrEdit::SetEdgeIsDirected 497
- NDVarGrEdit::SetEdgeProperty 497
- NDVarGrEdit::SetEdgeValue 496
- NDVarGrEdit::SetNodeCursor 499
- NDVarGrEdit::SetNodeHeight 493
- NDVarGrEdit::SetNodeId 491
- NDVarGrEdit::SetNodeProperty 494
- NDVarGrEdit::SetNodeValue 492
- NDVarGrEdit::SetNodeWidth 494
- NDVarGrEdit::SetNodeXOrigin 492
- NDVarGrEdit::SetNodeYOrigin 493
- NDVarGrEdit::SetTitle 481
- NDVarGrInEdgeAccessor::GoFirst 486
- NDVarGrInEdgeAccessor::GoId 486
- NDVarGrInEdgeAccessor::GoIndexed 486
- NDVarGrInEdgeAccessor::GoNext 486
- NDVarGrInEdgeAccessor::GoPrevious 486
- NDVarGrInEdgeAccessor::NDVarGrInEdge-
Accessor 483
- NDVarGrNode::GetChildNode 500
- NDVarGrNode::GetHeight 503
- NDVarGrNode::GetId 501
- NDVarGrNode::GetInEdge 501
- NDVarGrNode::GetNeighborNode 501
- NDVarGrNode::GetNumChildren 500
- NDVarGrNode::GetNumNeighbors 500
- NDVarGrNode::GetNumParents 500
- NDVarGrNode::GetOutEdge 501
- NDVarGrNode::GetParentNode 500
- NDVarGrNode::GetProperty 504
- NDVarGrNode::GetUndirectedEdge 501
- NDVarGrNode::GetValue 502
- NDVarGrNode::GetWidth 503
- NDVarGrNode::GetXOrigin 502
- NDVarGrNode::GetYOrigin 502
- NDVarGrNode::QueryValue 501
- NDVarGrNode::RemoveProperty 504
- NDVarGrNode::SetHeight 503
- NDVarGrNode::SetId 501
- NDVarGrNode::SetProperty 504
- NDVarGrNode::SetValue 502
- NDVarGrNode::SetWidth 503
- NDVarGrNode::SetXOrigin 502
- NDVarGrNode::SetYOrigin 502
- NDVarGrNode::StartEdit 507
- NDVarGrNodeAccessor class 61, 69, 70
- NDVarGrNodeAccessor::GoFirstChild 484
- NDVarGrNodeAccessor::GoFirstNeighbor 485
- NDVarGrNodeAccessor::GoFirstParent 484
- NDVarGrNodeAccessor::GoFirstRoot 484
- NDVarGrNodeAccessor::GoId 484
- NDVarGrNodeAccessor::GoIndexed 484
- NDVarGrNodeAccessor::GoNext 485
- NDVarGrNodeAccessor::GoNthChild 484
- NDVarGrNodeAccessor::GoNthNeighbor 485
- NDVarGrNodeAccessor::GoNthParent 484
- NDVarGrNodeAccessor::GoNthRoot 484
- NDVarGrNodeAccessor::GoPrevious 485
- NDVarGrNodeAccessor::NDVarGrNode Ac-
cessor 482
- NDVarGrNodeAccessor::NDVarGrNodeAc-
cessor 482
- NDVarGrNodeEdit::RemoveProperty 504
- NDVarGrNodeEdit::SetHeight 503
- NDVarGrNodeEdit::SetId 501
- NDVarGrNodeEdit::SetProperty 504
- NDVarGrNodeEdit::SetValue 502
- NDVarGrNodeEdit::SetWidth 503
- NDVarGrNodeEdit::SetXOrigin 502
- NDVarGrNodeEdit::SetYOrigin 503
- NDVarGrOutEdgeAccessor::Go Indexed 487
- NDVarGrOutEdgeAccessor::Go Previous 487
- NDVarGrOutEdgeAccessor::GoFirst 487
- NDVarGrOutEdgeAccessor::GoId 487
- NDVarGrOutEdgeAccessor::GoNext 487
- NDVarGrOutEdgeAccessor::NDVarGrOut-
EdgeAccessor 483
- NDVarGrUndirectedEdgeAccessor:: NDVarG-
rUndirectedEdge Access 483
- NDVarGrUndirectedEdgeAccessor:: NDVarG-
rUndirectedEdge Accessor 483
- NDVarGrUndirectedEdgeAccessor::GoFirst
487
- NDVarGrUndirectedEdgeAccessor::GoId 488
- NDVarGrUndirectedEdgeAccessor::GoIn-
dexed 488
- NDVarGrUndirectedEdgeAccessor::GoNext
487
- NDVarGrUndirectedEdgeAccessor::GoPrevi-
ous 487
- NDVarLs::AddRow 511
- NDVarLs::Class 479, 509
- NDVarLs::DefNfy 513
- NDVarLs::GetCursorRow 511
- NDVarLs::GetMaxRow 510
- NDVarLs::GetMaxStrLen 510
- NDVarLs::GetMods 513
- NDVarLs::GetNumRows 511
- NDVarLs::GetRowTitle 510
- NDVarLs::GetRowValue 510
- NDVarLs::GetTitle 509
- NDVarLs::QueryRowValue 510
- NDVarLs::RemoveRow 511
- NDVarLs::SetCursorRow 511
- NDVarLs::SetNumRows 511
- NDVarLs::SetRowTitle 510
- NDVarLs::SetRowValue 510
- NDVarLs::SetTitle 509

NDVarLs::StartRowEdit 512
NDVarLsEdit::AddRow 512
NDVarLsEdit::RemoveRow 512
NDVarLsEdit::SetCursorRow 512
NDVarLsEdit::SetNumRows 512
NDVarLsEdit::SetRowTitle 512
NDVarLsEdit::SetRowValue 512
NDVarLsEdit::SetTitle 513
NDVarTb::AddColumn 518
NDVarTb::AddRow 517
NDVarTb::Class 515
NDVarTb::GetCellValue 516
NDVarTb::GetColumnTitle 516
NDVarTb::GetCursorColumn 517
NDVarTb::GetCursorRow 517
NDVarTb::GetMaxColStrLen 516
NDVarTb::GetMods 520
NDVarTb::GetNumColumns 515
NDVarTb::GetNumRows 515
NDVarTb::GetRowTitle 516
NDVarTb::GetTitle 516
NDVarTb::QueryCellValue 516
NDVarTb::RemoveColumn 518
NDVarTb::RemoveRow 517
NDVarTb::SetColumnTitle 518
NDVarTb::SetColValue 518
NDVarTb::SetCursorColumn 518
NDVarTb::SetCursorRow 518
NDVarTb::SetNumRowColumns 517
NDVarTb::SetRowTitle 516, 518
NDVarTb::SetTitle 518
NDVarTb::StartCellEdit 517
NDVarTb::StartEdit 519
NDVarTb::StartRowEdit 517
NDVarTbEdit::AddColumn 519
NDVarTbEdit::AddRow 519
NDVarTbEdit::RemoveColumn 519
NDVarTbEdit::RemoveRow 519
NDVarTbEdit::SetCellValue 519
NDVarTbEdit::SetColumnTitle 520
NDVarTbEdit::SetCursorColumn 520
NDVarTbEdit::SetCursorRow 520
NDVarTbEdit::SetNumRowColumns 519
NDVarTbEdit::SetRowTitle 520
NDVarTbEdit::SetTitle 520
NDVarTr::AddNode 525
NDVarTr::AreNodesEqual 526
NDVarTr::Class 523
NDVarTr::DisposeCursor 527
NDVarTr::GetCursor 527
NDVarTr::GetMods 529
NDVarTr::GetNodeID 528
NDVarTr::GetNodeValue 527
NDVarTr::GetNumChildNodes 526
NDVarTr::GetNumRootNodes 526
NDVarTr::GetNumSiblingNodes 526
NDVarTr::GetTitle 523
NDVarTr::IsNodeValid 526
NDVarTr::QueryNodeID 528
NDVarTr::QueryNodeValue 527
NDVarTr::RemoveNode 525
NDVarTr::RemoveTree 525
NDVarTr::SetCursor 527
NDVarTr::SetNodeID 528
NDVarTr::SetNodeValue 527
NDVarTr::SetTitle 523
NDVarTr::StartEdit 528
NDVarTr::StartNodeEdit 528
NDVarTrEdit::AddNode 525
NDVarTrEdit::RemoveNode 525
NDVarTrEdit::RemoveTree 526
NDVarTrEdit::SetCursor 527
NDVarTrEdit::SetNodeID 529
NDVarTrEdit::SetNodeValue 528
NDVarTrEdit::SetTitle 523
NDVarTrNodeAccessor::Class 523
NDVarTrNodeAccessor::GoFirst Sibling 524
NDVarTrNodeAccessor::GoFirstChild 524
NDVarTrNodeAccessor::GoFirstRoot 524
NDVarTrNodeAccessor::GoNext Sibling 524
NDVarTrNodeAccessor::GoNthChild 525
NDVarTrNodeAccessor::GoNthRoot 525
NDVarTrNodeAccessor::GoNthSibling 525
NDVarTrNodeAccessor::GoParent 524
NDVarTrNodeAccessor::GoPrev Sibling 524
NDVarTrNodeEdit::SetID 529
NDVarTrNodeEdit::SetValue 528
NDVStr::Append 533
NDVStr::AppendChar 533
NDVStr::AppendStr 533
NDVStr::AppendStrSu 533
NDVStr::Clear 534
NDVStr::Cmp 534
NDVStr::CmpStr 534
NDVStr::Copy 532
NDVStr::GetLen 533
NDVStr::GetStr 533
NDVStr::ICmp 534
NDVStr::ICmpStr 534
NDVStr::QueryStrSub 533
NDVStr::Set 532
NDVStr::SetCtStr 532
NDVStr::SetCtStrSub 532
NDVStr::SetNatStr 532
NDVStr::SetNatStrSub 532
NDVStr::SetRes 535
NDVStr::SetStr 531
NDVStr::SetStrSub 532

NDVStr::TruncAt 534
 NDVStr::Truncate 534
 NDVStrArray::~~NDVStrArray 535
 NDVStrArray::NDVStrArray 535
 node accessors 61, 71, 124
 creating 114
 node cursors 71, 73
 view options 82
 node edit objects 76
 Node Edition dialog 83
 node ID values 61
 nodes
 binary trees 163
 file system entities 273
 graphs 60, 67
 creating 117–124
 properties 98, 99
 variant trees 523
 nonwindow-based applications 389
 notification variable 423
 notifications 339, 399
 See also notifier
 notifier 339
 add client 341
 associate client with data 341
 broadcast notification 340
 construct client 340
 default client constructor 340
 default client destructor 340
 default constructor 340
 default destructor 340
 get client data 341
 remove client 341
 NT file I/O 247
 NULL 181
 null values 181
 numeric sets *See* interval sets
 numeric values 131
 reading 184
 NxDataSource 12
 properties 12
 NxTableDataSource 12
 properties 12, 13

O

object collections 139
 object pointers 147
 object references 147
 object sets *See* set
 ODD 181
 OI Core data sources 9
 OI example 8
 Open Interface File I/O 248
 opening files 259, 260
 operating system selection 335
 origins 377
 See also rectangle

OS/2 file I/O 247
 OutEdgePtrArr 60, 62
 output 183, 247
 output buffer 269

P

pack objects
 decode with CCITT 347
 decode with LZW 347
 decode with PackBits 346
 decode with RLE algorithm 346
 decode with specified method 348
 default constructor 344
 default destructor 344
 encode with CCITT 347
 encode with LZW 347
 encode with PackBits 346
 encode with RLE algorithm 346
 encode with specified method 348
 PackBits algorithm 343, 346
 PackMethodEnum 348
 PackRec 344
 paging mechanism 183
 paging methods 185, 191
 path name 299
 PC code pages 209
 PC code type 218
 PerfEnum 174, 410
 persistent data 371
 persistent fields 349
 categories 350
 describe to resource manager 350
 enumerated types 349
 resources 387
 persistent objects 383
 persistent resources 9
 PFldCatEnum 350
 PFldRec
 warning 350
 PFldTypeEnum 349
 PM look
 huge buffer 172
 huge strings 172
 operating system selection 335
 portable far keyword 332
 portable near keyword 332
 windowing system selection 336
 point objects 351
 construct with values 351
 default constructor 351
 default destructor 351
 get distance between 353
 get X coordinate 352
 get Y coordinate 352
 increment coordinates 352
 regions 405
 reset coordinates 352
 set coordinates same values 352
 set X coordinate 352

- set Y coordinate 352
 - test for null 352
 - test values 353
 - point structures 377
 - pointer 361
 - See also* memory manager
 - client information 171
 - resource library 412
 - resource notification 397
 - polygonal regions 409
 - private stuff 238
 - program termination 231
 - properties 10
 - graph edges 495, 497, 504, 506
 - graph links 98, 106
 - graph nodes 98, 99, 490, 494, 503
 - variant trees 523, 526
 - PtrFailEnum 360
 - PtrStatsPtr 359
 - PtrStatsRec 359
- R**
- RangePtr 193
 - RangeRec 193
 - ranges 193
 - contain cells 194
 - rc format 383
 - RClasFlagsSetEnum 371
 - RClasRec 371
 - read-only graphs 83
 - record collections 157
 - RecordSet
 - defined 10
 - modifying 11
 - updating 11
 - RecordSetDataSource 10
 - implementation 10
 - methods 11
 - Rect16::Intersection 380
 - rectangle 377
 - call callback proc 410
 - callback method 410
 - contained within 379
 - contains specified rectangle 380
 - copy/reset origin 380
 - copying 380
 - create intersection 408
 - determine union 380
 - disjointed 405
 - exclusive Or with region 409
 - get begin/end coordinates 382
 - get distance between points 378
 - get origin/extent 381
 - increment origin/extent 378
 - intersects with 380
 - invalid 377
 - make coordinates valid 381
 - make equal 379
 - make fit 380
 - move 381
 - reposition 380
 - reset coordinates 379
 - set begin/end coordinates 382
 - set coordinates 379
 - set origin/extent 378, 382
 - set point location 379
 - subtract from region 409
 - test for empty 379
 - test for empty point 378
 - test for specified point 378
 - union with region 409
 - validate coordinates 381
 - region 405
 - associate with rectangle 408
 - check for empty 405
 - constructor 409
 - create intersection 407, 408
 - create union 408, 409
 - determine inside points 407
 - get boundaries 406
 - get rectangle position 407
 - perform exclusive Or 408, 409
 - perform offset 406
 - relative position 405
 - reset to empty 406
 - set coordinates 407, 408
 - subtract from another 408
 - subtract rectangle 409
 - test equality 406
 - translate by offset 406
 - register variables 333
 - RegisterEventRec 424
 - registering resource views 223
 - RegisterVerbRec 425
 - relative file names 296
 - ResNfyProc 397
 - resource 9, 383
 - attached vs. detached 385
 - check full name 394
 - check if initialized 396
 - check if named 394
 - clone persistent fields 388
 - count children 395
 - create 388
 - deallocate persistent fields 388
 - default command routing 403
 - default notification handler 397
 - define notifications 396
 - edit definitions 383
 - exit library 390
 - filter attributes 350
 - find 393
 - find by full name 393
 - find Nth resource 462
 - find string 462
 - finding 462
 - get class 396
 - get client data 395
 - get client notification 398, 399
 - get IsCommandSource flag 403
 - get name 394
 - get notification handler 398

- get notify data 401
 - get Nth child 395
 - handle command 404
 - increment reference count 388
 - initialize library 390
 - install library 389
 - issue execution command 403
 - issue update command 404
 - load and edit dat file 412
 - load and initialize attached 392
 - load and initialize detached 392
 - load and initialize library 390
 - load at run time 383
 - load by class name 393
 - load by full name 391, 393
 - load by module name 391
 - load children 393
 - load detached 392
 - loading 461
 - naming 384
 - nonwindow-based applications 389
 - notify and send 400
 - persistent fields 349, 387
 - query full name 394
 - reference count 385
 - remove notification handler 398
 - save at run time 383
 - save to binary file 389
 - save to text file 389
 - scripting functionality 421
 - send CtrlNfyData notification 402
 - send NfyEND 401
 - send NfyINIT 401
 - send NfyRESET 401
 - send notification 400
 - send vs. post 399
 - set client data 395
 - set client notification 398, 399
 - set notification handler 398
 - start command routing 403
 - trigger default notification 402
 - trigger default parent notification 402
 - verify class 404
 - verify inheritance 396
 - Resource Browser 371
 - resource class 371, 384
 - access 396
 - default allocation 372
 - default deallocation 372
 - determine subclasses 374
 - enumerated types 371
 - find by name 373
 - find library 411
 - get default notification 373
 - get fields 373
 - get first class 373
 - get library name 411
 - get next class 374
 - get pointer 388
 - notification codes 396
 - registering 387
 - registering new 372
 - set default notification 374
 - set fields 374
 - trigger default notification 375
 - resource library object 411
 - See also* library
 - resource manager 383
 - resource scripting 404
 - execute script 404
 - resource views 223
 - responder objects 383
 - response files 127
 - RETRY mechanism 229
 - RgnPosEnum 405
 - RLE algorithm 343, 346
 - RLibPtr 412
 - Run Length Byte Encoding 343, 346
- S**
- script 421
 - compile 429
 - compile string resource 429
 - compile to file 429
 - copy return value 430
 - data types 422
 - dispose 430
 - execute 429
 - get return type 429
 - handler format 421
 - initialize and load library 423
 - install and initialize library 423
 - install in resource 404
 - install library 423
 - language extensions 424
 - load, compile and execute 428
 - register constants 424
 - register events 425
 - register verbs 425, 427
 - set string return value 427
 - unload and uninstall library 423
 - script variables 422
 - scripting language *See* script
 - SCVRB_LibInstall 423
 - SCVRB_LibLoadInit 423
 - selection tables 15
 - SELF variable 423
 - services classes 2
 - set 431
 - See also* interval sets
 - add element 431
 - add multiple elements 432
 - combine 433
 - combine and extract 433
 - compare 433
 - count elements 432
 - default constructor 431
 - default destructor 431
 - empty 432
 - empty then copy 432
 - query elements 432
 - remove element 432
 - remove multiple elements 432

- set elements 432
- set shared empty set 431
- test equality 433
- test for elements 432
- SetMixPartSetEnum 433
- shared resources 385
- signal file name errors 302
- signed integer constants 171
- signed integers 333
- silent failures 241
- sorting large arrays 152
- Source Code Control System (SCCS) 178
- standalone applications 428
- string
 - See also* substrings; variable strings
 - append string 441
 - append to string butter 417
 - append to variable string 533
 - basic types 435
 - clone string 439
 - compare two strings 448
 - compare two strings for equality 449
 - compare with variable string 534
 - compile 429
 - convert ASCII character to lower case 461
 - convert ASCII character to upper case 460
 - convert decimal integer into string buffer 458
 - convert to machine format 368
 - convert to standard format 367
 - converting 458
 - copy to buffer 442
 - copying 439
 - define native string type 437
 - define UNICODE string pointer 439
 - defines native string pointer 438
 - defines UNICODE string type 439
 - disposes of string buffer 440
 - disposing 440
 - find first character 452
 - find first string 452
 - find first substring 452, 453
 - find last character 453
 - find last string 453
 - find last substring 454
 - finding last string 453
 - get character code 443, 444
 - get character code, native string 443
 - get code in front of string location 444
 - get decimal integer 454
 - get double real numeric value 456
 - get double real string 456
 - get hexadecimal integer 455
 - get integer radix 455
 - get length 442
 - get new copy 439
 - get radix integer 455
 - getting a character code 443
 - getting an integer 455
 - global string pointer 438
 - global string type 438
 - huge string type 172
 - load resource 461
 - load string resource 470
 - match character 450
 - match pattern 450
 - match string 450
 - memory allocation 367
 - native string types 435
 - put native code 447
 - set substring 441
 - set variable string 535
 - write ASCII character 446
 - write ASCII character into native string 447
 - write ASCII character into string 446
 - write ASCII character to buffer 460
 - write ASCII character to upper case 461
 - write character 447
 - write character code 446, 447
 - write character code into string 447
 - write into buffer 445
 - write native ASCII character 447
 - write native character code 448
 - write native code into native string 447
 - write substring to buffer 445
 - write to buffer 445
- string buffer 415
 - append contents 417
 - change contents 417
 - clear contents 417
 - convert byte offset 416
 - convert character count 416
 - convert decimal integers into 458
 - convert string to lower case 419
 - convert string to upper case 418
 - get byte 416
 - get character code 416
 - get contents 415
 - get length 415
 - get specified string 415
 - insert at index specified 417
 - match case-insensitive string 420
 - match case-sensitive string 419
 - match string with specified case 419
 - remove character 418
 - remove character range 418
 - replace character 418
 - truncate string 418
- string list resource
 - add string 466
 - find nth string 467
 - get length 465
 - get nth string 466
 - inserts new string 466
 - load nth string 467
 - remove string 466
 - set nth string 466
- string resource
 - find string resource 470
 - get id 470
 - get string 470
 - set id 470
 - set string 470
- StrIVal 198, 438
- StrIValPtr 438

- subclasses 374
- subresources 384
- substring
 - append substring 442
 - compare 449
 - compare two substrings 449, 451
 - convert ASCII character to lower case 461
 - convert substring to lower case 461
 - find first character 452
 - find last character 454
 - finding first substring 452
 - finding within a string 453
 - get double 458
 - get double real numeric value 458
 - get integer, decimal substring 456
 - get integer, given radix 457
 - get integer, hex substring 457
 - match subpattern 451
 - match substring containing pattern 451
 - match substrings 451
 - set new substring 440
 - set substring 441
- substrings
 - append to variable string 533
 - finding 533
 - get from buffer 415
 - replace variable string with 532
- symbolic constants 424
- synchronized access 6
- synchronous notifications 399

T

- T4 encoding *See* CCITT Group3 algorithm
- T6 encoding *See* CCIT Group4 algorithm
- table data sources 7, 12
- table types 14
- Tandem mainframes file I/O 248
- text files 247
 - get next character 266
 - query current position 263
 - read line 268
 - read N characters 267
 - read string 267
 - return current offset position 263
 - set current position 264
 - write character 266
 - write N characters 267
 - write null-terminated string 268
 - write string and line terminator 268, 269
- tracing mechanism 233, 239
- truncate file 270
- truncating strings 418, 533, 534
- types *See* data types

U

- UInt 170
- UInt16 170
- UInt32 170

- UInt64 170
- UInt8 170
- UnDirEdgePtrArr 60, 63
- UnDirPtrArr 63
- Unicode 198
- UNICODE character pointer 198
- UNICODE character type 196
- UNICODE code sets 210
- UNICODE code type 218
- UnicodePtr 198
- UniStr 198
- UniStrPtr 198
- UNIX exception handling 243–244
- Unix file I/O 247
- unsigned integer constants 171
- unsigned integer types 170
- update interfaces 225
- user input 12
- user-defined ClientData 395
- UTF2 code type 218
- UTF8 code type 218

V

- variable strings 531
 - append character to 533
 - append string 533
 - append substring 533
 - append to string buffer 417
 - clear contents 534
 - compare with string 534
 - comparing 534
 - construct array 535
 - copying 532
 - default array constructor 535
 - default array destructor 535
 - get length 532
 - get string 533
 - query substrings 533
 - replace content with copy 531
 - replace content with native 532
 - replace contents 532
 - replace contents with substring 532
 - replace copy with native 532
 - resetting 534
 - set contents 535
 - truncate 534
 - truncate exactly 533
- variables 422
 - assigning values 422
 - notification 423
- variant data
 - constructor from value 473
 - conversion methods 474–477
 - convert type 474
 - convert value 474
 - copy constructor 473
 - copy to type 474
 - destructor 473

- detect reference 477
- empty variant 477
- get type 477
- initialize class 473
- return empty variant 477
- return NULL variant 477
- return NULLOBJ object 477
- types 471, 471–472
- unload class 473
- variant data source 479
 - get values 479
 - notifications 479
 - query values 479
 - set values 479
- variant data source lists 509
 - add row 511, 512
 - count rows 511
 - external declarations 521
 - get cursor position 511
 - get modification descriptions 513
 - get row title 510
 - get row value 510
 - get title 509
 - implementation 521
 - notifications 513
 - query row value 510
 - remove row 511, 512
 - set cursor position 511
 - set edition cursor 512
 - set edition row title 512
 - set edition row values 512
 - set edition rows 512
 - set edition title 513
 - set number of rows 511
 - set row value 510
 - set title 509, 510
 - start row edit 512
- variant data source tables 515
 - add column 518
 - add edition column 519
 - add edition row 519
 - add row 517
 - count columns 515
 - count rows 515
 - get cell values 516
 - get column title 516
 - get cursor position 517
 - get longest column string 516
 - get modifications 520
 - get row title 516
 - get table title 516
 - open edition 519
 - query cell values 516
 - remove column 518
 - remove edition column 519
 - remove edition row 519
 - remove row 517
 - set column values 518
 - set edition cell values 519
 - set edition column title 518, 520
 - set edition current column 518, 520
 - set edition current row 518, 520
 - set edition row title 518, 520
 - set edition rows/columns 519
 - set edition table title 518, 520
 - set number rows/columns 517
 - set row title 516
 - start cell edition 517
 - start edit 519
 - start row edition 517
- variant graph datasources 481
 - accessors referring to same edge 495
 - accessors referring to same node 490
 - adding directed edge 489
 - adding nodes 488
 - adding undirected edge 489
 - construct edge accessor 482
 - construct inward edge accessor 483
 - construct node accessor 482
 - construct outward edge accessor 483
 - construct undirected edge 483
 - count child nodes 500
 - count edges 482, 495
 - count neighbor nodes 500
 - count parent nodes 500
 - creating 507
 - default edge-accessor constructor 482, 483
 - default node-accessor constructor 482
 - get child nodes 490, 500
 - get directed edge 497, 506
 - get edge 500
 - get edge boundaries 504
 - get edge cursor 499
 - get edge ID 496, 505
 - get edge origins 504
 - get edge properties 497, 506
 - get edge values 496, 505
 - get inward edge 501
 - get modification description 507
 - get neighbor nodes 490, 501
 - get node 500
 - get node cursor 498
 - get node height 493, 503
 - get node ID 491, 501
 - get node origins 492, 502
 - get node properties 494, 504
 - get node values 491, 502
 - get node width 494, 503
 - get number of nodes 482
 - get outward edge 501
 - get parent nodes 490, 500
 - get root nodes 482
 - get title 481
 - get undirected edge 501
 - go to edge between 486
 - go to first child node 484
 - go to first edge 485
 - go to first neighbor node 485
 - go to first parent node 484
 - go to first root node 484
 - go to identified edge 486
 - go to identified node 484
 - go to indexed edge 486
 - go to indexed node 484
 - go to inward edge 486
 - go to next edge 485
 - go to next node 485
 - go to Nth child node 484

- go to Nth neighbor node 485
 - go to Nth root node 484
 - go to outward edge 487
 - go to previous edge 486
 - go to previous node 485
 - go to undirected edge 487, 488
 - query cyclic result 499
 - query edge ID 495
 - query edge properties 497
 - query edge values 496, 505
 - query node height 493
 - query node ID 491
 - query node origins 492
 - query node properties 494
 - query node values 491, 501
 - query node width 493
 - remove edge 489
 - remove edge between 489, 490
 - remove edge properties 497, 498, 506
 - remove node properties 494, 495, 504
 - remove nodes 488
 - set directed edge 497, 506
 - set edge cursor 499
 - set edge ID 496, 505
 - set edge properties 497, 506
 - set edge values 496, 505
 - set node cursor 498, 499
 - set node height 493, 503
 - set node ID 491, 501
 - set node origins 492, 493, 502, 503
 - set node properties 494, 504
 - set node values 491, 492, 502
 - set node width 494, 503
 - set title 481
 - start edge edit 499, 507
 - start node edit 499, 507
 - testing node relationships 498
 - validating edge accessor 495
 - validating node accessor 490
 - variant tree datasources 523
 - accessors referring to same node 526
 - add node 525
 - construct 526
 - count nodes 526
 - describe modifications 529
 - destruct/deallocate cursor 527
 - get cursor 527
 - get node ID 528
 - get node value 527
 - go to child node 524, 525
 - go to parent node 524
 - go to root node 524, 525
 - go to sibling node 524, 525
 - labeling 523
 - properties 523, 526
 - query node ID 528
 - query node value 527
 - remove node 525
 - remove tree 525, 526
 - set cursor 527
 - set node ID 528
 - set node ID field 529
 - set node value 527, 528
 - start edit 528
 - validate node 526
 - VariantDataSource 9
 - VariantListDataSource 9
 - VariantTable
 - methods 11, 14
 - properties 10, 14
 - VariantTableDataSource 9
 - properties 10, 11
 - verbs 423
 - registering 425, 427
 - VersEnum 175
 - version numbers 176
 - VertEnum 175
 - view interfaces 223
 - views 5
 - updating 14
 - virtual data source 223
 - virtual interface implementation 521
 - VMS file I/O 248
 - VMS look
 - file and directory names 283
 - noshare keyword 332
 - operating system selection 335
 - readonly keyword 333
 - windowing system selection 336
 - VStr object 531
- ## W
- W16 API exception handling 244–245
 - warnings 177, 228
 - generating 231, 242
 - wide characters *See* multibyte characters
 - wildcards 291, 292
 - windowing system selection 336
- ## X
- x-axis grids 84
- ## Y
- y-axis grids 85