Neuron Data Elements Environment
Elements Application Services

Version 4.1

C Programmer’s Guide

Contents

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

Chapter 1

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.

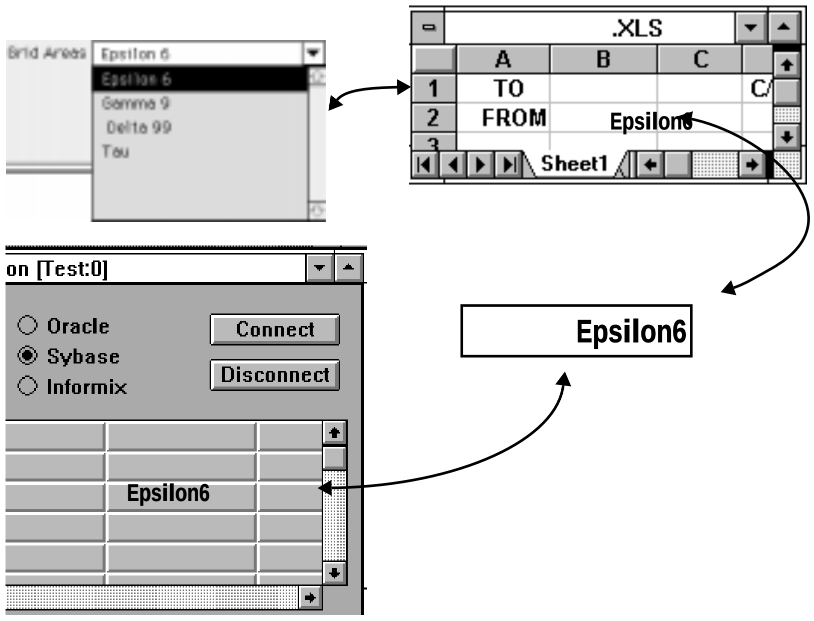
Building Block Mechanisms

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.

Chapter

2 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.RowColumnCount(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...
`RowCount`	Return the number of records in the RecordSet.
`ColumnCount`	Return the number of columns in the RecordSet
`ColumnTitle`	Return the name of the column in the RecordSet
`CursorRow`	Perform either a Get or Set CursorRow operation:

Use this property...	To...
	Use a CursorRow Set to set the current record position in the RecordSet.
	Use a CursorRow Get to return the current record position in the RecordSet.
`Cells`	Perform a Get or a Set:
	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.
`ColumnTitle`	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 RecordSetData Source.

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.

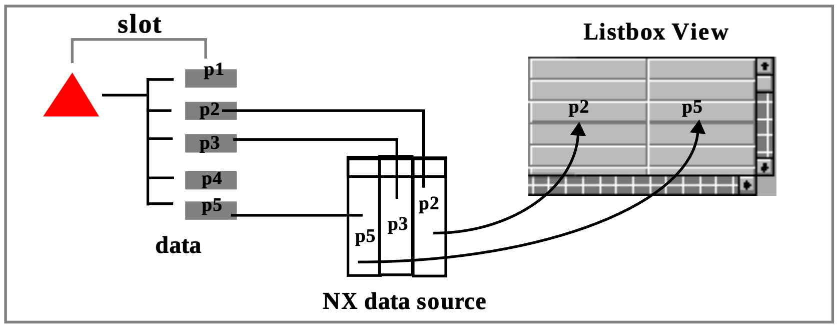
Creating IRE data sources are a little bit different from the Data Access data sources. IRE Data sources have additional properties to be set at creation:

- 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.
`ColumnTitle`	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 tot he 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 WGTSINITIALIZED
       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.RowColumnCount(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 Litsbox 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
```

Chapter

3 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 **TVIEW** and **BROWS** views, which are supplied by the Open Interface Element. The Elements Environment *datasource/views* mechanism supports the interface between the datasource and the **TVIEW** and **BROWS** views.

This chapter discusses these topics:

- Concepts
- Options for the TVIEW and BROWS 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 **TVIEW** and **BROWS** 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 33 to tell you how to program a tree datasource.

Tree Datasource

The *tree datasource*—an object of the **VARTR** 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 (**VARTR** 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 **VARTR** 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.

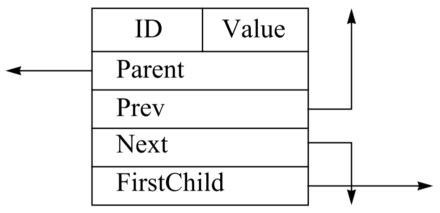


Figure 3–1 The Structure of a Node

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 **ID**s 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 23.

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:

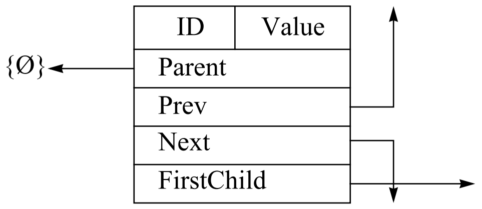


Figure 3–2 Unique Characteristics of a Root Node

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*:

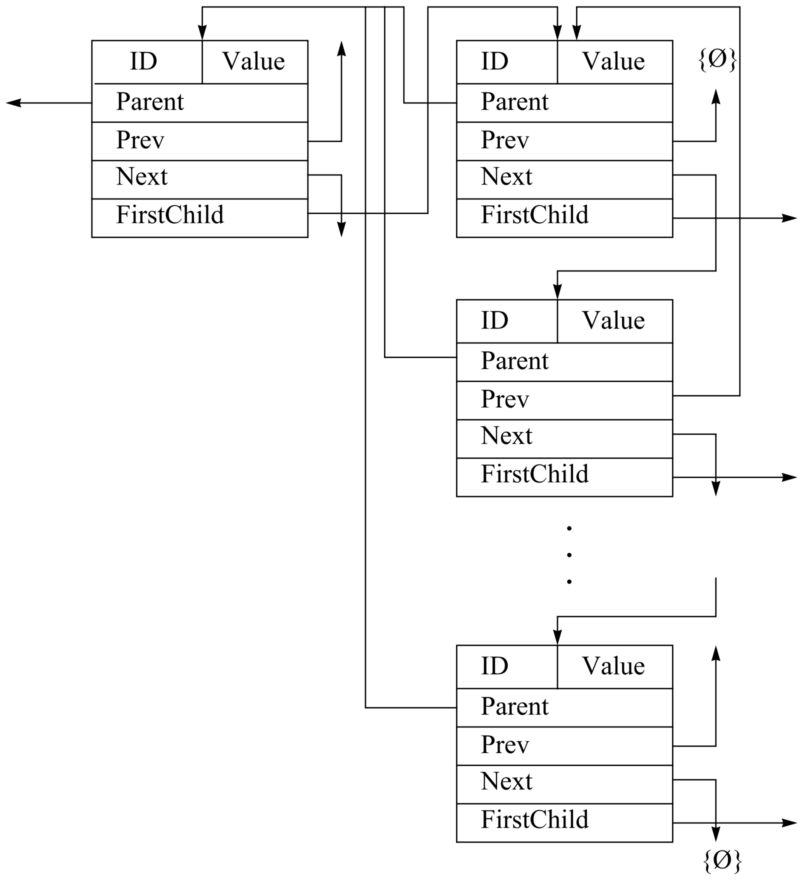


Figure 3–3 Parent-Child Node Relationship

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 than 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 **VARTRNODEACCESSOR** 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. */
VarTrPtr treeDs;

/* Declare a node-accessor pointer variable. */
VarTrNodeAccessorPtr nodeAccessor;
...
/* Assign a tree-datasource object to the tree-datasource
   pointer variable. */
treeDs = VARTR_Create();

/* Assign a node-accessor object to the node-accessor pointer
   variable. */
nodeAccessor = VARTRNODEACCESSOR_Create();
...
/* Destroy the node accessor. */
VARTRNODEACCESSOR_Dispose(nodeAccessor);
...
/* Destroy the tree datasource. */
RES_Release((ResPtr)treeDs);
```

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 36 and “Destroying a Node-Accessor Object” on page 46.

With a node accessor, you can traverse the node hierarchy using functions in the **VARTRNODEACCESSOR** 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:

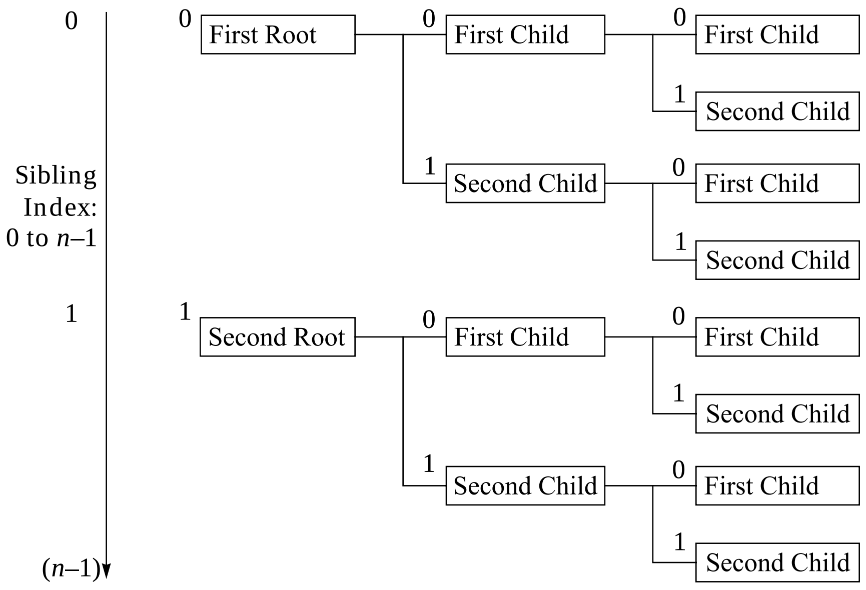


Figure 3–4 Node Indexing in the Tree Datasource

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 36 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 56.

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 **VARTR_SetCursor(***vartr*, *accessor***)** function
- Then access the node at the current cursor location using the **VARTR_GetCursor(***vartr***)** function

When a **TVIEW** or **BROWS** 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. */
VarTrPtr treeDs;

/* Declare a browser pointer variable. */
BrowsPtr browsWgt;

/* Declare a node-accessor pointer variable. */
VarTrNodeAccessorPtr nodeAccessor;

/* Declare two variant pointer variables. */
VarPtr varID, varValue;
...
/* Assign a browser object to the browser pointer variable. */
```

```
browsWgt = BROWS_Create();

/* Assign a tree-datasource object to the tree-datasource
   pointer variable. */
treeDs = VARTR_Create();

/* Register the browser with the tree datasource. */
DS_RegisterView((DsPtr)treeDs, (ResPtr)win->browsWgt);
DS_SetViewOptions((DsPtr)treeDs, (ResPtr)win->browsWgt,
                  "cursor", "CONTROLS");

/* Assign a node-accessor object to the node-accessor pointer
   variable. */
nodeAccessor = VARTRNODEACCESSOR_Create();

/* Assign variant objects to the variant-pointer variables. */
varID = VAR_New();
varValue = VAR_New();

/* Set a cursor at the location of the node accessor. */
VARTR_SetCursor(treeDs, nodeAccessor);

/* Position the node accessor and edit the tree. */
...
/* Use "convenience" API functions to edit the ID and Value
   properties of the node at the current cursor location. */
VAR_SetStr(varID, "0000");
VAR_SetStr(varValue, "Node");
VARTR_SetNodeID(treeDs, VARTR_GetCursor(treeDs), varID);
VARTR_SetNodeValue(treeDs, VARTR_GetCursor(treeDs), varValue);
...
// Destroy the variant objects.
VAR_Delete(varID);
VAR_Delete(varValue);

/* Destroy the node accessor. */
VARTRNODEACCESSOR_Dispose(nodeAccessor);

/* Destroy the tree datasource. */
RES_Release((ResPtr)treeDs);
```

"Destroying a Node-Accessor Object" on page 46 shows an alternative use of the **GetCursor()** function. For information about setting the cursor behavior, see "Options for the TVIEW and BROWS Views" on page 31.

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 29.

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 **VARTR** class executes the **StartEdit()** function. This is a public function inherited from the **DS** class. The tree datasource is unlocked when the **DSEDIT_End()** function executes, as shown in this example:

```
/* Declare pointer variables. */
VarTrPtr treeDs;
VarTrNodeAccessorPtr nodeAccessor;

/* Declare a datasource-level edit pointer. */
VarTrEditPtr editTreeDs;

/* Create objects and assign them to pointer variables. */
treeDs = VARTR_Create();
nodeAccessor = VARTRNODEACCESSOR_Create();

/* Execute the StartEdit() function to create a datasource edit
   object, and assign it to the datasource edit pointer. */
editTreeDs = VARTR_StartEdit(treeDs);

/* Position the node accessor and edit the tree. */
VARTRNODEACCESSOR_GoNthRoot(VARTR_GetNumRootNodes(treeDs));
VARTREDIT_AddNode(editTreeDs, nodeAccessor);
...
VARTRNODEACCESSOR_GoFirstRoot(nodeAccessor);
VARTREDIT_RemoveNode(editTreeDs, nodeAccessor);

/* Execute the DSEDIT_End() function. */
DSEDIT_End((DsEditPtr)editTreeDs);

/* Destroy other objects. */
VARTRNODEACCESSOR_Dispose(nodeAccessor);
RES_Release((ResPtr)treeDs);
```

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 **VARTR** class executes the **StartNodeEdit()** function. This is a public function inherited from the **DS** 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. */
VarTrPtr treeDs;
VarTrNodeAccessorPtr nodeAccessor;
VarPtr varID, varValue;

/* Declare a node-level edit pointer. */
VarTrNodeEditPtr editNode;

treeDs = VARTR_Create();
nodeAccessor = VARTRNODEACCESSOR_Create();
varID = VAR_New();
varValue = VAR_New();

/* Position the node accessor and edit nodes. */
VARTRNODEACCESSOR_GoFirstRoot(nodeAccessor);
while (VARTR_IsNodeValid(treeDs, nodeAccessor)) {
   VARTRNODEACCESSOR_GoNext(nodeAccessor);
}

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

/* Execute the StartNodeEdit() function to create a node-level
   edit object, and assign it to the node-level edit pointer. */
editNode = VARTR_StartNodeEdit(treeDs, nodeAccessor);

/* 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 (editNode != NULL) {
   VAR_SetStr(varID, "0000");
   VAR_SetStr(varValue, "New Node");
   VARTRNODEEDIT_SetID(editNode, varID);
   VARTRNODEEDIT_SetValue(editNode, varValue);
   ...
/* Commit the edit operations and dispose of the node-level edit
   object. */
   DSEDIT_End((DsEditPtr)editNode);
} /* End if. */
...
/* Dispose of other objects. */
VAR_Delete(varID);
VAR_Delete(varValue);
VARTRNODEACCESSOR_Dispose(nodeAccessor);
RES_Release((ResPtr)treeDs);
```

The code in the preceding example creates a node edit object after the **VARTR_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 **DSEDIT_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 **VARTR_AddNode(***vartr, 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, **VARTR_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. */
VarTrPtr treeDs;
VarTrNodeAccessorPtr nodeAccessor;

/* Initialize the pointer variables. */
treeDs = VARTR_Create();
nodeAccessor = VARTRNODEACCESSOR_Create();

/* Move the node accessor to the next empty root-node
   location. */
VARTRNODEACCESSOR_GoFirstRoot(nodeAccessor);
while (VARTR_IsNodeValid(treeDs, nodeAccessor)) {
   VARTRNODEACCESSOR_GoNext(nodeAccessor);
}

/* 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. */
VARTR_AddNode(treeDs, nodeAccessor);
...
```

```
/* Dispose of other objects. */
VAR_Delete(varID);
VAR_Delete(varValue);
VARTRNODEACCESSOR_Dispose(nodeAccessor);
RES_Release((ResPtr)treeDs);
```

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 27.

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, **VARTR_SetNodeID()** and **VARTR_SetNodeValue()**. Each of these functions creates a node edit object, commits its edit operation, and destroys the edit object.

```
/* Declare pointer variables. */
VarTrPtr treeDs;
VarTrNodeAccessorPtr nodeAccessor;
VarPtr varID, varValue;

/* Initialize the pointer variables. */
treeDs = VARTR_Create();
nodeAccessor = VARTRNODEACCESSOR_Create();
varID = VAR_New();
varValue = VAR_New();

/* Move the node accessor to the next empty root-node
   location. */
VARTRNODEACCESSOR_GoFirstRoot(nodeAccessor);
while (VARTR_IsNodeValid(treeDs, nodeAccessor)) {
   VARTRNODEACCESSOR_GoNext(nodeAccessor);
}

/* 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. */
VARTR_AddNode(treeDs, nodeAccessor);

/* Set the variant objects to some initializing values. */
VAR_SetStr(varID, "0000");
VAR_SetStr(varValue, "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. */
VARTR_SetNodeID(treeDs, nodeAccessor, varID);
VARTR_SetNodeValue(treeDs, nodeAccessor, varValue);
...
/* Dispose of other objects. */
VAR_Delete(varID);
VAR_Delete(varValue);
VARTRNODEACCESSOR_Dispose(nodeAccessor);
RES_Release((ResPtr)treeDs);
```

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 27.

Options for the TVIEW and BROWS Views

The tree datasource supports these view options for the **TVIEW** and **BROWS** views:

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

To set view options, use the third and fourth parameters of the **DS_SetViewOption()** function, as shown here:

```
DS_SetViewOption(<datasource>, (ResPtr)<win>-><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:

```
DS_SetViewOption(<datasource>, (ResPtr)<win>-><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 tviewWgt cursor reflects the current location of the datasource cursor, while browsWgt controls the position of the datasource cursor:

```
VarTrPtr treeDs;
BrowsPtr browsWgt;
TViewPtr tviewWgt;
...
DS_RegisterView((DsPtr)treeDs, (ResPtr)win->browsWgt)
DS_RegisterView((DsPtr)treeDs, (ResPtr)win->tviewWgt)
...
DS_SetViewOption((DsPtr)treeDs, (ResPtr)win->tviewWgt,
                  "cursor", "REFLECTS");
DS_SetViewOption((DsPtr)treeDs, (ResPtr)win->browsWgt,
                  "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:

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

where *n* is the number of levels of expansion from the root node in the treeDs hierarchy. The default expansion level depends on the type of view to which it applies. By default, **BROWS** views are fully expanded (*n* expansion levels displayed), while **TVIEW** 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, treeDs, has six expansion levels. Widgets tviewWgt and browsWgt share treeDs as a common datasource with browsWgt controlling the datasource cursor and tviewWgt reflecting it. However, if the browsWgt cursor is placed on a level 4 node, the tviewWgt cursor is unable to reflect its position in the display, because it is initially expanded only to two expansion levels.

```
VarTrPtr treeDs;
BrowsPtr browsWgt;
TViewPtr tviewWgt;
...
DS_RegisterView((DsPtr)treeDs, (ResPtr)win->browsWgt)
DS_RegisterView((DsPtr)treeDs, (ResPtr)win->tviewWgt)
...
DS_SetViewOption((DsPtr)treeDs, (ResPtr)win->tviewWgt,
                 "cursor", "REFLECTS");
DS_SetViewOption((DsPtr)treeDs, (ResPtr)win->browsWgt,
                 "cursor", "CONTROLS");
DS_SetViewOption((DsPtr)treeDs, (ResPtr)win->tviewWgt,
                  "initexpandlevel", "2");
DS_SetViewOption((DsPtr)treeDs, (ResPtr)win->browsWgt,
                 "initexpandlevel", "6");
```

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

```
VarTrPtr treeDs;
BrowsPtr browsWgt;
TViewPtr tviewWgt;
DS_RegisterView((DsPtr)treeDs, (ResPtr)win->browsWgt)
DS_RegisterView((DsPtr)treeDs, (ResPtr)win->tviewWgt)
...
DS_SetViewOption((DsPtr)treeDs, (ResPtr)win->tviewWgt,
                 "cursor", "REFLECTS");
DS_SetViewOption((DsPtr)treeDs, (ResPtr)win->browsWgt,
                 "cursor", "CONTROLS");
DS_SetViewOption((DsPtr)treeDs, (ResPtr)win->tviewWgt,
                  "initexpandlevel", "6");
DS_SetViewOption((DsPtr)treeDs, (ResPtr)win->browsWgt,
                 "initexpandlevel", "6");
```

autosize

For **BROWS** 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:

```
DS_SetViewOption(<datasource>, (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 treeDs;
BrowsPtr brows1;
DS_RegisterView((DsPtr)treeDs, (ResPtr)win->brows1)
...
DS_SetViewOption((DsPtr)treeDs, (ResPtr)win->brows1,
                 "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 **BROWS** and **TVIEW** views—probably a more realistic approach—see “Editing a Tree Datasource” on page 48 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 TVIEW and BROWS Views” on page 31.

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 treeDs as a **VarTrPtr** variable and initializes treeDs to the value returned by **VARTR_Create()**:

```
/* Declare a pointer variable for the tree datasource. */
VarTrPtr treeDs;
...
/* Create a tree-datasource object and assign it to the pointer
   variable. */
treeDs = VARTR_Create();
```

```
...
/* Destroy the tree-datasource object. */
RES_Release((ResPtr)treeDs);
```

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, treeDs.

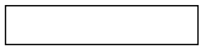


Figure 3–5 Untitled Tree Datasource

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 20.

After the datasource object has served its purpose, use the **RES_Release()** function to destroy it. For more information about memory management, see “Destroying the Tree-Datasource Object” on page 46.

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. */
VarTrPtr treeDs;

/* Declare a tree-edit pointer variable. */
VarTrEditPtr editTreeDs;

/* Declare a node-accessor pointer variable. */
VarTrNodeAccessorPtr nodeAccessor;
...
/* Assign a tree-datasource object to the tree-datasource
   pointer variable. */
treeDs = VARTR_Create();

/* Assign a tree-level edit object to the tree-edit pointer
   variable. */
editTreeDs = VARTR_StartEdit(treeDs);

/* Assign a node-accessor object to the node-accessor pointer
   variable. */
nodeAccessor = VARTRNODEACCESSOR_Create();
...
/* Destroy the node accessor. */
VARTRNODEACCESSOR_Dispose(nodeAccessor);
...
```

This creates *nodeAccessor* as a **VarTrNodeAccessorPtr** variable and initializes nodeAccessor to the value returned by **VARTRNODEACCESSOR_Create()**. When your application is finished with the node accessor, use the **VARTRNODEACCESSOR_Dispose(*accessor*)** function to free the memory allocated for it. For more information about memory management, see "Destroying a Node-Accessor Object" on page 46.

For more information about tree-datasource edit objects, see "Node Accessor" on page 23.

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, editTreeDs:

```
VarTrPtr treeDs;

/* Declare a datasource-edit pointer variable. */
VarTrEditPtr editTreeDs;
...
treeDs = VARTR_Create();

/* Assign an edit object to the datasource-edit pointer
   variable. */
editTreeDs = VARTR_StartEdit(treeDs);

/* Edit operations defined. */
...
/* Commit edit operations to the datasource and destroy the edit
   object. */
DSEDIT_End((DsEditPtr)editTreeDs);
...
```

In this example, editTreeDs is a **VarTrEditPtr** variable and is assigned a datasource edit object—the value returned by the **VARTR_StartEdit()** function. When the **DSEDIT_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 27.

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

```
VarTrPtr treeDs;
VarTrEditPtr editTreeDs;
...
treeDs = VARTR_Create();
editTreeDs = VARTR_StartEdit(treeDs);

/* Set the title of the tree datasource. */
VARTREDIT_SetTitle(editTreeDs, "Tree Datasource");

/* Commit editing operations to the datasource and destroy the
   edit object. */
DSEDIT_End((DsEditPtr)editTreeDs);
```

Note the use of the string literal in quotation marks. Unlike the node **ID** and **Value** properties, the **VARTREDIT_SetTitle()** function accepts a string as the datasource title. Building on the example from Figure 3–5, executing the preceding **VARTREDIT_SetTitle()** function adds a title to the tree datasource, as Figure 3–6 shows:

Tree Datasource

Figure 3–6 Titled Tree Datasource

Adding Nodes

After creating a node accessor, you use the **VARTRNODEACCESSOR** 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:

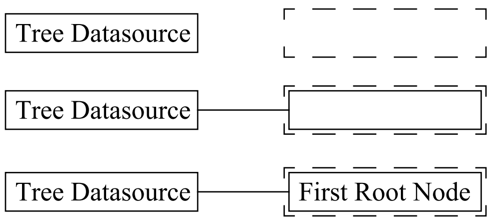


Figure 3–7 Creating the First Root Node in a Tree Datasource

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 treeDs;
VarTrNodeAccessorPtr nodeAccessor;
VarPtr varID, varValue;
VarTrEditPtr editTreeDs;

treeDs = VARTR_Create();
nodeAccessor = VARTRNODEACCESSOR_Create();
varID = VAR_New();
varValue = VAR_New();
editTreeDs = VARTR_StartEdit(treeDs);

/* Set the title of the tree datasource. */
VARTR_SetTitle(mEditTreeDs, "Tree Datasource");

/* Move the node accessor to the first unoccupied root-node
   location. */
VARTRNODEACCESSOR_GoNthRoot(nodeAccessor,
                            VARTR_GetNumRoots(treeDs));

/* Add a node at the first empty root-node location. */
VARTREDIT_AddNode(editTreeDs, nodeAccessor);
```

```
/* Set the node ID and Value properties. */
VAR_SetStr(varID, "New");
VAR_SetStr(varValue, "First Root Node");
VARTREDIT_SetNodeID(editTreeDs, nodeAccessor, varID);
VARTREDIT_SetNodeValue(editTreeDs, nodeAccessor, varValue);

DSEDIT_End((DsEditPtr)editTreeDs);
...
VAR_Delete(varID);
VAR_Delete(varValue);
VARTRNODEACCESSOR_Dispose(nodeAccessor);
RES_Release((ResPtr)treeDs);
```

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 treeDs;
VarTrNodeAccessorPtr nodeAccessor;
VarPtr varID, varValue;

treeDs = VARTR_Create();
nodeAccessor = VARTRNODEACCESSOR_Create();
varID = VAR_New();
varValue = VAR_New();

/* Set the title of the tree datasource using the "convenience"
   API. */
VARTR_SetTitle(treeDs, "Tree Datasource");

/* Move the node accessor to the first unoccupied root-node
   location. */
VARTRNODEACCESSOR_GoFirstRoot(nodeAccessor);
while (VARTR_IsNodeValid(treeDs, nodeAccessor) {
   VARTRNODEACCESSOR_GoNext(nodeAccessor);
}

/* Add a node. */
VARTR_AddNode(treeDs, nodeAccessor);

/* Set the node ID and Value properties using "convenience" API
   functions. */
VAR_SetStr(varID, "New");
VAR_SetStr(varValue, "First Root Node");
VARTR_SetNodeID(treeDs, nodeAccessor, varID);
VARTR_SetNodeValue(treeDs, nodeAccessor, varValue);
...

VAR_Delete(varID);
VAR_Delete(varValue);
VARTRNODEACCESSOR_Dispose(nodeAccessor);
RES_Release((ResPtr)treeDs);
```

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 **VARTRNODEACCESSOR** API move the accessor to the same node. The **VARTRNODEACCESSOR_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 treeDs;
VarTrNodeAccessorPtr nodeAccessor;
...
/* Assignments. */
treeDs = VARTR_Create();
nodeAccessor = VARTRNODEACCESSOR_Create();
...
/* Move the node accessor to the first unoccupied root-node
   location. */
VARTRNODEACCESSOR_GoNthRoot(nodeAccessor,
                            VARTR_GetNumRoots(treeDs));
...
```

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

Warning: The **VARTR_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 treeDs;
VarTrNodeAccessorPtr nodeAccessor;
...
/* Assignments. */
treeDs = VARTR_Create();
nodeAccessor = VARTRNODEACCESSOR_Create();
...
/* Move the node accessor to the first unoccupied root-node
   location. */
VARTRNODEACCESSOR_GoFirstRoot(nodeAccessor);
while (VARTR_IsNodeValid(treeDs, nodeAccessor))
   VARTRNODEACCESSOR_GoNext(nodeAccessor);
}
...
```

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 treeDs;
VarTrNodeAccessorPtr nodeAccessor;
```

```
...
/* Assignments. */
treeDs = VARTR_Create();
nodeAccessor = VARTRNODEACCESSOR_Create();
...
/* Move the node accessor to the first unoccupied root-node
   location. */
VARTRNODEACCESSOR_GoFirstRoot(nodeAccessor);
while (VARTR_IsNodeValid(treeDs, nodeAccessor))
   VARTRNODEACCESSOR_GoNext(nodeAccessor);
}
VARTR_AddNode(treeDs, nodeAccessor);
...
```

Note: You can also “insert” nodes in the hierarchy. For more information, see “Inserting Nodes versus Adding Nodes” on page 51.

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 **ID**s. 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—**VARTR_SetNodeID()** and **VARTR_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 55.

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:

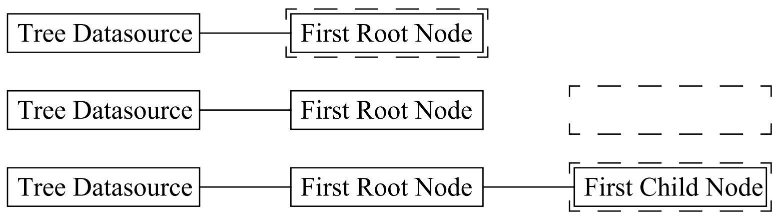


Figure 3–8 Creating the First Child Node

To create the first child node, use the **VARTRNODEACCESSOR** 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 **VARTRNODEACCESSOR_GoFirstChild()** function to position the node accessor:

```
VarTrPtr treeDs;
VarTrNodeAccessorPtr nodeAccessor;
VarPtr varID, varValue;
VarTrEditPtr editTreeDs;

treeDs = VARTR_Create();
nodeAccessor = VARTRNODEACCESSOR_Create();
varID = VAR_New();
varValue = VAR_New();
editTreeDs = VARTR_StartEdit(treeDs);

/* Move the node accessor to the first unoccupied root-node
   location. */
VARTRNODEACCESSOR_GoNthRoot(nodeAccessor,
                            VARTR_GetNumRoots(treeDs));

/* Add a node at the first empty root-node location. */
VARTREDIT_AddNode(editTreeDs, nodeAccessor);

/* Set the node ID and Value properties. */
VAR_SetStr(varID, "0");
VAR_SetStr(varValue, "First Root Node");
VARTREDIT_SetNodeID(editTreeDs, nodeAccessor, varID);
VARTREDIT_SetNodeValue(editTreeDs, nodeAccessor, varValue);

/* Relative to the first root node, move the node accessor to
   the first unoccupied child-node location. */
VARTRNODEACCESSOR_GoFirstChild(nodeAccessor);

/* Add a node at the first child-node location. */
VARTREDIT_AddNode(editTreeDs, nodeAccessor);

/* Set the node ID and Value properties. */
VAR_SetStr(varID, "0,0");
VAR_SetStr(varValue, "First Child Node");
VARTREDIT_SetNodeID(editTreeDs, nodeAccessor, varID);
VARTREDIT_SetNodeValue(editTreeDs, nodeAccessor, varValue);
DSEDIT_End((DsEditPtr)editTreeDs);

VAR_Delete(varID);
VAR_Delete(varValue);
VARTRNODEACCESSOR_Dispose(nodeAccessor);
RES_Release((ResPtr)treeDs);
```

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 treeDs;
VarTrNodeAccessorPtr nodeAccessor;

/* Assignments. */
treeDs = VARTR_Create();
nodeAccessor = VARTRNODEACCESSOR_Create();
...
/* Relative to the first root node, move the node accessor to
   the first unoccupied child-node location. */
VARTRNODEACCESSOR_GoFirstRoot(nodeAccessor);
if (VARTR_IsNodeValid(treeDs, nodeAccessor)) {
   while (VARTR_IsNodeValid(treeDs, nodeAccessor))
      VARTRNODEACCESSOR_GoNext(nodeAccessor);
   } /* End while. */
} /* End if. */
...
```

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

Warning: Like the **VARTR_GetNumRoots()** function, the **VARTR_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 treeDs;
VarTrNodeAccessorPtr nodeAccessor;

/* Assignments. */
treeDs = VARTR_Create();
nodeAccessor = VARTRNODEACCESSOR_Create();
...
/* Relative to the first root node, move the node accessor to
   the first unoccupied child-node location. */
VARTRNODEACCESSOR_GoFirstRoot(nodeAccessor);
if (VARTR_IsNodeValid(treeDs, nodeAccessor)) {
   VARTRNODEACCESSOR_GoNthChild(nodeAccessor,
                             VARTR_GetNumChildren(treeDs,
                                                     nodeAccessor));
} /* 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:

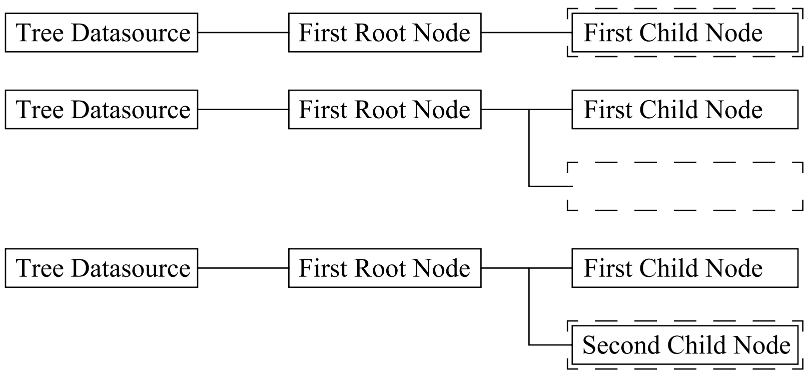


Figure 3–9 Creating the Second Child Node

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 **VARTRNODEACCESSOR_GoNext()** function to position the node accessor:

```
VarTrPtr treeDs;
VarTrNodeAccessorPtr nodeAccessor;
```

```
VarPtr varID, varValue;
VarTrEditPtr editTreeDs;

treeDs = VARTR_Create();
nodeAccessor = VARTRNODEACCESSOR_Create();
varID = VAR_New();
varValue = VAR_New();
editTreeDs = VARTR_StartEdit(treeDs);

/* Move the node accessor to the first unoccupied root-node
   location. */
VARTRNODEACCESSOR_GoNthRoot(nodeAccessor,
                            VARTR_GetNumRoots(treeDs));

/* Add a node at the first empty root-node location. */
VARTREDIT_AddNode(editTreeDs, nodeAccessor);

/* Set the node ID and Value properties. */
VAR_SetStr(varID, "0");
VAR_SetStr(varValue, "First Root Node");
VARTREDIT_SetNodeID(editTreeDs, nodeAccessor, varID);
VARTREDIT_SetNodeValue(editTreeDs, nodeAccessor, varValue);

/* Relative to the first root node, move the node accessor to
   the first child-node location. */
VARTRNODEACCESSOR_GoFirstChild(nodeAccessor);

/* Add a node at the first child-node location. */
VARTREDIT_AddNode(editTreeDs, nodeAccessor);

/* Set the node ID and Value properties. */
VAR_SetStr(varID, "0,0");
VAR_SetStr(varValue, "First Child Node");
VARTREDIT_SetNodeID(editTreeDs, nodeAccessor, varID);
VARTREDIT_SetNodeValue(editTreeDs, nodeAccessor, varValue);

/* Relative to the first child node, move the node accessor to
   the first unoccupied sibling-node location. */
VARTRNODEACCESSOR_GoNext(nodeAccessor);

/* Add a node at the first child-node location. */
VARTREDIT_AddNode(editTreeDs, nodeAccessor);

/* Set the node ID and Value properties. */
VAR_SetStr(varID, "0,1");
VAR_SetStr(varValue, "Second Child Node");
VARTREDIT_SetNodeID(editTreeDs, nodeAccessor, varID);
VARTREDIT_SetNodeValue(editTreeDs, nodeAccessor, varValue);
DSEDIT_End((DsEditPtr)editTreeDs);

VAR_Delete(varID);
VAR_Delete(varValue);
VARTRNODEACCESSOR_Dispose(nodeAccessor);
RES_Release((ResPtr)treeDs);
```

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 treeDs;
VarTrNodeAccessorPtr nodeAccessor;
Int16 i, maxNodes;
...
/* Assignments. */
treeDs = VARTR_Create();
nodeAccessor = VARTRNODEACCESSOR_Create();
...
/* Relative to the first root node, create nodes at, the first
   10 child-node locations. */
VARTRNODEACCESSOR_GoFirstRoot(nodeAccessor);
for (i = VARTR_GetNumChildren(treeDs, nodeAccessor),
     maxNodes = 10; i < maxNodes; i++) {
   VARTRNODEACCESSOR_GoFirstChild(nodeAccessor);
   while (VARTR_IsNodeValid(treeDs, nodeAccessor)) {
      VARTRNODEACCESSOR_GoNthChild(nodeAccessor,
                                 VARTR_GetNumChildren(treeDs,
                                                      nodeAccessor));
   }
   VARTR_AddNode(treeDs, nodeAccessor);
   VARTRNODEACCESSOR_GoParent(nodeAccessor);
}
...
```

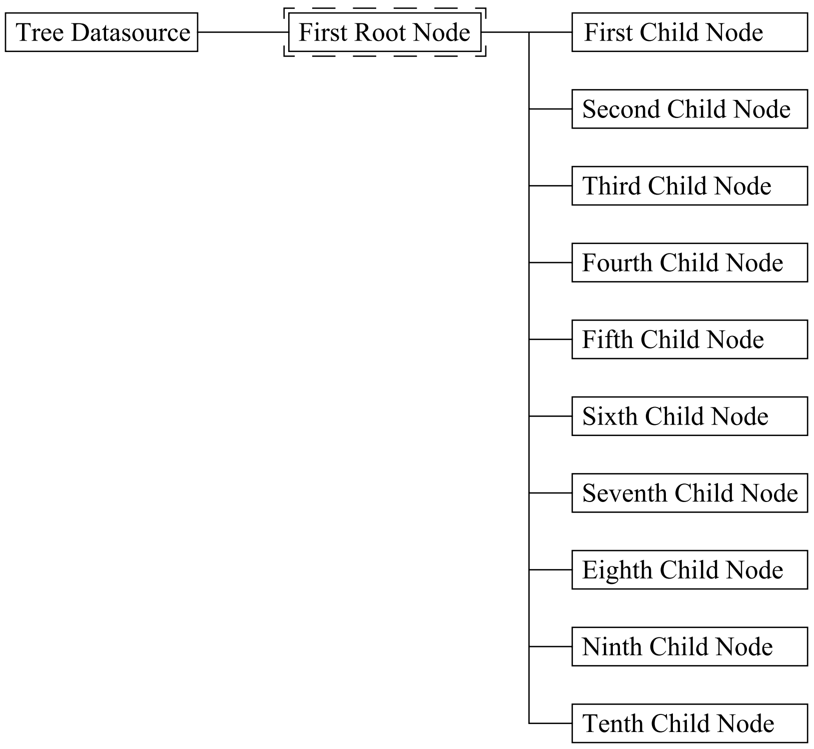


Figure 3–10 Creating the Next Sibling Node

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 39. Figure 3–11 shows the creation of a second root node, from which you can methodically build another tree:

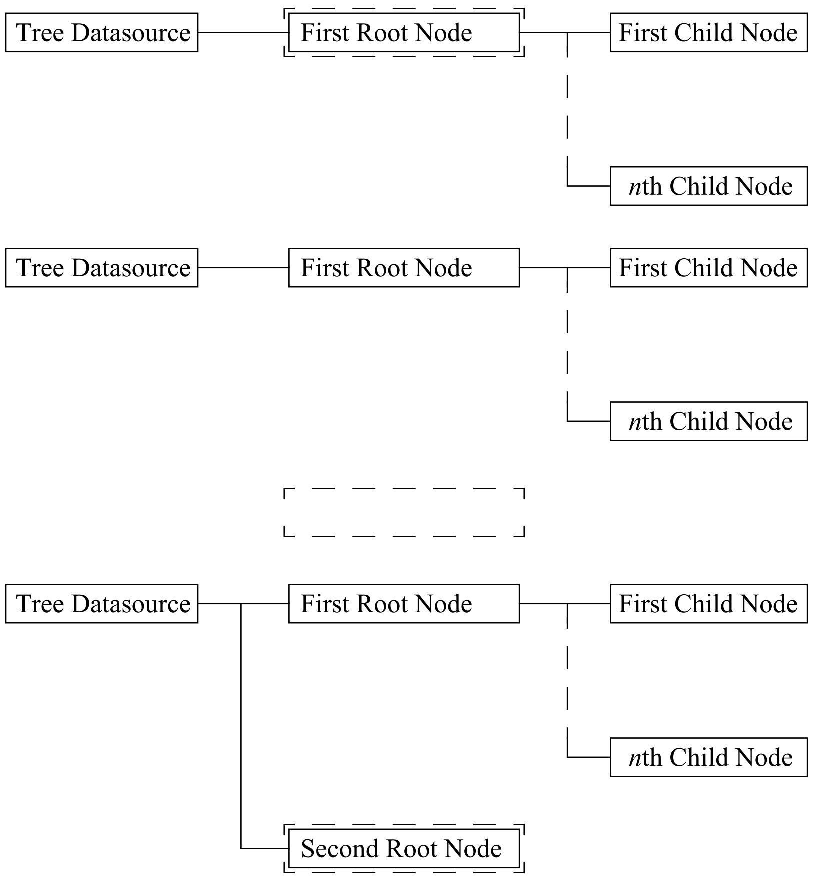


Figure 3–11 Creating Additional Trees

To create additional root nodes, move the node accessor to the first root node, then use the **VARTRNODEACCESSOR_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 treeDs;
VarTrNodeAccessorPtr nodeAccessor;
VarPtr varID, varValue;
VarTrEditPtr editTreeDs;

treeDs = VARTR_Create();
nodeAccessor = VARTRNODEACCESSOR_Create();
varID = VAR_New();
varValue = VAR_New();
editTreeDs = VARTR_StartEdit(treeDs);

/* Move the node accessor to the first unoccupied root-node
   location. */
VARTRNODEACCESSOR_GoFirstRoot(nodeAccessor);
while (VARTR_IsNodeValid(treeDs, nodeAccessor))
   VARTRNODEACCESSOR_GoNext(nodeAccessor);
}
VARTREDIT_AddNode(editTreeDs, nodeAccessor);

/* Set the node ID and Value properties. */
VAR_SetStr(varID, "1");
VAR_SetStr(varValue, "Second Root Node");
VARTREDIT_SetNodeID(editTreeDs, nodeAccessor, varID);
VARTREDIT_SetNodeValue(editTreeDs, nodeAccessor, varValue);
DSEDIT_((DsEditPtr)editTreeDs);
```

```
...
VAR_Delete(varID);
VAR_Delete(varValue);
VARTRNODEACCESSOR_Dispose(nodeAccessor);
RES_Release((ResPtr)treeDs);
```

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 treeDs;
VarTrNodeAccessorPtr nodeAccessor;
...
/* Assignments. */
treeDs = VARTR_Create();
nodeAccessor = VARTRNODEACCESSOR_Create();

/* Move the node accessor to the first unoccupied root-node
   location. */
VARTRNODEACCESSOR_GoNthRoot(nodeAccessor,
                            VARTR_GetNumRoots(treeDs));
...
```

In this case, the number of existing root nodes is used as an argument for the **VARTRNODEACCESSOR_GoNthRoot()** function. The value returned by the **VARTR_GetNumRoots()** function is a one-based counter, while the **VARTRNODEACCESSOR_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. */
   VarTrPtr treeDs;

   /* Declare an edit pointer variable. */
      VarTrEditPtr editTreeDs;

      /* Declare a node-accessor pointer variable. */
         VarTrNodeAccessorPtr nodeAccessor;
...
/* Assign a tree-datasource object to the tree-datasource
   pointer variable. */
   treeDs = VARTR_Create();

  /* Assign an edit object to the tree-edit pointer variable.
*/
      editTreeDs = VARTR_StartEdit(treeDs);

      /* Assign a node-accessor object to the node-accessor
         pointer variable. */
         nodeAccessor = VARTRNODEACCESSOR_Create();

         /* Position the node accessor and edit the tree. */
            ...
      /* Destroy the node-accessor object. */
         VARTRNODEACCESOR_Destruct(nodeAccessor);

      ...
   /* Commit the changes and destroy the edit object. */
      DSEDIT_End((DsEditPtr)editTreeDs);

   ...
/* Destroy the tree-datasource object. */
RES_Release((ResPtr)treeDs);
```

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—**DS_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. */
VarTrPtr treeDs;
...
/* Assign a tree-datasource object to the tree-datasource
   pointer variable. */
treeDs = VARTR_Create();
...
/* Destroy the tree-datasource object. */
RES_Release((ResPtr)treeDs);
```

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. */
VarTrPtr treeDs;

/* Declare a tree-edit pointer variable. */
VarTrEditPtr editTreeDs;

/* Declare three node-accessor pointer variables. */
VarTrNodeAccessorPtr nodeAccessor_1;
VarTrNodeAccessorPtr nodeAccessor_2;
VarTrNodeAccessorPtr nodeAccessor_3;
...
/* Assign a tree-datasource object to the tree-datasource
   pointer variable. */
treeDs = VARTR_Create();

/* Assign a node-accessor objects to the node-accessor
   pointer variables. */
nodeAccessor_1 = VARTRNODEACCESSOR_Create();
nodeAccessor_2 = VARTRNODEACCESSOR_Create();
nodeAccessor_3 = VARTRNODEACCESSOR_Create();

/* Move the node accessors and edit the tree datasource. */
...
/* Destroy each of the node accessors separately. */
VARTRNODEACCESOR_Dispose(nodeAccessor_1);
VARTRNODEACCESOR_Dispose(nodeAccessor_2);
VARTRNODEACCESOR_Dispose(nodeAccessor_3);
...
/* Destroy the tree-datasource object. */
RES_Release((ResPtr)treeDs);
```

You can also set aliases to a node accessor by assigning the node-accessor pointer returned by the **VARTR_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. */
VarTrPtr treeDs;

/* Declare multiple node-accessor pointer variables. */
VarTrNodeAccessorPtr nodeAccessor;
VarTrNodeAccessorPtr manager;
VarTrNodeAccessorPtr employee;
...
/* Assign a tree-datasource object to the tree-datasource
   pointer variable. */
treeDs = VARTR_Create();

/* Assign a node-accessor object to the node-accessor
   pointer variable. */
nodeAccessor = VARTRNODEACCESSOR_Create();

/* Set a cursor at the location of the node accessor. */
VARTR_SetCursor(treeDs, nodeAccessor);

/* Assign two node-accessor objects returned by the
   GetCursor() function to the extra node-accessor
   variables. */
manager = VARTR_GetCursor(treeDs);
employee = VARTR_GetCursor(treeDs);
...
/* Move the node accessor and aliases, and edit the tree
   datasource. */
...
/* Dispose of the node-accessor object. */
VARTRNODEACCESOR_Dispose(nodeAccessor);

/* Destroy the tree-datasource object. */
RES_Release((ResPtr)treeDs);
```

In the preceding example, only nodeAccessor must be destroyed, because manager and employee are only aliases to the node-accessor object, nodeAccessor. 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 34.

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 **DSEDIT_End()** function executes.

Note: See “Datasource Editing” on page 27 and “Node Editing” on page 27 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
`GoFirstRoot(accessor)`	Move *accessor* to the first root node.
`GoFirstChild(accessor)`	Move *accessor* to the first child node relative to the current *accessor* location.
`GoFirstSibling(accessor)`	Move *accessor* to the first sibling node of the current *accessor* location.
`GoParent(accessor)`	Move *accessor* to the parent node of the current *accessor* location.

Function	Description
`GoPrev(accessor)`	Move *accessor* to the previous sibling node of the current *accessor* location.
`GoNext(accessor)`	Move *accessor* to the next sibling node of the current *accessor* location.

The functions in Table 3–2 require the indexing system described in “Tree Datasource” on page 20. Note that nodes are indexed from 0.

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

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

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

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

```
VarTrPtr treeDs;
VarTrNodeAccessorPtr nodeAccessor;

/* Declare a datasource-level edit pointer. */
VarTrEditPtr editTreeDs;
...
treeDs = VARTR_Create();
nodeAccessor = VARTRNODEACCESSOR_Create();

/* Assign a datasource edit object to the edit pointer. */
editTreeDs = VARTR_StartEdit(treeDs);
...

/* Commit the edit operations, and destroy the edit object. */
DSEDIT_End((DsEditPtr)editTreeDs);
```

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 treeDs;
VarTrNodeAccessorPtr nodeAccessor;

/* Declare a datasource-level edit pointer. */
VarTrEditPtr editTreeDs;
...
treeDs = VARTR_Create();
nodeAccessor = VARTRNODEACCESSOR_Create();

/* Assign a node edit object to the edit pointer. */
editTreeDs = VARTR_StartNodeEdit(treeDs, nodeAccessor);
...
DSEDIT_End((DsEditPtr)editTreeDs);
...
```

Modifications are committed when you call the **DSEDIT_End()** function. This also releases the data lock that was created when the **VARTR_StartEdit()** function executed. If you call another **VARTR_StartEdit()** before the **DSEDIT_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 **VARTREDIT** API.

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

Function	Description
`SetTitle(treeDs)`	Sets the **Title** property of the tree datasource, *treeDs*.
`AddNode(accessor)`	Add a node at the current *accessor* location.
`RemoveNode(accessor)`	Remove the node at the current *accessor* location. Child nodes of the node to be removed become the children of the parent node of the removed node.
`RemoveTree(accessor)`	Remove the hierarchy beneath the current *accessor* 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 **VARTREDIT_SetTitle(***vartr*, *string***)** function. Likewise, you can retrieve the current title assigned to the tree datasource using the **VARTR_GetTitle(***vartr***)** 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.

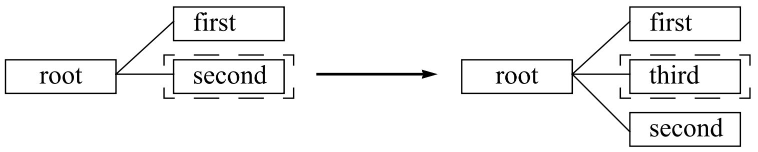


Figure 3–12 Inserting a Child Node

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

```
VarTrPtr treeDs;
VarTrNodeAccessorPtr nodeAccessor;
VarPtr varValue;

treeDs = VARTR_Create();
nodeAccessor = VARTRNODEACCESSOR_Create();
varValue = VAR_New();

/* Empty accessor location. */
VARTRNODEACCESSOR_GoNthRoot(VARTR_GetNumRoots(treeDs,
                                              nodeAccessor));

/* Add first root node. */
VARTR_AddNode(treeDs, nodeAccessor);
VAR_SetStr(varValue, "root");
VARTR_SetNodeValue(treeDs, nodeAccessor, varValue);

/* Empty accessor location. */
VARTRNODEACCESSOR_GoNthChild(VARTR_GetNumChildren(treeDs,
                                                  nodeAccessor));
```

```
/* Add first child node. */
VARTR_AddNode(treeDs, nodeAccessor);
VAR_SetStr(varValue, "first");
VARTR_SetNodeValue(treeDs, nodeAccessor, varValue);

/* Empty accessor location. */
VARTRNODEACCESSOR_GoNext(nodeAccessor);

/* Add second child node. */
VARTR_AddNode(treeDs, nodeAccessor);
VAR_SetStr(varValue, "second");
VARTR_SetNodeValue(treeDs, nodeAccessor, varValue);

/* Add the third child node at the "second" node location. */
VARTR_AddNode(treeDs, nodeAccessor);
VAR_SetStr(varValue, "third");
VARTR_SetNodeValue(treeDs, nodeAccessor, varValue);

VAR_Delete(varID);
VAR_Delete(varValue);
VARTRNODEACCESSOR_Dispose(nodeAccessor);
RES_Release((ResPtr)treeDs);
```

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 treeDs;
VarTrNodeAccessorPtr nodeAccessor;
VarPtr varValue;

treeDs = VARTR_Create();
nodeAccessor = VARTRNODEACCESSOR_Create();
varValue = VAR_New();

/* Empty root-level accessor location. */
VARTRNODEACCESSOR_GoNthRoot(VARTR_GetNumRoots(treeDs));

/* Add first root node. */
VARTR_AddNode(treeDs, nodeAccessor);
VAR_SetStr(varValue, "1st root");
VARTR_SetNodeValue(treeDs, nodeAccessor, varValue);

/* Empty root-level accessor location. */
VARTRNODEACCESSOR_GoNext(nodeAccessor);

/* Add second root node. */
VARTR_AddNode(treeDs, nodeAccessor);
VAR_SetStr(varValue, "2nd root");
VARTR_SetNodeValue(treeDs, nodeAccessor, varValue);
...
/* Empty root-level accessor location. */
VARTRNODEACCESSOR_GoNext(nodeAccessor);

/* Add nth root node. */
VARTR_AddNode(treeDs, nodeAccessor);
VAR_SetStr(varValue, "nth root");
VARTR_SetNodeValue(treeDs, nodeAccessor, varValue);
...
VAR_Delete(varID);
VAR_Delete(varValue);
VARTRNODEACCESSOR_Dispose(nodeAccessor);
RES_Release((ResPtr)treeDs);
```

Modifying Node Data Using the “Convenience” API

Using the “convenience” functions supplied with the **VARTREDIT** 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 **VARTREDIT_SetNodeID()** function. Likewise, you can set the value using the **VARTREDIT_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
`SetNodeID(editTreeDs, accessor, id)`	Set the **ID** property of the node at the current *accessor* location to *id.*
`SetNodeValue(editTreeDs, accessor, value)`	Set the **Value** property of the node at the current *accessor* location to *value.*

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 treeDs;
VarTrNodeAccessorPtr nodeAccessor;
VarTrEditPtr editTreeDs;
VarPtr varID, varValue;

treeDs = VARTR_Create();
nodeAccessor = VARTRNODEACCESSOR_Create();
varID = VAR_New();
varValue = VAR_New();
editTreeDs = VARTR_StartEdit(treeDs);

/* Create the first root node. */
VARTRNODEACCESSOR_GoFirstRoot(nodeAccessor);
VARTREDIT_AddNode(editTreeDs, nodeAccessor);

/* Go to the first child of the first root node. */
VARTRNODEACCESSOR_GoFirstChild(nodeAccessor);
VARTREDIT_AddNode(editTreeDs, nodeAccessor);

/* Set the node ID and node value. */
VAR_SetStr(varID, "0,0");
VAR_SetStr(varValue, "first child");
VARTREDIT_SetNodeID(editTreeDs, nodeAccessor, varID);
VARTREDIT_SetNodeValue(editTreeDs, nodeAccessor, varValue);
DSEDIT_End((DsEditPtr)editTreeDs);
...
VAR_Delete(varID);
VAR_Delete(varValue);
VARTRNODEACCESSOR_Dispose(nodeAccessor);
RES_Release((ResPtr)treeDs);
```

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.

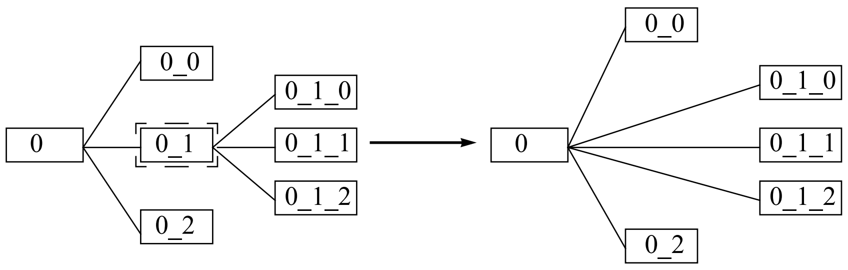


Figure 3–13 Removing a Node

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 treeDs;
VarTrNodeAccessorPtr nodeAccessor;
VarTrEditPtr editTreeDs;

treeDs = VARTR_Create();
nodeAccessor = VARTRNODEACCESSOR_Create();
editTreeDs = VARTR_StartEdit(treeDs);

/* Create the tree structure on the left side of
   Figure 3–13. */
...
/* Go to the second child node of the first root node. */
VARTRNODEACCESSOR_GoFirstRoot(nodeAccessor);
VARTRNODEACCESSOR_GoNthChild(nodeAccessor, 1);

/* Remove a node. */
VARTREDIT_RemoveNode(editTreeDs, nodeAccessor);
DSEDIT_End((DsEditPtr)editTreeDs);
...
VARTRNODEACCESSOR_Dispose(nodeAccessor);
RES_Release((ResPtr)treeDs);
```

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, **VARTREDIT_RemoveTree()** removes the entire tree. Figure 3–14 shows the nodes with labels that indicate their positions in the hierarchy:

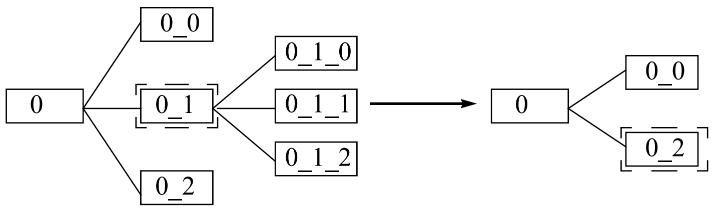


Figure 3–14 Removing a Tree

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 treeDs;
VarTrNodeAccessorPtr nodeAccessor;
VarTrEditPtr editTreeDs;

treeDs = VARTR_Create();
nodeAccessor = VARTRNODEACCESSOR_Create();
editTreeDs = VARTR_StartEdit(treeDs);

/* Create the tree structure on the left side of
   Figure 3–14. */
...
/* Go to the second child of the first root node. */
VARTRNODEACCESSOR_GoFirstRoot(nodeAccessor);
VARTRNODEACCESSOR_GoNthChild(nodeAccessor, 1);

/* Remove a tree. */
VARTREDIT_RemoveTree(editTreeDs, nodeAccessor);
DSEDIT_End((DsEditPtr)editTreeDs);
...
VARTRNODEACCESSOR_Dispose(nodeAccessor);
RES_Release((ResPtr)treeDs);
```

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 **VARTRNODEEDIT_SetID()** and **VARTRNODEEDIT_SetValue()** require you to declare a node edit object.

Table 3–6 Functions for Getting and Setting Node Data

Function	Description
`QueryNodeID(treeDs, accessor, idPtr)`	Copy the **ID** property of the node at the current *accessor* location to the address stored by *idPtr*.
`QueryNodeValue(treeDs, accessor, valuePtr)`	Copy the **Value** property of the node at the current *accessor* location to the address stored by *valuePtr*.
`GetNodeID(treeDs, accessor)`	Get a pointer to the variant object that stores the data of the **ID** property at the current *accessor* location.
`GetNodeValue(treeDs, accessor)`	Get a pointer to the variant object that stores the data of the **Value** property of the node at the current *accessor* location.
`SetID(editNode, id)`	Set the **ID** property of the node at the current *accessor* location to *id*.
`SetValue(editNode, value)`	Set the **Value** property of the node at the current *accessor* location to *value*.
`SetNodeID(editTreeDs, accessor, id)`	Set the **ID** property of the node at the current *accessor* location to *id*.
`SetNodeValue(editTreeDs, accessor, value)`	Set the **Value** property of the node at the current *accessor* location to *value*.
`IsNodeValid(treeDs, accessor)`	Get a boolean value indicating whether a node exists in the tree datasource at the current *accessor* location.

Setting the Node ID

You can:

- Get a copy of the node **ID** using **VARTR_QueryNodeID()**
- Get a pointer to the **ID** using **VARTR_GetNodeID()**
- Set the **ID** using **VARTRNODEEDIT_SetID()**

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

```
VarTrPtr treeDs;
VarTrNodeAccessorPtr nodeAccessor;
VarTrNodeEditPtr editNode;
VarPtr varID, varValue;

treeDs = VARTR_Create();
nodeAccessor = VARTRNODEACCESSOR_Create();
varID = VAR_New();
varValue = VAR_New();

/* Go to the first child of the first root node. */
VARTRNODEACCESSOR_GoFirstRoot(nodeAccessor);
VARTR_AddNode(treeDs, nodeAccessor)

VARTRNODEACCESSOR_GoFirstChild(nodeAccessor);
VARTR_AddNode(treeDs, nodeAccessor)

/* Set the node ID and node value. */
editNode = VARTR_StartNodeEdit(treeDs, nodeAccessor);
VAR_SetStr(varID, "0,0");
VAR_SetStr(varValue, "first child");
VARTRNODEEDIT_SetID(editTreeDs, nodeAccessor, varID);
VARTRNODEEDIT_SetValue(editTreeDs, nodeAccessor, varValue);
DSEDIT_End((DsEditPtr)editNode);
...
VAR_Delete(varID);
VAR_Delete(varValue);
VARTRNODEACCESSOR_Dispose(nodeAccessor);
RES_Release((ResPtr)treeDs);
```

Setting the Node Value

You can:

- Get a copy of the **Value** using **VARTR_QueryNodeValue()**
- Get a pointer to the **Value** using **VARTR_GetNodeValue()**
- Set the **Value** using **VARTRNODEEDIT_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 **VARTR** 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 **VARTR_GetNumRoots()** function returns the same value on each cycle of the loop, so the control criterion never evaluates **FALSE**.

```
VarTrPtr treeDs;
VarTrNodeAccessorPtr nodeAccessor;
VarTrEditPtr editTreeDs;
Int16 maxRoots;

treeDs = VARTR_Create();
nodeAccessor = VARTRNODEACCESSOR_Create();
editTreeDs = VARTR_StartEdit(treeDs);

/* Add nodes as long as the number of root nodes is less than
   maxRoots. */
maxRoots = 10;
while (VARTR_GetNumRoots(treeDs) < maxRoots) {
   VARTRNODEACCESSOR_GoNthRoot(VARTR_GetNumRoots(treeDs),
                                nodeAccessor);
   VARTREDIT_AddNode(editTreeDs, nodeAccessor);
}
...
DSEDIT_End((DsEditPtr)editTreeDs);
```

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 **VARTR_GetNumRoots()** function as a counter. In this example, the **VARTR_StartEdit()** and **DSEDIT_End()** functions are located, respectively, as the first and last executable lines of code within the **while** loop:

```
VarTrPtr treeDs;
VarTrNodeAccessorPtr nodeAccessor;
VarTrEditPtr editTreeDs;
Int16 maxRoots;

treeDs = VARTR_Create();
nodeAccessor = VARTRNODEACCESSOR_Create();
editTreeDs = VARTR_StartEdit(treeDs);

maxRoots = 10;
while (VARTR_GetNumRoots(treeDs) < maxRoots) {
   /* Create edit object. */
   editTreeDs = VARTR_StartEdit(treeDs);
   VARTRNODEACCESSOR_GoNthRoot(nodeAccessor,
                                VARTR_GetNumRoots(treeDs));
   VARTREDIT_AddNode(editTreeDs, nodeAccessor);
   /* Commit the edit operations and destroy the edit object. */
   DSEDIT_End((DsEditPtr)editTreeDs);
}
...
```

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

```
VarTrPtr treeDs;
VarTrNodeAccessorPtr nodeAccessor;
Int16 maxRoots;

treeDs = VARTR_Create();
nodeAccessor = VARTRNODEACCESSOR_Create();

maxRoots = 10;
while (VARTR_GetNumRoots(treeDs) < maxRoots) {
   VARTRNODEACCESSOR_GoNthRoot(nodeAccessor,
                               VARTR_GetNumRoots(treeDs));
   /* Create edit object, add a node, commit the edit operations
      and, destroy the edit object. */
   VARTR_AddNode(treeDs, nodeAccessor);
}
...
```

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 treeDs;
VarTrNodeAccessorPtr nodeAccessor;
VarTrEditPtr editTreeDs;
Int16 i, maxRoots;

treeDs = VARTR_Create();
nodeAccessor = VARTRNODEACCESSOR_Create();
editTreeDs = VARTR_StartEdit(treeDs);

i = 0;
maxRoots = 10;
VARTRNODEACCESSOR_GoFirstRoot(nodeAccessor);
while (VARTR_IsNodeValid(treeDs, nodeAccessor) &&
        (i < maxRoots))
   VARTREDIT_AddNode(editTreeDs, nodeAccessor);
   VARTRNODEACCESSOR_GoNext(nodeAccessor);
   i++;
}
...
DSEDIT_End((DsEditPtr)editTreeDs);
```

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 treeDs;
VarTrNodeAccessorPtr nodeAccessor;
VarPtr varID, varValue;
VarTrEditPtr editTreeDs;

treeDs = VARTR_Create();
nodeAccessor = VARTRNODEACCESSOR_Create();
varID = VAR_New();
varValue = VAR_New();

/* Using the "convenience" API, add a node at the next empty
   root-node location. */
VARTRNODEACCESSOR_GoNthRoot(nodeAccessor,
VARTR_GetNumRoots(treeDs));
VARTR_AddNode(treeDs, nodeAccessor);

/* Set variants and use them to set the node ID and Value
   properties. */
```

```
varID->SetStr("r0000");
varValue->SetStr("First Root Node");
VARTR_SetNodeID(treeDs, nodeAccessor, varID);
VARTR_SetNodeValue(treeDs, nodeAccessor, varValue);

/* Add nodes as long as the number of root nodes is less than
   maxNodes. */
Int16 maxNodes = 10;
while (VARTR_GetNumChildren(treeDs, nodeAccessor) < maxNodes)
{

/* Create a datasource edit object, and assign it to the edit
   pointer variable. */
   editTreeDs = VARTR_StartEdit(treeDs);

/* Descend from the parent to the next "empty" child-node
   location. */
   VARTRNODEACCESSOR_GoNthChild(nodeAccessor,
      VARTR_GetNumChildren(treeDs, nodeAccessor));

/* Add a node and set the node ID and Value properties. */
   VARTREDIT_AddNode(editTreeDs, nodeAccessor);
   varID->SetStr("c0000");
   varValue->SetStr("Child Node");
   VARTREDIT_SetNodeID(editTreeDs, nodeAccessor, varID);
   VARTREDIT_SetNodeValue(editTreeDs, nodeAccessor, varValue);

/* Commit the changes to the datasource, and destroy the edit
   object. */
   DSEDIT_End((DsEditPtr)editTreeDs);

/* Return the node accessor to the parent node. */
   VARTRNODEACCESSOR_GoParent(nodeAccessor);
}

/* Dispose of the accessor and datasource. */
VARTRNODEACCESSOR_Dispose(nodeAccessor);
RES_Release((ResPtr)treeDs);
```

In this case, the number of existing child nodes is used as an argument for the **VARTRNODEACCESSOR_GoNthChild()** function. The value returned by the **VARTR_GetNumChildren()** function is a one-based counter, while the **VARTRNODEACCESSOR_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
`SetCursor(treeDs, accessor)`	Set *accessor* as the datasource cursor.
`GetCursor(treeDs)`	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:

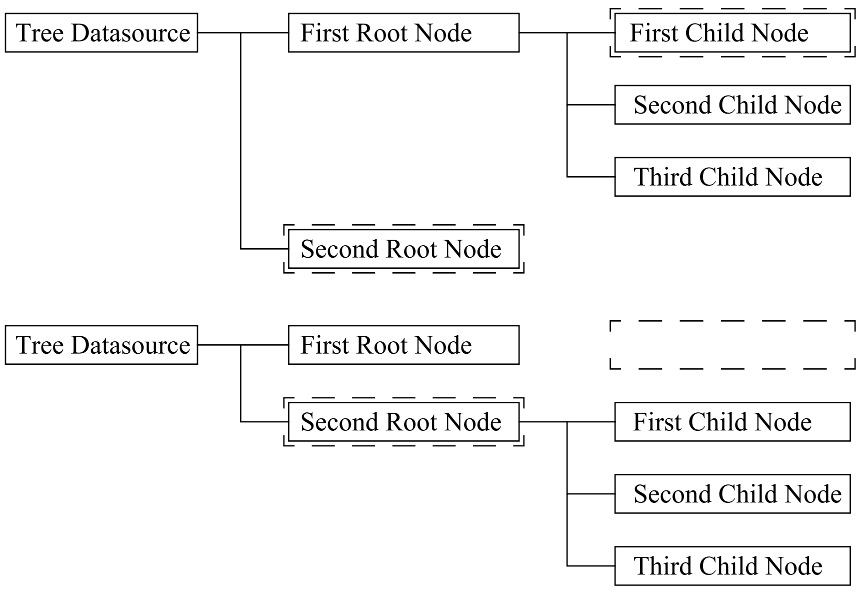


Figure 3–15 Assigning Child Nodes a New Parent Node

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

```
VarTrPtr treeDs;
VarTrNodeAccessorPtr nodeAccessorFrom;
VarTrNodeAccessorPtr nodeAccessorTo;
VarPtr varID, varValue;

treeDs = VARTR_Create();
nodeAccessorFrom = VARTRNODEACCESSOR_Create();
nodeAccessorTo = VARTRNODEACCESSOR_Create();
varID = VAR_New();
varValue = VAR_New();


/* Using the "convenience" APIs, create the initial hierarchy:
   one root root node with the three children and a second root
   node with no children. */
VARTRNODEACCESSOR_GoFirstRoot(nodeAccessorFrom);
VARTR_AddNode(treeDs, nodeAccessorFrom);
VAR_SetStr(varID, "0");
VAR_SetStr(varValue, "First Root Node");
VARTR_SetNodeID(treeDs, nodeAccessorFrom, varID);
VARTR_SetNodeValue(treeDs, nodeAccessorFrom, varValue);

VARTRNODEACCESSOR_GoFirstChild(nodeAccessorFrom);
VARTR_AddNode(treeDs, nodeAccessorFrom);
VAR_SetStr(varID, "0,0");
VAR_SetStr(varValue, "First Child Node");
VARTR_SetNodeID(treeDs, nodeAccessorFrom, varID);
VARTR_SetNodeValue(treeDs, nodeAccessorFrom, varValue);

VARTRNODEACCESSOR_GoNext(nodeAccessorFrom);
VARTR_AddNode(treeDs, nodeAccessorFrom);
```

```
VAR_SetStr(varID, "0,1");
VAR_SetStr(varValue, "Second Child Node");
VARTR_SetNodeID(treeDs, nodeAccessorFrom, varID);
VARTR_SetNodeValue(treeDs, nodeAccessorFrom, varValue);

VARTRNODEACCESSOR_GoNext(nodeAccessorFrom);
VARTR_AddNode(treeDs, nodeAccessorFrom);
VAR_SetStr(varID, "0,2");
VAR_SetStr(varValue, "Third Child Node");
VARTR_SetNodeID(treeDs, nodeAccessorFrom, varID);
VARTR_SetNodeValue(treeDs, nodeAccessorFrom, varValue);

VARTRNODEACCESSOR_GoNthRoot(nodeAccessorFrom,
                            VARTR_GetNumRoots(treeDs));
VARTR_AddNode(treeDs, nodeAccessorFrom);
VAR_SetStr(varID, "1");
VAR_SetStr(varValue, "Second Root Node");
VARTR_SetNodeID(treeDs, nodeAccessorFrom, varID);
VARTR_SetNodeValue(treeDs, nodeAccessorFrom, varValue);


/* Move the "From" accessor to the first root node. */
VARTRNODEACCESSOR_GoFirstRoot(nodeAccessorFrom);

/* Move the "To" accessor to the second root-node. */
VARTRNODEACCESSOR_GoNthRoot(nodeAccessorTo, 1);

/* Execute the while loop until all children of the first root
   node are relocated. */
while (VARTR_IsNodeValid(treeDs, nodeAccessorFrom)) {

   /* Move the "From" accessor to the FIRST child of the first
      root node. */
   VARTRNODEACCESSOR_GoFirstChild(nodeAccessorFrom);

   /* Move the "To" accessor to the LAST EMPTY child location of
      the second root node. */
   VARTRNODEACCESSOR_GoNthChild(nodeAccessorTo,
      mTreeDs->GetNumChildren(nodeAccessorTo));

   /* Add new child with ID and Value properties of the "From"
      node. */
   VARTR_AddNode(treeDs, nodeAccessorTo);
   VARTR_SetNodeID(treeDs, nodeAccessorTo,
                   VARTR_GetNodeID(treeDs, nodeAccessorFrom));
   VARTR_SetNodeValue(treeDs, nodeAccessorTo,
                   VARTR_GetNodeValue(treeDs, odeAccessorFrom));

   /* Remove the child node from the first root. */
   VARTR_RemoveNode(treeDs, nodeAccessorFrom);

   /* Move the "From" and "To" node accessors to the first and
      second root nodes, respectively. */
   VARTRNODEACCESSOR_GoParent(nodeAccessorFrom);
   VARTRNODEACCESSOR_GoParent(nodeAccessorTo);
}
VAR_Delete(varID);
VAR_Delete(varValue);
VARTRNODEACCESSOR_Dispose(nodeAccessorFrom);
VARTRNODEACCESSOR_Dispose(nodeAccessorTo);
RES_Release((ResPtr)treeDs);
```

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.

Chapter 4

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 a **DGRAM** view, which is supplied by the Open Interface Element. The Elements Environment *datasource/views* mechanism supports the interface between the datasource and the **DGRAM** view.

This chapter discusses these topics:

- Concepts
- Options for the DGRAM 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 **DGRAM** widget in the *Open Interface Element C User's Guide*. See Chapter 2 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 thee preceding concepts, which are then used in "Building a Graph Datasource" on page 117 to tell you how to program a graph datasource.

Graph Datasource

The *graph datasource*—an object of the **VARGR** 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 (**VARGR** object), your application can add and remove nodes and edges contained by the datasource object. You can also use the methods in the **VARGR** 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 67). 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

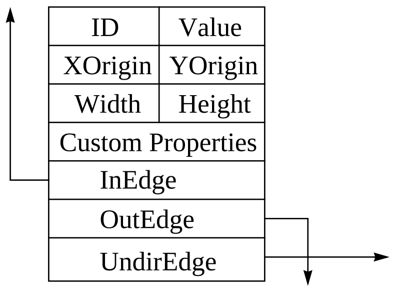


Figure 4–1 Structure of a Node

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:

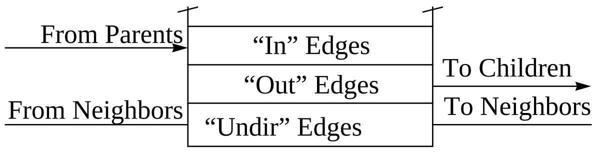


Figure 4–2 Edge-Reference Mechanisms

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 **VARGRNODEACCESSOR** 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 74.

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* (**VAR**) 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 **DGRAM** view. Specifically, **XOrigin** and **YOrigin** store the number of pixels from the left side and top of the **DGRAM** 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 **DGRAM** view.

If you do not set these properties, the size of the node is determined by the **DGRAM** 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 **DGRAM** view options, see “Options for the DGRAM View” on page 84.

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 **VARGR** and **VARGREDIT** APIs. For more information, see “Custom Node Properties” on page 103.

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 64). 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*th+1 neighbor nodes

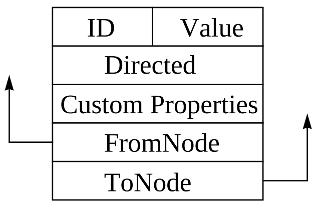


Figure 4–3 Edge Structure

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.

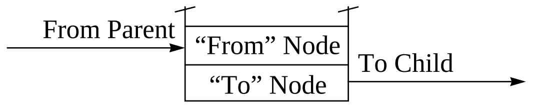


Figure 4–4 Node-Reference Mechanisms for Parent-Child Nodes

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.

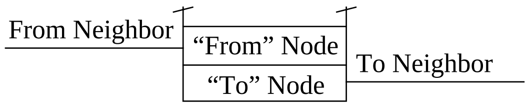


Figure 4–5 Node-Reference Mechanisms for Neighbor Nodes

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.

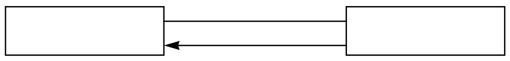


Figure 4–6 Not Supported: Combining Directed and Undirected Edges

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 72.

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.

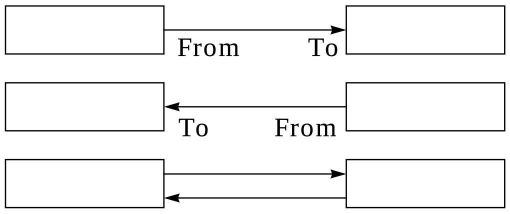


Figure 4–7 Single and Multiple Directed Edges

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(***vargr*, *source*, *target***)** function, and add another directed edge using the **AddDirEdge(***vargr*, *source*, *target***)** function from either the **VARGR** or **VARGREDIT** 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(***vargr*, *node1*, *node2***)** function from either the **VARGR** or the

VARGREDIT 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 73.

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.

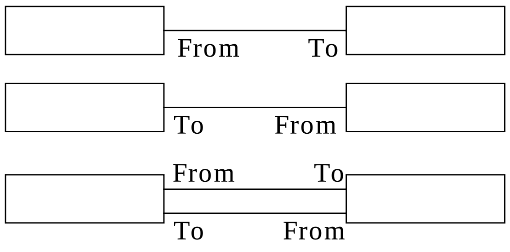


Figure 4–8 Single and Multiple Undirected Edges

For example, when traversing neighboring nodes, the **GoNext()** and **GoPrev()** functions from the **VARGRNODEACCESSOR** 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(***vargr*, *node1*, *node2***)** function, and add another undirected edge using the **AddUndirEdge(***vargr*, *node1*, *node2***)** function from the **VARGR** API. If you need to change several edges, use the functions from the **VARGREDIT** 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 **VARGR** and **VARGREDIT** APIs. For more information, see “Custom Link Properties” on page 110.

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 **VARGRNODEACCESSOR** 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.

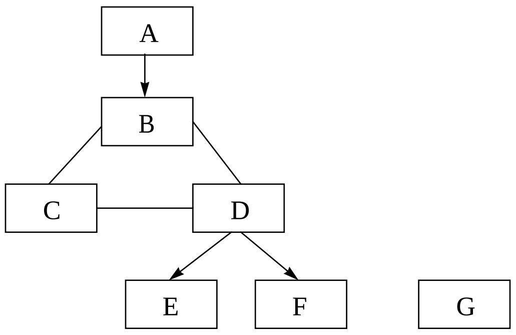


Figure 4–9 Node Relationships in a Graph

For more information about nodes, see “Node” on page 64. For more information about edges, see “Edge” on page 67.

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.

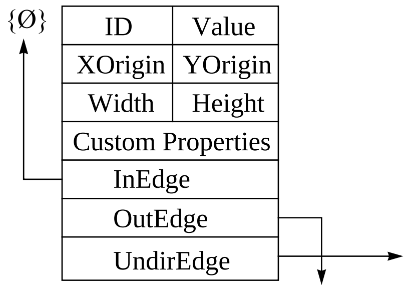


Figure 4–10 Unique Characteristics of a Root Node

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.

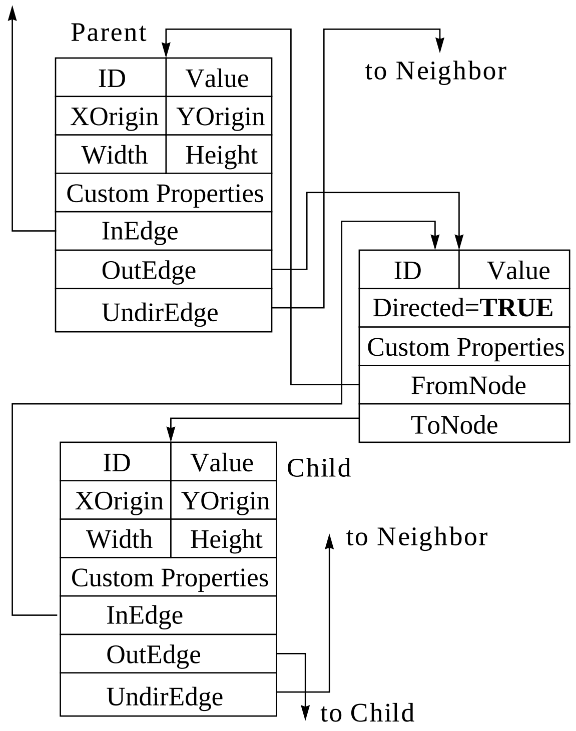


Figure 4–11 Parent-Child Node Relationship

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

The **GoNext(***node***)** 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(*node*)** function.
3. Using either the **VARGR** or **VARGREDIT** API, execute the **AddNode(*node*)** function.

Because a node can have multiple parents, you can use the **GoNext(*node*)** and **GoPrev(*node*)** 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.

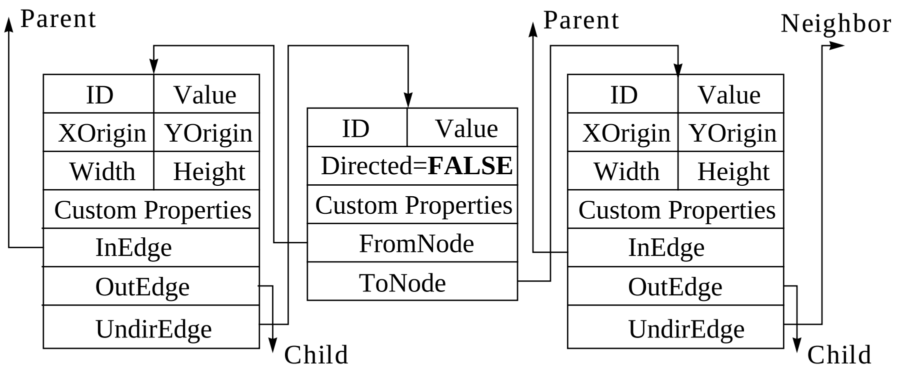


Figure 4–12 Neighbor Node Relationship

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

The **GoNext(*node*)** 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:

1. Position the accessor on a valid node.
2. Execute the **GoNext(*node*)** function.
3. Using the **VARGR** or **VARGREDIT** API, execute the **AddNode(*node*)** 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 76.

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 **VARGRNODEACCESSOR** 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:

```
/* Declare pointer variables. */
VarGrPtr graphDs;

/* Declare node-accessor pointers. */
VarGrNodeAccessorPtr nodeAccessorFrom;
VarGrNodeAccessorPtr nodeAccessorTo;
...

graphDs = VARGR_Create();

/* Create node accessors. */
nodeAccessorFrom = VARGRNODEACCESSOR_Create();
nodeAccessorTo = VARGRNODEACCESSOR_Create();
...
/* Destroy the node accessor. */
VARTRNODEACCESSOR_Dispose(nodeAccessor);
...
/* Destroy the graph datasource. */
RES_Release((ResPtr)graphDs);
```

The preceding code fragment declares two accessors—nodeAccessorFrom and nodeAccessorTo—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:

- **VARGRINEDGEACCESSOR**
- **VARGROUTEDGEACCESSOR**
- **VARGRUNDIREDGEACCESSOR**

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 **VARGRALLEDGEACCESSOR** 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:

```
/* Declare pointer variables. */
VarGrPtr graphDs;
VarPtr varValue;

/* Declare a datasource edit pointer. */
VarGrNodeEditPtr editNode;

/* Declare two node accessors for the two nodes at either end of
   the edge to be added. */
VarGrNodeAccessorPtr nodeAccessorFrom;
VarGrNodeAccessorPtr nodeAccessorTo;

/* Declare an undirected edge-accessor pointer. */
VarGrUndirEdgePtr undirEdgeAccessor;
...
graphDs = VARGR_Create();
editNode = VARGR_StartEdit(graphDs);
varValue = VAR_New();

/* Create two node-accessors, and assign them to
   nodeAccessorFrom and nodeAccessorTo. */
nodeAccessorFrom = VARGRNODEACCESSOR_Create();
nodeAccessorTo = VARGRNODEACCESSOR_Create();

/* Create an undirected edge accessor based on the node accessed
   by nodeAccessorFrom; assign it to undirEdgeAccessor. */
undirEdgeAccessor =
        VARGRUNDIREDGEACCESSOR_Create(nodeAccessorFrom);

/* Create several root nodes. */
...
/* Move nodeAccessorFrom to the first root node, and move
   nodeAccessorTo to the second root node. */
VARGRNODEACCESSOR_GoFirstRoot(nodeAccessorFrom);
VARGRNODEACCESSOR_GoFirstRoot(nodeAccessorTo);
VARGRNODEACCESSOR_GoNext(nodeAccessorTo);

/* Add an undirected edge between the nodes accessed by
   nodeAccessorFrom and nodeAccessorTo. */
```

```
VARGREDIT_AddUndirEdge(editGraphDs,
                            nodeAccessorFrom, nodeAccessorTo);

/* Use undirEdgeAccessor to set the Value property of the new
   edge to "Neighbor." */
VAR_SetStr(varValue, "Neighbor");
VARGREDIT_SetEdgeValue(editGraphDs,
                       undirEdgeAccessor, varValue);
VAR_Delete(varValue);
DSEDIT_End((DsEditPtr)editGraphDs);
```

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 **nodeAccessorFrom** 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 **DGRAM** 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 **VARGR_SetNodeCursor(***vargr*, *nodeaccessor***)** function.
- Access the node at the current cursor location using the **VARGR_GetNodeCursor(***vargr***)** function.

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

```
/* Declare a graph-datasource pointer variable. */
VarGrPtr graphDs;

/* Declare a browser pointer variable. */
DGramPtr win->dgramWgt;
WinPtr winDGram;

/* Declare a node-accessor pointer variable. */
VarGrNodeAccessorPtr nodeAccessor;

/* Declare two variant pointer variables. */
VarPtr varID, varValue;
...
/* Create the node and edge accessors. */
win->dgramWgt =
   (DGramPtr)PANEL_GetNamedWgt((PanelPtr)win, "DGram");
```

```
/* Assign a graph-datasource object to the graph-datasource
   pointer variable. */
graphDs = VARGR_Create();

/* Register the diagrammer with the graph datasource, and set
   the "cursor" view option. */
DS_RegisterView((DsPtr)graphDs, (ResPtr)win->dgramWgt);
DS_SetViewOptions((DsPtr)graphDs, (ResPtr)win->dgramWgt,
                  "cursor", "CONTROLS");

/* Create node and edge accessors. */
nodeAccessor = VARGRNODEACCESSOR_Create();

/* Set a cursor at the location of the node accessor. */
VARGR_SetNodeCursor(graphDs, nodeAccessor);

/* Position the node accessor. */
...
/* Use "convenience" API functions to edit the ID and Value
   properties of the node at the current cursor location. */
VAR_SetStr(varID, "0000");
VAR_SetStr(varValue, "Node");
VARTR_SetNodeID(graphDs, VARTR_GetNodeCursor(graphDs), varID);
VARTR_SetNodeValue(graphDs,
                   VARTR_GetNodeCursor(graphDs), varValue);
...
// Destroy the variant objects.
VAR_Delete(varID);
VAR_Delete(varValue);
...
/* Destroy the node accessor. */
VARTRNODEACCESOR_Dispose(nodeAccessor);
...
/* Destroy the graph datasource. */
RES_Release((ResPtr)graphDs);
```

For information about setting the cursor behavior, see "Options for the DGRAM View" on page 84.

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 **VARGR_SetEdgeCursor(***vargr*, *edgeaccessor***)** function.
- Access the node at the current cursor location using the **VARGR_GetEdgeCursor(***vargr***)** function.

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

```
/* Declare a graph-datasource pointer variable. */
VarGrPtr graphDs;

/* Declare a browser pointer variable. */
DGramPtr win->dgramWgt;

/* Declare a node-accessor pointer variable. */
VarGrAllEdgeAccessorPtr allEdgeAccessor;

/* Declare two variant pointer variables. */
VarPtr varID, varValue;
...
/* Create the node and edge accessors. */
win->dgramWgt =
   (DGramPtr)PANEL_GetNamedWgt((PanelPtr)win, "DGram");

/* Assign a graph-datasource object to the graph-datasource
   pointer variable. */
graphDs = VARGR_Create();
```

```
/* Register the diagrammer with the graph datasource, and set
   the "cursor" view option. */
DS_RegisterView((DsPtr)graphDs, (ResPtr)win->dgramWgt);
DS_SetViewOptions((DsPtr)graphDs, (ResPtr)win->dgramWgt,
                  "cursor", "CONTROLS");

/* Create node and edge accessors. */
allEdgeAccessor = VARGRALLEDGEACCESSOR_Create();

/* Set a cursor at the location of the edge accessor. */
VARGR_SetEdgeCursor(graphDs, allEdgeAccessor);

/* Position the edge accessor. */
...
/* Use "convenience" API functions to edit the ID and Value
   properties of the edge at the current cursor location. */
VAR_SetStr(varID, "0000");
VAR_SetStr(varValue, "Edge");
VARTR_SetEdgeID(graphDs, VARTR_GetEdgeCursor(graphDs), varID);
VARTR_SetEdgeValue(graphDs, VARTR_GetEdgeCursor(graphDs),
varValue);
...
// Destroy the variant objects.
VAR_Delete(varID);
VAR_Delete(varValue);
...
/* Destroy the node accessor. */
VARGRALLEDGEACCESSOR_Dispose(allEdgeAccessor);
...
/* Destroy the graph datasource. */
RES_Release((ResPtr)graphDs);
```

For information about setting the cursor behavior, see “Options for the DGRAM View” on page 84.

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 80.

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 a **VARGR** object executes the **StartEdit()** function. This is a public function inherited from the **DS** class. The graph datasource is unlocked when the **DSEDIT_End()** function executes, as shown in this example:

```
/* Declare pointer variables. */
VarGrPtr graphDs;
/* Declare node and edge accessors. */
VarGrNodeAccessorPtr nodeAccessor;
VarGrAllEdgeAccessorPtr allEdgeAccessor;
VarGrEditPtr editGraphDs;

/* Assign addresses to pointers. */
graphDs = VARGR_Create();
/* Assign node and edge accessors. */
nodeAccessor = VARGRNODEACCESSOR_Create();
allEdgeAccessor = VARGRALLEDGEACCESSOR_Create();
/* Create the datasource-level edit object. */
editGraphDs = VARGR_StartEdit(graphDs);
/* Position node accessor and edit the graph. */
...
/* Execute the DSEDIT_End() function. */
DSEDIT_End((DsEditPtr)editGraphDs);
```

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 **VARGR** class executes the **StartNodeEdit(***vargr, accessor***)** function. This is a public function inherited from the **DS** class. The accessed node is unlocked when the **DSEDIT_End()** function executes, as shown in this example:

```
VarGrPtr graphDs;
VarGrNodeAccessorPtr nodeAccessor;
VarGrNodeEditPtr editNode;
...
graphDs = VARGR_Create();
accessor = VARGRNODEACCESSOR_Create();

/* Execute the StartNodeEdit() function. */
editNode = VARGR_StartNodeEdit(graphDs, nodeAccessor);

/* Position the node accessor and edit node data. */
```

```
...

/* Execute the DSEDIT_End() function. */
DSEDIT_End((DsEditPtr)editNode);
```

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 **VARGR** class executes the **StartEdgeEdit(***vargr*, *accessor***)** function. This is a public function inherited from the **DS** class. The accessed node is unlocked when the **DSEDIT_End()** function executes, as shown in this example:

```
VarGrPtr graphDs;
VarGrAllEdgeAccessorPtr allEdgeAccessor;
VarGrEdgeEditPtr editEdge;
graphDs = VARGR_Create();
accessor = VARGRALLEDGEACCESSOR_Create();

/* Execute the StartEdgeEdit() function. */
editEdge = VARGR_StartEdgeEdit(graphDs, allEdgeAccessor);

/* Position the node accessor and edit edge data. */
...

/* Execute the DSEDIT_End() function. */
DSEDIT_End((DsEditPtr)editNode);
```

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 **VARGR_AddNode(***vargr*, *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, **VARGR_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. */
VarGrPtr graphDs;
VarGrNodeAccessorPtr nodeAccessor;

/* Initialize the pointer variables. */
graphDs = VARGR_Create();
nodeAccessor = VARGRNODEACCESSOR_Create();

/* Move the node accessor to the next empty root-node
   location. */
VARGRNODEACCESSOR_GoFirstRoot(nodeAccessor);
while (VARGR_IsNodeValid(graphDs, nodeAccessor)) {
   VARGRNODEACCESSOR_GoNext(nodeAccessor);
}

/* 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 VARGR_AddNode() function. */
VARGR_AddNode(graphDs, nodeAccessor);
...
/* Dispose of other objects. */
VAR_Delete(varID);
VAR_Delete(varValue);
VARGRNODEACCESSOR_Dispose(nodeAccessor);
RES_Release((ResPtr)graphDs);
```

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 79.

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, **VARGR_SetNodeID()** and **VARGR_SetNodeValue()**. Each of these functions creates a node edit object, commits its edit operation, and destroys the edit object.

```
/* Declare pointer variables. */
VarGrPtr graphDs;
VarGrNodeAccessorPtr nodeAccessor;
```

```
VarPtr varID, varValue;

/* Initialize the pointer variables. */
graphDs = VARGR_Create();
nodeAccessor = VARGRNODEACCESSOR_Create();
varID = VAR_New();
varValue = VAR_New();

/* Move the node accessor to the next empty root-node
   location. */
VARGRNODEACCESSOR_GoFirstRoot(nodeAccessor);
while (VARGR_IsNodeValid(graphDs, nodeAccessor)) {
   VARGRNODEACCESSOR_GoNext(nodeAccessor);
}

/* 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 VARGR_AddNode() function. */
VARGR_AddNode(graphDs, nodeAccessor);

/* Set the variant objects to some initializing values. */
VAR_SetStr(varID, "0000");
VAR_SetStr(varValue, "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. */
VARGR_SetNodeID(graphDs, editGraphDs, varID);
VARGR_SetNodeValue(graphDs, editGraphDs, varValue);
...
/* Dispose of other objects. */
VAR_Delete(varID);
VAR_Delete(varValue);
VARGRNODEACCESSOR_Dispose(nodeAccessor);
RES_Release((ResPtr)graphDs);
```

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 79.

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 create by each of the convenience API functions, **VARGR_SetEdgeID()** and **VARGR_SetEdgeValue()**. Each of these functions creates a edge edit object, commits its edit operation, and destroys the edit object.

```
/* Declare pointer variables. */
VarGrPtr graphDs;
VarGrNodeAccessorPtr nodeAccessorFrom;
VarGrNodeAccessorPtr nodeAccessorTo;
VarGrAllEdgeAccessorPtr allEdgeAccessor;
VarPtr varID, varValue;

/* Initialize the pointer variables. */
graphDs = VARGR_Create();
nodeAccessorFrom = VARGRNODEACCESSOR_Create();
nodeAccessorTo = VARGRNODEACCESSOR_Create();
allEdgeAccessor = VARGRALLEDGEACCESSOR_Create();
varID = VAR_New();
varValue = VAR_New();
```

```
/* Move the "source" node accessor to the next empty root-node
   location. */
VARGRNODEACCESSOR_GoFirstRoot(nodeAccessorFrom);
while (VARGR_IsNodeValid(graphDs, nodeAccessorFrom)) {
   VARGRNODEACCESSOR_GoNext(nodeAccessorFrom);
}

/* 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 VARGR_AddNode() function. */
VARGR_AddNode(graphDs, nodeAccessorFrom);

/* Set the variant objects to some initializing values. */
VAR_SetStr(varID, "n0000");
VAR_SetStr(varValue, "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. */
VARGR_SetNodeID(graphDs, nodeAccessorFrom, varID);
VARGR_SetNodeValue(graphDs, nodeAccessorFrom, varValue);

/* Move the node accessor to the next empty root-node
   location. */
VARGRNODEACCESSOR_GoNthRoot(VARGR_GetNumRoots(graphDs),
                            nodeAccessorTo);
while (VARGR_IsNodeValid(graphDs, nodeAccessorTo)) {
   VARGRNODEACCESSOR_GoNext(nodeAccessorTo);
}

/* 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 VARGR_AddNode() function. */
VARGR_AddNode(graphDs, nodeAccessorTo);

/* 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. */
VARGR_SetNodeID(graphDs, nodeAccessorTo, varID);
VARGR_SetNodeValue(graphDs, nodeAccessorTo, varValue);

/* 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 VARGR_AddEdge() function. */
VARGR_AddDirEdge(graphDs,
                 nodeAccessorFrom, nodeAccessorTo);

/* Set the variant objects to some initializing values. */
VAR_SetStr(varID, "e0000");
VAR_SetStr(varValue, "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. */
VARGRALLEDGEACCESSOR_GoBetween(allEdgeAccessor,
                               nodeAccessorFrom,
                               nodeAccessorTo);
VARGR_SetEdgeID(graphDs, allEdgeAccessor, varID);
VARGR_SetEdgeValue(graphDs, allEdgeAccessor, varValue);
...
/* Dispose of other objects. */
VAR_Delete(varID);
VAR_Delete(varValue);
VARGRNODEACCESSOR_Dispose(nodeAccessorFrom);
VARGRNODEACCESSOR_Dispose(nodeAccessorTo);
VARGRNODEACCESSOR_Dispose(allEdgeAccessor);
RES_Release((ResPtr)graphDs);
```

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 80.

Options for the DGRAM View

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

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

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

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

```
DS_SetViewOption(<datasource>, (ResPtr)<window>-><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 4–1 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>`
		DGRAM-specific

`Font`	`Font.<font_name>`
	`<module_name>.<font_name>`
		DGRAM-specific

`Color`	`Color.<color_name>`
	`<module_name>.<color_name>`
		DGRAM-specific

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

```
DS_SetViewOption(graphDs, (ResPtr)win->dgramWgt,
                "Diagrammer",
                "NodeW = 72; NodeH = 18;
                 XGrid = 36; YGrid = 36;
                 GridAlignment = TRUE;
                 BgColor = Color.Black;

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

                 Orientation = VERTICAL;
                 LinkStandardDData.Shape = RIGHTANGLE;
                 LinkStandardDData.LinkDirColor = Color.Blue;
                 LinkFocusDData.LinkDirColor = Color.Red;
                 LinkStandardDData.LinkUndirColor = Color.Navy;
                 LinkFocusDData.LinkUndirColor = Color.Maroon;
                 LinkStandardDData.LinkPen = Pen.Solid;
                 LinkStandardDData.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:

```
DS_SetViewOption(<datasource>, (ResPtr)<window>-><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 graphDs;
DGramPtr dgramWgt;
```

```
...
graphDs = VARGR_Create();
win->dgramWgt =
   (DGramPtr)PANEL_GetNamedWgt((PanelPtr)win, "DGram");
DS_RegisterView((DsPtr)graphDs, (ResPtr)win->dgramWgt)
DS_SetViewOption((DsPtr)graphDs, (ResPtr)win->dgramWgt,
                 "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:

```
DS_SetViewOption(<datasource>, (ResPtr)<window>-><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 **DGRAM** views, dgram1 and dgram2, registered to a common graph datasource, graphDs. The dgram1 cursor **reflects** the current location of the datasource cursor. The dgram2 cursor **controls** the position of the datasource cursor:

```
VarGrPtr graphDs;
WinPtr winDGram;
DGramPtr win->dgram1;
DGramPtr win->dgram2;
...
win->dgram1 =
   (DGramPtr)PANEL_GetNamedWgt((PanelPtr)win, "DGram");
win->dgram2 =
   (DGramPtr)PANEL_GetNamedWgt((PanelPtr)win, "DGram");
DS_RegisterView((DsPtr)graphDs, (ResPtr)win->dgram1)
DS_RegisterView((DsPtr)graphDs, (ResPtr)win->dgram2)
...
DS_SetViewOption((DsPtr)graphDs, (ResPtr)win->dgram1,
                 "cursor", "REFLECTS");
DS_SetViewOption((DsPtr)graphDs, (ResPtr)win->dgram2,
                 "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 graphDs;
DGramPtr win->dgram1;
DGramPtr win->dgram2;
...
```

```
win->dgram1 =
   (DGramPtr)PANEL_GetNamedWgt((PanelPtr)win, "DGram");
win->dgram2 =
   (DGramPtr)PANEL_GetNamedWgt((PanelPtr)win, "DGram");
DS_RegisterView((DsPtr)graphDs, (ResPtr)win->dgram1)
DS_RegisterView((DsPtr)graphDs, (ResPtr)win->dgram2)
...
DS_SetViewOption((DsPtr)graphDs, (ResPtr)win->dgram1,
                   "cursor", "CONTROLS");
DS_SetViewOption((DsPtr)graphDs, (ResPtr)win->dgram2,
                   "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.

```
DS_SetViewOption(<datasource>, (ResPtr)<window>-><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:

```
DS_SetViewOption(<datasource>, (ResPtr)<window>-><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:

```
DS_SetViewOption(<datasource>, (ResPtr)<window>-><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:

```
DS_SetViewOption(<datasource>, (ResPtr)<winres>-><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:

```
DS_SetViewOption(graphDs, (ResPtr)win->dgramWgt,
                 "Diagrammer", "NodeW = 72");
```

Type: Int32

Default: **DGRAM**-specific

Synonyms: **Width**

NodeH

This integer value specifies the width 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:

```
DS_SetViewOption(<datasource>, (ResPtr)<winres>-><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:

```
DS_SetViewOption(graphDs, (ResPtr)win->dgramWgt,
                 "Diagrammer", "NodeH = 24");
```

Type: Int32

Default: **DGRAM**-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:

```
DS_SetViewOption(<datasource>, (ResPtr)<window>-><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:

```
DS_SetViewOption(graphDs, (ResPtr)win->dgramWgt,
                 "Diagrammer", "XGrid = 36");
```

Type: Int32

Default: **DGRAM**-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:

```
DS_SetViewOption(<datasource>, (ResPtr)<window>-><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:

```
DS_SetViewOption(graphDs, (ResPtr)win->dgramWgt,
                 "Diagrammer", "YGrid = 36");
```

Type: Int32

Default: **DGRAM**-specific

Synonyms: None

GridAlignment

With **GridAlignment** set to **FALSE** (the default), you can move nodes and links in an unconstrained manner within the **DGRAM** view. If you set **GridAlignment** to **TRUE**, the diagram contents snap to the grid. This syntax shows how to set the **GridAlignment** parameter:

```
DS_SetViewOption(<datasource>, (ResPtr)<window>-><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:

```
DS_SetViewOption(graphDs, (ResPtr)win->dgramWgt,
                 "Diagrammer", "YGrid = 36; YGrid = 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:

```
DS_SetViewOption(<datasource>, (ResPtr)<window>-><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:

```
DS_SetViewOption(graphDs, (ResPtr)win->dgramWgt,
                 "Diagrammer", "ScaleFactor = 2.00");
```

Type: Double

Default: 1.00

Synonyms: **Scale**

Overview

The **DGRAM** 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:

```
DS_SetViewOption(<datasource>, (ResPtr)<window>-><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:

```
DS_SetViewOption(graphDs, (ResPtr)win->dgramWgt,
                 "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:

```
DS_SetViewOption(<datasource>, (ResPtr)<window>-><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:

```
DS_SetViewOption(graphDs, (ResPtr)win->dgramWgt,
                 "Diagrammer", "Orientation = 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:

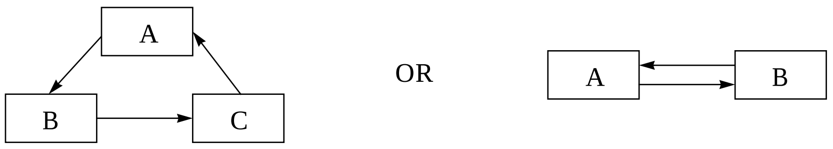


Figure 4–13 Cyclic Node Reference

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:

```
DS_SetViewOption(<datasource>, (ResPtr)<window>-><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 **DGRAM** view to display mapping information, such as floor plans for a computer network. This syntax shows how to set the **BitmapFile** parameter:

```
DS_SetViewOption(<datasource>, (ResPtr)<window>-><view>,
                 "Diagrammer",
                 "...;
                  BitmapFile = <filename>;
                  ..."
                );
```

This code assigns the filename “network.gif” to the **BitmapFile** parameter:

```
DS_SetViewOption(graphDs, (ResPtr)win->dgramWgt,
                 "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:

```
DS_SetViewOption(<datasource>, (ResPtr)<window>-><view>,
                 "Diagrammer",
                 "...;
                  BgColor = <color_resource>;
                  ..."
                );
```

This code assigns the color resource “res.blue” to the **BgColor** parameter:

```
DS_SetViewOption(graphDs, (ResPtr)win->dgramWgt,
                 "Diagrammer", "BgColor = res.blue");
```

Type: Color

Default: **DGRAM**-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:

```
DS_SetViewOption(<datasource>, (ResPtr)<window>-><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>;] |
                 [LinkStandardDData.LinkDirColor =
                              <color_resource>;] |
                 [LinkStandardDData.LinkUndirColor =
                              <color_resource>;] |
                 [LinkStandardDData.LabelColor =
                              <color_resource>;] |
                 [LinkStandardDData.LinkPen =
                              <pen_resource>;] |
                 [LinkStandardDData.LabelFont =
                              <font_resource>;] |
                 [LinkStandardDData.DrawLabel = {FALSE|TRUE};] |
                 [LinkStandardDData.Shape =
                              {0|DEFAULT|DIAGONAL|
                               1|RIGHTANGLE
                              };]"
                );
```

NodeStandardDData.FrameColor

NodeStandardDData.FrameColor sets the color of the frame around the node. This syntax shows how to set the **NodeStandardDData.FrameColor** parameter:

```
DS_SetViewOption(<datasource>, (ResPtr)<window>-><view>,
                 "Diagrammer",
                 "...;
                  NodeStandardDData.FrameColor =
                                           <color_resource>;
                  ..."
                );
```

Type: Color

Default: **DGRAM**-specific

Synonyms: **Node.FColor**, **Node.FrameColor**

NodeStandardDData.BgColor

NodeStandardDData.BgColor sets the color of the node. This syntax shows how to set the **NodeStandardDData.BgColor** parameter:

```
DS_SetViewOption(<datasource>, (ResPtr)<window>-><view>,
                 "Diagrammer",
                 "...;
                  NodeStandardDData.BgColor = <color_resource>;
                  ..."
                );
```

Type: Color

Default: **DGRAM**-specific

Synonyms: **Node.Bg**, **Node.Color**, **Node.BgColor**

NodeStandardDData.LabelColor

NodeStandardDData.LabelColor sets the color of the node label. This syntax shows how to set the **NodeStandardDData.LabelColor** parameter:

```
DS_SetViewOption(<datasource>, (ResPtr)<window>-><view>,
                 "Diagrammer",
                 "...;
                  NodeStandardDData.LabelColor =
                                         <color_resource>;
                  ..."
                );
```

Type: Color

Default: **DGRAM**-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:

```
DS_SetViewOption(<datasource>, (ResPtr)<window>-><view>,
                 "Diagrammer",
                 "...;
                  NodeStandardDData.FramePen = <pen_resource>;
                  ..."
                );
```

Type: Pen

Default: **DGRAM**-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:

```
DS_SetViewOption(<datasource>, (ResPtr)<window>-><view>,
                 "Diagrammer",
                 "...;
                  NodeStandardDData.LabelFont = <font_resource>;
                  ..."
                );
```

Type: Font

Default: **DGRAM**-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:

```
DS_SetViewOption(<datasource>, (ResPtr)<window>-><view>,
                 "Diagrammer",
                 "...;
                  NodeStandardDData.DrawLabel = {TRUE|FALSE};
                  ..."
                );
```

Type: Boolean

Default: TRUE

Synonyms: **Node.DrawLabel**

NodeStandardDData.Shape

NodeStandardDData.Shape specifies the default node shape. This syntax shows how to set the **NodeStandardDData.Shape** parameter:

```
DS_SetViewOption(<datasource>, (ResPtr)<window>-><view>,
                 "Diagrammer",
                 "...;
                  NodeStandardDData.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 **LinkStandardDData.LinkDirColor** or **LinkStandardDData.LinkUndirColor** parameter is not set. This syntax shows how to set the **Link.Color** parameter:

```
DS_SetViewOption(<datasource>, (ResPtr)<window>-><view>,
                 "Diagrammer",
                 "...;
                  Link.Color = <color_resource>;
                  ..."
                );
```

Type: Color

Default: **DGRAM**-specific

Synonyms: None

LinkStandardDData.LinkDirColor

LinkStandardDData.LinkDirColor sets the default color of all directed links. This syntax shows how to set the **LinkStandardDData.LinkDirColor** parameter:

```
DS_SetViewOption(<datasource>, (ResPtr)<window>-><view>,
                 "Diagrammer",
                 "...;
                  LinkStandardDData.LinkDirColor =
                                             <color_resource>;
                  ..."
                );
```

Type: Color

Default: **DGRAM**-specific

Synonyms: **Link.DirColor**

LinkStandardDData.LinkUndirColor

LinkStandardDData.LinkUndirColor sets the default color of undirected links. This syntax shows how to set **LinkStandardDData.LinkUndirColor**:

```
DS_SetViewOption(<datasource>, (ResPtr)<window>-><view>,
                 "Diagrammer",
                 "...;
                  LinkStandardDData.LinkUndirColor =
                                             <color_resource>;
                  ..."
                );
```

Type: Color

Default: **DGRAM**-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:

```
DS_SetViewOption(<datasource>, (ResPtr)<window>-><view>,
                 "Diagrammer",
                 "...;
                  LinkStandardDData.LabelColor =
                                           <color_resource>;
                  ..."
                );
```

Type: Color

Default: **DGRAM**-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:

```
DS_SetViewOption(<datasource>, (ResPtr)<window>-><view>,
                 "Diagrammer",
                 "...;
                  LinkStandardDData.LinkPen = <pen_resource>;
                  ..."
                );
```

Type: Pen

Default: **DGRAM**-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:

```
DS_SetViewOption(<datasource>, (ResPtr)<window>-><view>,
                 "Diagrammer",
                 "...;
                  LinkStandardDData.LabelFont = <font_resource>;
                  ..."
                );
```

Type: Font

Default: **DGRAM**-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:

```
DS_SetViewOption(<datasource>, (ResPtr)<window>-><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:

```
DS_SetViewOption(<datasource>, (ResPtr)<window>-><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:

```
DS_SetViewOption(<datasource>, (ResPtr)<window>-><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:

```
DS_SetViewOption(<datasource>, (ResPtr)<window>-><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:

```
DS_SetViewOption(<datasource>, (ResPtr)<window>-><view>,
                 "Diagrammer",
                 "...;
                  NodeFocusDData.BgColor = <color_resource>;
                  ..."
                );
```

Type: Color

Default: **DGRAM**-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:

```
DS_SetViewOption(<datasource>, (ResPtr)<window>-><view>,
                 "Diagrammer",
                 "...;
                  NodeFocusDData.LabelColor = <color_resource>;
                  ..."
                );
```

Type: Color

Default: **DGRAM**-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:

```
DS_SetViewOption(<datasource>, (ResPtr)<window>-><view>,
                 "Diagrammer",
                 "...;
                  NodeFocusDData.FramePen = <pen_resource>;
                  ..."
                );
```

Type: Pen

Default: **DGRAM**-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:

```
DS_SetViewOption(<datasource>, (ResPtr)<window>-><view>,
                 "Diagrammer",
                 "...;
                  NodeFocusDData.LabelFont = <font_resource>;
                  ..."
                );
```

Type: Font

Default: **DGRAM**-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:

```
DS_SetViewOption(<datasource>, (ResPtr)<window>-><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:

```
DS_SetViewOption(<datasource>, (ResPtr)<window>-><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:

```
DS_SetViewOption(<datasource>, (ResPtr)<window>-><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:

```
DS_SetViewOption(<datasource>, (ResPtr)<window>-><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**:

```
DS_SetViewOption(<datasource>, (ResPtr)<window>-><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:

```
DS_SetViewOption(<datasource>, (ResPtr)<window>-><view>,
                 "Diagrammer",
                 "...;
                  LinkFocusDData.LabelColor = <color_resource>;
                  ..."
                );
```

Type: Color

Default: **DGRAM**-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:

```
DS_SetViewOption(<datasource>, (ResPtr)<window>-><view>,
                 "Diagrammer",
                 "...;
                  LinkFocusDData.LinkPen = <pen_resource>;
                  ..."
                );
```

Type: Pen

Default: **DGRAM**-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:

```
DS_SetViewOption(<datasource>, (ResPtr)<window>-><view>,
                 "Diagrammer",
                 "...;
```

```
                  LinkFocusDData.LabelFont = <font_resource>;
                  ..."
                );
```

Type: Font

Default: **DGRAM**-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:

```
DS_SetViewOption(<datasource>, (ResPtr)<window>-><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:

```
DS_SetViewOption(<datasource>, (ResPtr)<window>-><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:

```
VarRec <value_variable>;
VAR_SetStr(<value_variable>, "<value>");
...
DS_SetNodeProperty(<datasource>, <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:

```
...
VarRec cityLarge;
VAR_SetStr(cityLarge, "Large");
...
DS_SetNodeProperty(graphDs, nodeAccessor,
                   "City", &cityLarge);
...
DS_SetViewOption(graphDs, (ResPtr)win->dgramWgt,
                  "City:Large", "Width = 144; Height = 48");
```

Here is the **SetViewOption()** syntax for the custom node properties:

```
DS_SetViewOption(<datasource>, (ResPtr)<window>-><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 graphDs;
VarGrNodeAccessorPtr nodeAccessor;
DGramPtr win->dgramWgt;
Str answer;
StrIVal len;

/* Define value variables for node properties. */
VarPtr citySmall, cityMedium, cityLarge;
enum citySize {
   Small = 1,
   Medium,
   Large
};
citySize theSize;

/* Assign objects to the pointers. */
graphDs = VARGR_Create();
nodeAccessor = VARGRNODEACCESSOR_Create();
citySmall = VAR_New();
cityMedium = VAR_New();
cityLarge = VAR_New();

win->dgramWgt =
   (DGramPtr)PANEL_GetNamedWgt((PanelPtr)win, "DGram");

/* Define value strings for the custom node properties. */
VAR_SetStr(citySmall, "Small");
VAR_SetStr(cityMedium, "Medium");
VAR_SetStr(cityLarge, "Large");

/* Set a node cursor using "nodeAccessor." */
VARGR_SetNodeCursor(graphDs, nodeAccessor);

/* Set the cursor option to "CONTROLS." */
DS_SetViewOption((DsPtr)graphDs, (ResPtr)win->dgramWgt,
                        "Cursor", "CONTROLS");

/* Set common node options. */
DS_SetViewOption((DsPtr)graphDs, (ResPtr)win->dgramWgt,
                 "Diagrammer",
                 "NodeStandardDData.Shape = RECTANGLE");
/* Set the parameters for the custom node options. */
DS_SetViewOption((DsPtr)graphDs, (ResPtr)win->dgramWgt,
                 "City:Small", "NodeW = 72; NodeH = 12");
DS_SetViewOption((DsPtr)graphDs, (ResPtr)win->dgramWgt,
                 "City:Medium", "NodeW = 108; NodeH = 18");
DS_SetViewOption((DsPtr)graphDs, (ResPtr)win->dgramWgt,
                 "City:Large", "NodeW = 144; NodeH = 24");

/* Define properties and values for individual nodes. */
if (ASKW_AskQuestion("1=Small, 2=Medium, 3=Large", answer)) {
   len = STR_GetLen(answer);
   theSize = (enum CitySize)STR_GetDecInt(answer, &len);
   switch (theSize) {
      case Small:
         VARGR_SetNodeProperty(graphDs, nodeAccessor,
                               "City", citySmall);
         break;
      case Medium:
         VARGR_SetNodeProperty(graphDs, nodeAccessor,
                               "City", cityMedium);
         break;
      case Large:
         VARGR_SetNodeProperty(graphDs, nodeAccessor,
                               "City", cityLarge);
         break;
      default:
         ALRTW_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:

```
DS_SetViewOption(<datasource>, (ResPtr)<winres>-><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:

```
DS_SetViewOption(graphDs, (ResPtr)win->dgramWgt,
                 "City:Large", "Width = 72");
```

Type: Int32

Default: **DGRAM**-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:

```
DS_SetViewOption(<datasource>, (ResPtr)<winres>-><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:

```
DS_SetViewOption(graphDs, (ResPtr)win->dgramWgt,
                 "City:Large", "Height = 24");
```

Type: Int32

Default: **DGRAM**-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:

```
DS_SetViewOption(<datasource>, (ResPtr)<window>-><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:

```
DS_SetViewOption(<datasource>, (ResPtr)<window>-><view>,
                 "<node_property>:<value>",
                 "...;
                  StandardDData.FrameColor =
                                         <color_resource>;
                  ..."
                );
```

Type: Color

Default: **DGRAM**-specific

Synonyms: **FrameColor**, **FColor**

StandardDData.BgColor

StandardDData.BgColor sets the color of the node. This syntax shows how to set the **StandardDData.BgColor** parameter:

```
DS_SetViewOption(<datasource>, (ResPtr)<window>-><view>,
                 "<node_property>:<value>",
                 "...;
                  StandardDData.BgColor = <color_resource>;
                  ..."
                );
```

Type: Color

Default: **DGRAM**-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:

```
DS_SetViewOption(<datasource>, (ResPtr)<window>-><view>,
                 "<node_property>:<value>",
                 "...;
                  StandardDData.LabelColor =
                                           <color_resource>;
                  ..."
                );
```

Type: Color

Default: **DGRAM**-specific

Synonyms: **LabelColor**, **FgColor**, **Fg**

StandardDData.FramePen

StandardDData.FramePen sets the pen for the node label. This syntax shows how to set the **StandardDData.FramePen** parameter:

```
DS_SetViewOption(<datasource>, (ResPtr)<window>-><view>,
                "<node_property>:<value>",
                "...;
                 StandardDData.FramePen = <pen_resource>;
                 ..."
               );
```

Type: Pen

Default: **DGRAM**-specific

Synonyms: **FramePen**, **Pen**

StandardDData.LabelFont

StandardDData.LabelFont sets the font for the node label. This syntax shows how to set the **StandardDData.LabelFont** parameter:

```
DS_SetViewOption(<datasource>, (ResPtr)<window>-><view>,
                "<node_property>:<value>",
                "...;
                 StandardDData.LabelFont = <font_resource>;
                 ..."
               );
```

Type: Font

Default: **DGRAM**-specific

Synonyms: **LabelFont**, **Font**

StandardDData.DrawLabel

StandardDData.DrawLabel specifies whether or not to display node labels. This syntax shows how to set the **StandardDData.DrawLabel** parameter:

```
DS_SetViewOption(<datasource>, (ResPtr)<window>-><view>,
                "<node_property>:<value>",
                "...;
                 StandardDData.DrawLabel = {TRUE|FALSE};
                 ..."
               );
```

Type: Boolean

Default: TRUE

Synonyms: **DrawLabel**

StandardDData.Shape

StandardDData.Shape specifies the default node shape. This syntax shows how to set the **StandardDData.Shape** parameter:

```
DS_SetViewOption(<datasource>, (ResPtr)<window>-><view>,
                 "<node_property>:<value>",
                 "...;
                  NodeStandardDData.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:

```
DS_SetViewOption(<datasource>, (ResPtr)<window>-><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:

```
DS_SetViewOption(<datasource>, (ResPtr)<window>-><view>,
                 "<node_property>:<value>",
                 "...;
                  FocusDData.BgColor = <color_resource>;
                  ..."
                );
```

Type: Color

Default: **DGRAM**-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:

```
DS_SetViewOption(<datasource>, (ResPtr)<window>-><view>,
                 "<node_property>:<value>",
                 "...;
                  FocusDData.LabelColor = <color_resource>;
                  ..."
                );
```

Type: Color

Default: **DGRAM**-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:

```
DS_SetViewOption(<datasource>, (ResPtr)<window>-><view>,
                 "<node_property>:<value>",
                 "...;
                  FocusDData.FramePen = <pen_resource>;
                  ..."
                );
```

Type: Pen

Default: **DGRAM**-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:

```
DS_SetViewOption(<datasource>, (ResPtr)<window>-><view>,
                 "<node_property>:<value>",
                 "...;
                  FocusDData.LabelFont = <font_resource>;
                  ..."
                );
```

Type: Font

Default: **DGRAM**-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:

```
DS_SetViewOption(<datasource>, (ResPtr)<window>-><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:

```
DS_SetViewOption(<datasource>, (ResPtr)<window>-><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:

```
VarPtr <value_variable>;
<value_variable> = VAR_New();
VAR_SetStr(<value_variable>, "<value>");
...
DS_SetEdgeProperty((DsPtr)<datasource>,
                   (VarGrEdgeAccessor)<edge_accessor>,
                   "<link_property>", <value_variable>);
...
VAR_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:

```
VarPtr roadFast;
roadFast = VAR_New();
VAR_SetStr(roadFast, "Fast");
...
DS_SetNodeProperty((DsPtr)graphDs,
                   (VarGrEdgeAccessorPtr)allEdgeAccessor,
                   "Road", roadFast);
...
DS_SetViewOption((DsPtr)graphDs, (ResPtr)win->dgramWgt,
                  "Road:Fast", "StandardDData.LinkPen =
Map.Fast");
...
VAR_Delete(roadFast);
```

Here is the **SetViewOption()** syntax for the custom node properties:

```
DS_SetViewOption(<datasource>, (ResPtr)<window>-><view>,
                 "<link_property>:<value>",
                 "[LinkColor = <color_resource>;] |
                  [onFocus = {FALSE|TRUE};] |
                  [StandardDData.LinkDirColor =
                                <color_resource>;] |
                  [StandardDData.LinkUndirColor =
                                <color_resource>;] |
                  [StandardDData.LabelColor = <color_resource>;] |
                  [StandardDData.LinkPen = <pen_resource>;] |
                  [StandardDData.LabelFont = <font_resource>;] |
                  [StandardDData.DrawLabel = {FALSE|TRUE};] |
                  [StandardDData.Shape =
                                {0|DEFAULT|DIAGONAL|
                                 1|RIGHTANGLE};] |
                  [FocusColor = <color_resource>;] |
                  [FocusDData.LinkDirColor = <color_resource>;] |
              [FocusDData.LinkUndirColor = <color_resource>;]
|
                  [FocusDData.LabelColor = <color_resource>;] |
                  [FocusDData.LinkPen = <pen_resource>;] |
                  [FocusDData.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 graphDs;
VarGrAllEdgeAccessorPtr edgeAccessor;
DGramPtr win->dgramWgt;
Str answer;
StrIVal len;

/* Define value variables for node properties. */
VarPtr roadSlow, roadModerate, roadFast;
enum roadSpeed {
   Slow = 1,
   Moderate,
   Fast
};

roadSpeed mRoadSpeed;

/* Assign objects to the pointers. */
graphDs = VARGR_Create();
nodeAccessor = VARGRNODEACCESSOR_Create();
roadSlow = VAR_New();
roadModerate = VAR_New();
roadFast = VAR_New();

win->dgramWgt =
   (DGramPtr)PANEL_GetNamedWgt((PanelPtr)win, "DGram");

/* Define value strings for the custom node properties. */
VAR_SetStr(roadSlow, "Slow");
VAR_SetStr(roadModerate, "Moderate");
VAR_SetStr(roadFast, "Fast");

/* Set a edge cursor using "mEdgeAccessor." */
VARGR_SetEdgeCursor(graphDs, mEdgeAccessor);

/* Set the cursor option to "CONTROLS."
DS_SetViewOption((DsPtr)graphDs, (ResPtr)win->dgramWgt,
                 "Cursor", "CONTROLS");
/* Set common edge options. */
DS_SetViewOption((DsPtr)graphDs, (ResPtr)win->dgramWgt,
                 "Diagrammer",
                 "LinkStandardDData.Shape = DIAGONAL");
/* Set the parameters for the custom edge options. */
DS_SetViewOption((DsPtr)graphDs, (ResPtr)win->dgramWgt,
                 "Road:Slow",
                 "StandardDData.LinkPen = Map.Slow");
DS_SetViewOption((DsPtr)graphDs, (ResPtr)win->dgramWgt,
                 "Road:Moderate",
                 "StandardDData.LinkPen = Map.Moderate");
DS_SetViewOption((DsPtr)graphDs, (ResPtr)win->dgramWgt,
                 "Road:Fast",
                 "StandardDData.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 (mRoadSpeed) {
      case Slow:
         VARGR_SetEdgeProperty(graphDs,
                               (VarGrEdgeAccessorPtr)mEdgeAccessor,
                               "Road", mRoadSlow);
         break;
      case Moderate:
         VARGR_SetEdgeProperty(graphDs,
                               (VarGrEdgeAccessorPtr)mEdgeAccessor,
                               "Road", mRoadModerate);
         break;
      case Fast:
         VARGR_SetEdgeProperty(graphDs,
                               (VarGrEdgeAccessorPtr)mEdgeAccessor,
                               "Road", mRoadFast);
         break;
      default:
         break;
   } /*End switch. */
} /* End if. */
```

LinkColor

LinkColor sets the link color if the **StandardDData.LinkDirColor** or **StandardDData.LinkUndirColor** parameter is not set. This syntax shows how to set the **LinkColor** parameter:

```
DS_SetViewOption(<datasource>, (ResPtr)<window>-><view>,
                 "<link_property>:<value>",
                 "...;
                  LinkColor = <color_resource>;
                  ..."
                );
```

Type: Color

Default: **DGRAM**-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:

```
DS_SetViewOption(<datasource>, (ResPtr)<window>-><view>,
                 "<link_property>:<value>",
                 "...;
                  onFocus = {TRUE|FALSE};
                  ..."
                );
```

Type: Boolean

Default: **FALSE**

Synonyms: None

StandardDData.LinkDirColor

StandardDData.LinkDirColor sets the default color of all directed links. This syntax shows how to set the **StandardDData.LinkDirColor** parameter:

```
DS_SetViewOption(<datasource>, (ResPtr)<window>-><view>,
                 "<link_property>:<value>",
                 "...;
                  StandardDData.LinkDirColor =
                                             <color_resource>;
                  ..."
                );
```

Type: Color

Default: **DGRAM**-specific

Synonyms: **DirColor**

StandardDData.LinkUndirColor

StandardDData.LinkUndirColor sets the default color of undirected links. This syntax shows how to set **StandardDData.LinkUndirColor**:

```
DS_SetViewOption(<datasource>, (ResPtr)<window>-><view>,
                 "<link_property>:<value>",
                 "...;
                  StandardDData.LinkUndirColor =
                                             <color_resource>;
                  ..."
                );
```

Type: Color

Default: **DGRAM**-specific

Synonyms: **UndirColor**

StandardDData.LabelColor

StandardDData.LabelColor sets the color of the node label. This syntax shows how to set the **StandardDData.LabelColor** parameter:

```
DS_SetViewOption(<datasource>, (ResPtr)<window>-><view>,
                 "<link_property>:<value>",
                 "...;
                  StandardDData.LabelColor =
                                           <color_resource>;
                  ..."
                );
```

Type: Color

Default: **DGRAM**-specific

Synonyms: **LabelColor**

StandardDData.LinkPen

StandardDData.LinkPen sets the pen for the link label. This syntax shows how to set the **StandardDData.LinkPen** parameter:

```
DS_SetViewOption(<datasource>, (ResPtr)<window>-><view>,
                 "<link_property>:<value>",
                 "...;
                  StandardDData.LinkPen = <pen_resource>;
                 ..."
               );
```

Type: Pen

Default: **DGRAM**-specific

Synonyms: **Pen**

StandardDData.LabelFont

StandardDData.LabelFont sets the font for the node label. This syntax shows how to set the **StandardDData.LabelFont** parameter:

```
DS_SetViewOption(<datasource>, (ResPtr)<window>-><view>,
                 "<link_property>:<value>",
                 "...;
                  StandardDData.LabelFont = <font_resource>;
                  ..."
               );
```

Type: Font

Default: **DGRAM**-specific

Synonyms: **Font**

StandardDData.DrawLabel

StandardDData.DrawLabel specifies whether or not to display node labels. This syntax shows how to set the **StandardDData.DrawLabel** parameter:

```
DS_SetViewOption(<datasource>, (ResPtr)<window>-><view>,
                 "<link_property>:<value>",
                 "...;
                  StandardDData.DrawLabel = {TRUE|FALSE};
                  ..."
               );
```

Type: Boolean

Default: **FALSE**

Synonyms: **DrawLabel**

StandardDData.Shape

StandardDData.Shape specifies the default shape for links. This syntax shows how to set the **StandardDData.Shape** parameter:

```
DS_SetViewOption(<datasource>, (ResPtr)<window>-><view>,
                 "<link_property>:<value>",
                 "...;
                  StandardDData.Shape =
                              {0|DEFAULT|DIAGONAL|
                               1|RIGHTANGLE
                              };
                  ..."
               );
```

Type: Enumerated

Default: **DIAGONAL**

Synonyms: **Shape**

FocusColor

FocusColor sets the link color if the **FocusDData.LinkDirColor** or **FocusDData.LinkUndirColor** parameter is not set. This syntax shows how to set the **FocusColor** parameter:

```
DS_SetViewOption(<datasource>, (ResPtr)<window>-><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:

```
DS_SetViewOption(<datasource>, (ResPtr)<window>-><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**:

```
DS_SetViewOption(<datasource>, (ResPtr)<window>-><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:

```
DS_SetViewOption(<datasource>, (ResPtr)<window>-><view>,
                 "<link_property>:<value>",
                 "...;
                  FocusDData.LabelColor = <color_resource>;
                  ..."
                );
```

Type: Color

Default: **DGRAM**-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:

```
DS_SetViewOption(<datasource>, (ResPtr)<window>-><view>,
                 "<link_property>:<value>",
                 "...;
                  FocusDData.LinkPen = <pen_resource>;
                  ..."
                );
```

Type: Pen

Default: **DGRAM**-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:

```
DS_SetViewOption(<datasource>, (ResPtr)<window>-><view>,
                 "<link_property>:<value>",
                 "...;
                  FocusDData.LabelFont = <font_resource>;
                  ..."
                );
```

Type: Font

Default: **DGRAM**-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:

```
DS_SetViewOption(<datasource>, (ResPtr)<window>-><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:

```
DS_SetViewOption(<datasource>, (ResPtr)<window>-><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.**DGRAM** For more information about options for the **DGRAM** view, see “Options for the DGRAM View” on page 84.

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 graphDs as a **VarGrPtr** variable and initializes graphDs to the value returned by **VARGR_Create()**:

```
VarGrPtr graphDs;
...
graphDs = VARGR_Create();
```

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, graphDs.

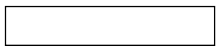


Figure 4–14 Untitled Graph Datasource

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 63.

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, editGraphDs:

```
VarGrPtr graphDs;

/* Declare a pointer variable for the edit object. */
VarGrEditPtr editGraphDs;
...
graphDs = VARGR_Create();
editGraphDs = VARGR_StartEdit(graphDs);

/* Edit operations defined. */
...

/* Commit edit operations to the datasource and destroy the edit
   object. */
DSEDIT_End((DsEditPtr)editGraphDs);
```

In this example, editGraphDs is a **VarGrEditPtr** variable and is assigned a datasource edit object—the value returned by the **VARGR_StartEdit()** function. When the **DSEDIT_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 79.

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

```
VarGrPtr graphDs;
VarGrEditPtr editGraphDs;
...
graphDs = VARGR_Create();
editGraphDs = VARGR_StartEdit(graphDs);

/* Set the title of the graph datasource. */
VARGREDIT_SetTitle(editGraphDs, "Graph Datasource");

/* Commit edit operations to the datasource and destroy the edit
   object. */
DSEDIT_End((DsEditPtr)editGraphDs);
```

Note the use of the string literal in quotation marks. Unlike the node **ID** and **Value** properties, the **VARGREDIT_SetTitle()** function accepts a string as the datasource title. Building on the example from Figure 4–14, executing the preceding **VARGREDIT_SetTitle()** function adds a title to the graph datasource, as shown in Figure 4–15:

Graph Datasource

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 the following code fragment declares a node-accessor pointer, nodeAccessor. In the last executable line of code, a node-accessor object is created and assigned to nodeAccessor:

```
VarGrPtr graphDs;
VarGrEditPtr editGraphDs;

/* Declare a node-accessor pointer and assign a node-accessor
   object to it. */
VarGrNodeAccessorPtr nodeAccessor;
...
graphDs = VARGR_Create();
editGraphDs = VARGR_StartEdit(graphDs);

/* Assign a node-accessor object to the node-accessor
   pointer. */
nodeAccessor = VARGRNODEACCESSOR_Create();
...
```

This creates *nodeAccessor* as a **VarGrNodeAccessorPtr** variable and initializes nodeAccessor to the value returned by **VARGRNODEACCESSOR_Create()**. For more information about node accessors, see “Node Accessor” on page 74.

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 fragment declares an “all” edge-accessor pointer, allEdgeAccessor, creates an “all” edge-accessor object, and assigns it to allEdgeAccessor:

```
VarGrPtr graphDs;

/* Declare an "all" edge-accessor pointer. */
```

```
VarGrAllEdgeAccessorPtr allEdgeAccessor;
VarGrEdgeEditPtr editEdge;
...
graphDs = VARGR_Create();

/* Assign an "all" edge-accessor object to the node-accessor
   pointer. */
allEdgeAccessor = VARGRALLEDGEACCESSOR_Create();

/* Assign an "all" edge-edit object to the edge-edit pointer
   based on the "all" edge accessor, allEdgeAccessor. */
editEdge = VARGR_StartEdgeEdit(allEdgeAccessor);
...
```

This declares a **VarGrAllEdgeAccessorPtr** variable, *allEdgeAccessor*, and assigns the value returned by **VARGRALLEDGEACCESSOR_Create()** to it. For more information about “all” edge accessors, see “Edge Accessor” on page 74.

“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 and fourth executable lines of the following code fragment declare inEdgeAccessor and outEdgeAccessor, respectively, as “in” and “out” edge-accessor pointers. Later in the example, “in” and “out” edge-accessor objects are created and assigned to their corresponding edge-accessor pointers.

```
VarGrPtr graphDs;

/* Declare a node-accessor pointer. */
VarGrNodeAccessorPtr nodeAccessor;

/* Declare "in" and "out" edge-accessor pointers. */
VarGrInEdgeAccessorPtr inEdgeAccessor;
VarGrOutEdgeAccessorPtr outEdgeAccessor;

/* Declare an edge-edit pointer. */
VarGrEdgeEditPtr editEdge;
...
graphDs = VARGR_Create();

/* Based on the node-accessor pointer, nodeAccessor, assign
   "in" and "out" edge-accessor objects, respectively, to the
   "in" and "out" edge-accessor pointers. */
inEdgeAccessor =
   VARGRNODEACCESSOR_CreateInEdgeAccessor(nodeAccessor);
outEdgeAccessor =
   VARGRNODEACCESSOR_CreateOutEdgeAccessor(nodeAccessor);

/* Assign an "in" edge-edit object to the edge-edit pointer
   based on the "in" edge accessor, inEdgeAccessor. */
editEdge = VARGR_StartEdgeEdit(inEdgeAccessor);
...
DSEDIT_End((DsEditPtr)editEdge);
...

/* Assign an "out" edge-edit object to the edge-edit pointer
   based on the "out" edge accessor, outEdgeAccessor. */
editEdge = VARGR_StartEdgeEdit(outEdgeAccessor);
...
DSEDIT_End((DsEditPtr)editEdge);
...
```

This declares **VarGrInEdgeAccessorPtr** and **VarGrOutEdgeAccessorPtr** variables, *inEdgeAccessor* and *outEdgeAccessor*, respectively, and assigns the values returned by **VARGRINEDGEACCESSOR_Create()** and **VARGROUTEDGEACCESSOR_Create()** to them. For more information about “in” and “out” edge accessors, see “Edge Accessor” on page 74.

“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 undirEdgeAccessor as “undirected” edge-accessor pointer. Later in the example, an “undirected” edge-accessor object is created and assigned to the “undirected” edge-accessor pointer.

```
VarGrPtr graphDs;

/* Declare a node-accessor pointer. */
VarGrNodeAccessorPtr nodeAccessor;

/* Declare an "undirected" edge-accessor pointer. */
VarGrUndirEdgeAccessorPtr undirEdgeAccessor;

/* Declare an edge-edit pointer. */
VarGrEdgeEditPtr editEdge;
...
graphDs = VARGR_Create();

/* Based on the node-accessor pointer, nodeAccessor, assign an
   "undirected" edge-accessor object to the "undirected"
   edge-accessor pointer. */
undirEdgeAccessor =
   VARGRNODEACCESSOR_CreateUndirEdgeAccessor(nodeAccessor);

/* Assign an "undirected" edge-edit object to the edge-edit
   pointer based on the "undirected" edge accessor,
   inEdgeAccessor. */
editEdge = VARGR_StartEdgeEdit(undirEdgeAccessor);
...
DSEDIT_End((DsEditPtr)editEdge);
...

/* Assign an "undirected" edge-edit object to the edge-edit
   pointer based on the "undirected" edge accessor,
   undirEdgeAccessor. */
editEdge = VARGR_StartEdgeEdit(undirEdgeAccessor);
...
DSEDIT_End((DsEditPtr)editEdge);
...
```

This declares a **VarGrUndirEdgeAccessorPtr** variable, *undirEdgeAccessor*, and assigns the value returned by **VARGRNODEACCESSOR_CreateUndirEdgeAccessor()** to it. For more information about “undirected” edge accessors, see “Edge Accessor” on page 74.

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:

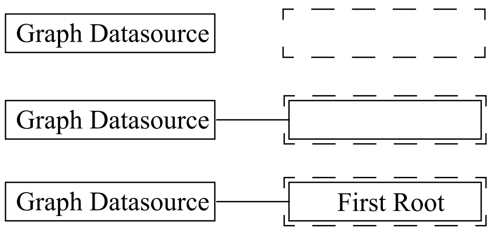


Figure 4–16 Creating the First Root Node in a Graph Datasource

This code fragment shows how to add the first node in the graph datasource:

```
VarGrPtr graphDs;
VarGrEditPtr editGraphDs;
VarPtr varID = VAR_New();
VarPtr varValue = VAR_New();

/* Declare a node-accessor pointer variable. */
VarGrNodeAccessorPtr nodeAccessor;
...
graphDs = VARGR_Create();

/* Assign a node-accessor object to the node-accessor
   pointer. */
nodeAccessor = VARGRNODEACCESSOR_Create();

/* Assign a datasource edit object to the VarGrEditPtr,
   editGraphDs. */
editGraphDs = VARGR_StartEdit(graphDs);

/* Move nodeAccessor to the first available node location. */
VARGRNODEACCESSOR_GoFirstRoot(nodeAccessor);

/* Add the first root node to begin the graph. */
VARGREDIT_AddNode(editGraphDs, nodeAccessor);

/* Set the node ID and Value properties. */
VAR_SetStr(varID, "0001"):
VAR_SetStr(varValue, "First Root"):
VARGREDIT_SetNodeID(editGraphDs, nodeAccessor, varID);
VARGREDIT_SetNodeValue(editGraphDs, nodeAccessor, varValue);

DSEDIT_End((DsEditPtr)editGraphDs);
```

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 graphDs;
VarGrNodeAccessorPtr nodeAccessor;
VarPtr varID = VAR_New();
VarPtr varValue = VAR_New();
...
graphDs = VARGR_Create();
nodeAccessor = VARGRNODEACCESSOR_Create();

/* Move nodeAccessor to the first available node location. */
VARGRNODEACCESSOR_GoFirstRoot(nodeAccessor);

/* Use the convenience API to create an edit object, add the
   first root node, and dispose of the edit object. */
VARGR_AddNode(graphDs, nodeAccessor);

/* Set the node ID and Value properties using "convenience" API
   functions. */
VAR_SetStr(varID, "0001"):
VAR_SetStr(varValue, "First Root"):
VARGR_SetNodeID(graphDs, nodeAccessor, varID);
VARGR_SetNodeValue(graphDs, nodeAccessor, varValue);
...
```

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.

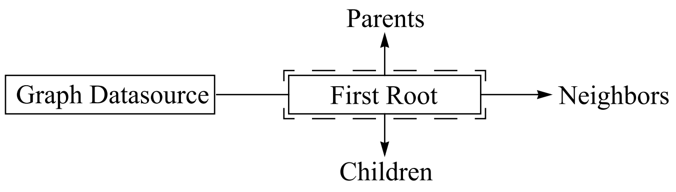


Figure 4–17 Illustration Scheme for Child, Parent, and Neighbor Nodes

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 graphDs;
VarGrNodeAccessorPtr nodeAccessor;
VarPtr varID = VAR_New();
VarPtr varValue = VAR_New();

/* Declare an edit pointer variable. */
...
graphDs = VARGR_Create();
nodeAccessor = VARGRNODEACCESSOR_Create();
VARGRNODEACCESSOR_GoFirstRoot(nodeAccessor);
VARGR_AddNode(graphDs, nodeAccessor);

/* Set the node ID and Value properties using "convenience" API
   functions. */
VAR_SetStr(varID, "0001"):
VAR_SetStr(varValue, "First Root"):
VARGR_SetNodeID(graphDs, nodeAccessor, varID);
VARGR_SetNodeValue(graphDs, nodeAccessor, varValue);
...
```

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 **VARGRNODEACCESSOR_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 **VARGRNODEACCESSOR_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:

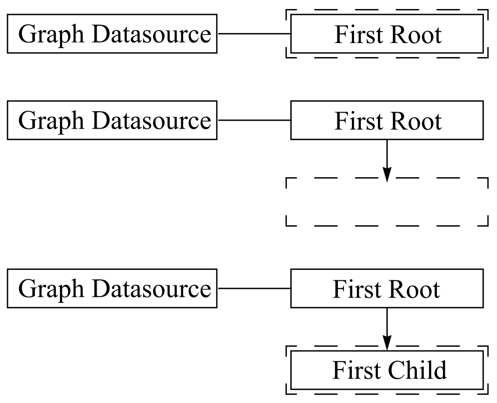


Figure 4–18 Adding the First Child Node

```
/* Create a graph datasource and the first root node. */
VarGrPtr graphDs;
```

```
VarGrNodeAccessorPtr nodeAccessor;
VarPtr varID;
VarPtr varValue;
...
graphDs = VARGR_Create();
nodeAccessor = VARGRNODEACCESSOR_Create();
varID = VAR_New();
varValue = VAR_New();

VARGRNODEACCESSOR_GoFirstRoot(nodeAccessor);
VARGR_AddNode(graphDs, nodeAccessor);

/* Set the node ID and Value properties using "convenience" API
   functions. */
VAR_SetStr(varID, "0001"):
VAR_SetStr(varValue, "First Root"):
VARGR_SetNodeID(graphDs, nodeAccessor, varID);
VARGR_SetNodeValue(graphDs, nodeAccessor, varValue);
...
/* Move the node accessor to the first child-node location. */
VARGRNODEACCESSOR_GoFirstChild(nodeAccessor);

/* Execute the GoNext() function repeatedly until an invalid
   node location is found. */
while (VARGR_IsNodeValid(graphDs, nodeAccessor)) {
   VARGRNODEACCESSOR_GoNext(nodeAccessor);
}

/* The following GoNthChild() function would replace the
   preceding GoFirstChild() function and while loop. */
/* VARGRNODEACCESSOR_GoNthChild(nodeAccessor,
                                 VARGR_GetNumChildren(graphDs,
                                                      nodeAccessor)); */

/* Add a node. */
VARGR_AddNode(graphDs, nodeAccessor);

/* Set the node ID and Value properties using "convenience" API
   functions. */
VAR_SetStr(varID, "0002"):
VAR_SetStr(varValue, "First Child"):
VARGR_SetNodeID(graphDs, nodeAccessor, varID);
VARGR_SetNodeValue(graphDs, nodeAccessor, varValue);
...
```

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 **VARGRNODEACCESSOR_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 *first* child 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:

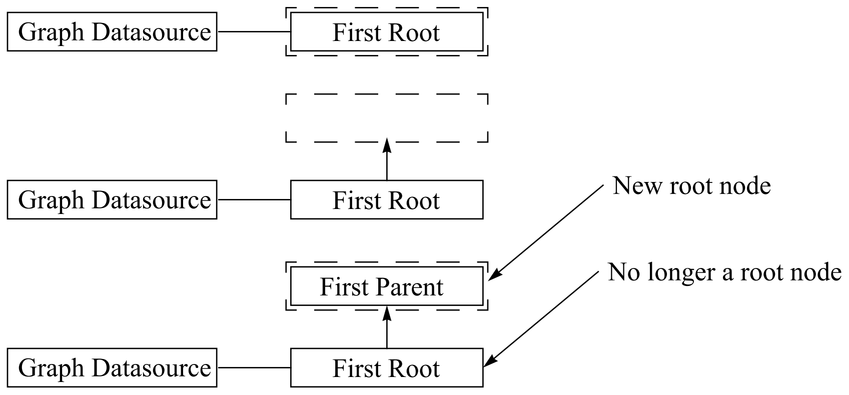


Figure 4–19 Adding the First Parent Node

```
/* Create a graph datasource and the first root node. */
VarGrPtr graphDs;
VarGrNodeAccessorPtr nodeAccessor;
VarPtr varID = VAR_New();
VarPtr varValue = VAR_New();

/* Declare an edit pointer variable. */
...
graphDs = VARGR_Create();
nodeAccessor = VARGRNODEACCESSOR_Create();
VARGRNODEACCESSOR_GoFirstRoot(nodeAccessor);
VARGR_AddNode(graphDs, nodeAccessor);

/* Set the node ID and Value properties using "convenience" API
   functions. */
VAR_SetStr(varID, "0001"):
VAR_SetStr(varValue, "First Root"):
VARGR_SetNodeID(graphDs, nodeAccessor, varID);
VARGR_SetNodeValue(graphDs, nodeAccessor, varValue);
...
/* Move the node accessor to the first parent-node location. */
VARGRNODEACCESSOR_GoFirstParent(nodeAccessor);

/* Execute the GoNext() function repeatedly until an invalid
   node location is found. */
while (VARGR_IsNodeValid(graphDs, nodeAccessor)) {
   VARGRNODEACCESSOR_GoNext(nodeAccessor);
}

/* The following GoNthParent() function would replace the
   preceding GoFirstParent() function and while loop. */
/* VARGRNODEACCESSOR_GoNthParent(nodeAccessor,
                          VARGRNODEACCESSOR_GetNumParents());
*/

/* Add a node. */
VARGR_AddNode(graphDs, nodeAccessor);

/* Set the node ID and Value properties using "convenience" API
   functions. */
VAR_SetStr(varID, "0003"):
VAR_SetStr(varValue, "First Parent"):
VARGR_SetNodeID(graphDs, nodeAccessor, varID);
VARGR_SetNodeValue(graphDs, nodeAccessor, varValue);
...
```

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 **VARGRNODEACCESSOR_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:

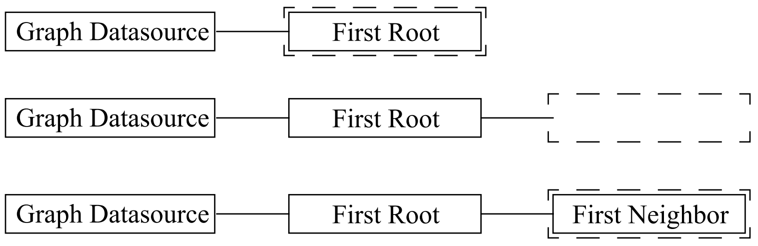


Figure 4–20 Adding the First Neighbor Node

```
/* Create a graph datasource and the first root node. */
VarGrPtr graphDs;
VarGrNodeAccessorPtr nodeAccessor;
VarPtr varID = VAR_New();
VarPtr varValue = VAR_New();

/* Declare an edit pointer variable
...
graphDs = VARGR_Create();
nodeAccessor = VARGRNODEACCESSOR_Create();
VARGRNODEACCESSOR_GoFirstRoot(nodeAccessor);
VARGR_AddNode(graphDs, nodeAccessor);

/* Set the node ID and Value properties using "convenience" API
   functions. */
VAR_SetStr(varID, "0001"):
VAR_SetStr(varValue, "First Root"):
VARGR_SetNodeID(graphDs, nodeAccessor, varID);
VARGR_SetNodeValue(graphDs, nodeAccessor, varValue);
...
/* Move the node accessor to the first neighbor-node
   location. */
VARGRNODEACCESSOR_GoFirstNeighbor(nodeAccessor);

/* Execute the GoNext() function repeatedly until an invalid
   node location is found. */
while (VARGR_IsNodeValid(graphDs, nodeAccessor)) {
   VARGRNODEACCESSOR_GoNext(nodeAccessor);
}

/* The following GoNthNeighbor() function would replace the
   preceding GoFirstNeighbor() function and while loop. */
/* VARGR_GoNthNeighbor(nodeAccessor, GetNumNeighbors()); */

/* Add a node. */
VARGR_AddNode(graphDs, nodeAccessor);

/* Set the node ID and Value properties using "convenience" API
   functions. */
VAR_SetStr(varID, "0004"):
VAR_SetStr(varValue, "First Neighbor"):
```

```
VARGR_SetNodeID(graphDs, nodeAccessor, varID);
VARGR_SetNodeValue(graphDs, nodeAccessor, varValue);
...
```

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 **VARGRNODEACCESSOR_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 128.

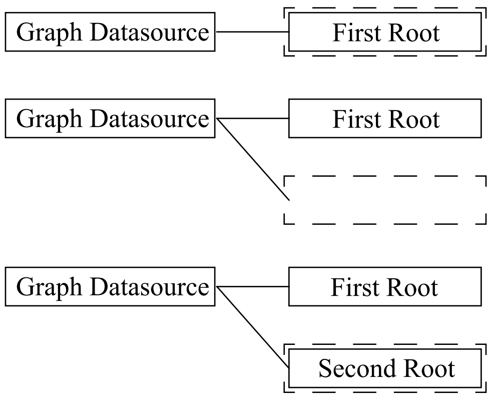


Figure 4–21 Adding an Unlinked Node

```
/* Create a graph datasource and the first root node. */
VarGrPtr graphDs;
VarGrNodeAccessorPtr nodeAccessor;
VarPtr varID = VAR_New();
VarPtr varValue = VAR_New();

/* Declare an edit pointer variable. */
...
graphDs = VARGR_Create();
nodeAccessor = VARGRNODEACCESSOR_Create();
VARGRNODEACCESSOR_GoFirstRoot(nodeAccessor);
VARGR_AddNode(graphDs, nodeAccessor);

/* Set the node ID and Value properties using "convenience" API
   functions. */
VAR_SetStr(varID, "0001"):
VAR_SetStr(varValue, "First Root"):
VARGR_SetNodeID(graphDs, nodeAccessor, varID);
VARGR_SetNodeValue(graphDs, nodeAccessor, varValue);
...
/* Move the node accessor to the first root-node location. */
VARGRNODEACCESSOR_GoFirstRoot(nodeAccessor);

/* Execute the GoNext() function repeatedly until an invalid
```

```
   node location is found. */
while (VARGR_IsNodeValid(graphDs, nodeAccessor)) {
   VARGRNODEACCESSOR_GoNext(nodeAccessor);
}

/* The following GoNthRoot() function would replace the
   preceding GoFirstRoot() function and while loop. */
/* VARGR_GoNthRoot(nodeAccessor, GetNumRoots()); */

/* Add a node. */
VARGR_AddNode(graphDs, nodeAccessor);

/* Set the node ID and Value properties using "convenience" API
   functions. */
VAR_SetStr(varID, "0003"):
VAR_SetStr(varValue, "First Parent"):
VARGR_SetNodeID(graphDs, nodeAccessor, varID);
VARGR_SetNodeValue(graphDs, nodeAccessor, varValue);
...
```

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 graphDs;
VarGrNodeAccessorPtr nodeAccessorFrom;
VarGrNodeAccessorPtr nodeAccessorTo;
VarGrInEdgeAccessorPtr inEdgeAccessor;
VarGrOutEdgeAccessorPtr outEdgeAccessor;
VarGrUndirEdgeAccessorPtr undirEdgeAccessor;
...
graphDs = VARGR_Create();

/* Create node accessors for both the "From" and "To" nodes. */
nodeAccessorFrom = VARGRNODEACCESSOR_Create();
nodeAccessorTo = VARGRNODEACCESSOR_Create();

/* Move the "From" and "To" accessors to two unlinked nodes. */
...

/* Create an "out" edge accessor for the node referenced by
   nodeAccessorFrom, the source node. */
outEdgeAccessor =
   VARGRNODEACCESSOR_CreateOutEdgeAccessor(nodeAccessorFrom);

/* Create an "in" edge accessor for the node referenced by
   nodeAccessorTo, the source node. */
inEdgeAccessor =
   VARGRNODEACCESSOR_CreateInEdgeAccessor(nodeAccessorTo);
```

```
/* Create an "undirected" edge accessor for the node referenced
   by nodeAccessorFrom, an arbitrarily chosen end node. */
undirEdgeAccessor =
  VARGRNODEACCESSOR_CreateUndirEdgeAccessor(nodeAccessorFrom);
...
```

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 nodeAccessorFrom and nodeAccessorTo. 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 nodeAccessorFrom. It could just as easily be an “in” edge accessor based on the node referenced by nodeAccessorTo.

```
/* Create a graph datasource and the first root node. */
VarGrPtr graphDs;
VarGrNodeAccessorPtr nodeAccessorFrom;
VarGrNodeAccessorPtr nodeAccessorTo;
VarGrOutEdgeAccessorPtr outEdgeAccessor;
VarPtr varID = VAR_New();
VarPtr varValue = VAR_New();
...
graphDs = VARGR_Create();

/* Create node accessors for both the "From" and "To" nodes. */
nodeAccessorFrom = VARGRNODEACCESSOR_Create();
nodeAccessorTo = VARGRNODEACCESSOR_Create();
...
/* Move the "From" and "To" accessors to two unlinked nodes. */
...

/* Add a directed edge using nodeAccessorFrom to identify the
   source of the edge, while using nodeAccessorTo to identify
   the target node. */
VARGR_AddDirEdge(graphDs, nodeAccessorFrom, nodeAccessorTo);

/* Create an "out" edge accessor for the node referenced by
   nodeAccessorFrom, the source node. */
outEdgeAccessor =
   VARGRNODEACCESSOR_CreateOutEdgeAccessor(nodeAccessorFrom);

/* 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. */
VAR_SetStr(varID, "0001"):
VAR_SetStr(varValue, "First Edge"):
VARGR_SetEdgeID(graphDs, outEdgeAccessor, varID);
VARGR_SetEdgeValue(graphDs, outEdgeAccessor, varValue);
...
```

Undirected Edges

This example uses undirected edges. The two node accessors are nodeAccessor1 and nodeAccessor2. 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 nodeAccessor1. It could just as easily be an “undirected” edge accessor based on the node referenced by nodeAccessor2.

```
/* Create a graph datasource and the first root node. */
VarGrPtr graphDs;
VarGrNodeAccessorPtr nodeAccessor1;
VarGrNodeAccessorPtr nodeAccessor2;
VarGrUndirEdgeAccessorPtr undirEdgeAccessor;
VarPtr varID = VAR_New();
VarPtr varValue = VAR_New();
...
graphDs = VARGR_Create();

/* Create node accessors for both the "1" and "2" nodes. */
nodeAccessor1 = VARGRNODEACCESSOR_Create();
nodeAccessor2 = VARGRNODEACCESSOR_Create();

/* Move the "1" and "2" accessors to two unlinked nodes. */
...

// Add an undirected edge using nodeAccessor1 and nodeAccessor2
   to identify its end points. */
VARGR_AddUndirEdge(graphDs, nodeAccessor1, nodeAccessor2);

/* Create an "undirected" edge accessor for the node referenced
   by nodeAccessor1. */
undirEdgeAccessor =
   VARGRNODEACCESSOR_CreateUndirEdgeAccessor(nodeAccessor1);

/* 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. */
VAR_SetStr(varID, "0001"):
VAR_SetStr(varValue, "First Edge"):
VARGR_SetEdgeID(graphDs, undirEdgeAccessor, varID);
VARGR_SetEdgeValue(graphDs, undirEdgeAccessor, varValue);
...
```

Chapter

5 Args Class

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 decides 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

void ARGS_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

ArrayPtr ARGS_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

ArgIVal ARGS_GetNum(void);

Returns the number of arguments (including the application name itself).

GetNth

CStr ARGS_GetNth(ArgIVal *n***);**

Returns the nth argument. It returns NULL if there are fewer arguments than n.

GetExecName

CStr ARGS_GetExecName(void);

Returns the application name.

GetFirst

CStr ARGS_GetFirst(void);

Returns the first argument after the application name. It returns NULL if there is no argument.

GetNext

CStr ARGS_GetNext(void);

Returns the next argument. GetNext can be called only after ARGS_GetFirst has been called at least once. It returns NULL when all arguments have been read.

RemoveNth

void ARGS_RemoveNth(ArgIVal *n*);

Extract the nth argument from the list.

InsertNth

void ARGS_InsertNth(ArgIVal *n*, CStr *arg*);

Inserts the new argument arg into the list at given index n.

Chapter

6 *ArNum Class*

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 refernces 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 otherwize 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

void ARNUM_Construct(ArNumPtr *arnum***);**

Default ARNUM construction.

void ARNUM_ConstructLen(ArNumPtr *arnum***, 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.

void ARNUM_ConstructAlloc(ArNumPtr *artpr***, ArrayIVal** *alloc***);**

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'.

void ARNUM_ConstructArnum(ArNumPtr *arnum***, ArNumPtr** *arnum2***);**

Constructs the ARNUM as a copy of `arnum2'. This is a deep copy.

Destructor

void ARNUM_Destruct(ArNumPtr *arnum***);**

Default ARNUM destruction. All the values stored in the arnum are lost at this point.

Clone, Copy, Reset

Reset

void ARNUM_Reset(ArNumPtr *arnum***);**

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 ARNUM_SetLen(ArNumPtr *arnum*, **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 ARNUM_SetAlloc(ArNumPtr *arnum*, **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 ARNUM_GetLen(ArNumCPtr *arnum***);**

Returns the number of elements in the ARNUM.

IsEmpty

BoolEnum ARNUM_IsEmpty(ArNumCPtr *arnum***);**

Returns whether the ARNUM is empty or not.

IsInRange

BoolEnum ARNUM_IsInRange(ArNumCPtr *arnum*, **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 ARNUM_GetNthElt(ArNumCPtr *arnum*, **ArrayIVal** *i***);**

Returns the element at index 'I'. Fails if the index is not in the [0, len-1] range.

UnboundedGetNthElt

ARNUM_ELT ARNUM_UnboundedGetNthElt(ArNumCPtr *arnum*, **ArrayIVal** *i***);**

Same as ARNUM_GetNthElt but returns 0 if 'i' is out of range instead of failing.

SetNthElt

void ARNUM_SetNthElt(ArNumPtr *arnum*, **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 ARNUM_UnboundedSetNthElt(ArNumPtr *arnum*, **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 ARNUM_ContainsElt(ArNumCPtr *arnum*, **ARNUM_ELT** *elt***);**

Returns whether or not the ARNUM contains ’elt’.

LookupElt

ArrayIVal ARNUM_LookupElt(ArNumCPtr *arnum*, **ARNUM_ELTCPtr** *elt***);**

Returns the index of the first occurence of ’elt’ in the ARNUM. Returns -1 if the ARNUM does not contain ’elt’.

FindElt

ArrayIVal ARNUM_FindElt(ArNumCPtr *arnum*, **ARNUM_ELTCPtr** *elt***);**

Same as ARNUM_Lookup routines but signal a failure if the ARNUM does not contain ’elt’.

SortedLookupElt

BoolEnum ARNUM_SortedLookupElt(ArNumCPtr *arnum*, **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 ARNUM_SortedFindElt(ArNumCPtr *arnum*, **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 occured. If no match is found, this routine signals a failure.

Adding Elements

AppendElt

void ARNUM_AppendElt(ArNumPtr *arnum*, **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 ARNUM_UniqAppendElt(ArNumPtr *arnum*, **ARNUM_ELT** *elt*);

Appends ’elt’ to the ARNUM if ’elt’ is not already in the ARNUM.

InsertNthElt

void ARNUM_InsertNthElt(ArNumPtr *arnum*, **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 ARNUM_SortedInsertElt(ArNumPtr *arnum*, **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 ARNUM_SortedUniqInsertElt(ArNumPtr *arnum*, **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 ARNUM_RemoveNthElt(ArNumPtr *arnum*, **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 ARNUM_RemoveElt(ArNumPtr *arnum*, **ARNUM_ELT** *elt*);

Removes the first occurence 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 ARNUM_ExtractNthElt(ArNumPtr arnum, ArrayIVal *i*);

Removes the element at index 'I'. This call preserves the relative ordering of the ARNUM elements.

ExtractElt

Same as corresponding ARNUM_Remove calls but preserve the relative ordering of the elements in the ARNUM.

SortedExtractElt

ArrayIVal ARNUM_SortedExtractElt(ArNumPtr *arnum*, **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 ARNUM_Sort(ArNumPtr *arnum*, **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 ARNUM_IsSorted(ArNumCPtr *arnumc*, **CmpProc** *proc*);

Returns whether a is sorted or not according to 'proc'.

Removing Duplicates

RemoveDupls

void ARNUM_RemoveDupls(ArNumPtr *arnum*);

Removes duplicate elements in the ARNUM.

SortedRemoveDupls

void ARNUM_SortedRemoveDupls(ArNumPtr *arnum***);**

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.

Chapter

7 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 MAXINT32 / sizeof(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
`AROBJ_DECLARE_CLASS(AROBJ_ELT, AROBJ_KEY)`	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.
`AROBJ_IMPLEMENT_CLASS(AROBJ_ELT, AROBJ_KEY)`	Implements 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.

AROBJ_ELT and AROBJ_KEY should be the same as the values passedto 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.
.
 }
```

Identifer	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 AROBJ_ConstructObjs(ArrayIVal *start*, ArrayIVal *num*);

Default AROBJ construction.

Constructs the array with `len' elements. The elements are initialized using the default constructor for AROBJ_ELT.

Destructor

void AROBJ_DeleteObjs(ArrayIVal *start*, ArrayIVal *num*);

Destroys the array and all its elements..

Clone, Copy, Reset

Reset

void AROBJ_Reset(void);

Resets the contents of the array. After this call, the length of the array will be 0.

Changing the Length

SetLen

void AROBJ_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 AROBJ_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 AROBJ_GetLen(void);

Returns the number of elements in the array.

IsEmpty

BoolEnum NDArObjOf*AROBJ_ELT*::IsEmpty(void);

BoolEnum AROBJ_IsEmpty(void);

Returns whether the array is empty.

IsInRange

BoolEnum AROBJ_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* AROBJ_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* AROBJ_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* AROBJ_GetNthEltRef(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 AROBJ_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 toan 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 AROBJ_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 AROBJ_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 AROBJ_FindElt(const *AROBJ_ELT elt*);

Same as AROBJ_Lookup routine but signal a failure if `elt'.

SortedLookupElt

BoolEnum AROBJ_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 AROBJ_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 AROBJ_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 AROBJ_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 AROBJ_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 AROBJ_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 AROBJ_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 AROBJ_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

void AROBJ_RemoveElt(const AROBJ_ELT *elt*);

Removes the first occurence 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 AROBJ_ExtractNthElt(ArrayIVal *i*);

Removes the element at index `i'. This call preserves the relative ordering of the array elements.

ExtractElt

void AROBJ_ExtractElt(const AROBJ_ELT *elt*);

Same as corresponding NDArObjOfAROBJ_ELT::RemoveElt call but preserves the relative ordering of the elements in the array.

SortedExtractElt

ArrayIVal AROBJ_SortedExtractElt(CmpProc *cmp*, const AROBJ_ELT *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 AROBJ_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 AROBJ_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 AROBJ_RemoveDupls(void);

Removes duplicate elements in the array. The elements are compared using the == operator of AROBJ_ELT.

SortedRemoveDupls

void AROBJ_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.

Chapter 8

ArPtr Class

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. ArPtrs grow as the number of elements stored in the array increases.

The ArPtr classes perform shallow copies: when an array object is copies 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
ARPTR	Type of the ArPtr
ARPTR_ELT	Type of the ArPtr element
ARPTR_KEY	Type of the search key

Macros

This API provides a set of macros that can be use 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

void ARPTR_Construct(ArPtrPtr *arptr***);**

Default ARPTR constructor.

void ARPTR_ConstructLen(ArPtrPtr *arptr***, 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.

void ARPTR_ConstructAlloc (ArPtrPtr *artpr***, ArrayIVal** *alloc***);**

Constructs the ARPTR with 0 elements but a buffer allocated for `alloc' elements.

Then, you may fill the array by calling ARPTR_AppendElt and the array logic will not need to reallocate the buffer as long as the number of elements does not exceed `alloc'.

void ARPTR_ConstructArPtr (ArPtrPtr *arptr***, 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

void ARPTR_Destruct(ArPtrPtr *arptr***);**

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 ARPTR_Reset(ArPtrPtr *arptr***);**

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 ARPTR_SetLen(ArPtrPtr *arptr*, **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 ARPTR_SetAlloc(ArPtrPtr *arptr*, **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 ARPTR_GetLen(ArPtrCPtr *arptr*);

Returns the number of elements in the ARPTR.

IsEmpty

BoolEnum ARPTR_IsEmpty(ArPtrCPtr *arptr*);

Returns whether the ARPTR is empty or not.

IsInRange

BoolEnum ARPTR_IsInRange(ArPtrCPtr *arptr*, **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 ARPTR_GetNthElt(ArPtrCPtr *arptr*, **ArrayIVal** *i*);

Returns the element at index `I'. Fails if the index is not in the [0, len-1] range.

GetNthEltAddr

ARPTR_ELTPtr ARPTR_GetNthEltAddr(ArPtrCPtr *arptr*, **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_ELT ARPTR_UnboundedGetNthElt(ArPtrCPtr *arptr*, **ArrayIVal** *i*);

Same as ARPTR_GetNthElt but returns 0 if `i' is out of range instead of failing.

SetNthElt

void ARPTR_SetNthElt(ArPtrPtr *arptr*, **ArrayIVal** *i*, **ARPTR_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 ARPTR_UnboundedSetNthElt(ArPtrPtr *arptr*, **ArrayIVal** *i*, **ARPTR_ELT** *elt*);

Same as ARPTR_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 (ARPTR_SortedLookup and ARPTR_SortedFind) uses a binary dichotomy, which makes it efficient even on large sorted arrays (search time grows in O(log(n))).

ContainsElt

BoolEnum ARPTR_ContainsElt(ArPtrCPtr *arptr*, **ARPTR_ELT** *elt*);

Returns whether or not the ARPTR contains `elt'.

LookupEltArrayIVal ARPTR_LookupElt(ArPtrCPtr *arptr*, **ARPTR_ELT** *elt*);

Returns the index of the first occurrence of `elt' in the ARPTR.
Returns -1 if the ARPTR does not contain `elt'.

FindElt

ArrayIVal ARPTR_FindElt(ArPtrCPtr *arptr*, **ARPTR_ELT** *elt*);

Same as ARPTR_Lookup routines but signal a failure if the ARPTR does not contain `elt'.

SortedLookupElt

BoolEnum ARPTR_SortedLookupElt(ArPtrCPtr *arptr*, **CmpProc** *proc*, **ARPTR_KEY** *key*, **ArrayIValPtr** *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

ArrayIVal ARPTR_SortedFindElt(ArPtrCPtr *arptr*, **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 mach occured. If no match is found, this routine signals a failure.

Adding Elements

AppendElt

void ARPTR_AppendElt(ArPtrPtr *arptr*, **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 ARPTR_UniqAppendElt(ArPtrPtr *arptr*, **ARPTR_ELT** *elt*);

Appends `elt' to the ARPTR if `elt' is not already in the ARPTR.

InsertNthElt

void ARPTR_InsertNthElt(ArPtrPtr *arptr*, **ArrayIVal** *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

ArrayIVal ARPTR_SortedInsertElt(ArPtrPtr *arptr*, **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 ARPTR_SortedUniqInsertElt(ArPtrPtr *arptr*, **CmpProc** *proc*, **ARPTR_ELT** *elt*);

Same as ARPTR_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 ARPTR_RemoveNthElt(ArPtrPtr *arptr*, **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. ARPTR_ExtractNthElt preserves the ordering but is less efficient.

RemoveElt

void ARPTR_RemoveElt(ArPtrPtr *arptr*, **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 ARPTR_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 ARPTR_ExtractNthElt(ArPtrPtr *arptr*, **ArrayIVal** *i***);**

Removes the element at index `I'. This call preserves the relative ordering of the ARPTR elements.

ExtractElt

void ARPTR_ExtractElt(ArPtrPtr *arptr*, **ARPTR_ELT** *elt***);**

Same as corresponding ARPTR_Remove calls but preserve the relative ordering of the elements in the ARPTR.

SortedExtractElt

ArrayIVal ARPTR_SortedExtractElt(ArPtrPtr *arptr*, **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 ARPTR_Sort(ArPtrPtr *arptr*, **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 ARPTR_IsSorted(ArPtrCPtr *arptrc*, **CmpProc** *proc***);**

Returns whether a is sorted or not according to `proc'.

Removing Duplicates

RemoveDupls

void ARPTR_RemoveDupls(ArPtrPtr *arptr***);**

Removes duplicate elements in the ARPTR.

SortedRemoveDupls

void ARPTR_SortedRemoveDupls(ArPtrPtr *arptr***);**

Removes duplicates in the ARPTR, assumes that it is sorted. This routine is more efficient than ARPTR_RemoveDupls because duplicates are necessarily contiguous in this case.

Chapter

9 ARRay Class

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.

Chapter

10 ARRec Class

The ArRec module implements the generic collection of records.

Overview

A n 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 anddeallocation 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
ARREC	Type of the array
ARREC_ELT	Type of the array element
ARREC_KEY	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

void ARREC_Construct(ArRecPtr *arrec***);**

Default ARREC construction

void ARREC_ConstructLen(ArRecPtr *arrec***, ArrayIval** *len***);**

Constructs the ARREC with len elements. The contents of the ARREC is initialized with zeros.

Destructor

void ARREC_Destruct(ArRecPtr *arrec***);**

Default ARREC destruction. All the records stored in the arrec are lost at this point.

Clone, Copy, Reset

Reset

void ARREC_Reset(ArRecPtr *arrec***);**

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 ARREC_SetLen(ArRecPtr *arrec***, 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 ARREC_SetAlloc(ArRecPtr *arrec***, 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 ARREC_GetLen(ArRecCPtr *arrec***);**

Returns the number of elements in the ARREC.

IsEmpty

BoolEnum ARREC_IsEmpty(ArRecCPtr *arrec***);**

Returns whether the ARREC is empty or not.

IsInRange

BoolEnum ARREC_IsInRange(ArRecCPtr *arrec***, ArrayIVal** *i***);**

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 ARREC_GetNthElt(ArRecCPtr *arrec***, ArrayIVal** *i***);**

Returns the address of the element at index I. Fails if the index is not in the [0, len-1] range.

SetNthElt

void ARREC_SetNthElt(ArRecPtr *arrec***, 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 ARREC_ContainsElt(ArRecCPtr *arrec***, ARREC_ELTPtr** *elt***);**

Returns whether or not the ARREC contains elt.

LookupElt

ArrayIVal ARREC_LookupElt(ArRecCPtr *arrec*, **ARREC_ELTCPtr** *elt***);**

Returns the index of the first occurrence of elt in the ARREC. Returns -1 if the ARREC does not contain elt.

FindElt **ArrayIVal ARREC_FindElt(ArRecCPtr** *arrec*, **ARREC_ELTCPtr** *elt***);**

Same as ARREC_Lookup routine but signal a failure if elt.

SortedLookupElt

BoolEnum ARREC_SortedLookupElt(ArRecCPtr *arrec*, **CmpProc** *proc*, **ARREC_KEY** *key*, **ArrayIValPtr** *result***);**

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 ARREC_SortedFindElt(ArRecCPtr *arrec*, **CmpProc** *proc*, **ARREC_KEY** *key***);**

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 ARREC_AppendElt(ArRecPtr *arrec*, **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 ARREC_UniqAppendElt(ArRecPtr** *arrec*, **ARREC_ELTPtr** *elt***);**

Appends elt to the ARREC if elt is not already in the ARREC.

InsertNthElt

void ARREC_InsertNthElt(ArRecPtr *arrec*, **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.

SortedInsertEltArrayIVal ARREC_SortedInsertElt(ArRecPtr *arrec*, **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 ARREC_SortedUniqInsertElt(ArRecPtr *arrec*, **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 ARREC_RemoveNthElt(ArRecPtr *arrec*, **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 ARREC_RemoveElt(ArRecPtr *arrec*, **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.

ExtractNthEltvoid ARREC_ExtractNthElt(ArRecPtr *arrec*, **ArrayIVal** *i***);**

Removes the element at index I. This call preserves the relative ordering of the ARREC elements.

ExtractEltvoid ARREC_ExtractElt(ArRecPtr *arrec*, **ARREC_ELTPtr** *elt***);**

Same as corresponding ARREC_Remove calls but preserve the relative ordering of the elements in the ARREC.

SortedExtractEltArrayIVal ARREC_SortedExtractElt(ArRecPtr *arrec*, **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 ARREC_Sort(ArRecPtr *arrec*, **CmpProc** *proc***);**

Sorts the ARREC by passing the address of the elements instead of the elements themselves to the comparison routine.

IsSorted

BoolEnum ARREC_IsSorted(ArRecCPtr *arrecc*, **CmpProc** *proc*);

Returns whether a is sorted or not according to proc.

Removing Duplicates

RemoveDupls

void ARREC_RemoveDupls(ArRecPtr *arrec*);

Removes duplicate elements in the ARREC.

SortedRemoveDupls

void ARREC_SortedRemoveDupls(ArRecPtr *arrec*);

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.

Chapter

11 Avl Class

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
`Nearest`	closest node found
`NearCmp`	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

Alloc

AvlNodePtr AVL_Node(void);

Default allocator. It returns a pointer to an allocated AvlNode. The AvlNode is at this point not yet constructed.

Constructors

void AVL_NodeConstruct(AvlNodePtr *avlnode***);**

Default construction.

void AVL_NodeConstructKey(AvlNodePtr *avlnode***, ClientPtr** *key***);**

Constructs the node, and assigns ’key’ to be its key.

Destructor

void AVL_NodeDestruct(AvlNodePtr *avlnode***);**

Default destructor for AvlNodes.

Dealloc

void AVL_NodeDealloc(AvlNodePtr *avlnode***);**

Deallocates the AvlNode. The AvlNode should have been destructed first.

Convenience Functions

These functions are provided for allocation and construction, destruction and deallocation.

NodeNewSetKey

AvlNodePtr AVL_NodeNewSetKey(ClientPtr *key***);**

Creates a new AvlNode with `key' as key.

NodeDispose

void AVL_NodeDispose(AvlNodePtr *avlnode***);**

Destroys `avlnode'. The contents of the AvlNodeKey field will not be disposed by this call and should be disposed (if necessary) just before issuing this call.

Accessing the AvlNode Key

SetKey

void AVL_NodeSetKey(AvlNodePtr *avlnode*, **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 AVL_NodeGetKey(AvlNodeCPtr *avlnode***);**

Returns the key stored in the AvlNode object.

Scanning AvlNodes

GetPrev
GetNext

AvlNodePtr AVL_NodeGetPrev(AvlNodePtr *avlnode***);**

AvlNodePtr AVL_NodeGetNext(AvlNodePtr *avlnode***);**

Return previous and next avlnode (sorted according to key comparison proc).

GetParent
GetLeftChild
GetRightChild

AvlNodePtr AVL_NodeGetParent(AvlNodePtr *avlnode***);**

AvlNodePtr AVL_NodeGetLeftChild(AvlNodePtr *avlnode***);**

AvlNodePtr AVL_NodeGetRightChild(AvlNodePtr *avlnode***);**

Return respectively the parent, left child or right child AvlNode of the current node.

GetFirstLeaf
GetLastLeaf

AvlNodePtr AVL_NodeGetFirstLeaf(AvlNodePtr *avlnode***);**

AvlNodePtr AVL_NodeGetLastLeaf(AvlNodePtr *avlnode***);**

Return respectively the leftmost and rightmost descendant node of the AvlNode

AvlTree Class

Constructors and Destructor

Alloc

AvlTreePtr AVL_TreeAlloc(void);

Returns a pointer to an allocated AvlTree. The Avl tree is at this point not yet constructed.

Constructors

void AVL_TreeConstruct(AvlTreePtr *avltree***);**

Default constructor for Avl trees.

void AVL_TreeConstructCmpProc(AvlTreePtr *avltree***, 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

void AVL_TreeDestruct(AvlTreePtr *avltree***);**

Default destructor for Avl trees.

Dealloc

void AVL_TreeDealloc(AvlTreePtr *avltree***);**

Default deallocation for an AvlTree.

Queries

GetLen

AvlIVal AVL_TreeGetLen(AvlTreeCPtr *avltreec***);**

Returns the number of nodes in the tree.

GetFirstNode
GetLastNode

AvlNodePtr AVL_TreeGetFirstNode(AvlTreeCPtr *avltreec***);**

AvlNodePtr AVL_TreeGetLastNode(AvlTreeCPtr *avltreec***);**

Return the first/last AvlNodes in the tree (sorted according to the key comparison proc).

CurFindKeyKey

AvlNodePtr AVL_TreeCurFindKeyKey(AvlTreePtr *avltree*, **ClientCPtr** *key***);**

This call returns the AvlNode which matches ’key’ and fails if no node matches key.

LookupKey

AvlNodePtr AVL_TreeLookupKey(AvlTreePtr *avltree*, **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 AVL_TreeInsertNode(AvlTreePtr *avltree*, **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 AVL_TreeExtractNode(AvlTreePtr *avltree*, **AvlNodePtr** *avlnode***);**

This call extracts ’avlnode’ from ’avltree’. This call rebalances the tree if necessary.

Propagating an Action

PerfProc

PerfEnum AVL_TreePerfProc(AvlNodeCPtr *avlnode*, **ClientPtr** *arg***);**

Callback function used when propagating an action.

PropagateAction

PerfEnum AVL_TreePropagateAction(AvlTreeCPtr *avltreec*, **AvlTreePerfProc** *proc*,
ClientPtr *arg***);**

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 AVL_TreeGoFirstNode(AvlTreePtr *avltree***);**

void AVL_TreeGoLastNode(AvlTreePtr *avltree***);**

Positions the current node on the first or last node of the tree (sorted according to the key comparison proc).

GoPrevNode
GoNextNode

void AVL_TreeGoPrevNode(AvlTreePtr *avltree***);**

void AVL_TreeGoNextNode(AvlTreePtr *avltree***);**

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 AVL_TreeGoNode(AvlTreePtr *avltree***, AvlNodePtr** *avlnode***);**

Sets the current node to be ’avlnode’ (which must belong to the AvlTree).

CurGetNode

AvlNodePtr AVL_TreeCurGetNode(AvlTreeCPtr *avltreec***);**

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 AVL_TreeCurGetNearestNode(AvlTreeCPtr *avltree***, CmpEnumPtr** *cmpp***);**

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. *cmpp 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 AVL_TreeCurFindKey(AvlTreePtr *avltree*, **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 AVL_TreeCurInsertNode(AvlTreePtr *avltree*, **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 AVL_TreeCurExtractNode(AvlTreePtr *avltree***);**

Extracts a node from the AvlTree. A call to AVL_TreeCurFindKey MUST have been done just before.

Chapter

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
`MAXINT8`	Maximum 8 bit signed integer (127).
`MAXINT16`	Maximum 16 bit signed integer (32,767).
`MAXINT32`	Maximum 32 bit signed integer (2,147,483,647).
`MAXINT64`	Maximum 64 bit signed integer (2,147,483,647). Not supported by all operating systems, specifically DOS.
`MAXUINT8`	Maximum 8 bit unsigned integer (255).
`MAXUINT16`	Maximum 16 bit unsigned integer (65535).
`MAXUINT32`	Maximum 32 bit unsigned integer (4,294,967,295).
`MAXUINT64`	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*);

IINT_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.

```
#ifdef 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**
`DBG_FILE`	Current file name.
`DBG_LINE`	Current line number.

See also

DBG_CHECK, DBG_ERROR, DBG_NIY, DBG_ON, DBG_SOURCE

DBG_NIY

Signals a warning to the error handler.

```
#ifdef 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.

```
#ifdef 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)
```

Chapter

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 accomodate 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 communcation 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 L1(Str, name)
{
      FilePtr             file = new NdFile(name);
      Uint32\        uint32;
      file->Open(FILE_IOREAD, FILE_FMTBINARY);
      NdBBuf         bbuf(file, 1, 512);
      bbfuf->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 L1(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 endianity:

   When reading multi-bytes integers (Int16, Int32, UInt16 or UInt32), the order of the bytes in a file is not necessary 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
`BBUF_ENDIAN`	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:
`BBUF_ENDIANBIG`	Most significant byte is stored first (example: MacPaint).
`BBUF_ENDIANLITTLE`	Least significant byte is stored first (example: Gif).
`BBUF_ENDIANNATIVE`	Bytes are stored in the same order as on the local machine (order given by MCH_ENDIAN).
`BBUF_ENDIANREVERSE`	Bytes are stored in the reverse order (relative to the native order given by MCH_ENDIAN).
`BBUF_ENDIANVARIABL E`	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 BBUF_SetEndianity (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
`SeekProc`	SeekProc(bbuf, pos) should move the current position to pos and load the page containing the byte at current position. The PageBeginPos, PageBeginPtr, PageEndPtr and CurPtr should be updated.
`ReadProc`	should just skip n bytes. The method should fail if an attemp to read past the end of data is made. The PageBeginPos, PageBeginPtr, PageEndPtr and CurPtr 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 attemp 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

Alloc

BBufPtr BBUF_Alloc(void);

Returns a pointer to an allocated BBuf. The BBuf is not yet constructed and needs to be constructed before being used.

NDBinBuf

void BBUF_Construct(BBufPtr *bbuf***);**

Default construction.

void BBUF_ConstructBuf(BBufPtr *bbuf***, BBufBytePtr** *data***, BBufOffsetVal** *len***);**

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

void BBUF_ConstructFile(BBufPtr *bbuf***, FilePtr** *file***, BBufPageVal** *maxbufs***, BBufOffsetVal** *bufsize***);**

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

void BBUF_ConstructData(BBufPtr *bbuf***, 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

void BBUF_Destruct(BBufPtr *bbuf***);**

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.

Dealloc

void BBUF_Dealloc(BBufPtr *bbuf***);**

Deallocates the bbuf. The bbuf must have been allocated using BBUF_Alloc.

Read and Write Operations

ReadNBytes

void BBUF_ReadNBytes(BBufPtr *bbuf***, 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 BBUF_ReadInt8(BBufPtr *bbuf***, Int8Ptr** *valptr***);**

void BBUF_ReadUInt8(BBufPtr *bbuf***, UInt8Ptr** *valptr***);**

void BBUF_ReadInt16(BBufPtr *bbuf***, Int16Ptr** *valptr***);**

void BBUF_ReadUInt16(BBufPtr *bbuf***, UInt16Ptr** *valptr***);**

void BBUF_ReadInt32(BBufPtr *bbuf***, Int32Ptr** *valptr***);**

void BBUF_ReadUInt32(BBufPtr *bbuf***, UInt32Ptr** *valptr***);**

Reads an Int8, Int16, Int32, UInt8, UInt16 or a UInt32 respectively and writes it into valptr.

WriteNBytes

void BBUF_WriteNBytes(BBufPtr *bbuf***, 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 BBUF_WriteInt8(BBufPtr *bbuf*, **Int8** *val***);**

void BBUF_WriteInt16(BBufPtr *bbuf*, **Int16** *val***);**

void BBUF_WriteInt32(BBufPtr *bbuf*, **Int32** *val***);**

void BBUF_WriteUInt8(BBufPtr *bbuf*, **UInt8** *val***);**

void BBUF_WriteUInt16(BBufPtr *bbuf*, **UInt16** *val***);**

void BBUF_WriteUInt32(BBufPtr *bbuf*, **UInt32** *val***);**

Writes an Int8, Int16, Int32, UInt8, UInt16, UInt32 respectively into the bbuf.

Flush

void BBUF_Flush(BBufPtr *bbuf***);**

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 BBUF_CurPos(BBufCPtr *bbuf***);**

Returns current position.

SeekTo

void BBUF_SeekTo(BBufPtr *bbuf*, **BBufOffsetVal** *pos***);**

Sets position to absolute offset. The new position must be between 0 and TotalSize-1.

SeekBy

void BBUF_SeekBy(BBufPtr *bbuf*, **BBufOffsetVal** *pos***);**

Sets position to offset relative to current position. The new position must be between 0 and TotalSize-1.

SkipRead

void BBUF_SkipRead(BBufPtr *bbuf*, **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 BBUF_SkipWrite(BBufPtr *bbuf*, **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 BBUF_LoadCurPage(BBufPtr *bbuf***);**

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

ClientPtr BBUF_GetClientData(BBufCPtr *bbuf***);**

void BBUF_SetClientData(BBufPtr *bbuf***, 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.

GetEndianity

EndianEnum BBUF_GetEndianity(BBufCPtr *bbuf***);**

void BBUF_SetEndianity(BBufPtr *bbuf***, EndianEnum** *endian***);**

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_SetEndianity to either ENDIAN_BIG or ENDIAN_LITTLE.

GetTotalSize
SetTotalSize

BBufOffsetVal BBUF_GetTotalSize(BBufCPtr *bbuf***);**

void BBUF_SetTotalSize(BBufPtr *bbuf***, 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 attemp 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 BBUF_GetPagingData(BBufCPtr *bbuf***);**

void BBUF_SetPagingData(BBufPtr *bbuf***, 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 BBUF_IsPageModified(BBufCPtr *bbuf***);**

void BBUF_SetPageModified(BBufPtr *bbuf***, BoolEnum** *mod***);**

Respectively, returns BOOL_TRUE if current page has been modified,BOOL_FALSE otherwise, and sets / unsets the PageModified flag.

GetPageBeginPos
SetPageBeginPos

BBufOffsetVal BBUF_GetPageBeginPos(BBufCPtr *bbuf***);**

void BBUF_SetPageBeginPos(BBufPtr *bbuf***, 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 BBUF_GetPageBeginPtr(BBufCPtr *bbuf***);**

void BBUF_SetPageBeginPtr(BBufPtr *bbuf***, 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 BBUF_GetPageEndPtr(BBufCPtr *bbuf***);**

void BBUF_SetPageEndPtr(BBufPtr *bbuf***, 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 BBUF_GetCurPtr(BBufCPtr *bbuf***);**

void BBUF_SetCurPtr(BBufPtr *bbuf***, 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 BBUF_QueryMethods(BBufCPtr *bbuf***, BBufMethodsPtr** *methods***);**

Fills methods with the methods installed in the bbuf.

SetMethods

void BBUF_SetMethods(BBufPtr *bbuf***, BBufMethodsPtr** *methods***);**

Installs the methods in methods in the bbuf.

Chapter 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.

BoolEnum RANGE_ContainsCell (RangeRec *range*, CellPtr *cell*);

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.

Chapter 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 CHAR_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.

StrIVal CHAR_GetLen (Char *ch***);**

CHAR_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

CHAR_CodeGetLen, CHAR_NatGetLen

CodeGetLen

Returns the length of the character whose first byte contains the specified 8-bit character code.

StrIVal CHAR_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

CHAR_GetLen, CHAR_NatGetLen

NatGetLen

Returns the length of the native character whose first byte contains the specified 8-bit character code.

StrIVal CHAR_NatGetLen (NatChar *natCh***);**

CHAR_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

CHAR_GetLen, CHAR_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.

Char CHAR_GetByte (ChCode *chcode*, **Int** *bytenum***);**

Char CHAR_GetByte1 (ChCode *chcode***);**

Char CHAR_GetByte2 (ChCode *chcode***);**

Char CHAR_GetByte3 (ChCode *chcode***);**

Returns the contents of the specified byte of a multibyte character.

The CHAR_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).

CHAR_GetByte1 obtains the first byte, CHAR_GetByte2 obtains the second byte, and CHAR_GetByte3 obtains the third byte of a multibyte character.

See also

CHAR_NatGetByte

NatGetByte...

Returns the contents the specified byte of a multibyte character.

NatChar CHAR_NatGetByte (NatCode *natcode*, **Int** *byteEnum***);**

NatChar CHAR_NatGetByte1 (NatCode *natcode***);**

NatChar CHAR_NatGetByte2 (NatCode *natcode***);**

NatChar CHAR_NatGetByte3 (NatCode *natcode***);**

Returns the contents of the specified byte of a native multibyte character.

The CHAR_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). CHAR_GetByte1 obtains the first byte, CHAR_GetByte2 obtains the second byte, and CHAR_GetByte3 obtains the third byte of a native multibyte character.

See also

CHAR_GetByte

Basic Character Classification

IsAscii...

Determines whether the character is ASCII.

BoolEnum CHAR_IsAscii (ChCode *chcode***);**

BoolEnum CHAR_IsAsciiAlpha (ChCode *chcode***);**

BoolEnum CHAR_IsAsciiUpper (ChCode *chcode***);**

BoolEnum CHAR_IsAsciiLower (ChCode *chcode***);**

BoolEnum CHAR_IsAsciiAlNum (ChCode *chcode***);**

BoolEnum CHAR_IsAsciiDigit (ChCode *chcode***);**

BoolEnum CHAR_IsAsciiHexDigit (ChCode *chcode***);**

BoolEnum CHAR_IsAsciiOctDigit (ChCode *chcode***);**

BoolEnum CHAR_IsAsciiSpace (ChCode *chcode***);**

BoolEnum CHAR_IsAsciiPunct (ChCode *chcode***);**

BoolEnum CHAR_IsAsciiControl (ChCode *chcode***);**

BoolEnum CHAR_IsAsciiPrint (ChCode *chcode***);**

BoolEnum CHAR_IsAsciiGraph (ChCode *chcode***);**

The CHAR_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 CHAR_IsAscii... macros automatically assume that the chcode argument is an EBCDIC character code, not an ASCII code.

The various versions of the CHAR_IsAscii... macros are described in the following table:

Macro	Inquiry
`CHAR_IsAscii(chcode)`	Does the character belong to the ASCII set?
`CHAR_IsAsciiAlpha(chcode)`	Is the character an ASCII letter?
`CHAR_IsAsciiUpper(chcode)`	Is the character an ASCII upper case letter?
`CHAR_IsAsciiLower(chcode)`	Is the character an ASCII lower case letter?
`CHAR_IsAsciiAlNum(chcode)`	Is the character an ASCII letter or a digit?
`CHAR_IsAsciiDigit(chcode)`	Is the character an ASCII digit?
`CHAR_IsAsciiHexDigit(chcode)`	Is the character an ASCII hexadecimal digit?
`CHAR_IsAsciiOctDigit(chcode)`	Is the character an ASCII octal digit?
`CHAR_IsAsciiSpace(chcode)`	Is the character an ASCII space character?
`CHAR_IsAsciiPunct(chcode)`	Is the character an ASCII punctuation?

`CHAR_IsAsciiControl(chcode)`	Is the character an ASCII control character?
`CHAR_IsAsciiPrint(chcode)`	Is the character an ASCII printable character?
`CHAR_IsAsciiGraph(chcode)`	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

CHAR_AsciiIs...

AsciiIs...

Same as CHAR_IsAscii... macros except that an error is generated if the character is not ASCII.

BoolEnum CHAR_AsciiIsAlpha (Char *ch*);

BoolEnum CHAR_AsciiIsUpper (Char *ch*);

BoolEnum CHAR_AsciiIsLower (Char *ch*);

BoolEnum CHAR_AsciiIsAlNum (Char *ch*);

BoolEnum CHAR_AsciiIsDigit (Char *ch*);

BoolEnum CHAR_AsciiIsHexDigit (Char *ch*);

BoolEnum CHAR_AsciiIsOctDigit (Char *ch*);

BoolEnum CHAR_AsciiIsSpace (Char *ch*);

BoolEnum CHAR_AsciiIsPunct (Char *ch*);

BoolEnum CHAR_AsciiIsControl (Char *ch*);

BoolEnum CHAR_AsciiIsPrint (Char *ch*);

BoolEnum CHAR_AsciiIsGraph (Char *ch*);

The CHAR_AsciiIs... functions are similar to the corresponding CHAR_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

CHAR_IsAscii

Basic Character Conversion

AsciiDigitGetInt
AsciiHexDigitGetInt
AsciiOctDigitGetInt

Returns the integer value of an ASCII digit.

Int CHAR_AsciiDigitGetInt (Char *ch*);

Int CHAR_AsciiHexDigitGetInt (Char *ch*);

Int CHAR_AsciiOctDigitGetInt (Char *ch*);

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

CHAR_AsciiDigitGetBase

AsciiAlphaGetBase

Returns the base value of an ASCII letter.

Int CHAR_AsciiAlphaGetBase (Char *ch*);

CHAR_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

CHAR_AsciiDigitGetInt

AsciiGetLower

Converts an ASCII character to lower case.

Char CHAR_AsciiGetLower (Char *ch*);

CHAR_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

CHAR_AsciiGetUpper

AsciiGetUpper

Converts an ASCII character to upper case.

Char CHAR_AsciiGetUpper (Char *ch*);

CHAR_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

CHAR_AsciiGetLower

AsciiGetControl

Converts a character to a control character.

Char CHAR_AsciiGetControl (Char *ch***);**

CHAR_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

CHAR_AsciiGetGraph

AsciiGetGraph

Converts a control character into a character.

Char CHAR_AsciiGetGraph (Char *ch***);**

CHAR_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

CHAR_AsciiGetControl

AsciiGetEbcdic

Converts an ASCII character to an EBCDIC character.

Byte CHAR_AsciiGetEbcdic (Byte *b***);**

CHAR_AsciiGetEbcdi converts an ASCII character to the first byte of an EBCDIC character.

See also

CHAR_EbcdicGetAscii

Conversions between ASCII and EBCDIc

EbcdicGetAscii

Converts an EBCDIC character to an ASCII character.

Byte CHAR_EbcdicGetAscii (Byte *b*);

CHAR_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

CHAR_AsciiGetEbcdic

ToAscii

Converts a native character to ASCII.

Byte CHAR_ToAscii (Char *ch*);

Converts a native character to ASCII. On an ASCII host, this function does nothing.

See also

CHAR_FromAscii

FromAscii

Converts an ASCII code to a native character.

Char CHAR_FromAscii (Byte *b*);

CHAR_FromAscii converts an ASCII code to a native character. On an ASCII host, this function does nothing.

See also

CHAR_ToAscii

Chapter

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_ZAPFDB`	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_1`

`CS_CNS11643_2`

`CS_CNS11643_3`

Creating and Destroying

Alloc

CsPtr CS_Alloc(void);

Returns a pointer to an allocated but not yet constructed code set object. The object should be constructed before being used.

Constructors

void CS_Construct(CsPtr *cs*);

Default code set object constructor.

void CS_ConstructId(CsPtr *cs*, CsIdEnum *csid*);

Constructs the code set object from the `csid' information.

Destructor

void CS_Destruct(CsPtr *cs*);

Default code set object destructor.

Dealloc

void CS_Dealloc(CsPtr *cs*);

Deallocates the notifier.

Convenience Functions

New

CsPtr CS_New(CsIdEnum *csid*);

Creates new code set object and constructs it with csid.

Dispose

void CS_Dispose(CsPtr *cs*);

Destructs and deallocates the code set object.

Dispose0

void CS_Dispose0(CsPtr *cs***);**

Disposes a code set object if it is not NULL.

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 CS_GetCsId(CsCPtr *cs***);**

Get the code set’s id.

GetCharLen

StrIVal CS_GetCharLen(CsCPtr *cs***);**

Get the character length for the code set.

GetCharInfo

CharInfoVal CS_GetCharInfo(CsCPtr *cs***, CsCode** *code***);**

Get the 'charinfo' value of the character `code'.

CvtChar

Converts a character within the code set.

BoolEnum CS_CvtChar(CsCPtr *cs***, 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 CS_TransChar(CsCPtr *cs*, **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 CS_ToUni(CsCPtr *cs*, **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 CS_FromUni(CsCPtr *cs*, **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

CsPtr CS_GetCsNative(void);

Returns a pointer to the native code set.

GetCsUnicode

CsPtr CS_GetCsUnicode(void);

Returns a pointer to the Unicode code set.

GetCsGlobal

CsPtr CS_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`	

Chapter

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

CT_GetCharLen, CT_GetFwrd, CT_GetBwrd, CT_GetInfo, CT_CvtChar, CT_CvtCtToCs, CT_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 CT_ISO_LATIN2 CT_ISO_LATIN3 CT_ISO_LATIN4 CT_ISO_CYRILLIc CT_ISO_ARABIc CT_ISO_GREEK CT_ISO_HEBREW CT_ISO_LATIN9	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.

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 CT_MSW_CYRILLIc CT_MSW_ANSI CT_MSW_GREEK CT_MSW_TURK CT_MSW_HEBREW CT_MSW_ARABIc	1252 is ASCII + MSW_ANSI + ISO_LATIN1 125X is ASCII + MSW_125X

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 <= X <= 7 are defined.

Creating and Disposing

Alloc

CtPtr CT_Alloc (void);

Returns a pointer to an allocated but not yet constructed Ct. The Ct should be constructed before being used.

Construct

void CT_Construct (CtPtr *ct*);

Default Ct construction.

ConstructId

void CT_ConstructId (CtPtr *ct*, 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.

Destruct

void CT_Destruct (CtPtr *ct*);

Default Ct destruction.

Dealloc

void CT_Dealloc (CtPtr *ct*);

Deallocates the notifier.

New

CtPtr CT_New (CtIdEnum *ctid*);

Creates new code type object and constructs it with `ctid'.

Dispose

void CT_Dispose (CtPtr *ct***);**

Destructs and deallocates the code type object.

Dispose0

void CT_Dispose0 (CtPtr *ct***); */**

Destructs and deallocates the code type object if it is not NULL.

Member Functions

GetCtId

Gets the code type id.

CtIdEnum CT_GetCtId (CtCPtr *ct***);**

CT_GetCtId returns the code type id from the code type data record structure.

See also

Ct.

GetFwrd

Returns the value of the character found at the beginning of a string.

ChCode CT_GetFwrd (CtCPtr *ct***, CStr** *str***, StrIValPtr** *lenp***);**

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

CT_GetBwrd, Ct.

GetBwrd

Returns the character code for the character found in front of a given index in a string.

ChCode CT_GetBwrd (CtCPtr *ct***, CStr** *str***, StrIVal** *pos***, StrIValPtr** *lenp***);**

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 CT_GetBwrd macro can be called for any code type. For more information, see Ct.

See also

CT_GetFwrd, Ct

GetInfo

Returns the CharInfoVal for a character.

CharInfoVal CT_GetInfo (CtCPtr *ct***, ChCode** *ch***);**

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 CT_CvtChar (CtCPtr *ct***, ChCode** *in***, CharCvtSet** *flags***, LgEnvCPtr** *lgenv***, ChCodePtr** *out***);**

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)
`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 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, CT_CvtCtToCs, CT_CvtCsToCt

CvtCtToCs

Converts a character code from its code type form to its code set form.

CsCode CT_CvtCtToCs (CtCPtr *ct***, ChCode** *ch***, CsPtr*** *cs***);**

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

CT_CvtCsToCt, Ct

CvtCsToCt

Converts a character code from its code set form to its code type form.

ChCode CT_CvtCsToCt (CtCPtr *ct***, CsCode** *code***, CsCPtr** *cs***);**

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

CT_CvtCtToCs, Ct

ToUni

Converts chcode to unicode.

BoolEnum CT_ToUni (CtCPtr *ct***, ChCode** *ch***, UniCodePtr** *uni***);**

If the chcode cannot be converted to UNICODE it returns FALSE; otherwise, TRUE.

FromUni

Converts unicode to chcode.

BoolEnum CT_FromUni (CtCPtr *ct***, 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 CT_GetMaxCharLen (CtCPtr *ct***);**

CT_GetMaxCharLen returns the maximum character length supported by a code type.

IsSingleOnly

Determines whether a code type defines single-byte characters only.

BoolEnum CT_IsSingleOnly (CtCPtr *ct***);**

CT_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 CT_GetUpper (CtCPtr *ct***, ChCode** *chcode***);**

CT_Get Upper returns the upper case form of a character.

See also

CT_GetLower

GetLower

Returns the lower case form of a character.

ChCode CT_GetLower (CtCPtr *ct*, **ChCode** *chcode***);**

CT_GetLower returns the lower case form of a character.

See also

CT_GetUpper

Chapter

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

RClasPtr DS_Class(void);

Returns a pointer to the DataSource class.

View Interface

RegisterView

void DS_RegisterView(DsPtr *ds*, **ResPtr** *view***);**

Register the resource view with the data source.

SetViewOption

void DS_SetViewOption(DsPtr *ds*, **ResPtr** *view*, **CStr** *option*, **CStr** *info***);**

Set info as the option for the view registered in the data source.

GetViewOption

CStr DS_GetViewOption(DsCPtr *ds*, **ResCPtr** *view*, **CStr** *option***);**

Returns the string corresponding to option for the view registered with the data source.

UnregisterView

void DS_UnregisterView(DsPtr *ds*, **ResPtr** *view***);**

Unregisters view from the data source.

ViewGetDs

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
`DSEDIT_COMPLETIONOK`	The edition was successfully completed
`DSEDIT_COMPLETIONFAILED`	The edition failed for some reason
`DSEDIT_COMPLETIONPREEMPTED`	The edition was preempted

StartEdit

DsEditPtr DS_StartEdit(DsPtr *ds***);**

Opens an edition on the whole data source. The operations are done through the edition object returned by this call.

End

DsEditCompletionEnum DSEDIT_End(DsEditPtr *dsEdit***);**

Commit the edition on the whole of the data source. The edition object is destroyed and deleted.

Abort

void DSEDIT_Abort(DsEditPtr *dsEdit***);**

Abort the edition on the data source. The edition object is destroyed and delete.

AddOperation

DsEditOpPtr DSEDIT_AddOperation(DsEditPtr *dsEdit***);**

Add an operation to the edition

SetOwner

void DSEDIT_SetOwner(DsEditPtr *dsEdit***, 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 DSEDIT_GetOwner(DsEditPtr *dsEdit***);**

Retrieve owner (if any) of the edition.

Default constructors and destructors for the base DsEdit and DsEditOp classes.

Update Interface

StartUpdateEdit

DsUpdateEditPtr DS_StartUpdateEdit(DsPtr *ds***);**

Opens an update on the whole data source. The operations are done through the edit object returned.

End

void DSUPDATEEDIT_End(DsUpdateEditPtr *dsEdit***);**

Commit the update on the whole of the data source. The edit object is destroyed and deleted.

Abort

void DSUPDATEEDIT_Abort(DsUpdateEditPtr *dsEdit***);**

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 DS_AddContDs(DsPtr *ds***, DsPtr** *contDs***);**

Adds contDs as a contained data source to the data source.

RemoveContDs

void DS_RemoveContDs(DsPtr *ds***, DsPtr** *contDs***);**

Removes contDs from the data source.

Creating and Disposing

Create

DsPtr DS_Create (RClasPtr *rclas***);**

Creates a datasource object.

Class

Edition Operation

DsEditOpEnum

Methods	Description
`DSEDIT_OPENUMINHERIT`	(DSEDITOP)

DsEditTypeEnum

Methods	Description
`DSEDIT_TYPEENUMINHERIT`	(DSEDITTYPE)

DsEditStateEnum

Methods	Description
`DSEDIT_STATECLOSED`	
`DSEDIT_STATEOPEN`	
`DSEDIT_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);

Chapter 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 descibed 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 enlightning 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 fullfill 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 fullfulling 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 fullfill 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 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 the 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 fullfill 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_RETRYIN;
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
`ERR_Fail`	Signals a failure. The error message will be loaded from the "list of strings" resource called Modul.Errors" where "Modul" is the first argument passed to ERR_Fail (usually you pass S_ModuleName, 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 (%d, %lx, ...) in which case you pass additional arguments.
`ERR_FailStr`	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
ERR_Warn	Generates a warning. The warning message will be loaded from the "list of strings" resource called Modul.Warnings" where "Modul" is the first argument passed to ERR_Warn (usually you pass S_ModuleName, 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 (%d, %lx, ...) in which case you pass additional arguments.
ERR_WarnStr	Generates a warning. Instead of loading the message from a resource, the message is hard-coded and passed as first argument. For now, printf 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 participe (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 developper 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
ERR_DECLARE	Declares ERR_LIB.
ERR_LIBCREATEINIT	Allocates, and initializes ERR_LIB for a library.
ERR_ISLIBCREATED	Is BOOL_FALSE if ERR_LIB has not been allocated for a library.
ERR_MAININIT	Initializes ERR_LIB for the `main' routine.

and also:

Item	Description
ERR_EXTERN	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
`Modul.Errors`	Messages associated with failures.
`Modul.Warnings`	Messages associated with warnings.
`Modul.Messages`	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 be 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 beacuse 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 aserts 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 ERR_GetErrFuncCallPtr (void)

Returns a pointer to the error reporting structure.

ErrFrame API for Error Reporting and Discrimination

FrameGetTop

ErrFramePtr ERR_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 ERR_FrameQueryMessage (ErrFrameCPtr *frame*, Str buf, StrIVal *size*)

Formats the frame's end user message into `buf', not writing more than size' characters.

FrameQueryTraceback

void ERR_FrameQueryTraceback (ErrFrameCPtr *frame*, **Str buf, StrIVal** *size***)**

Formats the frame's traceback message into `buf', not writing more than size' characters.

FrameQueryFullTraceback

void ERR_FrameQueryFullTraceback (ErrFrameCPtr *frame*, **VStrPtr** *vstr***)**

Formats the full traceback message into `vstr'.

FrameSetReported

void ERR_FrameSetReported (void)

Marks the error frames to indicate that the error has been reported to the user.

FrameIsReported

BoolEnum ERR_FrameIsReported (void)

Returns whether or not the error has already been reported to the user.

FrameReport

void ERR_FrameReport (ErrFrameCPtr *frame***)**

Invokes the global error reporting routine.

FrameDefReport

void ERR_FrameDefReport (ErrFrameCPtr *frame***)**

Invokes the default error reporting routine.

GetReportProc

ErrReportProc ERR_GetReportProc (void)

Returns the global error reporting procedure.

SetReportProc

void ERR_SetReportProc (ErrReportProc *proc***)**

Overrides the global error reporting procedure.

ErrFrame Class

ErrFrame

Private stuff.

Item	Description
`ErrFramePtr`	Next
`ErrJmpPtr`	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
`ERR_SETOPTINT(val)`	Sets up an optional integer argument for error reporting.
`ERR_SETOPTSTR(str)`	Sets up an optional string argument for error reporting.
`ERR_SETOPTVSTR(vstr)`	Sets up an optional variable string argument for error reporting.

SetReportPrint SetReportSilent Print

void ERR_SetReportPrint (ErrFramePtr *errframe***)**

void ERR_SetReportSilent (ErrFramePtr *errframe***)**

void ERR_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 ERR_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 ERR_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
`ERR_LIBDECLARE`	Declares global variable for error handling.
`ERR_ISLIBCREATED`	Has the library been created.
`ERR_LIBCREATEINIT`	Create and initialize the library.
`ERR_MAININIT`	Initialization of error handling in `main'.

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 ERR_Fatal (CStr *msg***)**

Exits with a message and produces a `core dump' (on UNIX).

Exit

void ERR_Exit (CStr *msg***)**

Exits with a message.

Signaling Failures

Fail

void ERR_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 ERR_FailStr (CStr *str***, ...)**

Generates a failure and displays `str' instead of loading a message from the resource file).

FailSilent

void ERR_FailSilent (void)

Generates a 'silent' failure (without message).

FailAssert

void ERR_FailAssert (CStr *cstr***, CStr** *fileName***, ErrLineVal** *line***)**

Use DBG_CHECK or ERR_CHECK instead.

FailError

void ERR_FailError (CStr *fileName*, **ErrLineVal** *line***)**

Use DBG_ERROR instead.

Generating Warnings

Warn

void ERR_Warn (CStr *modname*, **ErrIdVal** *msgId*, **...)**

Generates a warning and displays a warning message loaded from resource. Conversion specifications, if any, will apply.

WarnStr

void ERR_WarnStr (CStr *str*, **...)**

Generates a warning and displays `str' instead of loading a message from the resource file).

Querying the Error State

InError

BoolEnum ERR_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
`ERR_CHECK(t)`	Checks that <t> is true. If <t> is false, generates the error message: assertion <t> failed file ... line ... " This assertion is not controlled by the DBG_ON compilation flag. Usually, you will use DBG_CHECK rather than ERR_CHECK because you want assertions to disappear in production code. (See basepub.h for DBG_XXX macros).
`ERR_CHECKSTR(t, str)`	Same as ERR_CHECK except that it generates the message: assertion <str> failed file ... line This is to be used in the special cases when <t> is too long to fit on one line or if it contains a ("). You should use DBG_CHECKSTR instead.
`ERR_ASSERT(t)`	Is a synonym for ERR_CHECK ifdef DBG_NOCHECKSTR.

Error Reporting

TraceBack

void ERR_TraceBack (void)

Outputs error traceback starting from the top error frame for the current exception.

FrameTraceBack

void ERR_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).
`FAILERROR`	Failure generated by DBG_ERROR.
`FAILEXCEPTION`	Failure raised by an exception.

Exiting from the Application

ModExit

void ERR_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 ERR_SysExceptProc (ErrSigVal);

Data type for UNIX exception handler.

SetSysExceptHandler

void ERR_SetSysExceptHandler (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.

GetSysExceptHandler

ErrSysExceptProc ERR_GetSysExceptHandler (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 ERR_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 `SetSysExceptHandler' 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
`ERR_MswRegisterInterrupt(bool)`	Enable / Disable the native interrupt registration mechanism and set default action for ERR_LIBINIT, ERR_MAININIT.
`ERR_MswRegisterInterruptOnInit(bool)`	Set default action for ERR_LIBINIT, ERR_MAININIT.

MswRegisterInterrupt

void ERR_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 ERR_MswIsInterruptRegistered (void)

Return BOOL_TRUE if a ND native interrupt handler is registered.

MswRegisterInterruptOnInit

void ERR_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 ERR_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.

Chapter 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, FILE_ReadNBytes and FILE_WriteNBtyes 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 FILE_New), you can either check that the corresponding system file actually exists (with FILE_Find) and that it has the appropriate access rights (with FILE_GetAccess), before actually trying to open the file, or you can try to open the file directly (with FILE_Open or FILE_TryOpen).

Opening modes: FILE_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, FILE_ReadLine and FILE_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:

```
{
       FilePtr file = FILE_New("data");
       Byte   buf[200];

       FILE_Open(file, FILE_IOREAD, FILE_FMTBINARY);
       FILE_SeekBinaryTo(file, 300);
       FILE_ReadNBtyes(file, buf, 200);
       FILE_Close(file);
       FILE_Dispose(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”:

```
{
       FilePtr file = FILE_New("data");
       Byte         buf[200];

       FILE_Open(file, FILE_IOREAD, FILE_FMTBINARY);
       FILE_SeekBinaryTo(file, 300);
       FILE_ReadNBtyes(file, buf, 200);
       FILE_Close(file);
       FILE_Dispose(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(file)) {
              if (FILE_IsWritable(file)) {
                     FILE_Backup(file);
                     FILE_Open(file, FILE_IOREADWRITE,
FILE_FMTTEXT);
              } else {
                     FILE_Open(file, FILE_IOREAD,
FILE_FMTTEXT);
              }
       } else {
              FMgrCreateFileRec                info;
              info.Access = FMGR_ACCESSDEFAULTS;
              info.MacIds.Creator = FMGR_MACCREATOROIT;
              info.MacIds.Type = FMGR_MACTYPETEXT;
              FILE_CreateOpen(file, &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

FILE_New, FILE_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

FILE_CurLineNumber, FILE_QueryLinePos, FILE_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

FILE_QueryNatRef, FILE_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), FILE_CurSize, FILE_CurBinaryOffset, FILE_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

FILE_QueryTextPos, FILE_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 note 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

FILE_GetError, FILE_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`	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. 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. 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).
`FILE_FMTTEXT`	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. In this format, you can use the same Read/Write calls as in the FILE_FMTBINARY format, the difference is that native line delimitors ("\r" on Mac, "\r\n" on PC, record breaks on VMS and Mainframes) are automatically converted to "\n". You can not use a numeric offset to query and set the current position. You must use FILE_QueryTextPos and FILE_SetTextPos instead.
`FILE_FMTLINE`	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. For this mode, the Read/Write API is completely different and consists of two calls: FILE_ReadLine and FILE_WriteLine. The current position can only be at the beginning of a line. You can query and set the current position with FILE_QueryLinePos and FILE_SetLinePos. The current line number is managed automatically and can be queried or set by FILE_CurLineNumber and FILE_SetLinePos. Also, in this format the current line number is automatically maintained.

File I/O on Record-Oriented Binary files is not supported in this version of the library.

See also

FILE_Open, FILE_TryOpen, FILE_CreateOpen, FILE_GetOpenFormat, FILE_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 FILE_Open fails and FILE_TryOpen returns BOOL_FALSE).

See also

FILE_Open, FILE_TryOpen, FILE_GetOpenMode

Accessing File Attributes

GetSpecName

Get the file name used by FILE_Open.

CStr FILE_GetSpecName (FilePtr *fileobj***);**

FILE_GetSpecName returns the "specified name" of the fileobj. The "spec name" is the name used by FILE_Open to open the file. The spec name is initialized to the value passed to FILE_New. The spec name can be relative or absolute. It may be expressed in a foreign syntax, in which case

FILE_Open will convert it to the native syntax when it creates the “real name” for a file.

See also

FilePtr, FILE_SetSpecName, FILE_GetRealName, FILE_New

SetSpecName

Set the file name used by FILE_Open.

void FILE_SetSpecName (FilePtr *fileobj***, CStr** *specname***);**

FILE_SetSpecName changes the specname of the fileobj. The spec name is the name used by FILE_Open to open the file. The spec name is initialized to the value passed to FILE_New. The spec name can be relative or absolute. It may be expressed in a foreign syntax, in which case FILE_Open will convert it to the native syntax when it creates the “real name” for a file.

See also

FilePtr, FILE_SetSpecName, FILE_GetRealName, FILE_New

GetRealName

Get the real file name as set by FILE_Open.

CStr FILE_GetRealName (FileCPtr *fileobj***);**

FILE_GetRealName returns the “real name” of the fileobj. The real name is set by FILE_Find or FILE_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, FILE_GetSpecName, FILE_SetSpecName, FILE_New

GetSearchPath

Get the search path used by FILE_Open.

CStr FILE_GetSearchPath (FileCPtr *fileobj***);**

FILE_GetSearchPath returns the search path used by FILE_Open. The search path is a list of directory names where FILE_Open (or FILE_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, FILE_Open will use the global DefSearchPath defined by FILE_SetDefSearchPath instead.

See also

FILE_SetSearchPath, FILE_GetDefSearchPath, FILE_SetDefSearchPath

SetSearchPath

Set the search path used by FILE_Open.

void FILE_SetSearchPath (FilePtr *fileobj***, CStr** *path***);**

FILE_SetSearchPath sets the search path used by FILE_Open. The search path is a list of directory names where FILE_Open (or FILE_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, FILE_Open will use the global DefSearchPath instead. Use FILE_SetSearchPath (file, "") if FILE_Open should look only in the current directory.

See also

FILE_GetSearchPath, FILE_GetDefSearchPath, FILE_SetDefSearchPath

GetAutoBackup

Get the value of a file object's auto backup flag.

BoolEnum FILE_GetAutoBackup (FileCPtr *fileobj***);**

FILE_GetAutoBackup gets the value of a file object's auto backup flag. The auto backup flag is used by FILE_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

FILE_SetAutoBackup

SetAutoBackup

Set the value of a file object's auto backup flag.

void FILE_SetAutoBackup (FilePtr *fileob,***BoolEnum** *autobackup***);**

FILE_SetAutoBackup sets the value of a file object's auto backup flag. The auto backup flag is used by FILE_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

FILE_GetAutoBackup

GetFailIfNotFound

Get the value of a file structure's FailIfNotFound flag.

BoolEnum FILE_GetFailIfNotFound (FileCPtr *fileobj***);**

FILE_GetFailIfNotFound gets the value of a file object's FailIfNotFound flag. This flag is used by the FILE_Open and FILE_TryOpen routines. If the specified file cannot be found after looking up in the search path, and if this flag is set, FILE_TryOpen returns BOOL_FALSE and FILE_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

FILE_SetFailIfNotFound

SetFailIfNotFound

Set the value of a file structure's FailIfNotFound flag.

void File_SetFailIfNotFound (FilePtr *fileobj***, BoolEnum** *fail***);**

FILE_SetFailIfNotFound sets the value of a file object's FailIfNotFound flag. This flag is used by the FILE_Open and FILE_TryOpen routines. If the specified file cannot be found after looking up in the search path, and if this flag is set, FILE_TryOpen returns BOOL_FALSE and FILE_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

FILE_GetFailIfNotFound

SetFailOnEof

Set the value of a file structure's FailOnEOF flag.

void FILE_SetFailOnEof (FilePtr *fileobj***, BoolEnum** *fail***)**

FILE_SetFailOnEof sets the value of a file object's FailOnEOF flag. This flag is used by FILE_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

FILE_GetFailOnEof

GetFailOnEof

Get the value of a file structure's FailOnEOF flag.

BoolEnum FILE_GetFailOnEof (FileCPtr *file***);**

FILE_GetFailOnEof gets the value of a file object's FailOnEOF flag. This flag is used by FILE_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

FILE_SetFailOnEof

GetClientData

Gets the client data attached to a file object.

ClientPtr FILE_GetClientData (FileCPtr *fileobj***);**

FILE_GetClientData gets the client data attached to a file object.

SetClientData

Sets the client data attached to a file object.

void FILE_SetClientData (FilePtr *file***, ClientPtr** *clientdata***);**

FILE_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 FILE_Find (FilePtr *file***);**

FILE_Find searches for a file specified by its “spec name.” The spec name can be absolute or relative. If the spec name is relative, FILE_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 FILE_Find returns BOOL_TRUE. FILE_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 FILE_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 FILE_IsReadable to check this) or may not be a normal file (it could be a directory for instance. Use FILE_GetNodeType to check this).

IsReadable

Determines whether the given file has read access.

BoolEnum FILE_IsReadable (FileCPtr *file***);**

FILE_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 FILE_IsWritable (FileCPtr *file***);**

FILE_IsWritable determines whether the current user has write access to the file specified by the fileobj.

GetNodeType

Determines the type of a node.

FMgrNodeEnum FILE_GetNodeType (FileCPtr *fileobj***);**

FILE_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 FILE_Open (FilePtr *file***, FileIOEnum** *io***, FileFmtEnum** *format***);**

FILE_Open opens the file in the specified I/O mode and format. FILE_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 FILE_Open, you can check that the file exists with FILE_Find and then check the file access rights with FILE_IsReadable and FILE_IsWritable. You can also check the file type (normal file or directory) with FILE_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 FILE_CreateOpen, which lets you to specify creation parameters.

See also

FileIOEnum, FileFmtEnum, FILE_TryOpen, FILE_CreateOpen, FILE_Close, FILE_IsOpen…, FILE_Find, FILE_IsReadable, FILE_IsWritable, FILE_SetFailIfNotFound

CreateOpen

Creates a new file and opens it in the given format.

void FILE_CreateOpen (FilePtr *fileobj***, FMgrCreateFileCPtr** *create***, FileFmtEnum** *format***);**

FILE_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, FILE_TryCreateOpen, FILE_Open, FILE_TryOpen, FILE_Close

Close

Close a file.

void FILE_Close (FilePtr *fileobj***);**

FILE_Close closes the file referenced by the fileobj. This function does not release the memory used by the fileobj. Call FILE_Dispose to release that memory.

See also

FILE_TryClose, FILE_Open, FILE_Dispose

TryClose
TryCreateOpen
TryOpen

Non-asserting versions of the file close, create, and open functions.

BoolEnum FILE_TryClose (FilePtr *fileobj***);**

BoolEnum FILE_TryCreateOpen (FilePtr *file***, FMgrCreateFileCPtr** *createptr***, FileFmtEnum** *format***);**

BoolEnum FILE_TryOpen (FilePtr *fileobj***, FileIOEnum** *iomode***, FileFmtEnum** *format***);**

The FILE_TryXXX set of functions perform the same actions as their corresponding FILE_Close, FILE_CreateOpen, and FILE_Open counterparts, except that the FILE_TryXXX functions all return a BoolEnum value to indicate whether the function succeeded or failed. Refer to the FILE_XXX functions listed in the See Also section below for details about a corresponding FILE_TryXXX function.

The rationale for providing two sets of calls (FILE_TryXXX and FILE_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 FILE_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 FILE_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, FILE_Close, FILE_CreateOpen, FILE_Open, FILE_SetFailIfNotFound

IsOpen…

Determine if a file is open, and in which modes it is open.

BoolEnum FILE_IsOpen (FileCCPtr *fileobj***);**

BoolEnum FILE_IsOpenRead (FilecPtr *fileobj***);**

BoolEnum FILE_IsOpenWrite (FileCPtr *fileobj***);**

#define FILE_IsOpenBinary (file) (FILE_GetOpenFormat (file) == FILE_FMTBINARY)

#define FILE_IsOpenText (file) (FILE_GetOpenFormat (file) == FILE_FMTTEXT)

#define FILE_IsOpenLine (file) (FILE_GetOpenFormat (file) == FILE_FMTLINE)

The various FILE_IsOpen… functions and macros determine if a file is open, and in which modes it is open.

FILE_IsOpen returns BOOL_TRUE if the fileobj is open.

FILE_IsOpenRead returns BOOL_TRUE if the fileobj is open with read access (mode is READ, READWRITE, CLEARREADWRITE, or READAPPEND)

FILE_IsOpenWrite returns BOOL_TRUE if the fileobj is open with write access (mode is CLEARWRITE, OVERWRITE, APPEND, READWRITE, CLEARREADWRITE or READAPPEND)

FILE_IsOpenBinary returns BOOL_TRUE if the fileobj is open in binary mode.

FILE_IsOpenText returns BOOL_TRUE if the fileobj is open in text mode.

FILE_IsOpenLine returns BOOL_TRUE if the fileobj is open in line mode.

See also

FileIOEnum, FileFmtEnum, FILE_GetOpenMode, FILE_GetOpenFormat

GetOpenFormat
GetOpenMode

Determine the file format and I/O mode in which a file is open.

FileFmtEnum FILE_GetOpenFormat (FileCPtr *fileobj***);**

FileIOEnum FILE_GetOpenMode (FileCPtr *fileobj***);**

FILE_GetOpenFormat determines the file format in which a file has been opened.

FILE_GetOpenMode determines the I/O mode in which a file has been opened.

See also

FILE_IsOpen…, FileIOEnum, FileFmtEnum

Querying and Changing Position in a File

CurSize

Return the current size of a file.

FileOffsetVal FILE_CurSize (FileCPtr *fileobj***);**

FILE_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 FILE_GotoBeg (FilePtr *fileobj***);**

FILE_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 FILE_GotoEnd (FilePtr *fileobj***);**

FILE_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 FILE_IsAtEnd (FileCPtr *fileobj***);**

FILE_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 FILE_CurBinaryOffset (FileCPtr *fileobj***);**

FILE_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 FILE_SeekBinaryTo (FilePtr *fileobj***, FileOffsetVal** *position***);**

FILE_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 FILE_SeekBinaryBy (FilePtr *fileobj*, **FileOffsetVal** *position***);**

FILE_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 FILE_CurTextOffset (FileCPtr *fileobj***);**

FILE_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 FILE_QueryTextPos (FileCPtr *fileobj*, **FileTextPosPtr** *posptr***);**

FILE_QueryTextPos queries the current position structure for the text file referenced by fileobj. This posptr structure can later be passed to FILE_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 FILE_SetTextPos (FilePtr *fileobj*, **FileTextPosCPtr** *posptr***);**

FILE_SetTextPos sets the current position in fileobj to the posptr position saved by a previous call to FILE_QueryTextPos. This function is only valid for files opened in FILE_FMTBINARY mode.

CurLineNumber

Return the current line number in a file.

FileLineNbVal FILE_CurLineNumber (FileCPtr *fileobj***)**

FILE_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 FILE_QueryLinePos (FileCPtr *fileobj*, **FileLinePosPtr** *posptr***);**

FILE_QueryLinePos queries the current position structure for the file referenced by fileobj. This posptr structure can later be passed to FILE_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 FILE_SetLinePos (FilePtr *fileobj*, **FileLinePosCPtr** *posptr***);**

FILE_SetLinePos sets the current position in fileobj to the posptr position saved by a previous call to FILE_QueryLinePos. This function is only valid for files opened in FILE_FMTLINE mode.

See also

FILE_CurLineNumber, FileLinePosRec, FILE_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 FILE_ReadByte (FilePtr *file***);**

FILE_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

FILE_ReadChar, FILE_WriteByte, FILE_ReadNBytes, FILE_WriteNBytes

WriteByte

Writes a byte to a binary file.

void FILE_WriteByte (FilePtr *file*, **Int** *byte* **);**

FILE_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

FILE_WriteChar, FILE_ReadByte, FILE_ReadNBytes, FILE_WriteNBytes

ReadNBytes

Read a number of bytes from a binary file.

FileOffsetVal FILE_ReadNBytes (FilePtr *file*, **VoidPtr** *buffer*, **FileOffsetVal** *n***);**

FILE_ReadNBytes reads n bytes from the fileobj into buffer. The buffer must be allocated for at least n bytes. FILE_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, FILE_ReadNBytes and FILE_WriteNBtyes are limited to MAXINT32.

See also

FILE_ReadNChars, FILE_WriteByte, FILE_ReadByte, FILE_WriteNBytes

WriteNBytes

Writes a number of bytes to a binary file.

void FILE_WriteNBytes (FilePtr *file*, **VoidCPtr** *buffer*, **FileOffsetVal***n***);**

FILE_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, FILE_ReadNBytes and FILE_WriteNBtyes are limited to MAXINT32.

See also

FILE_WriteNChars, FILE_WriteByte, FILE_ReadByte, FILE_ReadNBytes

ReadChar

Return the next character in a text file.

Int FILE_ReadChar (FilePtr *fileobj***);**

FILE_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

FILE_ReadByte, FILE_WriteChar, FILE_ReadNChars, FILE_ReadStr, FILE_ReadTextLine, FILE_Printf

WriteChar

Writes a character to a text file.

void FILE_WriteChar (FilePtr *fileobj*, **Int** *char***);**

FILE_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

FILE_WriteByte, FILE_ReadChar, FILE_WriteNChars, FILE_WriteStr, FILE_WriteTextLine, FILE_Printf

ReadNChars

Read a number of characters from a text file.

FileOffsetVal FILE_ReadNChars (FilePtr *file*, **Str** *buffer*, **FileOffsetVal** *n***);**

FILE_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 FILE_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 increment by n.

See also

FILE_ReadNBytes, FILE_WriteNChars, FILE_ReadChar, FILE_ReadStr

WriteNChars

Writes a number of characters to a text file.

void FILE_WriteNChars (FilePtr *file*, **CStr** *buffer*, **FileOffsetVal** *n***);**

FILE_WriteNChars writes n characters from buffer into fileobj. This function is identical to FILE_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

FILE_WriteNBytes, FILE_WriteChar, FILE_ReadNChars

ReadStr

Read and return a string of characters from a text file.

CStr FILE_ReadStr (FilePtr *fileobj*, **FileOffsetVal** *n***);**

FILE_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 FILE_ReadStr is terminated by '\0'. This function is identical to FILE_ReadNChars except that it does not need any buffer and that the resulting string is terminated by '\0'. Also, FILE_ReadStr does not return how many characters have been read (use NDStr::Len on the string instead).

See also

FILE_ReadNChars, FILE_WriteNChars, FILE_ReadChar, FILE_ReadTextLine

WriteStr

Write a null terminated string to a text file.

void FILE_WriteStr (FilePtr *fileobj*, **CStr** *string***);**

FILE_WriteStr writes a NULL-terminated string into the fileobj FILE_FMTTEXT mode. This function is identical to FILE_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 FILE_ReadTextLine (FilePtr *file***);**

FILE_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, FILE_ReadTextLine will return the characters read up to this point as a normal line. At the next attempt, FILE_ReadTextLine will return NULL. If you want to check for this particular situation, you can call FILE_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 FILE_WriteTextLine (FilePtr *fileobj***, CStr** *string***);**

FILE_WriteTextLine writes a string to a text mode file and goes to the next line. This function is identical to FILE_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 FILE_ReadLine (FilePtr *fileobj***);**

FILE_ReadLine reads the next line from a file in FILE_FMTLINE format and returns it as a string. The '\n' is not included. FILE_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 FILE_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 FILE_WriteLine (FilePtr *fileobj***, CStr** *string***);**

FILE_WriteLine writes a string to a file and goes to the next line. The LineNumber is incremented by 1. This function is identical to FILE_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 FILE_Backup (FilePtr *fileobj***);**

FILE_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 FILE_Flush (FilePtr *fileobj***);**

FILE_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

FILE_Close, FILE_Write…

Truncate

Truncate a file at the current position.

void FILE_Truncate (FilePtr *fileobj***);**

FILE_Truncate truncates the fileobj at the current position. The file must be open at the time of the call.

See also

FILE_Close, FILE_Write…

Default Search Path

SetDefSearchPath

Set the default search path used by FILE_Open.

void FILE_SetDefSearchPath (CStr *path***);**

FILE_SetDefSearchPath sets the default search path used by FILE_Open. The default search path is the default value for the search path used by FILE_Find and FILE_Open to locate files specified as relative path names. The search path can be overridden globally with FILE_SetDefSearchPath or only for a specific file with FILE_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

FILE_GetSearchPath, FILE_SetSearchPath, FILE_GetDefSearchPath, FILE_GetDefSearchPathName, FILE_SetDefSearchPathName

GetDefSearchPath

Get the default search path used by FILE_Open.

CStr FILE_GetDefSearchPath (void);

FILE_GetDefSearchPath gets the default search path used by FILE_Open. The default search path is the default value for the search path used by FILE_Find and FILE_Open to locate files specified as relative path names. The search path can be overridden globally with FILE_SetDefSearchPath or only for a specific file with FILE_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

FILE_GetSearchPath, FILE_SetSearchPath, FILE_SetDefSearchPath, FILE_GetDefSearchPathName, FILE_SetDefSearchPathName

GetDefSearchPathName

Get the name of the environment variable containing the default search path.

CStr FILE_GetDefSearchPathName (void);

FILE_GetDefSearchPathName gets the name of the environment variable containing the default search path.

See also

FILE_GetSearchPath, FILE_SetSearchPath, FILE_GetDefSearchPath, FILE_SetDefSearchPath, FILE_SetDefSearchPathName

SetDefSearchPathName

Set the name of the environment variable containing the default search path.

void FILE_SetDefSearchPathName (CStr *name***);**

FILE_SetDefSearchPathName sets the name of the environment variable containing the default search path.

See also

FILE_GetSearchPath, FILE_SetSearchPath, FILE_GetDefSearchPath, FILE_SetDefSearchPath, FILE_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 FILE_QueryNatRef (FileCPtr *file***, FileNatRefPtr** *nat***);**

Returns in `nat' the native file handlers for the file.

SetNatRef

void FILE_SetNatRef (FilePtr *file***, 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 FILE_GetError (FileCPtr *file***);**

FILE_GetError returns the last error generated by a call to FILE_Find or FILE_TryOpen.

See also

FileErrEnum, FILE_Find, FILE_TryOpen

SetError

Sets an error for a file.

void FILE_SetError (FilePtr *file***, FileErrEnum** *fileerr***);**

Chapter

21 FMgr Class

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 FMGR_QueryNodeInfo function.

The various fields of this structure are summarized below:

Identifier	Description
`NodeType`	The type (file, directory, link, etc.) of the node.
`RefsNb`	The number of references (or hard links) to this node.
`Owner`	The user ID and group ID of the node’s owner.
`Times`	The node’s various creation and modification times.
`Access`	`The node’s allowed access rights.`
`MacIds`	The node’s Macintosh Type and Creator signatures.
`TotalSize`	The total size of the node.

See also

FMgrNodeEnum, FMgrRefsVal, FMgrOwnerRec, FMgrTimesRec, FMgrAccessSet, FMgrMacIdsRec, FMgrSizeVal, FMGR_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 FMGR_CreateDir function.

See also

FMgrNodeRec, FMgrAccessSet, FMGR_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 FMGR_CreateFile function and the NDFile::CreateOpen function (in the File class).

See also

FMgrNodeRec, FMgrAccessSet, FMgrMacIdsRec, FMGR_CreateFile, FILE_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, FMGR_GetMacCreator, FMGR_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…, FMGR_GetMacCreator, FMGR_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 note 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
FMGR_FILETYPEUNKNOWN	type unknown
FMGR_FILETYPESTATICLIB	static libraries (Unix:.a , DOS: .lib ..)
FMGR_FILETYPEDYNAMICLIB	dynamic libraries (.sl .so, .dll ..)
FMGR_FILETYPEAPPLICATION	executables (.exe on DOS and VMS)
FMGR_FILETYPEOBJECT	object file (.o, .obj)
FMGR_FILETYPEASSEMBLY	assembly source file (.asm, .s)
FMGR_FILETYPELINKOPTIONS	special options (.lnk, .opt)
FMGR_FILETYPESCRIPT	shell scripts (.sh .csh .bat .com)
FMGR_FILETYPECPLUSPLUS	C++ source file (.cc .cxx .cpp .C)
FMGR_FILETYPECFILE	C source file (.c)
FMGR_FILETYPEHFILE	C/C++ header file (.h)
FMGR_FILETYPELINT	Lint file (.ln)
FMGR_FILETYPEYACC	Yacc source file (.y)
FMGR_FILETYPELEX	Lex source file (.l)
FMGR_FILETYPESED	Sed source file (.sed)
FMGR_FILETYPEAWK	Awk source file (.awk)
FMGR_FILETYPEEMACSLISP	Emacs-Lisp source file (.el)
FMGR_FILETYPEEMACSLISPOBJ	Emacs-Lisp object file (.elc)
FMGR_FILETYPEARCHIVE	Archive file (.tar)
FMGR_FILETYPECOMPRESSED	Compressed file (.Z, .zip)
FMGR_FILETYPEHELP	Help document (.doc, .man, .doc)
FMGR_FILETYPERC	Rescomp source file (.rc)
FMGR_FILETYPERCO	Rescomp object file (.rco)
FMGR_FILETYPEDAT	Rescomp library file (.dat)
FMGR_FILETYPETEXT	Text file (.txt)
FMGR_FILETYPENXPTKB	Nexpert Text Knowledge-Base (.tkb)
FMGR_FILETYPENXPCKB	Nexpert Compiled Knowledge-Base (.ckb)
FMGR_FILETYPENXPEKB	Nexpert Text Knowledge-Base (.ekb)
FMGR_FILETYPENXPDB	Nexpert Text Database (.nxp)
FMGR_FILETYPESYLK	Sylk database file (.slk)
FMGR_FILETYPEDBASE	DBase database file (.dbf)
FMGR_FILETYPELOTUS	Lotus database file (.wks)
FMGR_FILETYPEORACLE	Oracle database file (.ora)
FMGR_FILETYPESYBASE	Sybase database file (.syb)
FMGR_FILETYPEINFORMIX	Informix database file (.inf)
FMGR_FILETYPEINGRES	Ingres database file (.ing)
FMGR_FILETYPEPICT	MacDraw file
FMGR_FILETYPEMOVIE	QuickTime Movie file
FMGR_FILETYPEMACPAINT	MacPaint file (.mcp)
FMGR_FILETYPEMSPAINT	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_FILETYPENDIMAGE`	Neuron Data image file (.ndi)
`FMGR_FILETYPESUNRASTER`	Sun Raster image file (.im8 .ras .rs)
`FMGR_FILETYPENXPIMAGE`	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 FMGR_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, FMGR_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_ACCESSSHIFTOTHERS	

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.
`FGMR_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, FMGR_GetMacCreator, FMGR_GetMacType

Querying and Changing File/Directory Attributes

Exists

Determines whether the specified node exists.

BoolEnum FMGR_Exists (CStr *name***)**

FMGR_Exists returns TRUE if the given node name (file, directory or volume) exists. It returns FALSE otherwise.

See also

FMGR_GetNodeType, FMGR_Is…

Is...

Functions for checking the access permissions of a node.

BoolEnum FMGR_IsExecutable (CStr *name***);**

BoolEnum FMGR_IsReadable (CStr *name***);**

BoolEnum FMGR_IsWritable (CStr *name***);**

These functions test whether a given node can be read from, written to, or executed. For directories, FMGR_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.

BoolEnum FMGR_IsDevConcealed (CStr *name***);**

The OpenVMS file system has the concept of a concealed device. FMGR_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 FMGR_IsDevConcealed returns BOOL_FALSE.

See also

FMGR_CheckDir, FMGR_CheckFile

QueryNodeInfo

Queries all the information for a node.

BoolEnum FMGR_QueryNodeInfo (CStr *name***, FMgrNodePtr** *fmgrnode***);**

FMGR_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

FMGR_GetNodeType, FMgrNodePtr

GetNodeType

Determines the type of the specified node.

FMgrNodeEnum FMGR_GetNodeType (CStr *name***);**

FMGR_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

FMGR_Is…, FMgrNodeEnum

GetMac...

Returns the Macintosh signatures of a file.

FMgrMacIdVal FMGR_GetMacCreator (CStr *name***);**

FMgrMacIdVal FMGR_GetMacType (CStr *name***);**

FMGR_GetMacCreator returns the Macintosh creator signature of the specified file. It returns type FMGR_MACCREATORNONE on non-Macintosh platforms. FMGR_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.

BoolEnum FMGR_IsDir (CStr *name***);**

BoolEnum FMGR_IsFile (CStr *name***);**

BoolEnum FMGR_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 FMGR_CheckFile or FMGR_CheckDir. These calls will not perform I/O operations, resulting in a significant performance increase over calls to FMGR_IsFile or FMGR_IsDir.

See also

FMGR_GetNodeType, FMgrNodeEnum, FMGR_CheckDir, FMGR_CheckFile

Check...

Checks the type of a node without performing I/O operations on VMS.

BoolEnum FMGR_CheckDir (CStr *name***);**

BoolEnum FMGR_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 FMGR_PerfDirFiles to trigger a callback procedure for a particular file or directory), you can call FMGR_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 FMGR_IsXXX counterparts.

These functions are intended for use on VMS systems. On all platforms except VMS these functions are equivalent to FMGR_IsXXX.

See also

FMGR_IsDevConcealed, FMGR_PerfDirFiles, FMGR_IsDir, FMGR_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_FILETYPEDYNAMICLIB`	dynamic libraries (.sl .so, .dll ..)
`FMGR_FILETYPEAPPLICATION`	executables (.exe on DOS and VMS)
`FMGR_FILETYPEOBJECT`	object file (.o, .obj)
`FMGR_FILETYPEASSEMBLY`	assembly source file (.asm, .s)
`FMGR_FILETYPELINKOPTIONS`	special options (.lnk, .opt)
`FMGR_FILETYPESCRIPT`	shell scripts (.sh .csh .bat .com)
`FMGR_FILETYPECPLUSPLUS`	C++ source file (.cc .cxx .cpp .C)
`FMGR_FILETYPECFILE`	C source file (.c)
`FMGR_FILETYPEHFILE`	C/C++ header file (.h)
`FMGR_FILETYPELINT`	Lint file (.ln)

`FMGR_FILETYPEYACC`	Yacc source file (.y)
`FMGR_FILETYPELEX`	Lex source file (.l)
`FMGR_FILETYPESED`	Sed source file (.sed)
`FMGR_FILETYPEAWK`	Awk source file (.awk)
`FMGR_FILETYPEEMACSLISP`	Emacs-Lisp source file (.el)
`FMGR_FILETYPEEMACSLISPOBJ`	Emacs-Lisp object file (.elc)
`FMGR_FILETYPEARCHIVE`	Archive file (.tar)
`FMGR_FILETYPECOMPRESSED`	Compressed file (.Z, .zip)
`FMGR_FILETYPEHELP`	Help document (.doc, .man, .doc)
`FMGR_FILETYPERC`	Rescomp source file (.rc)
`FMGR_FILETYPERCO`	Rescomp object file (.rco)
`FMGR_FILETYPEDAT`	Rescomp library file (.dat)
`FMGR_FILETYPETEXT`	Text file (.txt)
`FMGR_FILETYPENXPTKB`	Nexpert Text Knowledge-Base (.tkb)
`FMGR_FILETYPENXPCKB`	Nexpert Compiled Knowledge-Base (.ckb)
`FMGR_FILETYPENXPEKB`	Nexpert Text Knowledge-Base (.ekb)
`FMGR_FILETYPENXPDB`	Nexpert Text Database (.nxp)
`FMGR_FILETYPESYLK`	Sylk database file (.slk)
`FMGR_FILETYPEDBASE`	DBase database file (.dbf)
`FMGR_FILETYPELOTUS`	Lotus database file (.wks)
`FMGR_FILETYPEORACLE`	Oracle database file (.ora)
`FMGR_FILETYPESYBASE`	Sybase database file (.syb)
`FMGR_FILETYPEINFORMIX`	Informix database file (.inf)
`FMGR_FILETYPEINGRES`	Ingres database file (.ing)
`FMGR_FILETYPEPICT`	MacDraw file
`FMGR_FILETYPEMOVIE`	QuickTime Movie file
`FMGR_FILETYPEMACPAINT`	MacPaint file (.mcp)
`FMGR_FILETYPEMSPAINT`	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_FILETYPENDIMAGE`	Neuron Data image file (.ndi)
`FMGR_FILETYPESUNRASTER`	Sun Raster image file (.im8 .ras .rs)
`FMGR_FILETYPENXPIMAGE`	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.

void FMGR_AddFileType (FMgrFileTypeCPtr *fmgrfiletype***)**

RemoveFileType

Removes a file type.

void FMGR_RemoveFileType (FMgrFileTypeCPtr *fmgrfiletype***);**

GetNumFileTypes

Returns the number of registered file types.

ArrayIVal FMGR_GetNumFileTypes (void);

GetNthFileType

Returns the nth register file type description.

FMgrFileTypeCPtr FMGR_GetNthFileType (ArrayIVal *n***);**

FindFileTypeId

Returns the FileTypeId for the given file.

FMgrFileTypeEnum FMGR_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.

FMgrFileTypeCPtr FMGR_FindFileTypeInfo (CStr *name***);**

Same as 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.

void FMGR_CreateDir (CStr *name***, FMgrCreateDirCPtr** *createInfo***);**

FMGR_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…, FMGR_TryCreateDir, FMGR_CreateFile

CreateFile

Create a file with the specified permission rights.

void FMGR_CreateFile (CStr *name***, FMgrCreateFileCPtr** *createInfo***);**

FMGR_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 als

FMgrCreateFileRec, FMGR_ACCESS…, FMGR_TryCreateFile, FMGR_CreateDir

Copying

CopyFile

Copy a file.

void FMGR_CopyFile (CStr *orname***, CStr** *destname***);**

FMGR_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. FMGR_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

FNAME_MakeBackupName (FName class), FMGR_TryCopyFile, FMGR_CopyDir, FMGR_CopyNode, FMGR_MoveFile

CopyDir

Copy a directory and all of its content.

void FMGR_CopyDir (CStr *orname***, CStr** *destname***);**

FMGR_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

FMGR_TryCopyDir, FMGR_CopyFile, FMGR_CopyNode, FMGR_MoveDir

CopyNode

Copy a node.

void FMGR_CopyNode (CStr *orname***, CStr** *destname***);**

FMGR_CopyNode makes a copy of a node (file, link or directory). FMGR_CopyNode is more general than FMGR_CopyFile and FMGR_CopyDir, but may be slower.

See also

FMGR_TryCopyNode, FMGR_CopyFile, FMGR_CopyDir, FMGR_MoveNode

Moving

MoveFile

Rename and/or move a file.

void FMGR_MoveFile (CStr *orname***, CStr** *orname***);**

FMGR_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. FMGR_MoveFile can also be used for links.

FMGR_MoveFile will fail if a file of the same name already exists in the location passed.

See also

FMGR_TryMoveNode, FMGR_MoveDir, FMGR_MoveNode, FMGR_CopyFile

MoveDir

Rename and / or move a directory.

void FMGR_MoveDir (CStr *orname*, **CStr** *destname***);**

FMGR_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.

FMGR_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 FMGR_CopyDir to copy the directory to the new location and then FMGR_DeleteDir to delete the original directory.

Although DOS and Windows do not support moving directories, FMGR_MoveDir is implemented using the alternative method just described.

See also

FMGR_TryMoveDir, FMGR_MoveFile, FMGR_MoveNode, FMGR_CopyDir

MoveNode

Rename and / or move a node.

void FMGR_MoveNode (CStr *orname*, **CStr** *destname***);**

FMGR_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. FMGR_MoveNode can be used for either files or directories. It is more general than FMGR_MoveDir and FMGR_MoveFile, but may be slower.

See also

FMGR_TryMoveNode, FMGR_MoveDir, FMGR_MoveFile, FMGR_CopyNode

Deleting

DeleteFile

void FMGR_DeleteFile (CStr *name***);**

Delete a file.

FMGR_DeleteFile deletes the specified file. The file must exist.

See also

FMGR_TryDeleteFile, FMGR_DeleteDir, FMGR_DeleteNode, FMGR_DeleteDirContent

DeleteDir

Delete a directory and all its content.

void FMGR_DeleteDir (CStr *name***);**

FMGR_DeleteDir deletes the specified directory and all its content.

See also

FMGR_TryDeleteDir, FMGR_DeleteFile, FMGR_DeleteNode, FMGR_DeleteDirContent, FMGR_PurgeDir

DeleteNode

Delete a node.

void FMGR_DeleteNode (CStr *name***);**

FMGR_DeleteNode deletes the specified node. The node can be a file or a directory. This call is more general than FMGR_DeleteFile and FMGR_DeleteDir, but it is slower because it needs to test the type of the node first.

See also

FMGR_TryDeleteNode, FMGR_DeleteFile, FMGR_DeleteDir

DeleteDirContent

Delete the contents of a directory.

void FMGR_DeleteDirContent (CStr *name***);**

FMGR_DeleteDirContent deletes the content of the specified name directory (including sub-directories), but leaves the name directory itself intact.

See also

FMGR_TryDeleteDirContent, FMGR_DeleteDir, FMGR_PurgeDir

Try…

Non-asserting versions of the copy, create, delete, and move functions.

BoolEnum FMGR_TryCopyDir (CStr *orname*, **CStr** *destname***);**

BoolEnum FMGR_TryCopyFile (CStr *orname*, **CStr** *destname***);**

BoolEnum FMGR_TryCopyNode (CStr *orname*, **CStr** *destname***);**

BoolEnum FMGR_TryCreateDir (CStr *name*, **FMgrCreateDirPtr** *createInfo***);**

BoolEnum FMGR_TryCreateFile (CStr *name*, **FMgrCreateFilePtr** *createInfo***);**

BoolEnum FMGR_TryDeleteDir (CStr *name***);**

BoolEnum FMGR_TryDeleteFile (CStr *name***);**

BoolEnum FMGR_TryDeleteNode (CStr *name***);**

BoolEnum FMGR_TryDeleteDirContent (CStr *name***);**

BoolEnum FMGR_TryMoveDir (CStr *orname*, **CStr** *destname***);**

BoolEnum FMGR_TryMoveDir (CStr *orname*, **CStr** *destname***);**

BoolEnum FMGR_TryMoveFile (CStr *orname*, **CStr** *destname***);**

BoolEnum FMGR_TryMoveNode (CStr *orname*, **CStr** *destname***);**

The FMGR_TryXXX set of functions perform the same actions as their corresponding FMGR_XXX counterparts, except that the FMGR_TryXXX functions all return a BoolEnum value to indicate whether the function succeeded or failed. Refer to the FMGR_XXX functions listed in the See Also section below for details about a corresponding FMGR_XXX function.

The rationale for providing two sets of calls (FMGR_TryXXX and FMGR_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 FMGR_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 FMGR_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, FMGR_CopyDir, FMGR_CopyFile, FMGR_CopyNode, FMGR_CreateDir, FMGR_CreateFile, FMGR_DeleteDir, FMGR_DeleteFile, FMGR_DeleteNode, FMGR_DeleteDirContent, FMGR_MoveDir, FMGR_MoveFile, FMGR_MoveNode

PurgeDir

Purge specified files from a directory.

void FMGR_PurgeDir (CStr *dir*, **CStr** *pattern***);**

FMGR_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, FMGR_PurgeDir therefore purges only those files with a "." in the name. Use the functions FMGR_DirWildCard or FMGR_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

FMGR_DeleteDir, FMGR_DeleteDirContent

Performing an Action

PerfDirFiles

Call a user function for each matching file in a directory.

PerfEnum FMGR_PerfDirFiles (Str *dir*, **Str** *pattern*, **FMgrPerfFileProc** *func*, **ClientPtr** *data***);**

typedef PerfEnum (*FMgrPerfFileProc) (Str, Str, ClientPtr);

FMGR_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, FMGR_PerfDirFiles therefore limits its serarch to those files with a "." in the name. Use the functions FMGR_DirWildCard or FMGR_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 FMGR_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

FMGR_DirWildCard, FMGR_AllFilesWildCard, FMGR_PerfVolumes, Base class

DirWildCard

Return wildcard pattern that matches only directories.

CStr FMGR_DirWildCard (void);

FMGR_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, FMGR_AllFilesWildCard

AllFilesWildCard

Return wildcard pattern that matches all the files in a directory.

CStr FMGR_AllFilesWildCard (void);

FMGR_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, FMGR_DirWildCard

PerfVolumes

Call a user function for each volume in the system.

typedef PerfEnum (*FMgrPerfVolProc) (CStr, ClientPtr);

void FMGR_PerfVolumes (FMgrPerfVolProc *func*, **ClientPtr** *data***);**

FMGR_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 FMGR_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

FMGR_PerfDirFiles, Base class

Chapter
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 refering to the directory as a path.
`[dir1]dir2.dir`	Is used when refering 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 FNAME_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 avantageously). 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 _HOSTX might be absent as well as the _VOLY. 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    FNAME_UHOME:[dir1.dir2]file
DOS    $ (FNAME_UHOME)\dir1\dir2\file
MAC    $ (FNAME_UHOME):dir1:dir2:file

DOS    a:\dir1\dir2\file
UNIX   $ (FNAME_DVOLa)/dir1/dir2/file
VMS    FNAME_DVOLA:[dir1.dir2]file
MAC    $ (FNAME_DVOLa):dir1:dir2:file

DOS    a:dir1\dir2\file
UNIX   $ (FNAME_DVOLaDEF)/dir1/dir2/file
VMS    FNAME_DVOLADEF:[dir1.dir2]file
MAC    $ (FNAME_DVOLaDEF):dir1:dir2:file

DOS    \dir1\dir2\file
UNIX   /dir1/dir2/file
VMS    [dir1.dir2]file
MAC    $ (FNAME_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 FNAME_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 FNAME_FindSyntax or FNAME_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 FNAME_IsPortable. It should always contain at least FNAME_STXMASKOF (CurSyntax). By default, it is set to FNAME_STXMASKALL.
`CvtEvaluate`	If TRUE, FNAME_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, FNAME_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, FNAME_FindSyntax, FNAME_Convert, FNAME_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
FNAME_STXMASKDOS	The name could be a DOS, OS/2 or NT path name.
FNAME_STXMASKMAC	The name could be a Macintosh path name.
FNAME_STXMASKUNIX	The name could be a Unix path name.
FNAME_STXMASKVMS	The name could be a VMS path name.
FNAME_STXMASKW32	Name could be a Windows 95 or Windows NT path name
FNAME_STXMASKALL	The name could be in any syntax.

See also

FNameStxEnum, FNAME_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
FNAME_STATUSOK	Conversion has worked normally without any loss of information.
FNAME_STATUSCHARTRUNCATED	Result string has been truncated because it was over FNAME_MAXLEN.
FNAME_STATUSCOMPTRUNCATED	Some components have been dropped because there were more than 32 components.
FNAME_STATUSCHARSKIPPED	Some characters have been skipped because otherwise a component would be too long.
FNAME_STATUSCOMPSKIPPED	Some components have been skipped because they could not be translated.
FNAME_STATUSCHARALTERED	Some characters have been replaced by '_'. These characters would not have been valid.
FNAME_STATUSCOMPALTERED	Some components have been changed into components of different types. These components were not supported in the target syntax.

See also

FNAME_GetStatus, FNAME_SetStatus, FNAME_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
FNAME_STXBAD	Illegal or ambiguous syntax.
FNAME_STXDOS	The syntax in DOS, OS/2 and NT.
FNAME_STXMAC	The syntax in Macintosh.
FNAME_STXUNIX	The syntax in Unix.
FNAME_STXVMS	The syntax in VMS.
FNAME_STXW32	Syntax in Windows 95 or NT

See also

FNAME_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
FNAME_FAILUNIX	Invalid UNIX filename syntax.
FNAME_FAILDOS	Invalid DOS filename syntax.
FNAME_FAILMAC	Invalid Mac filename syntax.
FNAME_FAILVMS	Invalid VMS filename syntax.
FNAME_FAILVAR	Invalid variable substitution syntax.
FNAME_FAILAMBIGUOUS	Ambigous syntax.
FNAME_FAILDOSLEN	File name exceeds DOS length.
FNAME_FAILTOODEEP	Too many levels of directories.
FNAME_FAILVMSCLOSEBKT	Missing ‘]’ character in VMS filename.
FNAME_FAILNOTPATH	Directory name not in its path syntax.
FNAME_FAILSPLITPATH	Failed to split a path name.
FNAME_FAILFILETOPATH	Failed to convert a directory name into its path syntax.

See also

FNameStatusEnum, FNAME_GetStatus, FNAME_SetStatus, FNAME_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, FNAME_GetCompSet, FNAME_ReduceComps, FNAME_QueryComps

File Name Syntax

StxGetName

Returns the name of the specified syntax.

CStr FNAME_StxGetName (FNameStxEnum *syntax***);**

FNAME_StxGetName returns the name of the specified syntax.

GetSysSyntax

Determine the syntax of the native system.

FNameStxEnum FNAME_GetSysSyntax (void);

FNAME_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.

FNameStxEnum FNAME_GetCurSyntax (void);

FNAME_GetCurSyntax returns the current syntax. The current syntax is used by functions such as FNAME_Convert.

SetCurSyntax

Set a particular syntax as the current syntax.

void FNAME_SetCurSyntax (FNameStxEnum *syntax***);**

FNAME_SetCurSyntax sets the given syntax to be the current syntax. The current syntax is used by functions such as FNAME_Convert.

QueryCurParams

Query the current syntax conversion parameters.

void FNAME_QueryCurParams (FNameParamsPtr *params***);**

FNAME_QueryCurParams queries the currently set syntax conversion parameters and stores the results in params.

SetCurParams

Set the current syntax conversion parameters.

void FNAME_SetCurParams (FNameParamsCPtr *params***);**

FNAME_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.

void FNAME_ResetCurParams (void);

FNAME_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.

FNameStxEnum FNAME_FindSyntax (CStr *name***);**

FNAME_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, FNAME_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.

BoolEnum FNAME_IsValidIn (CStr *name***, FNameStxEnum** *syntax***);**

FNAME_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.

BoolEnum FNAME_IsValid (CStr *name***);**

FNAME_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

void FNAME_MakeValidIn (FNameBuf *name***, FNameStxEnum** *syntax***);**

FNAME_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

void FNAME_MakeValid (FNameBuf *name***);**

FNAME_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.

void FNAME_EvaluateIn (FNameBuf *name***, FNameStxEnum** *syntax***);**

FNAME_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, FNAME_Evaluate

Evaluate

Replace each variable expression by its value, using the current syntax.

void FNAME_Evaluate (FNameBuf *name***);**

FNAME_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.

void FNAME_Convert (CStr *source***, FNameBuf** *dest***);**

FNAME_Convert determines the source name syntax and converts it to the current syntax. The result is stored in dest.

This procedure calls FNAME_Evaluate and FNAME_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 FNAME_GetStatus.

See also

FNameBuf, FNAME_Evaluate, FNAME_MakeValid, FNAME_GetStatus, FNAME_SetCurSyntax, FNAME_ConvertFromTo, FNAME_ConvertInPlace

ConvertFromTo

Convert a name from one given syntax to another.

void FNAME_ConvertFromTo (CStr *source***, FNameBuf** *dest***, FNameStxEnum** *syntax1***, FNameStxEnum** *syntax2***);**

FNAME_ConvertFromTo converts the given source name from syntax1 to syntax2. The result is stored in dest.

This procedure calls FNAME_Evaluate and FNAME_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 FNAME_GetStatus.

See also

NDFNameStxEnum, FNameBuf, FNAME_Evaluate, FNAME_MakeValid, FNAME_GetStatus, FNAME_SetCurSyntax, FNAME_Convert, FNAME_ConvertInPlace

ConvertInPlace

Determine the syntax of a name and convert it to the current syntax.

void FNAME_ConvertInPlace (FNameBuf *name***);**

FNAME_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 FNAME_Evaluate and FNAME_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 FNAME_GetStatus.

See also

NDFNameBuf, FNAME_Evaluate, FNAME_MakeValid, FNAME_GetStatus, FNAME_SetCurSyntax, FNAME_ConvertFromTo, FNAME_Convert

IsConvertible

Determine if a name can be completely converted.

BoolEnum FNAME_IsConvertible (CStr *name***);**

FNAME_IsConvertible returns BOOL_TRUE if name can be converted without any loss of information (no truncation nor alteration).

See also

FNAME_IsPortable, FNAME_Convert

IsPortable

Determine if a name can be completely converted to all target syntaxes.

BoolEnum FNAME_IsPortable (CStr *name***);**

FNAME_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, FNAME_SetCurParams, FNAME_IsConvertible, FNAME_Convert

Conversion Status

GetStatus

Get the status of the most recent conversion.

FNameStatusEnum FNAME_GetStatus (void);

FNAME_GetStatus returns the status of the most recent file name conversion.

See also

FNameStatusEnum, FNAME_SetStatus, FNAME_StatusGetMsg

SetStatus

Set the status flag to the given value.

void FNAME_SetStatus (FNameStatusEnum *status***);**

FNAME_SetStatus sets the status flag to the given status value.

See also

NDFNameStatusEnum, FNAME_GetStatus, FNAME_StatusGetMsg

StatusGetMsg

Get the text description of the given status value.

CStr FNAME_StatusGetMsg (FNameStatusEnum *status***);**

FNAME_StatusGetMsg returns the text description of the given status value.

See also

NDFNameStatusEnum, FNAME_GetStatus, FNAME_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.

FNameCompSet FNAME_GetCompSet (CStr *name***);**

FNAME_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..., FNAME_ReduceComps, FNAME_QueryComps

QueryComps

Extract specified file name components and copy to a string.

void FNAME_QueryComps (CStr *name***, FNameCompSet** *components***, Str** *output***);**

FNAME_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…, FNAME_GetCompSet, FNAME_ReduceComps

ReduceComps

Reduce a file name to a specified set of components.

void FNAME_ReduceComps (Str *name***, FNameCompSet** *components***);**

FNAME_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…, FNAME_GetCompSet, FNAME_QueryComps

Directories Specified as Paths or as Files

IsDirAsFile

Determine if a directory is specified as a directory name or as a file name.

BoolEnum FNAME_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. FNAME_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.

void FNAME_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. FNAME_CvtDirPathToFile converts a directory string from path syntax to file syntax. It generates an error if the initial syntax in 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.

void FNAME_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. FNAME_CvtDirFileToPath converts a directory string from file syntax to path syntax. It generates an error if the initial syntax in 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.

void FNAME_SplitFile (CStr *name*, **FNameBuf** *path*, **FNameBuf** *file***);**

FNAME_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.

BoolEnum FNAME_SplitPath (CStr *name*, **FNameBuf** *path*, **FNameBuf** *subdirectory***);**

FNAME_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.

void FNAME_MergeFile (CStr *path*, **CStr** *file*, **FNameBuf** *name***);**

FNAME_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, FNAME_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.

void FNAME_MergePath (CStr *path***, CStr** *subdirectory***, FNameBuf** *name***);**

FNAME_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, FNAME_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.

CStr FNAME_TopDirStr (void);

FNAME_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.

void FNAME_QueryTopDir (FNameBuf *fnamebuf***);**

IsTopDir

Returns whether a directory path name is at the top level.

BoolEnum FNAME_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.

void FNAME_QueryCurDir (FNameBuf *directory***);**

FNAME_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

FNAME_QueryCurVolume, FNAME_VolumeQueryCurDir, FNAME_VolumeSetCurDir, FNAME_SetCurDir, FNAME_QueryHomeDir

SetCurDir

Set the current directory.

void FNAME_SetCurDir (CStr *directory***);**

FNAME_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

FNAME_QueryCurVolume, FNAME_VolumeQueryCurDir, FNAME_VolumeSetCurDir, FNAME_QueryCurDir, FNAME_QueryHomeDir

QueryCurVolume

Query the current volume string.

void FNAME_QueryCurVolume (FNameBuf *volume***);**

FNAME_QueryCurVolume queries the current volume string. It copies the string into the volume buffer argument.

VolumeQueryCurDir

Query the current directory of a given volume.

void FNAME_VolumeQueryCurDir (CStr *volume***, FNameBuf** *directory***);**

FNAME_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.

void FNAME_VolumeSetCurDir (CStr *volume***, CStr** *directory***);**

FNAME_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 passes as the second argument.

CurDirStr

Returns the string representation of the current directory.

CStr FNAME_CurDirStr (void);

FNAME_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.

void FNAME_QueryParentDir (FNameBuf *directory***);**

FNAME_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

FNAME_QueryCurVolume, FNAME_VolumeQueryCurDir, FNAME_VolumeSetCurDir, FNAME_SetCurDir, FNAME_QueryHomeDir

ParentDirStr

Returns the string representation of the parent directory.

CStr FNAME_ParentDirStr (void);

FNAME_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.

void FNAME_DirQueryParent (CStr *dir***, FNameBuf** *parent***);**

FNAME_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.

CStr FNAME_HomeDirStr (void);

FNAME_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.

void FNAME_QueryHomeDir (FNameBuf *home*);

FNAME_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

FNAME_QueryCurVolume, FNAME_VolumeQueryCurDir, FNAME_VolumeSetCurDir, FNAME_QueryCurDir, FNAME_SetCurDir, FNAME_QueryTopDir, FNAME_QueryParentDir

QueryTopDir

Query the top directory of the current volume.

void FNAME_QueryTopDir (FNameBuf *top*);

FNAME_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

FNAME_QueryCurVolume, FNAME_VolumeQueryCurDir, FNAME_VolumeSetCurDir, FNAME_QueryCurDir, FNAME_SetCurDir, FNAME_QueryHomeDir, FNAME_QueryParentDir

QueryParentDir

Query the parent directory of the current directory.

void FNAME_QueryParentDir (FNameBuf *parent*);

FNAME_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

FNAME_QueryCurVolume, FNAME_VolumeQueryCurDir, FNAME_VolumeSetCurDir, FNAME_QueryCurDir, FNAME_SetCurDir, FNAME_QueryHomeDir, FNAME_QueryTopDir

Absolute / Relative Parts

IsTopDir

Determine whether a directory path name is at the top level.

BoolEnum FNAME_IsTopDir (CStr *name***);**

FNAME_IsTopDir returns BOOL_TRUE if the given directory path name is at the top level, and it returns BOOL_FALSE otherwise name.

IsAbsolute

Determine if a file name is specified as absolute or as relative.

BoolEnum FNAME_IsAbsolute (CStr *name***);**

FNAME_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.

void FNAME_CvtToAbsolute (CStr *name***, FNameBuf** *absolute***);**

FNAME_CvtToAbsolute converts the given file name to an absolute file name.

Comparing File Names

Cmp

Compare two file names.

CmpEnum FNAME_Cmp (CStr *name1***, CStr** *name2***);**

FNAME_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.

BoolEnum FNAME_Equal (CStr *name1***, CStr** *name2***);**

FNAME_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.

CStr FNAME_GetTmpPath (void);

FNAME_GetTmpPath returns the path where temporary file names are created.

SetTmpPath

Change the path where temporary file names will be created.

void FNAME_SetTmpPath (CStr *directory***);**

FNAME_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.

CStr FNAME_SysTmpPath (void);

FNAME_SysTmpPath returns the pnative 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 "SYS$SCRATCH:" on VMS.

MakeTmpFileName

Generate a temporary file name.

void FNAME_MakeTmpFileName (CStr *prefix***, FNameBuf** *buffer***);**

FNAME_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 FNAME_GetTmpPath, on the prefix, and on a number generated by the system library.

MakeBackupName

Generate a name for a backup file.

void FNAME_MakeBackupName (CStr *filename***, FNameBuf** *backupname***);**

FNAME_MakeBackupName generates a backupname backup file name for the given input filename.

Chapter

23 Hash Class

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 threshhold. 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

`CloneKeyProc`	Entry key cloning procedure. If set to NULL, no cloning is carried out
`DisposeKeyProc`	Key disposing procedure. If set to NULL, no disposing is carried out. Used in conjunction with HashCloneKeyProc
`NumBins`	Initial number of bins. The number of bins can perfectly grow while the hash table is being used.
`MaxEntriesPerBin`	Maximum average number of entries per bin allowed before requesting an increase in the number of bins.
`GrowPercBins`	Percentage of NumBins that are allocated when MaxEntriesPerBin is reached.

Constructors and Destructor

Alloc

HashPtr HASH_Alloc(void);

Returns a pointer to an allocated but not yet constructed hash table. The hash table should be constructed before being used.

Constructors

void HASH_Construct(HashPtr *hash***);**

Default hash table construction

void HASH_ConstructInfo(HashPtr *hash*, **HashInfoCPtr** *info***);**

Constructs the hash table with the information obtained from ’info’.

Destructor

void HASH_Destruct(HashPtr *hash***);**

Default hash table destruction.

Dealloc

void HASH_Dealloc(HashPtr *hash***);**

Deallocates the hash table.

Convenience Functions

New

HashPtr HASH_New(HashInfoCPtr *hashInfo***);**

Creates and returns a pointer to a hash table parametrized by `hashInfo'.

NewForInt

HashPtr HASH_NewForInt(void);

Creates and returns a pointer to a hash table parametrized for integers.

NewForPt

HashPtr HASH_NewForPtr(void);

Creates and returns a pointer to a hash table parametrized for pointers.

NewForStr
NewForIStr

HashPtr HASH_NewForStr(void);

HashPtr HASH_NewForIStr(void);

Creates and returns a pointer to a hash table parametrized for strings.

Dispose

void HASH_Dispose(HashPtr *hash***);**

Disposes the hash table. It cannot be used again.

Resetting a Hash Table

Reset

void HASH_Reset(HashPtr *hash***);**

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 HASH_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 HASH_HashProc(HashKeyVal *key***, 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 HASH_DataCloneProc(HashDataVal);

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 HASH_DataDisposeProc(HashDataVal *data***);**

Procedure used to dispose data stored in the hash table for an entry (created through cloning).

KeyCloneProc

HashKeyVal HASH_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 HASH_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

void HASH_QueryDefInfo(HashInfoPtr *hashInfo***);**

Fills 'hashInfo' with the default settings for a hash table.

GetDefIntInfo

HashInfoCPtr HASH_GetDefIntInfo(void);

Returns the default settings for a hash table with integer keys.

GetDefPtrInfo

HashInfoCPtr HASH_GetDefPtrInfo(void);

Returns the default settings for a hash table with pointer keys.

GetDefStrInfo

HashInfoCPtr HASH_GetDefStrInfo(void);

HashInfoCPtr HASH_GetDefIStrInfo(void);

Returns the default settings for a hash table with string keys.

GetDefStrKeyClonedInfo

HashInfoCPtr HASH_GetDefStrKeyClonedInfo(void);

Returns the default settings for a hash table with cloned string keys.

QueryInfo

void HASH_QueryInfo(HashCPtr *hashc***, 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 HASH_Insert(HashPtr *hash*, **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 HASH_Extract(HashPtr *hash*, **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 HASH_Lookup(HashCPtr *hash*, **HashKeyVal** *key*, **HashDataValPtr** *valPtr***);**

Looks for an entry corresponding to ’key’. If it finds it, it updates valPtr with the corresponding value. Otherwize, it sets *valPtr to NULL. It returns BOOL_TRUE if an entry was found, BOOL_FALSE otherwise.

Note: If valPtr is NULL, it wonõt 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 HASH_Perf(HashCPtr *hash*, **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

HashLenVal HASH_DefHashInt(HashKeyVal *key*, **HashLenVal** *mod***);**

HashLenVal HASH_DefHashPtr(HashKeyVal *key*, **HashLenVal** *mod***);**

HashLenVal HASH_DefHashStr(HashKeyVal *key*, **HashLenVal** *mod***);**

HashLenVal HASH_DefHashIStr(HashKeyVal *key*, **HashLenVal** *mod***);**

Returns the bin index computed by the default hashing procedure provided by Open Interface.

Default Comparison

BoolEnum HASH_DefCompareInt(HashKeyVal *key1*, **HashKeyVal** *key2***);**

BoolEnum HASH_DefComparePtr(HashKeyVal *key1*, **HashKeyVal** *key2***);**

BoolEnum HASH_DefCompareStr(HashKeyVal *key1*, **HashKeyVal** *key2***);**

BoolEnum HASH_DefCompareIStr(HashKeyVal *key1*, **HashKeyVal** *key2***);**

Returns the result of the default Open Interface comparison of key1 and key2.

Default String Cloning

HashKeyVal HASH_DefStrKeyClone(HashKeyVal *key***);**

void HASH_DefStrKeyDispose(HashKeyVal *key***);**

Hash Table Entries

The following API is provided to users who want to improve performances is some special cases.

AddGetEntry

HashEntryPtr HASH_AddGetEntry(HashPtr *hash*, **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 HASH_InsertGetEntry(HashPtr *hash*, **HashKeyVal** *key*, **HashDataVal** *value***);**

Same as above but tests whether there was an entry there before or not.

GetEntry

HashEntryPtr HASH_GetEntry(HashPtr hash, 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

HashEntryVal HASH_GetEntry(HashEntryCPtr *entry***);**

Returns the key stored in the entry.

EntryGetValue

HashDataVal HASH_EntryGetValue(HashEntryCPtr *entry***);**

Returns the value stored in the entry.

EntrySetValue

void HASH_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 HASH_RemoveEntry(HashPtr *hash***, 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 HASH_QueryStats(HashCPtr *hash***, 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.

Chapter

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

Alloc

HeapPtr HEAP_Alloc(void);

Returns a pointer to an allocated but not yet constructed heap. The heap should be constructed before being used.

Constructor

void HEAP_Construct(HeapPtr *heap***);**

Default heap construction.

Destructor

void HEAP_Destruct(HeapPtr *heap***);**

Default heap destruction.

Dealloc

void HEAP_Dealloc(HeapPtr *heap);*

Deallocates the heap.

Convenience Functions

New

HeapPtr HEAP_New(void);

Creates and constructs a heap.

Dispose

void HEAP_Dispose(HeapPtr *heap***);**

Disposes a heap.

Heap Size

GetSize

HeapIndexVal HEAP_GetSize(HeapPtr *heap***);**

Returns the number of entries in the heap.

Heap Manipulation

Add

void HEAP_Add(HeapPtr *heap***, HeapKeyVal** *key***, ClientPtr** *client***);**

Insertion with no reorder: the heap structure is temporarily incorrect, a call to HEAP_Correct will be necessary before the heap can actually be used.

Correct

void HEAP_Correct(HeapPtr *heap***);**

Heap correction: the structure of the key is corrected. This call will be made in general after a series of calls to HEAP_Add.

Insert

void HEAP_Insert(HeapPtr *heap***, 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 HEAP_QueryFirst(HeapPtr *heap***, 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 HEAP_Perf(HeapPtr *heap***, HeapPerfProc** *proc***, ClientPtr** *clientData***);**

Performs `proc' on each entry of the heap. `proc' gets called with `clientData' as last argument.

Chapter

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 infinum (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

Alloc

IsetPtr ISET_Alloc(void);

Returns a pointer to an allocated but not yet constructed iset. The iset should be constructed before being used.

Constructor

void ISET_Construct(ISetPtr *iset***);**

Default iset construction.

Destructor

void ISET_Destruct(ISetPtr *iset***);**

Default iset destruction.

Dealloc

void ISET_Dealloc(ISetPtr *iset***);**

Deallocates the iset.

Special Intervals

UniversalSet

ISetPtr ISET_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 ISET_AddIntervals(ISetPtr *iset***, ISetLenVal** *n***, ISetIntervalPtr** *intervals***);**

Adds n internals to an ISet. It builds the union.

RemoveIntervals

void ISET_RemoveIntervals(ISetPtr *iset***, ISetLenVal** *numIntervals***,**
ISetIntervalPtr *intervals***);**

Removes ’n’intervals from an ISet. It builds the difference.

QueryIntervals

void ISET_QueryIntervals(ISetPtr *iset***, ISetLenVal** *numIntervals***, ISetIntervalPtr** *intervals***);**

Fills intervals with the intervals in the iset.

SetIntervals

void ISET_SetIntervals(ISetPtr *iset***, ISetLenVal** *numIntervals***, ISetIntervalPtr** *intervals***);**

Defines the intervals in the iset to be those specified by intervals.

GetNumIntervals

ISetLenVal ISET_GetNumIntervals(ISetPtr *iset***);**

Returns the number of intervals in the set.

IsAll

BoolEnum ISET_IsAll(ISetPtr *iset***);**

Returns BOOL_TRUE if the set contains all possible elements, i.e. is the interval [ISET_ELTVALINF, ISET_ELTVALSUP].

GetMinElt

ISetEltVal ISET_GetMinElt(ISetPtr *iset***);**

Returns the smallest element in the set.

GetMaxElt

ISetEltVal ISET_GetMaxElt(ISetPtr *iset***);**

Returns the biggest element in the set.

ContainsElt

BoolEnum ISET_ContainsElt(ISetPtr *iset***, ISetEltVal** *elt***);**

Returns BOOL_TRUE if the set contains ’elt’.

ContainsIntervals

BoolEnum ISET_ContainsIntervals(ISetPtr *iset***, ISetLenVal** *numIntervals***,**
ISetIntervalPtr *intervals***);**

Returns BOOL_TRUE if the set includes ’interval’.

QueryComplement

void ISET_QueryComplement(ISetPtr *iset***, 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

ISetMixPartSet ISET_MixGetPartSet(ISetPtr *A***, ISetPtr** *B***);**

void ISET_MixQueryParts(ISetPtr A, ISetPtr *B***, ISetMixPartSet** *part***, ISetPtr** *C***);**

Same as the equivalent calls in the Set package, but for ISet objects.

Chapter

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.

```
#if defined(C_ISANSI) || defined(VMS)
#      define C_CONST       const
#else
#      define C_CONST
#endif

#ifndef C_CONST
#define C_CONST
#endif
```

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.

void C_EXPORT (library *lib***, version** *vers***, return type** *ret***, routine name** *proc***, arguments** *args***);**

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

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
#endif
```

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

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
#endif
```

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
#endif
```

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

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 `char`s, for example you can execute the following:

```
printf(((int)(char)-1) == -1 ? "char is signed" : "char is
unsigned");
```

to check the sign of `char`s.

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

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.

```
#define MCH_CHIPMC68K;1/* Motorola 68000 family: 68000, 68010, 68020 */
#define MCH_CHIPI386;2/* Intel 80386 family */
#define MCH_CHIPSPARC;3/* SUN SPARC architecture */
#define MCH_CHIPVAX;4/* DEC VAX architecture */
#define MCH_CHIPMIPS;5/* MIPS RISC architecture */
#define MCH_CHIPHPPA;6/* HP precision architecture */
#define MCH_CHIPIBMRT;7/* IBM RT architecture */
#define MCH_CHIPRS6000;8/* IBM RS/6000 architecture */
#define MCH_CHIPMC88K9/* Motorola 88000 family */
#define MCH_CHIPALPHA10/* DEC Alpha AXP architecture */
#define MCH_CHIPIBM37011/* IBM 370 architecture */
```

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:

```
#ifdef 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 (source code *code*);

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 MSWOS_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 particuliar 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 particuliar 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 offically 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.

Chapter 27 Nfier Class

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 you 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

void NFIER_Construct(NfierPtr *nfier***);**

Default notifier construction.

void NFIER_ClientConstruct(NfierClientPtr *nfier***);**

Default notifier client construction.

void NFIER_ClientConstructProc(NfierClientPtr *infer***, NfierClientProc** *proc***);**

Constructs the notifier client with `proc' as call-back.

Alloc

NfierPtr NFIER_Alloc(void);

Returns a pointer to an allocated but not yet constructed notifier. The notifier should be constructed before being used.

Destructors

void NFIER_Destruct(NfierPtr *nfier***);**

Default notifier destruction.

void NFIER_ClientDestruct(NfierClientPtr *nfier***);**

Default notifier client destruction.

ClientDealloc

void NFIER_ClientDealloc(NfierClientPtr *nfier***);**

Deallocates the notifier client.

Broadcasting a Notification

Broadcast

void NFIER_Broadcast(NfierCPtr *nfier***, ClientPtr** *clientData***);**

Broadcasts a notification (`clientData' contains the notification information) to all the clients which have registered.

Notifier Client Creation and Destruction

ClientAlloc

NfierClientPtr NFIER_ClientAlloc(void);

Returns a pointer to an allocated but not yet constructed client notifier. The client notitifier should be constructed before being used.

ClientConstruct

void NFIER_ClientConstruct(NfierClientPtr *nfier***);**

Default notifier client construction

ClientConstructProc

void NFIER_ClientConstructProc(NfierClientPtr *nfier***, NfierClientProc** *proc***);**

Constructs the notifier client with `proc' as call-back.

ClientDestruct

void NFIER_ClientDestruct(NfierClientPtr *nfier***);**

Default notifier client destruction.

ClientDealloc

void NFIER_ClientDealloc(NfierClientPtr *nfier***);**

Deallocates the notifier client.

Associating Client Data with the Notifier Client Pointer

ClientSetClientData

void NFIER_ClientSetClientData(NfierClientPtr *nfierClient***, ClientPtr** *clientData***);**

Associates `clientData' with `nfierClient'.

ClientGetClientData

ClientPtr NFIER_ClientGetClientData(NfierClientCPtr *nfierClient***);**

Retrieves the client data previously associated with `nfierClient'.

Notifier Client Registration and Unregistration

RegisterNfierClient

void NFIER_RegisterNfierClient(NfierPtr *nfier***, 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.

Convienience: Unregistration, destruction and deallocation

UnregisterNfierClient

void NFIER_UnregisterNfierClient(NfierPtr *nfier***, NfierClientPtr** *client***);**

Removes a notifier client from the list of clients of a notifier.

Convienience: Allocation, construction and registration

ClientNewRegister

NfierClientPtr NFIER_ClientNewRegister(NfierPtr *nfier***, NfierClientProc proc);**

Allocates, constructs with `proc', registers with the notifier a notifier client that is returned.

Convienience: Unregistration, destruction and deallocation

ClientUnregisterDispose

void NFIER_ClientUnregisterDispose(NfierPtr *nfier***, NfierClientPtr** *client***);**

Unregisters, destructs and deallocates a notifier client from a notifier.

Chapter

28 Pack Class

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 encode 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 purposedly 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

PackPtr PACK_Alloc(void);

Returns a pointer to an allocated but not yet constructed pack object. The object should be constructed before being used.

void PACK_Construct(PackPtr *pack***);**

Default pack object construction.

Destructor

void PACK_Destruct(PackPtr *pack***);**

Default pack object destruction.

void PACK_Dealloc(PackPtr *pack***);**

Deallocates the pack object.

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 accomodate 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
`PACK_WORSTCASERLE(n)( n + n / 2)`	Happens with an alternance of 128 (encoded as [128, 0]) and of a different byte.
`PACK_WORSTCASEPKB(n)( n + 1 + (n - 1) / 128)`	Happens when there is no repetition of characters. Each strip is encoded as [N-1, byte1, .., byteN], with N <= 128.
`PACK_WORSTCASECCITT(n)( n * 9 / 2)`	Happens when 0s and 1s alternate ('01' encoded as '000111010').
`PACK_WORSTCASELZW(n)( n * 3 / 2 + 4)`	Happens with true random data.
`PACK_WORSTCASE PACK_WORSTCASECCITT`	Worst case of all worst cases: worst case for the CCITT encoding.

RLE (Run Length Encoding)

RleEncode

void PACK_RleEncode(PackPtr *pack***);**

Encodes with RLE algorithm.

RleDecode

void PACK_RleDecode(PackPtr *pack***);**

Decodes with RLE algorithm.

PackBits

PkbEncode

void PACK_PkbEncode(PackPtr *pack***);**

Encodes with PackBits.

PkbDecode

void PACK_PkbDecode(PackPtr *pack***);**

Decodes with PackBits.

The decoding should use the same depth and endianity 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 PACK_LzwEncode(PackPtr *pack*, **PackDepthVal** *depth*, **PackLzwFlags** *flags*);

Encodes with LZW.

LzwDecode

void PACK_LzwDecode(PackPtr *pack*, **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 PACK_CcittEncode(PackPtr *pack*, **PackSizeVal** *width*, **PackCcittFlags** *ccittFlags*);

void PACK_CcittDecode(PackPtr *pack*, **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 PACK_Encode(PackPtr *pack*, **PackMethodEnum** *method***);**

Encodes with the method specified by `method'.

Decode

void PACK_Decode(PackPtr *pack*, **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.

Chapter

29 Plfd Class

The PFld 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

PFldTypeEnum

Enumerated type describing the possible types for persistent fields

Identifier	Description
`PFLD_TYPEBAD,`	invalid type.
`PFLD_TYPEINT16,`	16 bit signed integer.
`PFLD_TYPEUINT16,`	16 bit unsigned integer.
`PFLD_TYPEINT32,`	32 bit signed integer.
`PFLD_TYPEUINT32,`	32 bit unsigned integer.
`PFLD_TYPESTR,`	string stored as a Str.
`PFLD_TYPESTRARRAY,`	array of strings (not implemented yet).
`PFLD_TYPEVSTR,`	string stored as a VStrPtr.
`PFLD_TYPEVSTRARRAY,`	array of VStrPtrs.
`PFLD_TYPERES,`	pointer to resource. NULL is illegal.
`PFLD_TYPERES0,`	pointer to resource, NULL is legal.
`PFLD_TYPERESARRAY,`	array of ResPtr.
`PFLD_TYPEDATA32,`	arbitrary length binary data (undoc.).
`PFLD_TYPEDATA8,`	internal: do not us.
`PFLD_TYPEDATA16,`	internal: do not use.

Field Categories

PFldCatEnum

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

NDPFld

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 rclaspub.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 PFldRec 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).

Chapter

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

void POINT16_Construct(Point16Ptr *p***);**

void POINT32_Construct(Point32Ptr *p***);**

Default construction.

ConstructWithValues

void POINT16_ConstructWithValues(Point16Ptr *p***, Int16** *xVal***, Int16** *yVal***);**

void POINT32_ConstructWithValues(Point32Ptr *p***, Int32** *xVal***, Int32** *yVal***);**

Construction with values.

Destruct

void POINT16_Destruct(Point16Ptr *p***);**

void POINT32_Destruct(Point32Ptr *p***);**

Default destruction.

Sets and Queries

GetX
SetX

Int16 POINT16_GetX(Point16CPtr *p***);**

void POINT16_SetX(Point16Ptr *p***, Int16** *val***);**

Int32 POINT32_GetX(Point32CPtr *p***);**

void POINT32_SetX(Point32Ptr *p***, Int32** *val***);**

Gets / sets the X coordinates of point `p'.

GetY
SetY

Int16 POINT16_GetY(Point16CPtr *p***);**

void POINT16_SetY(Point16Ptr *p***, Int16** *val***);**

Int32 POINT32_GetY(Point32CPtr *p***);**

void POINT32_SetY(Point32Ptr *p***, Int32** *val***);**

Gets / sets the Y coordinates of point `p'.

SetXY

void POINT16_SetXY(Point16Ptr *p***, Int16** *xVal***, Int16** *yVal***);**

void POINT32_SetXY(Point32Ptr *p***, Int32** *xVal***, Int32** *yVal***);**

Sets the X and Y coordinates of point `p' to `xVal' and `yVal' respectively.

IncXY

void POINT16_IncXY(Point16Ptr *p***, Int16** *dx***, Int16** *dy***);**

void POINT32_IncXY(Point32Ptr *p***, Int32** *dx***, Int32** *dy***);**

Increments the X and Y coordinates by `dx' and `dy' respectively.

SetSameXY

void POINT16_SetSameXY(Point16Ptr *p***, Int16** *val***);**

void POINT32_SetSameXY(Point32Ptr *p***, Int32** *val***);**

Sets both X and Y coordinates to `val'.

Reset

void POINT16_Reset(Point16Ptr *p***);**

void POINT32_Reset(Point32Ptr *p***);**

Resets both X and Y coordinates to 0.

IsNull

BoolEnum POINT16_IsNull(Point16Ptr *p***);**

BoolEnum POINT32_IsNull(Point32Ptr *p***);**

Returns BOOL_TRUE if both coordinates are 0.

Equals

BoolEnum POINT16_Equals(Point16Ptr *p1*, **Point16Ptr** *p2***);**

BoolEnum POINT32_Equals(Point32Ptr *p1*, **Point32Ptr** *p2***);**

Returns BOOL_TRUE if `p1' and `p2' have same coordinates.

AbsDist

Int16 POINT16_AbsDist(Point16Ptr *p1*, **Point16Ptr** *p2***);**

Int32 POINT32_AbsDist(Point32Ptr *p1*, **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

BoolEnum POINT16_IsInRectExt(Point16Ptr *p*, **Point16Ptr** *ext***);**

BoolEnum POINT32_IsInRectExt(Point32Ptr *p*, **Point32Ptr** *ext***);**

Returns BOOL_TRUE if `p' is in the rectangle with origin (0, 0) and extension `ext'.

Chapter

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 dealloactions 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 uneeded 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 threshhold number of free cells over which unecessary 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

Alloc

PoolPtr POOL_Alloc(void);

Returns a pointer to an allocated but not yet constructed memory pool. The memory pool should be constructed before being used.

void POOL_Construct(PoolPtr *pool***);**

Default memory pool construction.

void POOL_ConstructInfo(PoolPtr *pool***, 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

void POOL_Destruct(PoolPtr *pool***);**

Default memory pool destruction.

Dealloc

void POOL_Dealloc(PoolPtr *pool***);**

Deallocates the memory pool.

Setting/Querying the Information on a Memory Pool

SetInfo

void POOL_SetInfo(PoolPtr *pool***, PoolInfoCPtr** *poolInfo***);**

Updates the memory pool with the information from poolInfo.

QueryInfo

void POOL_QueryInfo(PoolCPtr *pool***, PoolInfoPtr** *poolInfo***);**

Fills poolInfo with the information with which the memory pool was created.

Allocating and Deallocating

NewPtr

HugePtr POOL_NewPtr(PoolPtr *pool***);**

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 POOL_DisposePtr(PoolPtr *pool***, 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

`NumCells`	Number of cells in fragment .
`NumAllocCells`	Number of allocated cells .

NDPoolStatsInfo

`NumCells`	Total number of cells .
`NumAllocCells`	Number of allocated cells .
`Frags`	Array of FragStatsInfoPtr .

QueryStats

void POOL_QueryStats(PoolCPtr *pool***, PoolStatsInfoPtr** *stats***);**

Fills stats with the statistics information on the pool. The Frags array in stats should be created by the caller, but it will get filled by the call.

ResetStats

void POOL_ResetStats(PoolStatsInfoPtr *stats***);**

Resets stats.

Chapter

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, PTR_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 PTR_NEW or PTR_SetSize is invalid (i.e., negative size). You must use the PTR_Huge… routines to allocate buffers larger than HUGELIMIT (65520), otherwise your code won't be portable to MS Windows.
`PTR_FAILNULL`	PTR_Dispose or PTR_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 PTR_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 PTR_SetFailProc.

See also

PTR_SetFailProc, PTR_DefFailProc, PTR_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.

PtrSizeVal PTR_GetAlignedSize (PtrSizeVal *size***);**

PTR_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.

void PTR_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 PTR_New or PTR_SetSize for a size which is greater than HUGELIMIT. If ever this happens, PTR_New or PTR_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 developping 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 dereferenciation of a huge pointer is significantly slower than the dereferenciation 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 PTR_New or PTR_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.

PtrSizeVal PTR_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, PTR_SetSize calls PTR_New(size) and returns the result.

VoidPtr PTR_SetSize (VoidPtr *ptr***, PtrSizeVal** *size***);**

New

Allocates a new pointer of the size specified or for named structure.

VoidPtr PTR_New (PtrSizeVal *size***);**

PTR_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.

void PTR_Dispose (VoidPtr *ptr***);**

When DBG_ON is not defined, PTR_Dispose is the same as using standard C free routine — it frees the memory allocated for the ptr passed. When DBG_ON is defined, PTR_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:

HugePtr PTR_HugeNew (PtrHugeSizeVal *size***);**

void PTR_HugeDispose (HugePtr *hugeptr***);**

PtrHugeSizeVal PTR_HugeGetSize (HugePtr *ptr***);**

HugePtr PTR_HugeSetSize (HugePtr *ptr***, PtrHugeSizeVal** *size***);**

Same as before except that the sizes can be larger than HUGELIMIT. You must use the PTR_Huge… routines to allocate buffers larger than HUGELIMIT (65520), otherwise you 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.

void PTR_Clear (VoidPtr *ptr***, PtrSizeVal** *number***);**

PTR_Clear sets the number of bytes starting at ptr to zero.

See also

PTR_Copy, PTR_Move, PTR_Set

Set

Sets the specified number of bytes of a buffer to a value specified.

void PTR_Set (VoidPtr *dest***, Byte** *byte***, PtrSizeVal** *number***);**

PTR_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.

VoidPtr PTR_SetSize (VoidPtr *ptr***, PtrSizeVal** *size***);**

PTR_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.

void PTR_Copy (VoidPtr *dest*, **VoidCPtr** *src*, **PtrSizeVal** *number***);**

PTR_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.

void PTR_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.

void PTR_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.

CmpEnum PTR_Cmp (VoidCPtr *p1*, **VoidCPtr** *p2*, **PtrHugeSizeVal** *size***);**

Matches

Returns whether `p1' and `p2' match exactly on their first `size'.

BoolEnum PTR_Matches (VoidCPtr *p1*, **VoidCPtr** *p2*, **PtrHugeSizeVal** *size***);**

Move

Copies the number of bytes specified to an overlapping location.

void PTR_Move (VoidPtr *dest*, **VoidCPtr** *src*, **PtrSizeVal** *size***);**

PTR_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:

void PTR_HugeClear (HugePtr *hugeptr***, PtrHugeSizeVal** *size***);**

void PTR_HugeSet (HugePtr *dest***, Byte** *i***, PtrHugeSizeVal** *size***);**

HugePtr PTR_HugeSetSize (HugePtr *hugeptr***, PtrHugeSizeVal** *size***);**

void PTR_HugeCopy (HugePtr *dest***, HugeCPtr** *src***, PtrHugeSizeVal** *size***);**

void PTR_HugeMove (HugeCPtr *p1***, HugeCPtr** *p2***, PtrHugeSizeVal** *size***);**

void PTR_HugeSwap (HugeCPtr *p1***, HugeCPtr** *p2***, PtrHugeSizeVal** *size***);**

CmpEnum PTR_HugeCmp (HugeCPtr *p1***, HugeCPtr** *p2***, PtrHugeSizeVal** *size***);**

BoolEnum PTR_HugeMatches (HugeCPtr *p1***, HugeCPtr** *p2***, PtrHugeSizeVal** *size***);**

void PTR_HugeMove (HugePtr *dest***, HugeCPtr** *src***, PtrHugeSizeVal** *size***);**

You must use the PTR_Huge… routines to allocate buffers larger than HUGELIMIT (65520), otherwise you code won't be portable to MS/Windows.

PTR_HugeSet is similar to PTR_Set, but fills with an integer i instead of a character.

PTR_HugeSetSize reallocates ptr to the size specified. Similar to PTR_SetSize, but PTR_HugeSetSize you can set a pointer for sizes much larger than the defined HUGELIMIT (65520).

Statistics

QueryStats

Determines the current memory manager statistics.

void PTR_QueryStats (PtrStatsPtr *ptr***);**

PTR_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.

void PTR_StatsOutput (void);

PTR_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'.

Byte PTR_GetByte (VoidCPtr *ptr***, PtrSizeVal** *offset***);**

SetByte

Sets new value at address ptr+'offset' to 'byte'.

void PTR_SetByte(VoidPtr *ptr***, PtrSizeVal** *offset***, Byte** *byte***);**

CopyByte

Copies value stored at ‘src’+’srcOffset’ to dst+’dstOffset’.

void PTR_CopyByte(VoidPtr *dst***, PtrSizeVal** *dstOffset***, VoidCPtr** *src***, PtrSizeVal** *srcOffset***);**

SwapByte

Exchanges values at ptr1+’offset1’ and ‘ptr2’+’offset2’.

void PTR_SwapByte (VoidPtr *p1***, PtrSizeVal** *offset1***, VoidPtr** *p2***, PtrSizeVal** *offset2***);**

See also PTR_WriteInt8, PTR_WriteInt16, PTR_WriteInt32

HugeCopyByte
HugeGetByte
HugeSetByte
HugeSwapByte

The following functions are analogous to the above, but they operate on huge pointers:

Byte PTR_HugeGetByte(HugeCPtr *ptr***, PtrHugeSizeVal offset);**

void PTR_HugeSetByte(HugePtr *ptr***, PtrHugeSizeVal** *offset***, Byte** *byte***);**

void PTR_HugeCopyByte(HugePtr *dst***, PtrHugeSizeVal** *dstOffset***, HugeCPtr** *src***, PtrHugeSizeVal** *srcOffset***);**

void PTR_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 sigificant byte is stored in lowest address. On Little-Endian machines, most sigificant 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.

void PTR_Int8ToStd(Int8Ptr *valp***);**

void PTR_Int16ToStd(Int16Ptr *valp***);**

void PTR_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.

void PTR_Int8ToMch(Int8Ptr *valp***);**

void PTR_Int16ToMch(Int16Ptr *valp***);**

void PTR_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.

void PTR_ReadInt8(VoidCPtr *ptr***, Int8Ptr** *valp***);**

void PTR_ReadInt16(VoidCPtr *ptr***, Int16Ptr** *valp***);**

void PTR_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.

void PTR_WriteInt8(VoidPtr *ptr***, Int8CPtr** *valp***);**

void PTR_WriteInt16(VoidPtr *ptr***, Int16CPtr** *valp***);**

void PTR_WriteInt32(VoidPtr *ptr***, Int32CPtr** *valp***);**

See also

PTR_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.

void PTR_StrToStd (Str *str***, StrIVal** *len***);**

StrToMch

Converts in place a string from machine-dependent format to standard format.

void PTR_StrToMch (Str *str***, StrIVal** *len***);**

Converts in place a string from standard format to machine-dependent format.

See also PTR_ReadStr, PTR_WriteStr.

ReadStr

Read and write machine-dependent strings to and from standard format.

void PTR_ReadStr (HugeCPtr *ptr***, Str** *str***, StrIVal** *len***);**

WriteStr

Reads a machine-dependent string from a memory buffer where strings are stored in standard format.

void PTR_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 PTR_StrToStd, PTR_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 >
PTR_SetFailProc(S_MyFailProc);
< in the caller code>
ptr = PTR_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.

PtrFailProc PTR_GetFailProc (void);

PTR_GetFailProc retrieves the previously installed custom failure procedure.

See also PTR_SetFailProc, PTR_DefFailProc, PtrFailEnum…

SetFailProc

Sets a custom failure callback procedure.

void PTR_SetFailProc (PtrFailProc *failProc***);**

PTR_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 PTR_GetFailProc, PTR_DefFailProc, PtrFailEnum…

DefFailProc

Default method to trap memory management failures.

HugePtr PTR_DefFailProc(PtrFailEnum *fail***, PtrHugeSizeVal** *size***);**

Chapter

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 necesary 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).
`Contruct`	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 exported by default.
`RCLAS_FLAGCOMPOSITE`	Resource has widgets in its children field.
`RCLAS_FLAGUNKNOWN`	Resource loaded from a .dat file and belongs to an 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.

Class Registration

Register

void RCLAS_Register (RClasPtr *rclas***);**

Registers the class and its default notification procedure with the resource manager.

CPlusRegister

Registers a new resource class with the resource manager.

RClasPtr RCLAS_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, RCLASS_Add.

Add

Registers the class and its default notification procedure with the resource manager.

void RCLAS_Add (RClasPtr *rclas***, ResNfyProc** *nfyProc***);**

See also RCLAS_Register,RCLAS_SetDefNfy.

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 RCLAS_OperatorNew (RClasPtr *rclas*, **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 RCLAS_OperatorDelete (RClasPtr *rclas*, **VoidPtr** *obj***);**

Member Functions

Accessing the Class Callbacks

These functions return the corresponding installed class method. (Recommended for advanced programmers.)

GetNew
GetDelete
GetConstruct
GetDestruct

RClasNewCBack RCLAS_GetNew (RClasPtr *rclas***);**
RClasDeleteCBack RCLAS_GetDelete (RClasPtr *rclas***);**
RClasConstructCBack RCLAS_GetConstruct (RClasPtr *rclas***);**
RClasDestructCBack RCLAS_GetDestruct (RClasPtr *rclas***);**

Get…

Functions that get the various fields of an RClas structure.

PtrSizeVal RCLAS_GetSizeOfRes (RClasCPtr *rclas***);**

CStr RCLAS_GetName (RClasCPtr *rclas***);**

PFldPtr RCLAS_GetFields (RClasPCtr *rclas***);**

RClasFlagsSet RCLAS_GetFlags (RClasCPtr *rclas***);**

RClasPtr RCLAS_GetParentClass (RClasCPtr *rclas***);**

ResPtr RCLAS_GetTemplate (RClasPtr *rclas***);**

CStr RCLAS_GetModName (RClasCPtr *rclas***);**

RClasVersionVal RCLAS_GetVersion (RClasCPtr *rclas***);**

The RCLASS_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

RCLASS_Set…

GetDefNfy

Get the default notification handler of an RClas.

ResNfyProc RCLAS_GetDefNfy (RClasCPtr *rclas***);**

The RCLASS_GetDefNfy procedure returns the defualt notification handler (the DefNotify field) of the given rclas structure.

See also

RCLASS_SetDefNfy, RCLASS_Get…

Querying Database of Resource Classes

FindByName

Returns a class by name.

RClasPtr RCLAS_FindByName (CStr *name***);**

RCLASS_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.

static RClasPtr RCLAS_GetFirst (void);

The RCLASS_GetFirst function returns the first alphabetical resource class.

See also

RCLASS_GetNext, RCLASS_FindByName

GetNext

Returns the class that is alphabetically after the class specified.

RClasPtr RCLAS_GetNext (RClasPtr *class***);**

RCLASS_GetNext returns the class that is alphabetically after class.

See also

RCLASS_GetFirst, RCLASS_FindByName.

Testing Inheritance

IsSubClassOf

Determines whether one class is a subclass of another.

BoolEnum RCLAS_IsSubClassOf (RClasCPtr *childclass***, RClasCPtr** *parentclass***);**

RCLASS_IsSubClassOf returns BOOL_TRUE if childclass inherits (directly or indirectly) from parentclass.

See also

RClasRec

Setting the Class Callbacks

These functions set the corresponding installed class method. (Recommended for advanced programmers.) These calls will typically be used only after the class is created and before any instance is created.

SetNewProc
SetDeleteProc
SetConstructProc
SetDestructProc

void RCLAS_SetNewProc(RClasPtr *rclas*, **RClasNewProc** *newProc***);**

void RCLAS_SetDeleteProc(RClasPtr *rclas*, **RClasDeleteProc** *delProc***);**

void RCLAS_SetConstructProc(RClasPtr *rclas*, **RClasConstructProc** *constProc***);**

void RCLAS_SetDestructProc(RClasPtr *rclas*, **RClasDestructProc** *destProc***);**

Set…

Functions that set the various fields of an RClas structure.

void RCLAS_SetSizeOfRes (RClasPtr *rclas*, **PtrSizeVal** *size***);**

void RCLAS_SetName (RClasPtr *rclas*, **Str** *name***);**

void RCLAS_SetFields (RClasPtr *rclas*, **PFldPtr** *fields***);**

void RCLAS_SetFlags (RClasPtr *rclas*, **RClasFlagsSet** *flags***);**

void RCLAS_SetParentClass (RClasPtr *rclas*, **RClasPtr** *parent***);**

void RCLAS_SetVersion (RClasPtr *rclas*, **RClasVersionVal** *version***);**

void RCLAS_SetModName (RClasPtr *rclas*, **CStr** *modname***);**

The RCLASS_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

RCLASS_Get…

SetDefNfy

Set the default notification handler for an RClas.

void RCLAS_SetDefNfy (RClasPtr *rclas*, **ResNfyProc** *defnfy***);**

The RCLASS_SetDefNfy procedure sets the DefNotify field of the given rclas structure to the given defnfy handler.

See also

RCLASS_GetDefNfy, RCLASS_Set…

ProcessDefNfy

Trigger the default notification procedure on an instance.

void RCLAS_ProcessDefNfy (RClasCPtr *rclas***, ResPtr** *res***, ResNfyEnum** *code***);**

RCLASS_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 RCLASS_ProcessParentDefNfy

ProcessParentDefNfy

Trigger the parent default notification procedure on an instance.

void RCLAS_ProcessParentDefNfy (RClasCPtr *rclas***, ResPtr** *res***, ResNfyEnum** *code***);**

RCLASS_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.

Chapter

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:

Ori.x <= x < Ori.x + Ext.x

Ori.y <= y < Ori.y + 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:

```
BegX = Ori.x  EndX = Ori.x + Ext.x
BegY = Ori.x  EndY = Ori.y + 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, RECT_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 POINT16_AbsDist(Point16Ptr *p1*, **Point16Ptr** *p2***);**

Int32 POINT32_AbsDist(Point32Ptr *p1*, **Point32Ptr** *p2***);**

POINT_ABSDIST returns the absolute distance between the points p1 and 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 RECT16_ContainsPoint(Rect16CPtr *r*, **Point16CPtr** *p***);**

BoolEnum RECT32_ContainsPoint(Rect32CPtr *r*, **Point32CPtr** *p***);**

RECT_ContainsPoint returns BOOL_TRUE if rectangle r contains point p. (There are 16-bit and 32-bit versions of this function.)

See also RECT_16IsEmpty, RECT_32IsEmpty.

SetOriExtXY

Sets all Origin / Extent coordinates.

void RECT16_SetOriExtXY(Rect16Ptr *r*, **Int16** *orix*, **Int16** *oriy*, **Int16** *extx*, **Int16** *exty***);**

void RECT32_SetOriExtXY(Rect32Ptr *r*, **Int32** *orix*, **Int32** *oriy*, **Int32** *extx*, **Int32** *exty***);**

RECT_SetOriExtXY sets the x, y coordinates and extent.

IncOriExtXY

Increments all Origin / Extent coordinates.

void RECT16_IncOriExtXY(Rect16Ptr *r*, **Int16** *orix*, **Int16** *oriy*, **Int16** *extx*, **Int16** *exty***);**

void RECT32_IncOriExtXY(Rect32Ptr *r*, **Int32** *orix*, **Int32** *oriy*, **Int32** *extx*, **Int32** *exty***);**

IsEmpty

Determines whether a point is empty.

BoolEnum EXT_IsEmpty (Point16Ptr *point***);**

EXT_IsEmpty returns BOOL_TRUE if point is empty; otherwise, BOOL_FALSE.

SetXY

Sets point coordinates.

void POINT16_SetXY(Point16Ptr *p*, **Int16** *xVal*, **Int16** *yVal***);**

void POINT32_SetXY(Point32Ptr *p*, **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 POINT16_IncXY(Point16Ptr p, Int16 *dx*, **Int16** *dy***);**

void POINT32_IncXY(Point32Ptr p, 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 RECT16_SetByPoints(Rect16Ptr *r*, **Point16CPtr** *beg*, **Point16CPtr** *end***);**

void RECT32_SetByPoints(Rect32Ptr *r*, **Point32CPtr** *beg*, **Point32CPtr** *end***);**

RECT_SetByPoints sets the coordinates of rect with `beg' and `end' as opposite corners.

IsEmpty

Determines whether a rectangle is empty.

BoolEnum RECT16_IsEmpty(Rect16CPtr *r***);**

BoolEnum RECT32_IsEmpty(Rect32CPtr *r***);**

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 RECT16_Reset(Rect16Ptr *r***);**
void RECT32_Reset(Rect32Ptr *r***); */**

Makes `r' empty by changing its extent to (0,0).

Rect Functions

Equals

Returns BOOL_TRUE if ‘r1’ and ‘r2’ are identical.

BoolEnum RECT16_Equals (Rect16CPtr *r1*, **Rect16CPtr** *r2***);**

BoolEnum RECT32_Equals (Rect32CPtr *r1*, **Rect32CPtr** *r2***);**

IncludesNonEmptyRect

Returns BOOL_TRUE if ‘r1’ is included in ‘r2’ (assuming ‘r1’ is not empty).

BoolEnum RECT16_IncludesNonEmptyRect (Rect16CPtr *r2*, **Rect16CPtr** *r1***);**

BoolEnum RECT32_IncludesNonEmptyRect (Rect32CPtr *r2*, **Rect32CPtr** *r1***);**

Copy

Copies a rectangle.

void RECT16_Copy (Rect16Ptr *dst*, **Rect16CPtr** *src***);**

void RECT32_Copy (Rect32Ptr *dst*, **Rect32CPtr** *src***);**

RECT_Copy copies the src rectangle to the dest rectangle.

CopyResetOri

Copies ‘src’ into ‘dst’, but then sets dst->Ori to (0,0).

void RECT16_CopyResetOri(Rect16Ptr *dst*, **Rect16CPtr** *src***);**

void RECT32_CopyResetOri(Rect32Ptr *dst*, **Rect32CPtr** *src***);**

Intersects

Determines whether two rectangles intersect.

BoolEnum RECT16_Intersects (Rect16CPtr *r1*, **Rect16CPtr** *r2***);**

BoolEnum RECT32_Intersects (Rect32CPtr *r1*, **Rect32CPtr** *r2***);**

Returns BOOL_TRUE if `r1' and `r2' intersect; otherwise, BOOL_FALSE.

See also

RECT_IsEmpty, RECT_IsEmpty32, RECT_IncludesRect.

IncludesRect

Determines whether a rectangle contains the rectangle specified.

BoolEnum RECT16_IncludesRect (Rect16CPtr *r2*, **Rect16CPtr** *r1***);**

BoolEnum RECT32_IncludesRect (Rect32CPtr *r2*, **Rect32CPtr** *r1***);**

Returns BOOL_TRUE if `r1' is included in `r2' (`r1' can be empty).

Union

Determines the union of two rectangles.

void RECT16_Union (Rect16Ptr *dst*, **Rect16CPtr** *src***);**

void RECT32_Union (Rect32Ptr *dst*, **Rect32CPtr** *src***); */**

Sets `dst' to the union of `dst' and `src'.

Intersection

void RECT16_Intersection (Rect16Ptr *dst*, **Rect16CPtr** *src***);**

void RECT32_Intersection (Rect32Ptr *dst*, **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 RECT16_MakeFit (Rect16Ptr *in*, **Rect16CPtr** *out*);

void RECT32_MakeFit (Rect32Ptr *in*, **Rect32CPtr** *out*);

MakeFit repositions in so that it is contained within out.

MoveInside

Moves one rectangle inside another.

void RECT16_MoveInside (Rect16Ptr *in*, **Rect16CPtr** *out*);

void RECT32_MoveInside (Rect32Ptr *in*, **Rect32CPtr** *out*);

MoveInside moves rect so that it is inside outrect.

See also

RECT_ContainsPoint, RECT_IncludesRect, RECT_MakeFit.

IsValid

Determines whether a rectangle has valid coordinates.

BoolEnum RECT16_IsValid(Rect16Ptr *r*);

BoolEnum RECT32_IsValid(Rect32Ptr *r*);

IsValid determines whether a rectangle r has valid coordinates.

MakeValid

Changes the coordinates of the rectangle specified to make them valid.

void RECT16_MakeValid(Rect16Ptr *r*);

void RECT32_MakeValid(Rect32Ptr *r*);

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

RECT_IsValid

Rectangles Defined by Origin and Extent

Get...

Int16 RECT16_GetOriX(Rect16CPtr *r*);

Int16 RECT16_GetOriY(Rect16CPtr *r*);

Int16 RECT16_GetExtX(Rect16CPtr *r*);

Int16 RECT16_GetExtY(Rect16CPtr *r*);

Int32 RECT32_GetOriX(Rect32CPtr *r*);

Int32 RECT32_GetOriY(Rect32CPtr *r*);

Int32 RECT32_GetExtX(Rect32CPtr *r*);

Int32 RECT32_GetExtY(Rect32CPtr *r*);

Gets one Origin / Extent coordinate.

Set...

void RECT16_SetOriX(Rect16Ptr *r*, **Int16** *val***);**

void RECT16_SetOriY(Rect16Ptr *r*, **Int16** *val***);**

void RECT16_SetExtX(Rect16Ptr *r*, **Int16** *val***);**

void RECT16_SetExtY(Rect16Ptr *r*, **Int16** *val***);**

void RECT32_SetOriX(Rect32Ptr *r*, **Int32** *val***);**

void RECT32_SetOriY(Rect32Ptr *r*, **Int32** *val***);**

void RECT32_SetExtX(Rect32Ptr *r*, **Int32** *val***);**

void RECT32_SetExtY(Rect32Ptr *r*, **Int32** *val***);**

Sets one Origin / Extent coordinate.

Rectangles Defined by Beginning and End

Get...

Int16 RECT16_GetBegX(Rect16CPtr *r***);**

Int16 RECT16_GetBegY(Rect16CPtr *r***);**

Int16 RECT16_GetEndX(Rect16CPtr *r***);**

Int16 RECT16_GetEndY(Rect16CPtr *r***);**

Int32 RECT32_GetBegX(Rect32CPtr *r***);**

Int32 RECT32_GetBegY(Rect32CPtr *r***);**

Int32 RECT32_GetEndX(Rect32CPtr *r***);**

Int32 RECT32_GetEndY(Rect32CPtr *r***);**

Gets one Begin / End coordinate.

Set...

void RECT16_SetBegX(Rect16Ptr *r*, **Int16** *val***);**

void RECT16_SetBegY(Rect16Ptr *r*, **Int16** *val***);**

void RECT16_SetEndX(Rect16Ptr *r*, **Int16** *val***);**

void RECT16_SetEndY(Rect16Ptr *r*, **Int16** *val***);**

void RECT32_SetBegX(Rect32Ptr *r*, **Int32** *val***);**

void RECT32_SetBegY(Rect32Ptr *r*, **Int32** *val***);**

void RECT32_SetEndX(Rect32Ptr *r*, **Int32** *val***);**

void RECT32_SetEndY(Rect32Ptr *r*, **Int32** *val***);**

Sets one Begin / End coordinate.

Chapter

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 parrallels 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 RES_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 RES_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) explicitely, the Res level does the lookup and finds the persistent description. When we do not pass the name(s) explicitely, 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 explicitely 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
`ResClass`	Pointer to the RClasRec structure, which describes the class to which the resource belongs.
`ResInitialized`	Indicates whether volatile fields have been initialized.
`ClientData`	32 bit storage for any data or pointer you wish to associate with the resource.
`NfyData`	32 bit storage for information used by some notifications.
`Notify`	Pointer to the class-specific notification procedure.

See Also:

RLib, Wgt classes.

Creating and Disposing

Create

Creates a resource of class `rclas'.

ResPtr RES_Create(RClasPtr *rclas***);**

Persistent fields are initialized with the values defined by the template for the class. The resource does not have a name and is thus detached. The reference count is set to 1.

Clone

ResPtr RES_Clone(ResCPtr *sourceRes***, 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 RES_LoadDetach instead of RES_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 RES_Release(ResPtr *res***);**

RES_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 RES_NFYEND notification) if the ResInitialized flag was set. Then the persistent fields and the resource structure itself are freed. RES_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

RES_Create, RES_Clone

Class

Returns a pointer to the Res class.

RClasPtr RES_Class (void);

RES_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 RES_Use(ResPtr *res***);**

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 RES_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 RES_Use after such calls.

Saving To a Resource Database

SaveDat

Saves attached resources to library database (.DAT) file.

void RES_SaveDat (ResPtr *resource***);**

RES_SaveDat saves resource to a library database (.DAT) file.

Output to a Text Resource File

FilenameOutputRc

Output to text resource file.

void RES_FilenameOutputRc (ResPtr *res***, CStr** *filename***);**

RES_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 RES_LibInit instead of the usual RES_LibInit from the GW module. This will give you a fasterstartup 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.

void RES_LibInstall (void);

RES_LibInstall installs the Res library.

See also

RES_LibInit, RES_LibLoadInit, GW_LibInstall, TKIT_LibInstall

LibLoadInit

Loads and initializes the Res library, making all Open Interface libraries available, except Genwin.

void RES_LibLoadInit (void);

RES_LibLoadInit loads and initializes the Res library.

See also

RES_LibInstall, RES_LibInit, GW_LibLoadInit, TKIT_LibLoadInit

LibExit

Exits the Res library, making all Open Interface libraries unavailable.

void RES_LibExit (void);

RES_LibExit exits the RES library.

See also RES_LibInit, RES_LibInstall, RES_LibLoadInit, GW_LibExit, TKIT_LibExit

LibInit

Installs, loads, and initializes the Res library, making all Open Interface libraries available, except Genwin.

void RES_LibInit (void);

RES_LibInit initializes the Res library.

If your application does not use any windows, you can initialize it with RES_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

RES_LibInstall, RES_LibLoadInit, RES_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 RES_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 WIN_LoadInit and menus through MENU_LoadInit.

Usually, you do not load the widgets of a window yourself, they are loaded automatically when you load the window through the corresponding WIN_LoadInit or WIN_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.

ResPtr RES_LoadByFullName (CStr *modres***);**

RES_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.

RES_LoadByFullName loads the resource without initializing it. Therefore, you should ordinarily use RES_LoadInit or RES_LoadInitDetach rather than RES_LoadByFullName to load your applications.

RES_LoadByFullName never returns NULL; instead, it signals a failure if resource does not exist or cannot be loaded

See also RES_Load, RES_LoadInit, RES_LoadInitDetach, RES_FindByFullName

Load

Loads a resource using two parameters.

ResPtr RES_Load (CStr *module***, CStr** *resource***);**

RES_Load loads and returns resource, which is resource contained in class.

RES_Load loads an attached resource given its module name `modName' and its resource name `resName' (relative to the module name).

Calling RES_Load twice will return the SAME resource pointer.

RES_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 RES_NfyInit notification to be processed by a notification procedure other than the default class notification procedure. The problem with RES_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 RES_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 RES_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

RES_Load, RES_LoadInit, RES_LoadInitDetach

LoadDetach

Loads a detached resource.

ResPtr RES_LoadDetach (CStr *module***, CStr** *resource***);**

The RES_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

RES_LoadInitDetach, RES_Load, RES_LoadInit, RES_Load, RES_FindByFullName, WIN_LoadInit, MENU_LoadInit

LoadInit

Loads and initializes an attached resource.

ResPtr RES_LoadInit (CStr *module***, CStr** *resource***);**

The RES_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 RES_LoadInit twice returns the same resource pointer, so use this function for resources that will need to accessed several times and shared among client applications, for example: fonts, colors, cursors. (WIN_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

RES_LoadInitDetach, RES_FindByFullName, RES_Load, RES_LoadDetach, WIN_LoadInit, MENU_LoadInit.

LoadInitDetach

Loads and initializes a detached resource.

ResPtr RES_LoadInitDetach (CStr *module***, CStr** *resource***);**

The RES_LoadInitDetach function loads, initializes, and returns a detached resource.

Each time you call RES_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 WIN_LoadInit and NDMnu::LoadInit, which both call RES_LoadInitDetach

This function never returns NULL; it signals a failure for a non-existent resource or a resource that cannot be loaded.

See also

RES_LoadDetach, RES_Load, RES_LoadInit, RES_Load, RES_FindByFullName, WIN_LoadInit, MENU_LoadInit

LoadChildren

Loads all of the children resources for a resource.

void RES_LoadChildren (ResPtr *resource***);**

The RES_LoadChildren loads all of the children resources of the specified resource.

See also

RES_GetNumChildren, RES_GetNthChild

FindByFullName

Loads the attached resource specified, returning NULL if the resource does not exist.

ResPtr RES_FindByFullName (CStr *Mod.Res***);**

RES_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 RES_Load, this function loads a resource without initializing it. Unlike RES_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

RES_Load

Find

Loads a resource by class name and resource name, returning NULL if the resource does not exist.

ResPtr RES_Find (CStr *modname***, CStr** *resname***);**

RES_Find loads the resource specified by modname and resname, returning NULL if the resource specified does not exist.

See also

RES_Load

Accessing the Name of a Resource

IsNamed

Determines whether a resource has a name or not.

BoolEnum RES_IsNamed (ResCPtr *res***);**

RES_IsNamed returns BOOL_TRUE if the resource has a name; otherwise, it returns BOOL_FALSE. If a resource was created dynamically with RES_Create, it will not have a name.

See also

RES_GetName, RES_QueryFullName

GetName

Returns the name of a resource as a NULL-terminated string.

CStr RES_GetName (ResCPtr *res***);**

RES_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 RES_Create to create the widget, a default name.

See also

RES_QueryFullName, RES_Create

QueryFullName

Determines the full name of a resource.

void RES_QueryFullName (ResCPtr *res***, Str** *name***, StrIVal** *length***);**

RES_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 RES_SetClientData (ResPtr *res*, **ClientPtr** *data***);**

RES_SetClientData sets the data passed into the resource.

GetClientData

Returns the client data of a resource.

ClientPtr RES_GetClientData (ResCPtr *res***);**

RES_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 RES_GetNumChildren (ResCPtr *res***);**

The RES_GetNumChildren function returns the total number of children belonging to res as an integer.

See also

RES_GetNthChild, RES_LoadChildren

GetNthChild

Returns the child resource specified for a resource.

ResPtr RES_GetNthChild (ResPtr *resource*, **ArrayIVal** *child***);**

The RES_GetNthChild returns a pointer to the child resource specified by its number.

Note that the children of the resource must already be loaded. RES_GetNthChild does not increment the reference count of the child resource returned.

See also

RES_GetNumChildren, RES_LoadChildren

Accessing the Class of A Resource

GetClass

Returns the class of the resource specified.

RClasCPtr RES_GetClass (ResCPtr *class***);**

RES_GetClass returns the class of the resource specified. The same pointer is returned if called on different instances of the same class.

See also

RCLAS_GetFirst, RCLAS_GetNext, RCLAS_FindByName

InheritsFrom

Determines whether one class inherits from another class.

BoolEnum RES_InheritsFrom (ResCPtr *childclass***, RClasCPtr** *parentclass***);**

RES_InheritsFrom returns BOOL_TRUE if childclass inherits (directly or indirectly) from parentclass.

See also

RES_CheckClass, RCLAS_IsSubClassOf

Resource States

IsInitialized

Determines whether a resource has been initialized already.

BoolEnum RES_IsInitialized (ResCPtr *res***);**

RES_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
`RES_NFYINIT`	Persistent fields have been initialized.	Initialize volatile fields of resource.

`RES_NFYEND`	Resource is no longer needed.	Deallocate volatile fields of resource that you allocated with NFYINIT, and, if the value is not overwritten by an new NFYINIT, you should RESET the volatile fields to NULL.
`RES_NFYRESET`	Persistent fields have been modified.	Reinitialize volatile fields, that is, deallocate and reallocate them according to current state of the persistent fields.
`RES_NFYRESMGR`	Not documented yet.	
`RES_NFYCTRLDATA`	Not documented yet.	
`RES_NFYSETDATA`	Not documented yet.	
`RES_NFYGETDATA`	Not documented yet.	
`RES_NFYDESTRUCTED`		Sent just before the resource is destructed by Open Interface
`RES_NFYDEALLOCATE`		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 RES_NFYINIT and frees the allocated fields on receipt of a RES_NFYEND. RES_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.

RES_NFYEND deallocates the volatile fields of a resource that you allocated with RES_NFYINIT, and if the value is not overwritten by an new RES_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 RES_NFYINHERIT, RES_SendNfyInit, PANEL_SetSubNfyProc

ResNfyProc

Pointer to a resource notification procedure.

typedef void (*ResNfyProc) L((ResPtr, ResNfyEnum));

Pointer to a resource notification procedure. The notification procedure receives a subclass of ResPtr and a subclass of ResNfyEnum as arguments

See also RES_SetNfyProc, RES_GetNfyProc, RClasRec

DefNfy

Default notification handler for a resource class.

void RES_DefNfy (ResPtr *res***, ResNfyEnum** *notif***);**

The RES_DefNfy procedure is the default notification handler for all resource classes. All resource classes, as subclasses of Res, use the RES_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 RES_DefNfy to initialize notification codes the handler does not process itself.

See also RES_SendNfyInit

SetNfyProc

Sets a client notification routine for a resource.

void RES_SetNfyProc (ResPtr *res***, ResNfyProc** *nfyproc***);**

The RES_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 RES_GetNfyProc, ResNfyProc

GetNfyProc

Returns the currently installed notification routine for a resource.

ResNfyProc RES_GetNfyProc (ResPtr *res***);**

RES_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 RES_SetNfyHandler(ResPtr *res***, 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 RES_SetNfyHandlerProc(), NULL is returned.

ResNfyHandlerProc RES_GetNfyHandlerProc(ResPtr *res***, 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 RES_RemoveNfyHandler (ResPtr *res***, ResNfyEnum** *nfy***);**

SetNfyHandlerClientData

Associates ‘data’ with the call-back defined for the resource and the notification ‘code’. ‘data’ can later be retrieved using RES_GetNfyHandlerClientData if needed.

void RES_SetNfyHandlerClientData (ResPtr *res***, ResNfyEnum** *nfy***, ClientPtr** *data***);**

GetNfyHandlerClientData

Returns the ClientData installed for ‘res’ to process the ‘nfy’ message.

ClientPtr RES_GetNfyHandlerClientData (ResPtr *res***, ResNfyEnum** *nfy***);**

See also RES_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 RES_NFYINIT to a Res instance will activate the corresponding NfyInit virtual method)
- the function is reponsible 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 RES_SendNfy (ResPtr *resource***, ResNfyEnum** *notif***);**

RES_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 RES_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 RES_LockedSendNfyData (ResPtr *resource***, ResNfyEnum** *notif***, ClientPtr** *data***);**

RES_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, WIN_SendNfy.

Ordinarily you will not use RES_LockedSendNfyData directly, unless you are creating a custom widget. If so, you may call this routine to trap notifications.

See also

RES_SendNfyData, RES_GetNfyData, RES_SetNfyData

SendNfyData

Notifies resource clients and sends data specified.

void RES_SendNfyData (ResPtr *resource***, ResNfyEnum** *notif***, ClientPtr** *data***);**

RES_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, WIN_SendNfy.

Ordinarily you will not use RES_SendNfyData directly, unless you are creating a custom widget. If so, you may call this routine to trap notifications.

See also

RES_LockedSendNfyData, RES_SetNfyData, RES_GetNfyData

GetNfyData

Returns the notify data of a resource.

ClientPtr RES_GetNfyData (ResPtr *res***);**

RES_GetNfyData retrieves the notify data for the resource specified.

SendNfyInit

Sends a RES_NFYINIT to the resource specified.

void RES_SendNfyInit (ResPtr *res***);**

RES_SendNfyInit sends a RES_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, WIN_SendNfyInit.

Ordinarily you will not use RES_SendNfyInit directly, unless you are creating a custom widget. If so, you may call this macro to trap the RES_NFYINIT notification to initialize the volatile fields after the persistent fields have been initialized.

See also RES_SendNfyEnd, RES_SendNfyReset, RES_SendNfy

SendNfyEnd

Sends a RES_NFYEND to the resource specified.

void RES_SendNfyEnd (ResPtr *res***);**

RES_SendNfyEnd sends a RES_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 RES_SendNfyEnd directly, unless you are creating a custom widget. If so, you may call this macro to trap the RES_NFYEND notification to deallocate the volatile fields before destroying or resetting the widget.

See also RES_SendNfyInit, RES_SendNfyReset, RES_SendNfy

SendNfyReset

Sends a RES_NFYRESET to the resource specified.

void RES_SendNfyReset (ResPtr *res***);**

RES_SendNfyReset sends a RES_NFYRESET notification to the resource. In case the resource has already been initialized (the ResInitialized flag was already set), it sends a RES_NFYEND notification followed by a RES_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, WIN_SendNfyReset.

Ordinarily you will not use RES_SendNfyReset directly, unless you are creating a custom widget. If so, you may call this macro to trap the RES_NFYRESET notification to update the volatile fields after the widget received changes to its persistent fields.

See also

RES_SendNfyEnd, RES_SendNfyInit, RES_SendNfy

Responding to a Notification

ClassDefNfy

Trigger the default notification procedure on an instance.

void RCLAS_ClassDefNfy (ResPtr *res*, **ResNfyEnum** *code***);**

RES_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

RES_ParentClassDefNfy

ParentClassDefNfy

Trigger the parent default notification procedure on an instance.

void RCLAS_ProcessParentDefNfy (ResPtr *res*, **ResNfyEnum** *code***);**

RES_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

RES_ClassDefNfy

Control Data

SendCtrlNfyData

void RES_SendCtrlNfyData (ResPtr *res*, **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 RES_GetNfyCmd(ResCPtr *res***);**

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 RES_IsCmdSource(ResCPtr *res***);**

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.

startvoid RES_CmdSend(ResPtr *res***, ResPtr** *start***, CmdCtlEnum** *ctl***);**

Start routing of the resource’s command at given ’start’ point. (start is sent a the RES_NFYCOMMANDROUTE notification with ‘ctl’ as argument).

CmdIssue

void RES_CmdIssue(ResPtr *res***);**

Issue the resource’s command for execution.

CmdUpdate

void RES_CmdUpdate(ResPtr *res***);**

Issue the resource’s command as command query for self updating. This call is the default answer to the RES_NFYUPDATEVIEW notification when the resource has the RES_FLAGISCOMMANDSOURCE flag set.

Handling Command Notifications

CmdTableHandle

void RES_CmdTableHandle(ResPtr *res***, CmdTablePtr** *table***);**

To be used upon RES_NfyCommand notifications fetches the command object and searches the given command table, using the default RES_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 RES_ExecuteScript(ResPtr *res***, 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 SCRPT_DefNfy(documented in the file scrptpub.h) This is because this SCRPT_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 RES_CheckClass (ResPtr *res***, RClasCPtr** *class***);**

RES_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 RES_VERIFY (ResCPtr *res***, RClasCPtr** *class***);**

RES_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 RES_CheckClass if DBG_ON is defined. This macro is disabled if DBG_ON is not defined.

Chapter

36 *Rgn Class*

The Rgn 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

NDRegion is a subclass of the NDRes (resource) class; the region class inherits the same characteristics defined for resources.

Res->Rgn

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.

RGN_POSINSIDE

RGN_POSCROSS

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

RGN_RectPos

Empty Region

IsEmpty

Determines whether a region is empty.

BoolEnum RGN_IsEmpty (RegionCPtr *region***);**

RGN_IsEmpty returns BOOL_TRUE if the region is empty; otherwise, it returns BOOL_FALSE.

See also

RGN_IsEqual, RGN_IsPointInside

Reset

Resets a region to empty.

void RGN_Reset (RegionPtr *region***);**

RGN_Reset resets region to an empty region.

See also

RGN_Clone, RGN_Translate, RGN_QueryBounds

Region Rectangular Bounds

QueryBounds

Determines the boundaries of a region.

void RGN_QueryBounds (RegionCPtr *region***, Rect16Ptr** *rectangle***);**

RGN_QueryBounds determines the geometry of the smallest rectangle that encloses region. It then places the result in rectangle.

See also

RGN_Clone, RGN_Reset, RGN_Translate

Region Translation

Translate

Translates a region by the offset specified.

void RGN_Translate (RegionPtr *region***, Point16CPtr** *point***);**

RGN_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 RGN_Clone, RGN_Reset, RGN_QueryBounds

Comparisons with other Regions

IsEqual

Determines whether two regions are equal.

BoolEnum RGN_IsEqual (RegionCPtr *region1***, RegionCPtr** *region2***);**

RGN_IsEqual returns BOOL_TRUE if region1 and region2 are equal; otherwise, it returns BOOL_FALSE.

See also

RGN_IsEmpty, RGN_IsPointInside

RectPos

Returns a code indicating the position of a rectangle relative to a region.

RgnPosEnum RGN_RectPos (RegionCPtr *region***, Rect16CPtr** *rectangle***);**

RGN_RectPos returns the position of the region.

See also

RgnPosEnum

IsPointInside

Determines whether a point is inside a region.

BoolEnum RGN_IsPointInside (RegionCPtr *region*, **Point16CPtr** *point***);**

RGN_IsPointInside returns BOOL_TRUE if the point is inside the region; otherwise, it returns BOOL_FALSE.

See also

RGN_IsEqual, RGN_IsEmpty

Operations between Two Regions

RgnSet

Sets the coordinates of one region as specified by another.

void RGN_RgnSet (RegionPtr *dest***, RegionCPtr** *src***);**

RGN_RgnSet sets the coordinates of dest as specified by src. In other words, it copies the coordinates of src to dest.

See also RGN_RgnIntersect, RGN_RgnUnion, RGN_RgnSubtract, RGN_RgnXOr

RgnIntersect

Creates an intersection of two regions.

void RGN_RgnIntersect (RegionPtr *region1***, RegionCPtr** *region2***);**

RGN_RgnIntersect creates the region representing the intersection of region1 and region2 and places the result in region1.

See also

RGN_RgnSet, RGN_RgnUnion, RGN_RgnSubtract, RGN_RgnXOr

RgnUnion

Creates a union of two regions.

void RGN_RgnUnion (RegionPtr *region1***, RegionCPtr** *region2***);**

RGN_RgnUnion creates a union between region1 and region2 and places the result in region1.

See also

RGN_RgnIntersect, RGN_RgnSet, RGN_RgnSubtract, RGN_RgnXOr

RgnSubtract

;Subtracts one region from another.

void RGN_RgnSubtract (RegionPtr *region1***, RegionCPtr** *region2***);**

RGN_RgnSubtract subtracts region2 from region1 and places the result in region1.

See also

RGN_RgnIntersect, RGN_RgnUnion, RGN_RgnSet, RGN_RgnXOr

RgnXOr

Performs an exclusive Or on two regions.

void RGN_RgnXOr (RegionPtr *region1***, RegionCPtr** *region2***);**

RGN_RgnXOr performs an exclusive Or on region1 and region2 and places the result in region1.

See also

RGN_RgnIntersect, RGN_RgnUnion, RGN_RgnSubtract, RGN_RgnSet

Operations between a Region and a Rectangle

RectSet

Associates a region with a rectangle.

void RGN_RectSet (RegionPtr *region***, Rect16CPtr** *rectangle***);**

RGN_RectSet sets region to the coordinates in rectangle.

See also

RGN_RectIntersect, RGN_RectUnion, RGN_RectSubtract, RGN_RectXOr

RectIntersect

Creates an intersection of a region and a rectangle.

void RGN_RectIntersect (RegionPtr *region*, **Rect16CPtr** *rectangle***);**

RGN_RectIntersect creates a new region that is the intersection of region and rectangle. It places the result in region.

See also

RGN_RectSet, RGN_RectUnion, RGN_RectSubtract, RGN_RectXOr

RectUnion

Creates a union of a rectangle and a region.

void RGN_RectUnion (RegionPtr *region*, **Rect16CPtr** *rectangle***);**

RGN_RectUnion creates a union between region and rectangle and places the result in region.

See also

RGN_RectIntersect, RGN_RectSet, RGN_RectSubtract, RGN_RectXOr

RectSubtract

Subtracts a rectangle from a region.

void RGN_RectSubtract (RegionPtr *region*, **Rect16CPtr** *rectangle***);**

RGN_RectSubtract subtracts rectangle from region and places the result in region.

See also

RGN_RectIntersect, RGN_RectUnion, RGN_RectSet, RGN_RectXOr

RectXOr

Performs an exclusive Or between a region and a rectangle.

void RGN_RectXOr (RegionPtr *region*, **Rect16CPtr** *rectangle***);**

RGN_RectXOr performs an exclusive Or on region and rectangle and places the result in region.

See also

RGN_RectIntersect, RGN_RectUnion, RGN_RectSubtract, RGN_RectSet

Regions Specified by a Polygon

void RGN_Construct(RegionPtr *region***);**

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 RGN_PropagateAction(RegionPtr *rgn*, **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.

Chapter 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.

RLibPtr RLIB_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. RLIB_GetLibName is the opposite API

GetLibName

Returns the name of a library.

CStr RLIB_GetLibName(RLibCPtr *lib***);**

Returns the name of the library lib. This is the opposite of RLIB_Find.

GetFirst

Returns the first library in the list.

RLibPtr RLIB_GetFirst (void);

Returns the first library in the list of resource libraries already loaded.

GetNext

Returns the next library in the list.

RLibPtr RLIB_GetNext (RLibPtr *lib***);**

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.

RLibPtr RLIB_LoadEdit (CStr *libname***, CStr** *filename***);**

RLIB_LoadEdit loads a libname in read-write mode and returns its pointer. You should open your library with RLIB_LoadEdit (instead of RLIB_LoadLibFile) if you intend to use RES_SaveDat.

See also

RES_SaveDat

LoadFile

Loads a library database file by full name and returns it.

RLibPtr RLIB_LoadFile (CStr *fullname***);**

RLIB_LoadFile loads the library database file identified by fullname, then returns a pointer to it.

See also

RLIB_LoadLibFile

LoadLibFile

Loads a library.

RLibPtr RLIB_LoadLibFile (CStr *libname*, **CStr** *filename***);**

RLIB_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.

void RLIB_Unload(RLibPtr *lib***);**

This call is the "safe" version of RLIB_Dispose because all resources from lib are first detached before the lib structure is deleted from memory. (You must explicitely release each resource with RES_Release to free the memory that it occupies)

Dispose

Unloads library 'lib' and all the resources it contains and closes the file.

void RLIB_Dispose(RLibPtr *lib***);**

WARNING: THIS IS THE UNSAFE VERSION OF RLIB_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). RLIB_Dispose doesn't perform any checking before deleting the content of lib!

Open

Opens a file.

void RLIB_Open(RLibPtr *lib***);**

Opens the file associated with 'lib'.

Close

Closes a file.

void RLIB_Close(RLibPtr *lib***);**

Closes the file associated with 'lib'.

Chapter 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 SBUF_GetLen(SBufCPtr *sbuf***);**

Returns the length in bytes of the string contained in sbuf (not including the final zero).

See also

SBUF_GetStr, SBUF_GetSubStr

GetStr

Returns the contents of the string buffer.

CStr SBUF_GetStr(SBufPtr *sbuf***);**

Returns the string contained in the sbuf.

See also

SBUF_GetLen, SBUF_GetSubStr

GetSubStr

Return the string specified by index range.

CStr SBUF_GetSubStr(SBufPtr *sbuf*, **StrIVal** *i1*, **StrIVal** *i2***);**

Returns the substring between index1 and index2 (index1 included, index2 not included), properly terminated by a zero.

See also

SBUF_GetStr, SBUF_GetLen

Iteration

GetBwrd
GetFwrd

Return character code before of after index specified.

ChCode SBUF_GetFwrd(SBufCPtr *sbuf*, **StrIVal** *i*, **StrIValPtr** *wp***);**

ChCode SBUF_GetBwrd(SBufCPtr *sbuf*, **StrIVal** *i*, **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

SBUF_GetByte

GetByte

Byte SBUF_GetByte (SBufCPtr *sbuf*, **StrIVal** *i***);**

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

SBUF_GetFwrd, SBUF_GetBwrd

Miscellaneous Queries

CountToIndex

Convert character count to index.

StrIVal SBUF_CountToIndex (SBufCPtr *sbuf*, **StrIVal** *n***);**

Converts between a character count and the corresponding offset in bytes in the sbuf.

See also

SBUF_IndexToCount

IndextoCount

Convert byte offset to character count.

StrIVal SBUF_IndexToCount(SBufCPtr *sbuf*, **StrIVal** *i***);**

Converts between a character count and the corresponding offset in bytes index in the sbuf.

See also

SBUF_CountToIndex

Changing Contents

Set…

Replace the contents of the string buffer.

void SBUF_SetStrSub(SBufPtr *sbuf*, **CStr** *str*, **StrIVal** *slen***);**

void SBUF_SetVStr(SBufPtr *sbuf*, **VStrCPtr** *vstr***);**

void SBUF_SetSBuf(SBufPtr *sbuf*, **SBufCPtr** *sbuf2***);**

Changes the contents of the sbuf with a copy of str, vstr or sbuf2.

See also

SBUF_Clear, SBUF_Insert

Clear

Clear context of string buffer specified.

void SBUF_Clear(SBufPtr *sbuf***);**

Resets the contents of the sbuf.

See also

SBUF_Set, SBUF_Insert

Insert…

Insert character, string, variable string or string buffer at index specified.

StrIVal SBUF_InsertChar(SBufPtr *sbuf*, **StrIVal** *i1*, **ChCode** *chcode***);**

StrIVal SBUF_InsertStr(SBufPtr *sbuf*, **StrIVal** *i1*, **CStr** *str***);**

StrIVal SBUF_InsertStrSub(SBufPtr *sbuf*, **StrIVal** *i1*, **CStr** *str*, **StrIVal** *slen***);**

StrIVal SBUF_InsertVStr(SBufPtr *sbuf*, **StrIVal** *i1*, **VStrCPtr** *vstr***);**

StrIVa SBUF_InsertSBuf(SBufPtr *sbuf*, **StrIVal** *i1*, **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

SBUF_Clear, SBUF_Set, SBUF_Append

Append…

Append string, variable string or string buffer.

void SBUF_AppendChar(SBufPtr *sbuf*, **ChCode** *chcode***);**

void SBUF_AppendStr(SBufPtr *sbuf*, **CStr** *str***);**

void SBUF_AppendStrSub(SBufPtr *sbuf*, **CStr** *str*, **StrIVal** *slen***);**

void SBUF_AppendVStr(SBufPtr *sbuf*, **VStrCPtr** *vstr***);**

void SBUF_AppendSBuf(SBufPtr *sbuf*, **SBufCPtr** *sbuf2***);**

Appends str, vstr or sbuf2 to the end of the sbuf.

RemoveRange

Remove range of characters specified.

void SBUF_RemoveRange(SBufPtr *sbuf***, StrIVal** *i1***, StrIVal** *i2***);**

Removes the range of characters between i1 and i2 (i1 included, i2 not included). i1 and i2 should verify: i1 < i2.

See also

SBUF_RemoveChar

ReplaceChar

Replace string character.

StrIVal SBUF_ReplaceChar (SBufPtr *sbuf***, StrIVal** *index1***, ChCode** *ch***);**

Replaces character at index1 by ch. Returns the index after the inserted character.

See also

SBUF_RemoveChar, SBUF_RemoveRange

Truncate

Truncate string at index specified.

void SBUF_Truncate(SBufPtr *sbuf***, StrIVal** *index***);**

Truncates sbuf at index.

See also

SBUF_Clear, SBUF_Set, SBUF_Append

RemoveChar

Remove character at index specified.

void SBUF_RemoveChar (SBufPtr *sbuf***, StrIVal** *i1***);**

Removes the character at index i.

See also

SBUF_RemoveRange

Case Conversion

UpCase…

Convert string to upper case.

void SBUF_UpCase (SBufPtr *sbuf***);**

void SBUF_UpCaseSub (SBufPtr *sbuf***, StrIVal** *i1***, StrIVal** *i2***);**

These routines perform upper case conversion on the specified range of characters.

See also

SBUF_DownCase...

DownCase…

Convert string to lower case.

void SBUF_DownCase (SBufPtr *sbuf***);**

void SBUF_DownCaseSub (SBufPtr *sbuf***, StrIVal** *i1***, StrIVal** *i2***);**

These routines perform lower case conversion on the specified range of characters.

See also

SBUF_UpCase...

Matching

MatchesI…

Returns whether a string match is found with case specified.

BoolEnum SBUF_MatchesIChar (SBufPtr *sbuf***, StrIVal** *i***, ChCode** *ch***, BoolEnum** *icase***, StrIValPtr** *endp***);**

BoolEnum SBUF_MatchesIStr (SBufPtr *sbuf***, StrIVal** *i***, CStr** *str***, BoolEnum** *icase***, StrIValPtr** *endp***);**

BoolEnum SBUF_MatchesIStrSub(SBufPtr *sbuf***, StrIVal** *i***, CStr** *str***, StrIVal** *slen***, BoolEnum** *icase***, StrIValPtr** *endp***);**

BoolEnum SBUF_MatchesISBuf (SBufPtr *sbuf***, StrIVal** *i***, 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

SBUF_Matches..., SBUF_IMatches...

Matches…

Returns whether a case sensitive string match is found.

BoolEnum SBUF_MatchesChar(SBufPtr *sbuf***, StrIVal i, ChCode** *ch***, StrIValPtr** *endp***);**

BoolEnum SBUF_MatchesStr(SBufPtr *sbuf***, StrIVal i, CStr** *str***, StrIValPtr** *endp***);**

BoolEnum SBUF_MatchesStrSub(SBufPtr *sbuf***, StrIVal i, CStr** *str***, StrIVal** *slen***, StrIValPtr** *endp***);**

BoolEnum SBUF_MatchesSBuf(SBufPtr *sbuf***, StrIVal i, 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 SBUF_IMatches, with icase set to BOOL_FALSE.

See also

SBUF_MatchesI..., SBUF_IMatches...

IMatches…

Returns whether a case insensitive string match is found.

BoolEnum SBUF_IMatchesChar(SBufPtr *sbuf*, **StrIVal i, ChCode** *ch*, **StrIValPtr** *endp***);**

BoolEnum SBUF_IMatchesStr(SBufPtr *sbuf*, **StrIVal i, CStr** *str*, **StrIValPtr** *endp***);**

BoolEnum SBUF_IMatchesStrSub(SBufPtr *sbuf*, **StrIVal i, CStr** *str*, **StrIVal** *slen*, **StrIValPtr** *endp***);**

BoolEnum SBUF_IMatchesSBuf(SBufPtr *sbuf*, **StrIVal i, 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 SBUF_IMatches, with icase set to BOOL_TRUE.

See also

SBUF_MatchesI..., SBUF_Matches...

Chapter

39 Scrpt 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 hance 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
`integer`	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.
`float`	A single precision floating point number.
`double`	A double precision floating point number.
`pointer`	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.
`string`	This is a nul-termina ted 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 Scrpt 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

void SCRPT_LibInstall (void);

Installing the script library.

LibLoadInit

void SCRPT_LibLoadInit (void);

Initialization and loading the script library.

LibInit

void SCRPT_LibInit (void);

Installing and initializing the script library.

LibExit

void SCRPT_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

void SCRPT_RegisterConstants (ScrptRegisterConstPtr *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.

NDScrptRegisterConst

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

void SCRPT_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.

NDScrptRegisterEvent

Item	Description
`Name`	Name of the event, referred to .
`Code`	Notification code (ex: TBUT_NFYHIT) .
`Class`	Resource class for which the event is registered. If NULL, 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	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`

NDScrptRegisterVerb

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

void SCRPT_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

void SCRPT_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 SCRPT_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

void SCRPT_RunApp (CStr *fileName***);**

void SCRPT_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_LoadFild("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

ScrptPtr SCRPT_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

ScrptPtr SCRPT_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

ScrptPtr SCRPT_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

BoolEnum SCRPT_Execute(ScrptPtr *scrpt***);**

This function executes the compiled script `scrpt'. The function returns control to the caller when the script has finished executing. This can happen for one of three resons, summarised below:

GetReturnType

Int32 SCRPT_GetReturnType(ScrptPtr *scrpt***);**

Obtains the type of the value which was returned by execution of the script `scrpt'. The value returned is one of the predefined SCRPT_VALUEXXX constants.

QueryReturnValue

void SCRPT_QueryReturnValue(ScrptPtr *scrpt*, **ClientPtr** *value***);**

Copies the value which was returned by execution of the script `scrpt' 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 in memory into which will be written the address of the beginning of the string value.

Dispose

void SCRPT_Dispose(ScrptPtr *scrpt***);**

This function disposes the compiled bare script `scrpt', freeing all of the memory used by the compiled form.

Return type	Reason
`SCRPT_VALVOID`	The script executed a `return;' statement, or the last line of thescript 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. `return 1;'.
`N/A`	The script was terminated prematurely due to a runtime error.

Chapter

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

NDSet

void SET_Construct(SetPtr *set***);**

Default set construction.

Alloc

SetPtr SET_Alloc(void);

Returns a pointer to an allocated but not yet constructed set. The set should be constructed before being used.

NDSet

void SET_Destruct(SetPtr *set***);**

Default set destruction.

Dealloc

void SET_Dealloc(SetPtr *set***);**

Deallocates the set.

Special Shared Sets

EmptySet

SetPtr SET_EmptySet(void);

Returns a pointer to a shared empty set.
This set should not be modified.

Adding, Removing, Accessing Elements

AddElt

void SET_AddElt(SetPtr *set*, **SetEltVal** *elt***);**

Adds ’elt’ to the set (unless it is already in).

RemoveElt

void SET_RemoveElt(SetPtr *set*, **SetEltVal** *elt***);**

Removes ’elt’ from the set (unless it is not in).

AddElts

void SET_AddElts(SetPtr *set*, **SetLenVal** *n*, **SetEltValPtr** *elts***);**

Adds elts[0], elts[1], ..,elts[n-1] to the set (unless they are already in).

RemoveElts

void SET_RemoveElts(SetPtr *set*, **SetLenVal** *n*, **SetEltValPtr** *elts***);**

Removes elts[0], elts[1], ..,elts[n-1] from the set (unless they are not in the set).

GetNumElts

SetLenVal SET_GetNumElts(SetCPtr *set***);**

Returns the number of elements in set.

Reset

void SET_Reset(SetPtr *set***);**

Empties the set.

Copy

void SET_Copy(SetPtr *dst*, **SetCPtr** *src***);**

Empties ’dst’, then copies the contents of ’src’ into ’dst’.

QueryElts

void SET_QueryElts(SetCPtr *set*, **SetLenVal** *n*, **SetEltValPtr** *elts***);**

Queries the first ’n’ elements of the set and put them into ’elts’. Use SET_GetNumElts to get the number of elements in the set.

SetElts

void SET_SetElts(SetPtr *set*, **SetLenVal** *n*, **SetEltValPtr** *elts***);**

Sets the first ’n’ elements of the set to the values taken from ’elts’. Use SET_GetNumElts to get the number of elements in the set.

ContainsElt

BoolEnum SET_ContainsElt(SetCPtr *set*, **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

SetMixPartSet SET_MixGetPartSet(SetCPtr *A*, **SetCPtr** *B***);**

Compares two sets A and B and returns the set of parts which are not empty.

AreEqual

BoolEnum SET_AreEqual(SetCPtr *A*, **SetCPtr** *B***);**

Returns BOOL_TRUE if sets A and B are equal.

MixQueryParts

void SET_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).

Chapter

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.

- StrIVal 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.

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

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.

typedef NatStr C_FAR* NatStrPtr; typedef C_INVAR NatStr C_FAR* NatStrCPtr;

typedef NatCStr C_FAR* NatCStrPtr;

typedef C_INVAR NatCStr C_FAR* NatCStrCPtr;

Data type for a string pointer.

See also

Str

UniStr

Defines a UNICODE string type.

typedef UniCode C_HUGE* UniStr;

typedef C_INVAR UniCode C_HUGE* UniCStr;

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.

Str STR_NewSet(CStr *str1***);**

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

STR_Clone

Clone

Returns a new copy of a string.

Str STR_Clone(CStr *str1***);**

Definition

STR_Clone returns a new copy of a string. This function is an alias for STR_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

STR_NewSet

NewSetSub

Returns a new substring.

Str STR_NewSetSub(CStr *str1*, **StrIVal** *len)*;

STR_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

STR_SetNew

Dispose

Disposes of a string buffer.

void STR_Dispose(Str *str*);

STR_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

STR_Dispose0

Dispose0

Disposes of a string buffer if the buffer is not NULL.

void STR_Dispose0(Str *str*);

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

STR_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.

void STR_Set(StrPtr *str***, CStr** *cstr***);**

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

STR_SetSub

SetSub

Assigns a substring as the contents of an existing string.

void STR_SetSub(StrPtr *str***, CStr** *cstr***, StrIVal** *str***ival);**

STR_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

STR_Set

Append

Appends a string.

void STR_Append(StrPtr *str***, CStr** *cstr***);**

STR_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

STR_AppendSub

AppendSub

Appends a substring.

void STR_AppendSub(StrPtr *str***, CStr** *cstr***, StrIVal** *strival***);**

STR_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

STR_Append

String Length

GetLen

Returns the length of a string.

StrIVal STR_GetLen(CStr *str***);**

STR_GetLen returns the length of string in bytes. The length does not include any null terminators. Replaces STR_Len.

See also

STR_GetTruncLen

GetTruncLen

Returns the number of bytes in a string that can be copied into a buffer of a given size.

StrIVal STR_GetTruncLen(CStr *cstr***, StrIVal** *size***);**

STR_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

STR_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 STR_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 STR_GetFwrd 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 STR_GetLen only on non ASCII characters).

To iterate backwards, you have no other choice than using STR_GetBwrd 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.

ChCode STR_GetCode(CStr *str***);**

STR_GetCode returns the character code located at the beginning of string. The length of the character is not set.

See also

STR_NatGetCode

GetFwrd

Returns the character code at the beginning of a string and sets its length.

ChCode STR_GetFwrd(CStr *str***, StrIValPtr** *lenp***);**

STR_GetFwrd returns the character code at the beginning of string and sets lengthptr to the length of the character. STR_GetFwrd always sets lengthptr. STR_GetFwrd does not test whether lengthptr is NULL. When STR_GetFwrd encounters a NULL character, it sets the length pointer to one. STR_GetFwrd returns zero when the end of the string is reached.

See also

STR_NatGetFwrd, STR_GetBwrd

GetBwrd

Returns the code found in front of the specified location.

ChCode STR_GetBwrd(CStr *str***, StrIVal** *i***, StrIValPtr** *lenp***);**

STR_GetBwrd returns the character code found in front of the location in string given by index and sets lengthptr to the length of the character. STR_GetBwrd does not test whether lengthptr is zero and always sets the length pointer at the end of the operation.

See also

STR_GetFwrd

CtGetCode

Returns the character code found at the beginning of a string.

NatCode STR_CtGetCode(NatCStr *nStr***, CtCPtr** *ct***);**

STR_CtGetCode returns the native character code located at the beginning of a native string. The length of the character is not set. See STR_CtGetFwrd.

See also

STR_GetCode, STR_NatGetCode, STR_CtGetFwrd

CtGetFwrd

Returns the character code found at the beginning of a string and sets the length.

NatCode STR_CtGetFwrd(NatCStr *nStr*, **CtCPtr** *ct***, StrIValPtr** *lenp***);**

STR_CtGetFwrd returns the character code found at the beginning of the string and sets lengthptr to the length of the character. STR_CtGetFwrd does not test whether lengthptr is zero and always sets lengthptr at the end of the operation.

See also

STR_GetFwrd, STR_NatGetFwrd

CtGetBwrd

Returns the code found in front of a location in a string.

NatCode STR_CtGetBwrd(NatCStr *nStr***, CtCPtr** *ct***, StrIVal** *i*, **StrIValPtr** *lenp***);**

STR_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. STR_CtGetBwrd does not test whether lengthptr is zero and always sets lengthptr at the end of the operation. If the index is zero, STR_CtGetBwrd returns zero and sets lengthptr to zero.

See also

STR_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 occured 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 STR_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 STR_Put calls will return immediately.

If you want to check the overflow condition, you can compare len and size after every STR_Put call (or compare size with 0 after the increment / decrement operations). Then, you may reallocate the buffer and retry the STR_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.

StrIVal STR_Put(Str *buf*, **StrIVal** *size*, **CStr** *str*, **StrIValPtr** *endp***);**

STR_Put writes a string into a buffer and truncates the string if it is too large. STR_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

STR_PutAscii, STR_PutCode, STR_PutSub

PutSub

Writes a substring into a string buffer.

StrIVal STR_PutSub(Str *buf*, **StrIVal** *size*, **CStr** *s*, **StrIVal** *slen*, **StrIValPtr** *endp***);**

Writes a substring into a string buffer. After the writing the character code, STR_PutSub terminates the buffer with NULL.

See also

STR_Put

PutAscii

Writes an ASCII character into a string.

StrIVal STR_PutAscii(Str *buf*, **StrIVal** *size*, **Char** *ch***);**

STR_PutAscii writes the ASCII character into the string buffer. After the writing the character, STR_PutAscii terminates the string buffer with NULL. In debugging mode, STR_PutAscii signals a failure if the character is not an ASCII character.

Use STR_PutAscii to append a single character to a string. To write many characters sequentially, use STR_WriteAscii instead.

See also

STR_WriteAscii, STR_NatPutAscii

PutCode

Writes a character code into a string.

StrIVal STR_PutCode(Str *buf*, **StrIVal** *size*, **ChCode** *chcode***);**

STR_PutCode writes a character code into the string buffer. After the writing the character code, STR_PutCode terminates the buffer with NULL.

Use STR_PutCode to append a single character code to a string. To write many character codes sequentially, use STR_WriteCode.

See also

STR_Put, STR_WriteCode, STR_NatPutCode

WriteAscii

Writes an ASCII character into a string without terminating the string with NULL.

StrIVal STR_WriteAscii(Str *str*, **StrIVal** *size*, **Char** *ch***);**

STR_WriteAscii writes an ASCII character into a string without terminating the string with NULL. If writing the character would overflow the buffer, STR_WriteAscii writes a NULL and returns size.

STR_WriteAscii is used for writing a sequence of character codes into a string. To write a single character, use STR_PutAscii.

See also

STR_PutAscii, STR_NatWriteAscii

WriteCode

Writes a character code into a string without terminating the string with NULL.

StrIVal STR_WriteCode(Str *str***, 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, STR_WriteCode writes a NULL and returns size.

STR_WriteCode is used for writing a sequence of character codes into a string. To write a single character, use STR_PutCode.

See also

STR_PutCode , STR_NatWriteAscii

NatPutAscii

Writes an ASCII character into a native string.

StrIVal STR_NatPutAscii(NatStr *nStr***, StrIVal** *size***, NatChar** *nch***);**

Writes an ASCII character into a native string. After the writing the character, STR_NatPutAscii terminates the native string with NULL. In debugging mode, STR_NatPutAscii signals a failure if the character is not an ASCII character.

STR_NatPutAscii is useful for appending a single character to a native string. To write many characters sequentially, use STR_NatWriteAscii.

See also

STR_PutAscii, STR_NatWriteAscii

NatWriteAscii

Writes an ASCII character into a native string without terminating the string with NULL.

SStrIVal STR_NatWriteAscii(NatStr *nStr***, StrIVal** *size***, NatChar** *nch***);**

Writes an ASCII character into a native string without terminating the string with NULL. If writing the character would overflow the native buffer, STR_NatWriteAscii writes a NULL and returns size.

STR_NatWriteAscii is used for writing a sequence of characters into a native string. For writing single characters, use STR_NatPutAscii.

See also

STR_WriteAscii, STR_NatPutAscii

NatPutCode

Writes a native character code into a native string.

StrIVal STR_NatPutCode(NatStr *nStr***, StrIVal** *size***, NatCode** *ncode)***;**

Writes a native character code into a native string. After the writing the character code, STR_NatPutCode terminate s the string buffer with NULL. STR_NatPutCode is useful for appending a single native character code to a native string. To write many native character codes sequentially, use STR_NatWriteCode.

See also

STR_PutCode, STR_NatWriteCode

NatWriteCode

Writes a native character code into a native string without terminating the string with NULL.

StrIVal STR_NatWriteCode(NatStr *nStr***, StrIVal** *size***, NatCode** *ncode)***;**

STR_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, STR_NatWriteCode writes a NULL and returns size.

STR_NatWriteCode is used for writing a sequence of character codes into a string. For writing single characters, use STR_NatPutCode.

See also

STR_WriteCode, STR_PutCode

Basic String Comparisons

Cmp
ICmp

Compares two strings.

CmpEnum STR_ICmp(CStr *s1***, CStr** *s2***);**

CmpEnum STR_Cmp(CStr *s1***, CStr** *s2***);**

STR_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, STR_Cmp returns CMP_UNDER; if they are equal, it returns CMP_EQUAL; and if string1 is greater than string2, STR_Cmp returns CMP_OVER.

STR_ICmp is exactly the same, but it ignores case differences in the ASCII range.

See also

STR_CmpSub

CmpSub
ICmpSub

Compares two substrings.

CmpEnum STR_CmpSub(CStr *s1***, StrIVal** *l1***, CStr** *s2***, StrIVal** *l2***);**

CmpEnum STR_ICmpSub(CStr *s1***, StrIVal** *l1***, CStr** *c2***, StrIVal** *l2l)***;**

STR_CmpSub compares two substrings and returns a CmpEnum as a result. See STR_Cmp.

STR_ICmpSub is exactly the same, but it ignores case differences in the ASCII range.

See also

STR_Cmp

Equals
IEquals

Compares two strings for equality.

BoolEnum STR_Equals(CStr *s1***, CStr** *s2***);**

BoolEnum STR_IEquals(CStr *s1***, CStr** *s2***);**

STR_Equals returns a boolean indicating whether or not the two strings are equal.

STR_IEquals performs the same function but ignores case differences in the ASCII range.

See also

STR_Cmp, STR_EqualsSub

EqualsSub
IEqualsSub

Compares two strings for equality.

BoolEnum STR_EqualsSub(CStr *s1***, StrIVal** *l1***, CStr c2, StrIVal** *l2***);**

BoolEnum STR_IEqualsSub(CStr *s1***, StrIVal** *l1***, CStr c2, StrIVal** *l2***);**

STR_EqualsSub returns a Boolean value indicating whether the two substrings are equal.

STR_IEqualsSub performs the same function but ignores case differences in the ASCII range.

See also

STR_Cmp, STR_Equals

Testing Matches

MatchesChar

Tests whether a string matches a character.

BoolEnum STR_MatchesChar(CStr *s1***, ChCode** *chcode***, StrIValPtr** *lenp***);**

STR_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.

BoolEnum STR_Matches(CStr *s1*, **ChCode** *chcode*, **StrIValPtr** *lenp***);**

BoolEnum STR_IMatches(CStr *s1*, **CStr** *s2*, **StrIValPtr** *lenp***);**

STR_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.

STR_IMatches is the same as STR_Matches but ignores case differences in the ASCII range only.

MatchesPat
IMatchesPat

Tests whether one string matches another string containing a pattern.

BoolEnum STR_MatchesPat(CStr *s1*, **CStr** *s2*, **StrIValPtr** *lenp***);**

BoolEnum STR_IMatchesPat(CStr *s1*, **CStr** *s2*, **StrIValPtr** *lenp***);**

STR_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 be set to the end of the match.

pattern contains a very simple pattern which can accept an question mark (?) to indicate an optional character and an asterisk (*) to indicate any substring. Regular expressions are not supported.

STR_IMatchesPat is the same as STR_MatchesPat but it ignores case differences in the ASCII range only.

See also

STR_MatchesPatSub

MatchesPatSub
IMatchesPatSub

Tests whether one substring matches another substring which contains a pattern.

BoolEnum STR_MatchesPatSub(CStr *s1*, **StrIVal** *l1*, **CStr** *s2*, **StrIVal** *l2*, **StrIValPtr** *lenp***);**

BoolEnum STR_IMatchesPatSub(CStr *s1*, **StrIVal** *l1*, **CStr** *s2*, **StrIVal** *l2*, **StrIValPtr** *lenp***);**

STR_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.

STR_IMatchesPatSub is the same as STR_MatchesPatSub but it ignores case differences in the ASCII range only.

See also

STR_MatchesPat

MatchesSub

IMatchesSub

Tests whether two substrings match.

BBoolEnum STR_MatchesSub(CStr *s1*, **StrIVal** *l1*, **CStr** *s2*, **StrIVal** *l2*, **StrIValPtr** *lenp*);

BoolEnum STR_IMatchesSub(CStr *s1*, **StrIVal** *l1*, **CStr** *s2*, **StrIVal** *l2*, **StrIValPtr** *lenp*);

STR_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.

STR_IMatchesSub ignores case differences in the ASCII range only.

See also

STR_Matches

Searching

FindFirst
IFindFirst

Finds the first occurrence of a string.

StrIVal STR_FindFirst(CStr *s1*, **CStr** *s2*);

StrIVal STR_IFindFirst(CStr *s1*, **CStr** *s2*);

STR_FindFirst finds the first occurrence of string2 within string1 and returns its index. Returns -1 if the search fails.

STR_IFindFirst performs the same function but ignores case differences in the ASCII range only.

See also

STR_FindLast

FindFirstChar

Finds the first occurrence of a character.

StrIVal STR_FindFirstChar(CStr *s1*, **ChCode** *chcode*);

STR_FindFirstChar finds the first occurrence of chcode within string and returns the index. Returns -1 if the search fails.

See also

STR_FindLastChar

FindFirstCharSub

Find the first occurrence of a character in a substring.

StrIVal STR_FindFirstCharSub(CStr *s1*, **StrIVal** *l1*, **ChCode** *chcode*);

STR_FindFirstCharSub finds the first occurrence of chcode in a substring and returns its index. Returns -1 if the search fails.

See also

STR_FindFirstChar, STR_FindLastCharSub

FindFirstSub
IFindFirstSub

Switchable case-independent search for the first occurrence of a substring.

StrIVal STR_FindFirstSub(CStr *s1***, StrIVal** *l1***, CStr** *s2***, StrIVal** *l2***);**

StrIVal STR_IFindFirstSub(CStr *s1***, StrIVal** *l1***, CStr** *s2***, StrIVal** *l2***);**

:STR_FindFirstSub finds the first occurrence of a substring within a substring and returns the index. Returns -1 if the search fails.

STR_IFindFirstSub performs the same function but ignores case differences in the ASCII range only.

See also

STR_FindFirst, STR_FindLastSub

FindIFirst

Switchable case-independent search for the first occurrence of a string.

StrIVal STR_FindIFirst(CStr *s1***, CStr** *s2***, BoolEnum** *caseI***);**

STR_FindFirst 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

STR_FindILast

FindIFirstSub

Switchable case-independent search for the first occurrence of a substring.

StrIVal STR_FindIFirstSub(CStr *s1***, StrIVal** *l1***, CStr** *s2***, StrIVal** *l2***, BoolEnum** *caseI***);**

STR_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

STR_FindIFirst, STR_FindILastSub

FindILast

Switchable case-independent search for the last occurrence of a string.

StrIVal STR_FindILast(CStr *s1***, CStr** *s2***, BoolEnum** *caseI***);**

STR_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

STR_FindIFirst

FindLast
IFindLast

Finds the last occurrence of a string.

StrIVal STR_FindLast(CStr *s1*, **CStr** *s2***);**

StrIVal STR_IFindLast(CStr *s1*, **CStr** *s2***);**

STR_FindLast finds the last occurrence of string2 within string1 and returns the index. Returns -1 if the search fails.

STR_IFindLast performs the same function but ignores case differences in the ASCII range only.

See also

STR_FindFirst

FindLastChar

Finds the last occurrence of a character within a string

StrIVal STR_FindLastChar(CStr *s1*, **ChCode** *chcode***);**

STR_FindLastChar finds the last occurrence of a character in string and returns its index. Returns -1 if the search fails.

See also

STR_FindFirstChart

FindLastCharSub

Finds the last occurrence of a character in a substring.

StrIVal STR_FindLastCharSub(CStr *s1*, **StrIVal** *l1*, **ChCode** *chcode***);**

STR_FindLastCharSub finds the last occurrence of chcode in a substring and returns its index. Returns -1 if the search fails.

See also

STR_FindLastChar, STR_FindFirstCharSub

FindLastSub
FindILastSub

Switchable case-independent search for the last occurrence of a substring.

StrIVal STR_FindLastSub(CStr *s1*, **StrIVal** *l1*, **CStr** *s2*, **StrIVal** *l2***);**

StrIVal STR_FindILastSub(CStr *s1*, **StrIVal** *l1*, **CStr** *s2*, **StrIVal** *l2*, **BoolEnum** *casei***);**

STR_FindILastSub finds the last occurrence of a substring given by subptr2 within a substring given by subptr1, 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.

STR_FindLastSub finds the last occurrence of a substring given by subptr2 within a substring, given by subptr1.

STR_IFindLastSub ignores case differences in the ASCII range only.

See also

STR_FindLast, STR_FindFirstSub

Scanning of Numeric Values

GetDec...

Returns the integer value found at the beginning of a decimal integer string.

Int STR_GetDecInt(CStr s, StrIValPtr *lenp***);**

Int16 STR_GetDecInt16(CStr s, StrIValPtr *lenp***);**

Int32 STR_GetDecInt32(CStr s, StrIValPtr *lenp***);**

UInt STR_GetDecUInt(CStr s, StrIValPtr *lenp***);**

UInt16 STR_GetDecUInt16(CStr s, StrIValPtr *lenp***);**

UInt32 STR_GetDecUInt32(CStr s, StrIValPtr *lenp***);**

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

STR_SubGetDec..., STR_GetHex...,STR_GetRadix..., STR_GetDouble

GetHex...

Returns the integer value found at the beginning of a hexadecimal integer string.

Int STR_GetHexInt (CStr s, StrIValPtr *lenp***);**

Int16 STR_GetHexInt16 (CStr s, StrIValPtr *lenp***);**

Int32 STR_GetHexInt32 (CStr s, StrIValPtr *lenp***);**

UInt16 STR_GetHexUInt16 (CStr s, StrIValPtr *lenp***);**

UInt32 STR_GetHexUInt32 (CStr *s,* **StrIValPtr** *lenp***);**

UInt STR_GetHexUInt(CStr s, StrIValPtr *lenp***);**

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

STR_SubGetHex..., STR_GetDec..., STR_GetRadix..., STR_GetDouble

GetRadix...

Returns the integer value found at the beginning of an integer string.

Int STR_GetRadixInt(CStr *s*, **Int** *i*, **StrIValPtr** *lenp***);**

Int16 STR_GetRadixInt16(CStr *s*, **Int** *radix*, **StrIValPtr** *lenp***);**

Int32 STR_GetRadixInt32(CStr *s*, **Int** *radix*, **StrIValPtr** *lenp***);**

Int STR_GetRadixInt(CStr s, Int *i*, **StrIValPtr** *lenp***);**

UInt STR_GetRadixUInt(CStr s, Int *radix*, **StrIValPtr** *lenp***);**

UInt16 STR_GetRadixUInt16(CStr *s*, **Int** *radix*, **StrIValPtr** *lenp***);**

UInt32 STR_GetRadixUInt32(CStr s, Int *radix*, **StrIValPtr** *lenp***);/**

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

STR_GetDec..., STR_GetHex..., STR_GetDouble

GetDouble

Returns the double real numeric value found at the beginning of a double real string.

Double STR_GetDouble(CStr *string*, **StrIValPtr** *endptr***);**

These functions return the double real value found at the beginning of a double real integer string 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

STR_SubGetDouble..., STR_GetDec..., STR_GetHex..., STR_GetRadix....

SubGetDec...

Returns the integer value found at the beginning of a decimal integer substring.

Int STR_SubGetDecInt(CStr *s,* **StrIVal l, StrIValPtr** *lenp***);**

Int16 STR_SubGetDecInt16(CStr *s,* **StrIVal** *l***, StrIValPtr** *lenp***);**

Int32 STR_SubGetDecInt32(CStr *s***, StrIVal** *l,* **StrIValPtr** *lenp***);**

UInt STR_SubGetDecUInt(CStr s, StrIVal *l***, StrIValPtr** *lenp***);**

UInt16 STR_SubGetDecUInt16(CStr *s***, StrIVal** *l,* **StrIValPtr** *lenp***);**

UInt32 STR_SubGetHexUInt32(CStr *s***, StrIVal** *l***, StrIValPtr** *lenp***);**

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

STR_GetDec..., STR_SubGetHex..., STR_SubGetRadix..., STR_SubGetDouble

SubGetHex...

Returns the integer value found at the beginning of a hexadecimal integer substring.

Int STR_SubGetHexInt(CStr *s***, StrIVal** *l***, StrIValPtr** *lenp)***;**

Int16 STR_SubGetHexInt16(CStr *s***, StrIVal** *l,* **StrIValPtr** *lenp***);**

Int32 STR_SubGetHexInt32(CSt*r s***, StrIVal** *l***, StrIValPtr** *lenp***);**

UInt STR_SubGetHexUInt(CSt*r s***, StrIVal** *l***, StrIValPtr** *lenp***);**

UInt16 STR_SubGetHexUInt16(CStr *s***, StrIVal** *l***, StrIValPtr** *lenp)***;**

UInt32 STR_SubGetHexUInt32(CStr *s***, StrIVal** *l,* **StrIValPtr** *lenp***);**

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

STR_GetHex..., STR_SubGetDec..., STR_SubGetRadix..., STR_SubGetDouble

SubGetRadix...

Returns the integer value found at the *beg*inning of an integer substring.

Int STR_SubGetRadixInt(CStr *s*, **StrIVal** *l*, **Int** *radix*, **StrIValPtr** *lenp***);**

Int16 STR_SubGetRadixInt16(CStr *s*, **StrIVal** *l*, **Int** *radix*, **StrIValPtr** *lenp***);**

Int32 STR_SubGetRadixInt32(CStr s, StrIVal *l*, **Int** *radix*, **StrIValPtr** *lenp***);**

UInt STR_SubGetRadixUInt(CStr s, StrIVal l, Int *radix*, **StrIValPtr** *lenp***);**

UInt16 STR_SubGetRadixUInt16(CStr s, StrIVal l, Int *radix*, **StrIValPtr** *lenp***);**

Int32 STR_SubGetRadixInt32(CStr s, StrIVal *l*, **Int** *radix*, **StrIValPtr** *lenp***);**

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

STR_GetRadix..., STR_SubGetDec..., STR_SubGetHex..., STR_SubGetDouble

SubGetDouble

Returns the double real numeric value found at the beginning of a double real substring.

Double STR_SubGetDouble(CStr *s*, **StrIVal** *l*, **StrIValPtr** *lenp***);**

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

STR_GetDouble, STR_SubGetHex.., STR_SubGetRadix..., STR_SubGetDec

Formating the Numeric Values

PutDec...

Converts a decimal integer into its textual representation in a string buffer.

SStrIVal STR_PutDecInt(Str *buf*, **StrIVal** *size*, **Int** *i***);**

StrIVal STR_PutDecInt16(Str *buf*, **StrIVal** *size*, **Int16** *i***);**

StrIVal STR_PutDecInt32(Str *buf*, **StrIVal** *size*, **Int32** *i);*

StrIVal STR_PutDecUInt(Str *buf*, **StrIVal** *size*, **UInt** *i***);**

StrIVal STR_PutDecUInt16(Str *buf*, **StrIVal** *size*, **UInt16** *i***);**

StrIVal STR_PutDecUInt32(Str *buf*, **StrIVal** *size*, **UInt32** *i***);**

The STR_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 STR_PutDecU... functions convert unsigned decimal integers.

See also

STR_Put, STR_PutHex..., STR_PutRadix..., STR_PutDouble

PutHex....

Converts a hexadecimal integer into its textual representation in a string buffer.

StrIVal STR_PutHexInt(Str *buf*, **StrIVal** *size*, **Int** *i***);**

StrIVal STR_PutHexInt16(Str *buf*, **StrIVal** *size*, **Int16** *i***);**

StrIVal STR_PutHexUInt32(Str *buf*, **StrIVal** *size*, **UInt32** *i***);**

StrIVal STR_PutHexUInt(Str *buf*, **StrIVal** *size*, **UInt** *i***);**

StrIVal STR_PutHexUInt16(Str *buf*, **StrIVal** *size*, **UInt16** *i***);**

StrIVal STR_PutHexUInt32(Str *buf*, **StrIVal** *size*, **UInt32** *i***);**

Converts a hexadecimal integer into its textual representation in a string buffer. The STR_PutHexU... functions convert unsigned hexadecimal integers. These functions convert 8-bit, 16-bit, and 32-bit hexadecimal integers.

See also

STR_Put, STR_PutDec..., STR_PutRadix..., STR_PutDouble

PutRadix...

Using the radix, converts an integer into its textual representation in the string buffer.

StrIVal STR_PutRadixInt(Str *buf*, **StrIVal** *size*, **Int** *radix*, **Int** *i***);**

StrIVal STR_PutRadixInt16(Str *buf*, **StrIVal** *size*, **Int** *radix*, **Int16** *i***);**

StrIVal STR_PutRadixInt32(Str *buf*, **StrIVal** *size*, **Int** *radix*, **Int32** *i***);**

StrIVal STR_PutRadixUInt(Str *buf*, **StrIVal** *size*, **Int** *radix*, **UInt** *i***);**

StrIVal STR_PutRadixUInt16(Str *buf*, **StrIVal** *size*, **Int** *radix*, **UInt16** *i***);**

StrIVal STR_PutRadixUInt32(Str *buf*, **StrIVal** *size*, **Int** *radix*, **UInt32** *i***);**

Using the radix, converts an integer into its textual representation in the string buffer. For example, STR_PutRadixInt converts 100 to "64" if the radix is 16.

The STR_PutRadixU... functions convert unsigned hexadecimal integers. These functions convert 8-bit, 16-bit, and 32-bit integers.

See also

STR_Put, STR_PutDec..., STR_PutHex..., STR_PutDouble

PutDouble

Converts a double precision value into its textual representation in the string buffer.

StrIVal STR_PutDouble(Str *buf*, **StrIVal** *size*, **Double** *d***);**

STR_PutDouble converts a double precision value into its textual representation in the string buffer.

See also

STR_Put, STR_PutDec..., STR_PutHex...

Basic Conversions

AsciiUpCase

Converts a string to upper case.

void STR_AsciiUpCase(Str *s***);**

STR_AsciiUpCase converts the ASCII characters in the string to upper case. Non-ASCII characters are ignored. string is converted in place.

See also

STR_AsciiUpCaseSub, STR_AsciiDownCase, STR_AsciiDownCaseSub

AsciiUpCaseSub

Converts a substring to upper case.

void STR_AsciiUpCaseSub(Str *s*, **StrIVal** *l***);**

STR_AsciiUpCaseSub converts the ASCII characters in the substring to upper case. Non-Ascii characters are ignored. The substring is converted in place.

See also

STR_AsciiUpCase, STR_AsciiDownCase, STR_AsciiDownCaseSub

AsciiDownCase

Converts a string to lower case.

void STR_AsciiDownCase(Str *s***);**

STR_AsciiDownCase converts the ASCII characters in the string to lower case. Non-ASCII characters are ignored. string is converted in place.

See also

STR_AsciiUpCase, STR_AsciiDownCaseSub

AsciiDownCaseSub

Converts a substring to lower case.

void STR_AsciiDownCaseSub(Str *s*, **StrIVal** *l***);**

STR_AsciiDownCaseSub converts the ASCII characters in the substring to lower case. Non-ASCII characters are ignored. The substring is converted in place.

See also

STR_AsciiUpCase, STR_AsciiDownCase, STR_AsciiDownCaseSub

PutAsciiUpper

Same as STR_Put, but also converts the ASCII characters in the string to upper case.

StrIVal STR_PutAsciiUpper(Str *buf***, StrIVal** *len***, CStr** *str***, StrIValPtr** *lenp***);**

Same as STR_Put, but also converts the ASCII characters in the string to upper case. The string is converted in place. STR_PutAsciiUpper permits ASCII conversion.

See also

STR_PutAsciiLower, STR_PutAsciiUpperSub

PutAsciiLower

Same as STR_Put, but also converts the ASCII characters in the string to lower case.

StrIVal STR_PutAsciiLower(Str *buf***, StrIVal** *len***, CStr** *str***, StrIValPtr** *lenp***);**

Same as STR_Put, but also converts the ASCII characters in the string to lower case. The string is converted in place. STR_PutAsciiLower permits ASCII conversion or simple code conversion only.

See also

STR_Put, STR_PutAsciiUpper, STR_PutAsciiLowerSub

PutAsciiUpperSub

Same as STR_PutSub, but also converts the ASCII characters in the substring to upper case.

StrIVal STR_PutAsciiUpperSub(Str *buf***, StrIVal** *len***, CStr** *str***, StrIVal** *slen***, StrIValPtr** *lenp***);**

Same as STR_PutSub, but also converts the ASCII characters in the substring to upper case. The substring is converted in place. STR_PutAsciiUpperSub permits ASCII conversion.

See also

STR_PutSub, STR_PutAsciiUpperSub

PutAsciiLowerSub

Same as STR_PutSub, but also converts a substring to lower case.

StrIVal STR_PutAsciiLowerSub(Str *buf***, StrIVal** *len***, CStr** *str***, StrIVal** *slen***, StrIValPtr** *lenp***);**

Same as STR_PutSub, but also converts the ASCII characters in the substring to lower case. The substring is converted in place. STR_PutAsciiLowerSub permits ASCII conversion

See also

STR_PutSub, STR_PutAsciiUpperSub

Loading from Resources

ResLoad
ResLoadNth

Returns a string from a StrR or StrL resource.

CStr STR_ResLoad(CStr *mod*, **CStr** *res*, **StrIValPtr** *lenp***);**

CStr STR_ResLoadNth(CStr *mod*, **CStr** *res*, **ArrayIVal** *n*, **StrIValPtr** *lenp***);**

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.

STR_ResLoadNth returns the nth string in the resource.

See also

STR_ResFind

ResFind
ResFindNth

Finds and returns a string from a StrR or StrL resource.

CStr STR_ResFind(CStr *mod*, **CStr** *res*, **StrIValPtr** *lenp***);**

CStr STR_ResFindNth(CStr *mod*, **CStr** *res*, **ArrayIVal** *n*, **StrIValPtr** *lenp***);**

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.

STR_ResLoadNth finds and returns the nth string in the resource.

See also

STR_ResLoad.

Conversions Between Code Types

FromCt

StrIVal STR_FromCt(Str *buf*, **StrIVal** *size*, **NatCStr** *ct***str, CtCPtr** *ct*, **StrCvtCtxPtr** *ctx***);**

StrIVal STR_FromCtSub(Str *buf*, **StrIVal** *size*, **NatCStr** *ctbuf*, **StrIVal** *ctslen*, **CtCPtr** *ct*, **StrCvtCtxPtr** *ctx***);**

ToCt

StrIVal STR_ToCt (NatStr *ctbuf*, **StrIVal** *size*, **CStr** *str*, **CtCPtr** *ct*, **StrCvtCtxPtr** *ctx***);**

StrIVal STR_ToCtSub(NatStr *ctbuf*, **StrIVal** *size*, **CStr** *str*, **StrIVal** *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.

STR_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

StrIVal STR_FromUni (Str *buf*, **StrIVal** *size*, **UniCStr** *unistr*, **CharCvtSet** *flags*, **StrCvtCtxPtr** *ctx*);

StrIVal STR_FromUniSub(Str *buf*, **StrIVal** *size*, **UniCStr** *unistr*, **StrIVal** *unilen*, **CharCvtSet** *flags*, **StrCvtCtxPtr** *ctx*);

ToUni

StrIVal STR_ToUni (UniStr *unistr*, **StrIVal** *unisize*, **CStr** *str*, **CharCvtSet** *flags*, **StrCvtCtxPtr** *ctx*);

StrIVal STR_ToUniSub(UniStr *unistr*, **StrIVal** *unisize*, **CStr** *str*, **StrIVal** *slen*, **CharCvtSet** *flags*, **StrCvtCtxPtr** *ctx*);

Routines to convert between internal string (Str) and UNICODE (UniStr). The `unisize’ 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.

Chapter

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.

RClasPtr STRL_Class (void);

STRL_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 STRL_GetLen(StrLCPtr *stringList***);**

STRL_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 STRL_GetNthStr (StrLCPtr *stringList*, **ArrayIVal** *stringIndex***);**

STRL_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

STRL_SetNthStr

SetNthStr

Replaces an existing string in a string list resource with a new string.

void STRL_SetNthStr(StrLPtr *stringList*, **ArrayIVal** *stringIndex*, **CStr** *string***);**

STRL_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

STRL_GetNthStr, STRL_AddStr, STRL_AddStrAtIndex

AddStr

Adds a new string to a string list resource.

void STRL_AddStr(StrLPtr *stringList*, **CStr** *string***);**

STRL_AddStr adds string to the string list resource specified by strlist. The new string appears last in the list of strings.

See also

STRL_AddStrAtIndex, STRL_RemoveIndex, STRL_SetNthStr

AddStrAtIndex

Inserts a new string in a string list resource at the position specified.

void STRL_AddStrAtIndex(StrLPtr *stringList*, **ArrayIVal** *stringIndex*, **CStr** *string***);**

STRL_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

STRL_AddStr, STRL_RemoveIndex, STRL_SetNthStr

RemoveIndex

Removes a string from a string list resource at the position specified.

void STRL_RemoveIndex(StrLPtr *stringList*, **ArrayIVal** *stringIndex***);**

STRL_RemoveIndex removes a string from the string list resource specified by strlist. Index specifies which string to remove from the list.

See also

STRL_AddStr, STRL_AddStrAtIndex, STRL_SetNthStr

LoadNthStr

Returns a string from a string list resource by resource name and index.

CStr STRL_LoadNthStr(CStr *mod***, CStr** *res***, ArrayIVal** *stringIndex***);**

STRL_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

STRL_LOADNTHSTR, STRL_FindNthStr

FindNthStr

Returns the string of a string list resource.

CStr STRL_FindNthStr(CStr *mod***, CStr** *res***, ArrayIVal** *stringIndex***);**

STRL_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

STRL_FindNthStr, STRL_LoadNthStr

Chapter

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.

RClasPtr STRR_Class(void);

STRR_Class returns a pointer to a string resource class data structure.

Loading a String Resource

LoadStr

Loads the string resource by resource name.

CStr STRR_LoadStr (CStr *modname*, **CStr** *resname***);**

STRR_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.

CStr STRR_FindStr (CStr *modname*, **CStr** *resname***);**

STRR_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 STRR_GetStr (StrRCPtr *stringRes***);**

STRR_GetStr returns the string from the string resource specified by strres.

SetStr

Replaces the string in a string resource with a new string.

void STRR_SetStr (StrRPtr *stringRes*, **CStr** *string***);**

STRR_SetStr substitutes the string in the string resource specified by strres with a new string.

Accessing the Id

GetId

Returns the id of a string resource.

StrRIdVal STRR_GetId (StrRCPtr *stringRes***);**

STRR_GetStr returns the Id contained in the string resource.

SetId

Changes the id value of a string resource.

void STRR_SetId (StrRPtr *stringRes*, **StrRIdVal** *id***);**

STRR_SetId changes the Id contained in the string resource.

Chapter

44 Var Class

This file defines the variant data type.

Type System

VarTypeEnum

Methods	Description
VAR_TYPE_NONE	No type. Used for initializing type tags. Equivalent to void.
VAR_TYPE_UNSUPPORTED	Unsupported type. Used when there is no OA type which matches a particular native type.
VAR_TYPE_VOID_PTR	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	Description
VAR_TYPE_NAT_STR	An array of char (regardless of code set & of the encoding system)
VAR_TYPE_NAT_WSTR	An array of wchar_t (to be defined)

“Power” Types

Methods	Description
VAR_TYPE_VAR	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.
VAR_TYPE_OAOBJ	Reference to an Object Access object
VAR_TYPE_NULL	The variant contains a NULL value
VAR_TYPE_NULLOBJ	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)

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 VAR_Convert(VarRef *var***, VarTypeEnum** *destType***);**

BoolEnum VAR_TryConvert(VarRef *var***, VarTypeEnum** *destType***);**

Converts the type of this any to the type specified by `destType'.

ConvertToValue
TryConvertToValue

void VAR_ConvertToValue(VarRef *var***);**

BoolEnum VAR_TryConvertToValue(VarRef *var***);**

If this any contains a reference the method converts this any to a value obtained by dereferencing the reference.

CopyToType
TryCopyToType

VarPtr VAR_CopyToType(VarCRef *var***, VarTypeEnum** *destType***);**

BoolEnum VAR_TryCopyToType(VarCRef *var***, VarTypeEnum** *destType***,**
VarPtrPtr *valuePtr***);**

Returns an NDVar which contains this any converted to the type specified by `destType'.

CopyToValue
TryCopyToValue

VarPtr VAR_CopyToValue(VarCRef *var***);**

BoolEnum VAR_TryCopyToValue(VarCRef *var***, VarPtr** *valuePtr***);**

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 VAR_CopyToInt(VarCRef *var***);**

BoolEnum VAR_TryCopyToInt(VarCRef *var***, IntPtr** *valuePtr***);**

Int8 VAR_CopyToInt8(VarCRef *var***);**

BoolEnum VAR_TryCopyToInt8(VarCRef *var***, Int8Ptr** *valuePtr***);**

Int16 VAR_CopyToInt16(VarCRef *var***);**

BoolEnum VAR_TryCopyToInt16(VarCRef *var***, Int16Ptr** *valuePtr***);**

Int32 VAR_CopyToInt32(VarCRef *var***);**

BoolEnum VAR_TryCopyToInt32(VarCRef *var***, Int32Ptr** *valuePtr***);**

Int64 VAR_CopyToInt64(VarCRef *var***);**

BoolEnum VAR_TryCopyToInt64(VarCRef *var***, 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 VAR_CopyToUInt(VarCRef *var***);**

BoolEnum VAR_TryCopyToUInt(VarCRef *var***, UIntPtr** *valuePtr***);**

UInt8 VAR_CopyToUInt8(VarCRef *var***);**

BoolEnum VAR_TryCopyToUInt8(VarCRef *var***, UInt8Ptr** *valuePtr***);**

UInt16 VAR_CopyToUInt16(VarCRef *var***);**

BoolEnum VAR_TryCopyToUInt16(VarCRef *var***, UInt16Ptr** *valuePtr***);**

UInt32 VAR_CopyToUInt32(VarCRef *var***);**

BoolEnum VAR_TryCopyToUInt32(VarCRef *var***, UInt32Ptr** *valuePtr***);**

UInt64 VAR_CopyToUInt64(VarCRef *var***);**

BoolEnum VAR_TryCopyToUInt64(VarCRef *var***, 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 VAR_CopyToLong(VarCRef *var***);**

BoolEnum VAR_TryCopyToLong(VarCRef *var***, LongPtr** *valuePtr***);**

ULong VAR_CopyToULong(VarCRef *var***);**

BoolEnum VAR_TryCopyToULong(VarCRef *var***, ULongPtr** *valuePtr***);**

CopyToFloat
TryCopyToFloat

Float VAR_CopyToFloat(VarCRef *var***);**

BoolEnum VAR_TryCopyToFloat(VarCRef *var***, FloatPtr** *valuePtr***);**

CopyToDouble
TryCopyToDouble

Double VAR_CopyToDouble(VarCRef *var***);**

BoolEnum VAR_TryCopyToDouble(VarCRef *var***, DoublePtr** *valuePtr***);**

CopyToChar
TryCopyToChar

Char VAR_CopyToChar(VarCRef *var***);**

BoolEnum VAR_TryCopyToChar(VarCRef *var***, CharPtr** *valuePtr***);**

CopyToVARWChar
TryCopyToVARWChar

VARWChar VAR_CopyToVARWChar(VarCRef *var***);**

BoolEnum VAR_TryCopyToVARWChar(VarCRef *var***, VARWCharPtr** *valuePtr***);**

CopyToBoolean
TryCopyToBoolean

BoolEnum VAR_CopyToBoolean(VarCRef *var***);**

BoolEnum VAR_TryCopyToBoolean(VarCRef *var***, BoolEnumPtr** *valuePtr***);**

CopyToByte
TryCopyToByte

Byte VAR_CopyToByte(VarCRef *var***);**

BoolEnum VAR_TryCopyToByte(VarCRef *var***, BytePtr** *valuePtr***);**

CopyToChCode
TryCopyToChCode

ChCode VAR_CopyToChCode(VarCRef *var***);**

BoolEnum VAR_TryCopyToChCode(VarCRef *var***, ChCodePtr** *valuePtr***);**

CopyToStr
TryCopyToStr

Str VAR_CopyToStr(VarCRef *var***);**

BoolEnum VAR_TryCopyToStr(VarCRef *var***, StrPtr** *valuePtr***);**

Caller must delete the returned Str.

CopyToVARWStr
TryCopyToVARWStr

VARWStr VAR_CopyToVARWStr(VarCRef *var***);**

BoolEnum VAR_TryCopyToVARWStr(VarCRef *var***, VARWStrPtr** *valuePtr***);**

Caller must delete the returned VARWStr.

CopyToClientPtr
TryCopyToClientPtr

ClientPtr VAR_CopyToClientPtr(VarCRef *var***);**

BoolEnum VAR_TryCopyToClientPtr(VarCRef *var***, ClientPtrPtr** *valuePtr***);**

Caller must NOT delete the returned ClientPtr..

Update
TryUpdate

void VAR_Update(VarRef *var*, **VarCRef** *ori***);**

BoolEnum VAR_TryUpdate(VarRef *var***, VarCRef** *ori***);**

Information Methods

Clear

void VAR_Clear(VarRef *var***);**

Empties this variant. After this call the variant will be of type VAR_TYPE_NONE.

GetType

VarTypeEnum VAR_GetType(VarCRef *var***);**

Returns the type contained in this variant.

ContainsRef

BoolEnum VAR_ContainsRef(VarCRef *var***);**

Returns BOOL_TRUE if this variant contains a reference. Returns BOOL_FALSE if this variant contains a value.

IsEmpty

BoolEnum VAR_IsEmpty(VarCRef *var***);**

Returns BOOL_TRUE if the type of this variant is VAR_TYPE_NONE. Returns BOOL_FALSE otherwise.

IsNULL

BoolEnum VAR_IsNULL(VarCRef *var***);**

Returns BOOL_TRUE if the type of this variant is VAR_TYPE_NULL. Returns BOOL_FALSE otherwise.

IsNULLObj

BoolEnum VAR_IsNULLObj(VarCRef *var***);**

Returns BOOL_TRUE if the type of this variant is VAR_TYPE_NULLOBJ. Returns BOOL_FALSE otherwise.

Chapter

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

RClasPtr VARDS_Class(void);

Returns a pointer to the variant list data source resource class.

QueryValue

void VARDS_QueryValue(VarDsPtr *varDs*, **VarPtr** *var***);**

Returns the value associated to the variant data source in `var'.

GetValue

VarPtr VARDS_GetValue(VarDsPtr *varDs***);**

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 VARDS_SetValue(VarDsPtr *varDs*, **VarPtr** *var***);**

Sets the value of the data source by internally creating an edition object and committing the changes created through it.

void VARDSEDIT_SetValue(VarDsEditPtr *edit*, **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 VARDS_DefNfy(VarDsPtr *varDs*, **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);*

Chapter

46 *VarGr*

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

RClasPtr VARGR_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 VARGR_GetTitle(VarGrPtr *varGr***);**

SetTitle

BoolEnum VARGR_SetTitle(VarGrPtr *varGr***, 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 VARGREDIT_SetTitle(VarGrEditPtr *edit***, 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 VARGR_GetNumNodes(VarGrPtr *varGr***);**

Returns the number of nodes in the graph.

GetNumRootNodes

VarGrIndexVal VARGR_GetNumRootNodes(VarGrPtr *varGr***);**

Returns the number of root nodes in the graph. A *root node* is any node with no parent.

GetNumEdges

VarGrIndexVal VARGR_GetNumEdges(VarGrPtr *varGr***);**

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.

Create

VarGrNodeAccessorPtr VARGRNODEACCESSOR_Create(void);

Allocates and constructs a node accessor for navigation through nodes in a graph datasource.

Alloc

VarGrNodeAccessorPtr VARGRNODEACCESSOR_Alloc(void);

Returns a pointer to an allocated, but not yet constructed, node accessor. The node accessor should be constructed before being used.

Construct

void VARGRNODEACCESSOR_Construct(
VarGrNodeAccessorPtr *node***);**

Default node-accessor construction.

ConstructCopy

void VARGRNODEACCESSOR_ConstructCopy(
VarGrNodeAccessorPtr *node***, VarGrNodeAccessorCPtr** *cnode***);**

Constructs the node accessor with the information obtained from “cnode.”

Destruct

void VARGRNODEACCESSOR_Destruct(VarGrNodeAccessorPtr *node***);**

Default node-accessor destruction.

Dealloc

void VARGRNODEACCESSOR_Dealloc(VarGrNodeAccessorPtr *node***);**

Deallocates the node accessor.

Dispose

void VARGRNODEACCESSOR_Dispose(VarGrNodeAccessorPtr *node***);**

Disposes (destroys and deallocates) the node accessor.

Clone a node accessor

It is sometimes necessary to clone a node accessor so that its navigation state can be transferred to a new accessor.

Clone

VarGrNodeAccessorPtr VARGRNODEACCESSOR_Clone(
VarGrNodeAccessorCPtr *node***);**

Creates a node accessor for navigation through nodes in a graph datasource using “node” as a template.

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

Alloc

VarGrAllEdgeAccessorPtr VARGRALLEDGEACCESSOR_Alloc(void);

Returns a pointer to an allocated, but not yet constructed, edge accessor. The edge accessor should be constructed before being used.

Alloc

void VARGRALLEDGEACCESSOR_Construct(
VarGrAllEdgeAccessorPtr *edge***);**

Default edge-accessor construction.

ConstructCopy

void VARGRALLEDGEACCESSOR_ConstructCopy(
VarGrAllEdgeAccessorPtr *edge***, VarGrAllEdgeAccessorCPtr** *cedge***);**

Constructs the edge accessor with the information obtained from “cedge.”

Destruct

void VARGRALLEDGEACCESSOR_Destruct(
VarGrAllEdgeAccessorPtr *edge***);**

Default edge-accessor destruction.

Dealloc

void VARGRALLEDGEACCESSOR_Dealloc(
VarGrAllEdgeAccessorPtr *edge***);**

Deallocates the edge accessor.

Dispose

void VARGRALLEDGEACCESSOR_Dispose(
VarGrAllEdgeAccessorPtr *edge***);**

Disposes (destroys and deallocates) the edge accessor.

Create

VarGrAllEdgeAccessorPtr VARGRALLEDGEACCESSOR_Create(void);

Creates an edge accessor for navigation through the edges of a graph-datasource node.

Clone an Edge Accessor

It is sometimes necessary to clone an edge accessor so that its navigation state can be transferred to a new accessor.

Clone

VarGrAllEdgeAccessorPtr VARGRALLEDGEACCESSOR_Clone(
VarGrAllEdgeAccessorCPtr *edge***);**

Creates an edge accessor for navigation through edges in a graph datasource using “edge” as a template.

“In” edge accessor

CreateInEdgeAccessor

VarGrInEdgeAccessorPtr(
VARGRNODEACCESSOR_CreateInEdgeAccessor(
VarGrNodeAccessorPtr *node***);**

Creates an edge accessor for navigation through the inwardly directed edges of a graph-datasource node.

Alloc

VarGrInEdgeAccessorPtr VARGRINEDGEACCESSOR_Alloc(void);

Returns a pointer to an allocated, but not yet constructed, “in” edge accessor. The “in” edge accessor should be constructed before being used.

Construct

void VARGRINEDGEACCESSOR_Construct(
VarGrInEdgeAccessorPtr *edge***, VarGrNodeAccessorPtr** *node***);**

Default “in” edge-accessor construction.

ConstructCopy

void VARGRINEDGEACCESSOR_ConstructCopy(
VarGrInEdgeAccessorPtr *edge***,**
VarGrInEdgeAccessorCPtr *cedge***);**

Constructs the "in" edge accessor with the information obtained from "cedge."

Destruct

void VARGRINEDGEACCESSOR_Destruct(
VarGrInEdgeAccessorPtr *edge***);**

Default "in" edge-accessor destruction.

Dealloc

void VARGRINEDGEACCESSOR_Dealloc(
VarGrInEdgeAccessorPtr *edge***);**

Deallocates the "in" edge accessor.

Dispose

void VARGRINEDGEACCESSOR_Dispose(
VarGrInEdgeAccessorPtr *edge***);**

Disposes (destroys and deallocates) the "in" edge accessor.

Create

VarGrInEdgeAccessorPtr VARGRINEDGEACCESSOR_Create(
VarGrNodeAccessorPtr *node***);**

Creates an edge accessor for navigation through the inwardly directed edges of a graph-datasource node.

Clone an "in" edge accessor

It is sometimes necessary to clone an "in" edge accessor so that its navigation state can be transferred to a new accessor.

Clone

VarGrInEdgeAccessorPtr VARGRINEDGEACCESSOR_Clone(
VarGrInEdgeAccessorCPtr *edge***);**

Creates an "in" edge accessor for navigation through edges in a graph datasource using "edge" as a template.

"Out" Edge Accessor

CreateOutEdgeAccessor

VarGrOutEdgeAccessorPtr VARGRNODEACCESSOR_CreateOutEdgeAccessor(
VarGrNodeAccessorPtr *node***);**

Creates an edge accessor for navigation through the outwardly directed edges of a graph-datasource node.

Alloc

VarGrOutEdgeAccessorPtr VARGROUTEDGEACCESSOR_Alloc(void);

Returns a pointer to an allocated, but not yet constructed, “out” edge accessor. The “out” edge accessor should be constructed before being used.

Construct

**void VARGROUTEDGEACCESSOR_Construct(VarGrOutEdgeAccessorPtr edge,
VarGrNodeAccessorPtr node);**

Default “out” edge-accessor construction.

ConstructCopy

**void VARGROUTEDGEACCESSOR_ConstructCopy(
VarGrOutEdgeAccessorPtr** *edge*, **VarGrOutEdgeAccessorCPtr** *cedge***);**

Constructs the “out” edge accessor with the information obtained from “cedge.”

Destruct

**void VARGROUTEDGEACCESSOR_Destruct(
VarGrOutEdgeAccessorPtr** *edge***);**

Default “out” edge-accessor destruction.

Dealloc

**void VARGROUTEDGEACCESSOR_Dealloc(
VarGrOutEdgeAccessorPtr** *edge***);**

Deallocates the “out” edge accessor.

Dispose

**void VARGROUTEDGEACCESSOR_Dispose(
VarGrOutEdgeAccessorPtr** *edge***);**

Disposes (destroys and deallocates) the “out” edge accessor.

Create

**VarGrOutEdgeAccessorPtr VARGROUTEDGEACCESSOR_Create(
VarGrNodeAccessorPtr** *node***);**

Creates an edge accessor for navigation through the outwardly directed edges of a graph-datasource node.

Clone an “Out” Edge Accessor

It is sometimes necessary to clone an “out” edge accessor so that its navigation state can be transferred to a new accessor.

Clone

**VarGrOutEdgeAccessorPtr VARGROUTEDGEACCESSOR_Clone(
VarGrOutEdgeAccessorCPtr** *edge***);**

Creates an “out” edge accessor for navigation through edges in a graph datasource using “edge” as a template.

Undirected Edge Accessor

CreateUndirEdgeAccessor

VarGrUndirEdgeAccessorPtr
VARGRNODEACCESSOR_CreateUndirEdgeAccessor(
VarGrNodeAccessorPtr *node***);**

Creates an edge accessor for navigation through the undirected edges of a graph-datasource node.

Alloc

VarGrUndirEdgeAccessorPtr
VARGRUNDIREDGEACCESSOR_Alloc(void);

Returns a pointer to an allocated, but not yet constructed, undirected edge accessor. The undirected edge accessor should be constructed before being used.

Construct

void VARGRUNDIREDGEACCESSOR_Construct(
VarGrUndirEdgeAccessorPtr *edge***,**
VarGrNodeAccessorPtr *node***);**

Default undirected-edge-accessor construction.

ConstructCopy

void VARGRUNDIREDGEACCESSOR_ConstructCopy(
VarGrUndirEdgeAccessorPtr *edge***,**
VarGrUndirEdgeAccessorCPtr *cedge***);**

Constructs the undirected edge accessor with the information obtained from “cedge.”

Destruct

void VARGRUNDIREDGEACCESSOR_Destruct(
VarGrUndirEdgeAccessorPtr *edge***);**

Default undirected-edge-accessor destruction.

Dealloc

void VARGRUNDIREDGEACCESSOR_Dealloc(
VarGrUndirEdgeAccessorPtr *edge***);**

Deallocates the undirected edge accessor.

Dispose

void VARGRUNDIREDGEACCESSOR_Dispose(
VarGrUndirEdgeAccessorPtr *edge***);**

Disposes (destroys and deallocates) the undirected edge accessor.

Create

VarGrUndirEdgeAccessorPtr VARGRUNDIREDGEACCESSOR_Create(
VarGrNodeAccessorPtr *node***);**

Creates an edge accessor for navigation through the undirected edges of a graph-datasource node.

Clone an Undirected Edge Accessor

It is sometimes necessary to clone an undirected edge accessor so that its navigation state can be transferred to a new accessor.

Clone

VarGrUndirEdgeAccessorPtr VARGRUNDIREDGEACCESSOR_Clone(
   VarGrUndirEdgeAccessorCPtr *edge***);**

Creates an undirected edge accessor for navigation through edges in a graph datasource using “edge” as a template.

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 VARGRNODEACCESSOR_GoFirstRoot(
   VarGrNodeAccessorPtr *node***);**

Positions the node accessor on the first root node in the graph.

GoNthRoot

void VARGRNODEACCESSOR_GoNthRoot(
   VarGrNodeAccessorPtr *node***, VarGrIndexVal** *index***);**

Positions the node accessor on the Nth root node identified by index “index” in the graph.

GoIndexed

void VARGRNODEACCESSOR_GoIndexed(
   VarGrNodeAccessorPtr *node***, VarGrIndexVal** *index***);**

Positions the node accessor on the node identified by index “index.”

GoID

void VARGRNODEACCESSOR_GoID(
VarGrNodeAccessorPtr *node***, VarPtr** *id***);**

Positions the node accessor on the node identified by ID “id.”

GoFirstParent

void VARGRNODEACCESSOR_GoFirstParent(
VarGrNodeAccessorPtr *node***);**

Positions the node accessor on the first parent node of the node where it is currently positioned.

GoNthParent

void VARGRNODEACCESSOR_GoNthParent(
VarGrNodeAccessorPtr *node***, 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 VARGRNODEACCESSOR_GoFirstChild(
VarGrNodeAccessorPtr *node***);**

Positions the node accessor on the first child node of the node where it is currently positioned.

GoNthChild

void VARGRNODEACCESSOR_GoNthChild(
VarGrNodeAccessorPtr *node***, 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 VARGRNODEACCESSOR_GoFirstNeighbor(
VarGrNodeAccessorPtr *node***);**

Positions the node accessor on the first neighbor node of the node where it is currently positioned.

GoNthNeighbor

void VARGRNODEACCESSOR_GoNthNeighbor(
VarGrNodeAccessorPtr *node***, 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 VARGRNODEACCESSOR_GoNext(VarGrNodeAccessorPtr *node***);**

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 VARGRNODEACCESSOR_GoPrev(
VarGrNodeAccessorPtr *node***);**

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 VARGRALLEDGEACCESSOR_GoFirst(
VarGrAllEdgeAccessorPtr *edge***);**

Positions the edge accessor on the first edge in the graph datasource.

GoNext

void VARGRALLEDGEACCESSOR_GoNext(
VarGrAllEdgeAccessorPtr *edge***);**

Positions the edge accessor on the next edge in the graph datasource.

GoPrev

void VARGRALLEDGEACCESSOR_GoPrev(
VarGrAllEdgeAccessorPtr *edge***);**

Positions the edge accessor on the previous edge in the graph datasource.

GoIndexed

void VARGRALLEDGEACCESSOR_GoIndexed(
VarGrAllEdgeAccessorPtr *edge***, VarGrIndexVal** *index***);**

Positions the edge accessor on the edge identified by the index "index" in the graph datasource.

GoID

void VARGRALLEDGEACCESSOR_GoID(
VarGrAllEdgeAccessorPtr *edge*, **VarPtr** *id***);**

Positions the edge accessor on the edge identified by the ID “id” in the graph datasource.

GoBetween

void VARGRALLEDGEACCESSOR_GoBetween(
VarGrAllEdgeAccessorPtr *edge*,
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 VARGRINEDGEACCESSOR_GoFirst(
VarGrInEdgeAccessorPtr *edge***);**

Positions the edge accessor on the first edge of the node it was created for.

GoNext

void VARGRINEDGEACCESSOR_GoNext(
VarGrInEdgeAccessorPtr *edge***);**

Positions the edge accessor on the next edge of the node it was created for.

GoPrev

void VARGRINEDGEACCESSOR_GoPrev(
VarGrInEdgeAccessorPtr *edge***);**

Positions the edge accessor on the previous edge of the node it was created for.

GoIndexed

void VARGRINEDGEACCESSOR_GoIndexed(
VarGrInEdgeAccessorPtr *edge*, **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 VARGRINEDGEACCESSOR_GoID(
VarGrInEdgeAccessorPtr *edge*, **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 VARGROUTEDGEACCESSOR_GoFirst(
VarGrOutEdgeAccessorPtr *edge***);**

Positions the edge accessor on the first edge of the node it was created for.

GoNext

void VARGROUTEDGEACCESSOR_GoNext(
VarGrOutEdgeAccessorPtr *edge***);**

Positions the edge accessor on the next edge of the node it was created for.

GoPrev

void VARGROUTEDGEACCESSOR_GoPrev(
VarGrOutEdgeAccessorPtr *edge***);**

Positions the edge accessor on the previous edge of the node it was created for.

GoIndexed

void VARGROUTEDGEACCESSOR_GoIndexed(
VarGrOutEdgeAccessorPtr *edge***, 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 VARGROUTEDGEACCESSOR_GoID(
VarGrOutEdgeAccessorPtr *edge***, 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 VARGRUNDIREDGEACCESSOR_GoFirst(
VarGrUndirEdgeAccessorPtr *edge***);**

Positions the edge accessor on the first edge of the node it was created for.

GoNext

void VARGRUNDIREDGEACCESSOR_GoNext(
VarGrUndirEdgeAccessorPtr *edge***);**

Positions the edge accessor on the next edge of the node it was created for.

GoPrev

void VARGRUNDIREDGEACCESSOR_GoPrev(
VarGrUndirEdgeAccessorPtr *edge***);**

Positions the edge accessor on the previous edge of the node it was created for.

GoIndexed

void VARGRUNDIREDGEACCESSOR_GoIndexed(
VarGrUndirEdgeAccessorPtr *edge*, **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 VARGRUNDIREDGEACCESSOR_GoID(
VarGrUndirEdgeAccessorPtr *edge*, **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 VARGR_AddNode(
VarGrPtr *varGr*, **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 VARGREDIT_AddNode(
VarGrEditPtr *edit*, **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

BoolEnum VARGR_RemoveNode(
VarGrPtr *varGr*, **VarGrNodeAccessorPtr** *node***);**

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 VARGREDIT_RemoveNode(
VarGrEditPtr *edit*, **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 VARGR_AddDirEdge(VarGrPtr *varGr*,
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 VARGREDIT_AddDirEdge(VarGrEditPtr *edit*,
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 VARGR_AddUndirEdge(VarGrPtr *varGr*,
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 VARGREDIT_AddUndirEdge(VarGrEditPtr *edit*,
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 VARGR_RemoveEdge(
VarGrPtr *varGr*, **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 VARGREDIT_RemoveEdge(
VarGrEditPtr *edit*, **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 VARGR_RemoveEdgeBetween(VarGrPtr *varGr*,

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 VARGREDIT_RemoveEdgeBetween(VarGrEditPtr *edit*,
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 VARGR_IsNodeValid(
VarGrPtr *varGr*, **VarGrNodeAccessorPtr** *node***);**

Returns BOOL_TRUE if the node accessor is currently positioned on an existing node.

AreNodesEqual

BoolEnum VARGR_AreNodesEqual(VarGrPtr *varGr*,
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 VARGR_GetNodeNumParents(
VarGrPtr *varGr*, **VarGrNodeAccessorPtr** *node***);**

Returns the number of parent nodes for the node identified by accessor “node.”

GetNodeNumChildren

VarGrIndexVal VARGR_GetNodeNumChildren(
VarGrPtr *varGr*, **VarGrNodeAccessorPtr** *node***);**

Returns the number of child nodes for the node identified by accessor “node.”

GetNodeNumNeighbors

VarGrIndexVal VARGR_GetNodeNumNeighbors(
VarGrPtr *varGr*, **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 VARGR_QueryNodeID(VarGrPtr *varGr*,
VarGrNodeAccessorPtr *node*, **VarPtr** *value***);**

Returns the ID of the node referenced by node accessor “node” into the variant “value.”

GetNodeID

VarPtr VARGR_GetNodeID(
VarGrPtr *varGr*, **VarGrNodeAccessorPtr** *node***);**

SetNodeID

BoolEnum VARGR_SetNodeID(VarGrPtr *varGr*,
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 VARGREDIT_SetNodeID(VarGrEditPtr *edit*,
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 VARGR_QueryNodeValue(VarGrPtr *varGr*,
VarGrNodeAccessorPtr *node*, **VarPtr** *value***);**

Returns the value of the node referenced by node accessor “node” into the variant “value.”

GetNodeValue

VarPtr VARGR_GetNodeValue(
VarGrPtr *varGr*, **VarGrNodeAccessorPtr** *node***);**

SetNodeValue

BoolEnum VARGR_SetNodeValue(VarGrPtr *varGr*,
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 VARGREDIT_SetNodeValue(VarGrEditPtr *edit*,
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 VARGR_QueryNodeXOrigin(VarGrPtr *varGr*,
VarGrNodeAccessorPtr *node*, **VarPtr** *value***);**

Returns the *x* origin of the node referenced by node accessor “node” into the variant “value.”

GetNodeXOrigin

VarPtr VARGR_GetNodeXOrigin(
VarGrPtr *varGr*, **VarGrNodeAccessorPtr** *node***);**

SetNodeXOrigin

BoolEnum VARGR_SetNodeXOrigin(VarGrPtr *varGr*,
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 VARGREDIT_SetNodeXOrigin(VarGrEditPtr *edit*,
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 VARGR_QueryNodeYOrigin(VarGrPtr *varGr***,**
VarGrNodeAccessorPtr *node***, VarPtr** *value***);**

Returns the *y* origin of the node referenced by node accessor “node” into the variant “value.”

GetNodeYOrigin

VarPtr VARGR_GetNodeYOrigin(
VarGrPtr *varGr***, VarGrNodeAccessorPtr** *node***);**

SetNodeYOrigin

BoolEnum VARGR_SetNodeYOrigin(VarGrPtr *varGr***,**
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 VARGREDIT_SetNodeYOrigin(VarGrEditPtr *edit***,**
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 VARGR_QueryNodeHeight(VarGrPtr *varGr***,**
VarGrNodeAccessorPtr *node***, VarPtr** *value***);**

Returns the height of the node referenced by node accessor “node” into the variant “value.”

GetNodeHeight

VarPtr VARGR_GetNodeHeight(
VarGrPtr *varGr***, VarGrNodeAccessorPtr** *node***);**

SetNodeHeight

BoolEnum VARGR_SetNodeHeight(VarGrPtr *varGr***,**
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 VARGREDIT_SetNodeHeight(VarGrEditPtr edit,
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 VARGR_QueryNodeWidth(VarGrPtr *varGr***,**
VarGrNodeAccessorPtr *node*, **VarPtr** *value***);**

Returns the width of the node referenced by node accessor "node" into the variant "value."

GetNodeWidth

VarPtr VARGR_GetNodeWidth(
VarGrPtr *varGr*, **VarGrNodeAccessorPtr** *node***);**

SetNodeWidth

BoolEnum VARGR_SetNodeWidth(VarGrPtr *varGr***,**
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 VARGREDIT_SetNodeWidth(VarGrEditPtr *edit***,**
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 VARGR_QueryNodeProperty(VarGrPtr *varGr***,**
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 VARGR_GetNodeProperty(VarGrPtr *varGr***,**
VarGrNodeAccessorPtr *node***, CStr** *key***);**

SetNodeProperty

BoolEnum VARGR_SetNodeProperty(VarGrPtr *varGr***,**
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 VARGREDIT_SetNodeProperty(VarGrEditPtr *edit***,**
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 VARGR_RemoveNodeProperty(VarGrPtr *varGr***,**
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 VARGREDIT_RemoveNodeProperty(VarGrEditPtr *edit***,**
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 VARGR_IsEdgeValid(VarGrPtr *varGr***,**
VarGrEdgeAccessorPtr *edge***);**

Returns BOOL_TRUE if the edge accessor is currently positioned on an existing edge.

AreEdgesEqual

BoolEnum VARGR_AreNodesEqual(VarGrPtr *varGr***,**
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 VARGR_GetEdgeNumEdges(
VarGrPtr *varGr***, 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 VARGR_QueryEdgeID(VarGrPtr *varGr***,**
VarGrEdgeAccessorPtr *edge***, VarPtr** *value***);**

Returns the ID of the edge referenced by edge accessor “edge” into the variant “value.”

GetEdgeID

VarPtr VARGR_GetEdgeID(
VarGrPtr *varGr***, VarGrEdgeAccessorPtr** *edge***);**

SetEdgeID

BoolEnum VARGR_SetEdgeID(VarGrPtr *varGr***,**
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 VARGREDIT_SetEdgeID(VarGrEditPtr *edit***,**
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 VARGR_QueryEdgeValue(VarGrPtr *varGr***,**
VarGrEdgeAccessorPtr *edge***, VarPtr** *value***);**

Returns the value of the edge referenced by edge accessor “edge” into the variant “value.”

GetEdgeValue

VarPtr VARGR_GetEdgeValue(
VarGrPtr *varGr***, VarGrEdgeAccessorPtr** *edge***);**

SetEdgeValue

BoolEnum VARGR_SetEdgeValue(VarGrPtr *varGr***,**
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 VARGREDIT_SetEdgeValue(VarGrEditPtr *edit***,**
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.

GetEdgeIsDirected

BoolEnum VARGR_GetEdgeIsDirected(
VarGrPtr *varGr***, VarGrEdgeAccessorPtr** *edge***);**

SetEdgeIsDirected

BoolEnum VARGR_SetEdgeIsDirected(VarGrPtr *varGr***,**
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.

SetEdgeIsDirected

void VARGREDIT_SetEdgeIsDirected(VarGrEditPtr *edit***,**
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 VARGR_QueryEdgeProperty(VarGrPtr *varGr***,**
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 VARGR_GetEdgeProperty(VarGrPtr *varGr***,**
VarGrEdgeAccessorPtr *edge***, CStr** *key***);**

SetEdgeProperty

BoolEnum VARGR_SetEdgeProperty(VarGrPtr *varGr***,**
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 VARGREDIT_SetEdgeProperty(VarGrEditPtr *edit***,**
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 VARGR_RemoveEdgeProperty(VarGrPtr *varGr***,**
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 VARGREDIT_RemoveEdgeProperty(VarGrEditPtr *edit***,**
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 VARGR_IsChildNode(VarGrPtr *varGr***,**
VarGrNodeAccessorPtr *source***, VarGrNodeAccessorPtr** *target***);**

Returns BOOL_TRUE if “target” is a child of “source.”

IsParentNode

BoolEnum VARGR_IsParentNode(VarGrPtr *varGr***,**
VarGrNodeAccessorPtr *node***, VarGrNodeAccessorPtr** *target***);**

Returns BOOL_TRUE if “target” is a parent of “source.”

IsNeighborNode

BoolEnum VARGR_IsNeighborNode(VarGrPtr *varGr***,**
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 VARGR_GetNodeCursor(VarGrPtr *varGr***);**

SetNodeCursor

BoolEnum VARGR_SetNodeCursor(
VarGrPtr *varGr***, 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 VARGREDIT_SetNodeCursor(
VarGrEditPtr *edit***, 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 VARGR_GetEdgeCursor(VarGrPtr *varGr***);**

SetEdgeCursor

BoolEnum VARGR_SetEdgeCursor(
VarGrPtr *varGr***, 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 VARGREDIT_SetEdgeCursor(
VarGrEditPtr *edit***, 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 VARGR_StartNodeEdit(
VarGrPtr *varGr***, 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 VARGR_StartEdgeEdit(
VarGrPtr *varGr***, 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 VARGR_QueryCyclicResult(VarGrPtr *varGr***,**
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 VARGR_GetNode(
  VarGrPtr *varGr*, **VarGrNodeAccessorPtr** *node***);**

Get a node from the graph. Nodes are retrieved from the graph by using a node accessor.

GetEdge

VarGrEdgePtr VARGR_GetEdge(
  VarGrPtr *varGr*, **VarGrEdgeAccessorPtr** *edge***);**

Get an edge from the graph. Edges are retrieved from the graph by using an edge accessor.

Node-Object Properties

GetNumChildren

VarGrIndexVal VARGRNODE_GetNumChildren(VarGrNodePtr *node***);**

Returns the number of child nodes for the node.

GetChildNode

VarGrNodePtr VARGRNODE_GetChildNode(
  VarGrNodePtr *node*, **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 VARGRNODE_GetNumParents(VarGrNodePtr *node***);**

Returns the number of parent nodes for the node.

GetParentNode

VarGrNodePtr VARGRNODE_GetParentNode(
  VarGrNodePtr *node*, **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 VARGRNODE_GetNumNeighbors(VarGrNodePtr *node***);**

Returns the number of neighbor nodes for the node.

GetNeighborNode

VarGrNodePtr VARGRNODE_GetNeighborNode(
  VarGrNodePtr *node*, **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 VARGRNODE_GetOutEdge(
VarGrNodePtr *node***, 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 VARGRNODE_GetInEdge(
VarGrNodePtr *node***, 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 VARGRNODE_GetUndirEdge(
VarGrNodePtr *node***, 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 VARGRNODE_GetID(VarGrNodePtr *node***);**

SetID

BoolEnum VARGRNODE_SetID(VarGrNodePtr *node***, 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 VARGRNODEEDIT_SetID(VarGrNodeEditPtr *edit***, 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 VARGRNODE_QueryValue(VarGrNodePtr *node***, VarPtr** *value***);**

Returns the value of the node into the variant “value.”

GetValue

VarPtr VARGRNODE_GetValue(VarGrNodePtr *node***);**

SetValue

BoolEnum VARGRNODE_SetValue(VarGrNodePtr *node***, 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 VARGRNODEEDIT_SetValue(
VarGrNodeEditPtr *edit***, 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 VARGRNODE_GetXOrigin(VarGrNodePtr *node***);**

SetXOrigin

BoolEnum VARGRNODE_SetXOrigin(
VarGrNodePtr *node***, 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 VARGRNODEEDIT_SetXOrigin(
VarGrNodeEditPtr *edit***, 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 VARGRNODE_GetYOrigin(VarGrNodePtr *node***);**

SetYOrigin

BoolEnum VARGRNODE_SetYOrigin(
VarGrNodePtr *node***, 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 VARGRNODEEDIT_SetYOrigin(
VarGrNodeEditPtr *edit***, 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 VARGRNODE_GetHeight(VarGrNodePtr *node***);**

SetHeight

BoolEnum VARGRNODE_SetHeight(
VarGrNodePtr *node***, 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 VARGRNODEEDIT_SetHeight(
VarGrNodeEditPtr *edit***, 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 VARGRNODE_GetWidth(VarGrNodePtr *node***);**

SetWidth

BoolEnum VARGRNODE_SetWidth(VarGrNodePtr *node***, 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 VARGRNODEEDIT_SetWidth(
VarGrNodeEditPtr *edit***, 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 VARGRNODE_GetProperty(VarGrNodePtr *node*, **CStr** *key***);**

SetProperty

BoolEnum VARGRNODE_SetProperty(
VarGrNodePtr *node*, **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 VARGRNODEEDIT_SetProperty(
VarGrNodeEditPtr *edit*, **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 VARGRNODE_RemoveProperty(
VarGrNodePtr *node*, **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 VARGRNODEEDIT_RemoveProperty(
VarGrNodeEditPtr *edit*, **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 VARGREDGE_GetFromNode(VarGrEdgePtr *edge***);**

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 VARGREDGE_GetToNode(VarGrEdgePtr *edge***);**

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 VARGREDGE_GetID(VarGrEdgePtr *edge***);**

SetID

BoolEnum VARGREDGE_SetID(VarGrEdgePtr *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.

SetID

void VARGREDGEEDIT_SetID(VarGrEdgeEditPtr *edit***, 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 VARGREDGE_QueryValue(VarGrEdgePtr *edge***, VarPtr** *value***);**

Returns the value of the edge into the variant “value.”

GetValue

VarPtr VARGREDGE_GetValue(VarGrEdgePtr *varGr***);**

SetValue

BoolEnum VARGREDGE_SetValue(VarGrEdgePtr *edge***, 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 VARGREDGEEDIT_SetValue(VarGrEdgeEditPtr *edit***, 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 VARGREDGE_GetIsDirected(VarGrEdgePtr *edge***);**

SetIsDirected

BoolEnum VARGREDGE_SetIsDirected(
VarGrEdgePtr *edge***, 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 VARGREDGEEDIT_SetIsDirected(
VarGrEdgeEditPtr *edit***, 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 VARGREDGE_GetProperty(VarGrEdgePtr *edge***, CStr** *key***);**

SetProperty

BoolEnum VARGREDGE_SetProperty(
VarGrEdgePtr *edge***, 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 VARGREDGEEDIT_SetProperty(
VarGrEdgeEditPtr *edit***, 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 VARGREDGE_RemoveProperty(
VarGrEdgePtr *edge***, 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 VARGREDGEEDIT_RemoveProperty(
VarGrEdgeEditPtr *edit*, **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 VARGRNODE_StartEdit(VarGrNodePtr *node***);**

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 VARGREDGE_StartEdit(VarGrEdgePtr *edge***);**

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 VARGR_GetMods(VarGrPtr *varGr***);**

Get a description of the last modifications committed on the graph datasource.

Class Operations

Create

VarGrPtr VARGR_Create(void);

Creates and constructs a variant graph datasource.

Chapter 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

RClasPtr VARLS_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.

CStr VARLS_GetTitle(VarLsPtr *varLs***);**

Returns the title for the list, if any.

SetTitle

BoolEnum VARLS_SetTitle(VarLsPtr *varLs***, 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 VARLS_GetRowTitle(VarLsPtr *varLs*, **VarLsIndexVal** *index*);

Returns the title for the row `index' in the list, if any.

SetRowTitle

BoolEnum VARLS_SetRowTitle(VarLsPtr *varLs*, **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 VARLS_GetMaxRowTitleStrLen(VarLsPtr *varLs*);

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 VARLS_QueryRowValue(VarLsPtr *varLs*, **VarLsIndexVal** *index*, **VarPtr** *value*);

Returns the value of the row `index' in the list.

GetMaxStrLen

StrIVal VARLS_GetMaxStrLen(VarLsPtr *varLs*);

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 VARLS_GetRowValue(VarLsPtr *varLs*, **VarLsIndexVal** *index*);

Returns the value of the row `index' in the list. The caller is responsible for freeing the returned variant. VARLS_QueryRowValue is a preferable call.

SetRowValue

BoolEnum VARLS_SetRowValue(VarLsPtr *varLs*, **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 VARLS_GetNumRows(VarLsPtr *varLs***);**

Returns the number of rows in the list data source.

SetNumRows

BoolEnum VARLS_SetNumRows(VarLsPtr *varLs***, 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 VARLS_AddRow(VarLsPtr *varLs***, 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 VARLS_RemoveRow(VarLsPtr *varLs***, 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 VARLS_GetCursorRow(VarLsPtr *ls***);**

Returns the current position of the cursor of the list.

SetCursorRow

BoolEnum VARLS_SetCursorRow(VarLsPtr *varLs***, 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 VARLS_StartRowEdit(VarLsPtr *varLs*, **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 VARLSEDIT_AddRow(VarLsEditPtr *edit*, **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 VARLSEDIT_RemoveRow(VarLsEditPtr *edit*, **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 VARLSEDIT_SetCursorRow(VarLsEditPtr *edit*, **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 VARLSEDIT_SetNumRows(VarLsEditPtr *lsEdit*, **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 VARLSEDIT_SetRowValue(VarLsEditPtr *edit*, **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 VARLSEDIT_SetRowTitle(VarLsEditPtr *edit*, **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 VARLSEDIT_SetTitle(VarLsEditPtr *edit*, **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 VARLS_GetMods(VarLsPtr *varLs***);**

Get a description of the last modifications made on the list through an edition object.

Notifications

DefNfy

void VARLS_DefNfy(VarLsPtr *varLs*, **VarLsNfyEnum** *code***);**

Default notification procedure for the VarLs class.

Chapter

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

RClasPtr VARTB_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 VARTB_GetNumRows(VarTbPtr *varTb***);**

Returns the number of rows in the table.

GetNumColumns

VarTbIndexVal VARTB_GetNumColumns(VarTbPtr *varTb***);**

Returns the number of columns in the table.

QueryCellValue

void VARTB_QueryCellValue(VarTbPtr *varTb*, **VarTbIndexVal** *row*,
VarTbIndexVal *col*, **VarPtr** *value***);**

Returns the value of the cell at row `row' and column `col' the list.

GetCellValue**VarPtr VARTB_GetCellValue(VarTbPtr** *varTb*,
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 VARTB_GetMaxColStrLen(VarTbPtr *varTb*, **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 VARTB_GetRowTitle(VarTbPtr *varTb*, **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 VARTB_SetRowTitle(VarTbPtr *varTb*,
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 VARTB_GetColumnTitle(VarTbPtr *varTb*, **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 VARTB_GetTitle(VarTbPtr *varTb***);**

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 VARTB_GetCursorRow(VarTbPtr *varTb***);**

Returns the current position of the row cursor

GetCursorColumn

VarTbIndexVal VARTB_GetCursorColumn(VarTbPtr *varTb***);**

Returns the current position of the column cursor

Edition Support

StartRowEdit

VarTbEditPtr VARTB_StartRowEdit(VarTbPtr *varTb***, 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 VARTB_StartCellEdit(VarTbPtr *varTb***,**
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 VARTB_SetNumRowColumns(VarTbPtr *varTb***, 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 VARTB_AddRow(VarTbPtr *varTb***, 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 VARTB_RemoveRow(VarTbPtr *varTb***, VarTbIndexVal** *index***);**

Remove a row by internally creating and committing a edition. Returns BOOL_TRUE if the internal edition could take place.

AddColumn

BoolEnum VARTB_AddColumn(VarTbPtr *varTb*, **VarTbIndexVal** *index*);

Add a column by internally creating and committing a edition. Returns BOOL_TRUE if the internal edition could take place.

RemoveColumn

BoolEnum VARTB_RemoveColumn(VarTbPtr *varTb*, **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 VARTB_SetColValue(VarTbPtr *varTb*,
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 VARTB_SetRowTitle(VarTbPtr *varTb*,
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.

SetColumnTitle

BoolEnum VARTB_SetColumnTitle(VarTbPtr *varTb*,
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 VARTB_SetTitle(VarTbPtr *varTb*, **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 VARTB_SetCursorRow(VarTbPtr *varTb*, **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 VARTB_SetCursorColumn(
VarTbPtr *varTb*, **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 VARTB_StartEdit(VarTbPtr *varTb***);**

Opens an edition on the whole table data source. The operations are done through the edition object returned by this call.

SetNumRowColumns

void VARTBEDIT_SetNumRowColumns(VarTbEditPtr *edit***,
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

**BoolEnum VARTBEDIT_AddRow(
VarTbPtr** *varTb***, 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 VARTBEDIT_RemoveRow(
VarTbEditPtr** *edit***, 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 VARTBEDIT_AddColumn(
VarTbEditPtr** *edit***, 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 VARTBEDIT_RemoveColumn(
VarTbEditPtr** *edit***, VarTbIndexVal** *index***);**

Remove a column through a edition.

SetCellValue

void VARTBEDIT_SetCellValue(VarTbEditPtr *edit***,**

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 VARTBEDIT_SetRowTitle(
VarTbEditPtr *edit*, **VarTbIndexVal** *index*, **CStr** *title***);**

Sets the title corresponding to the row identified by "index" through an edition.

SetColumnTitle

void VARTBEDIT_SetColumnTitle(
VarTbEditPtr *edit*, **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 VARTBEDIT_SetTitle(VarTbEditPtr *edit*, **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 VARTBEDIT_SetCursorRow(
VarTbEditPtr *edit*, **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 VARTBEDIT_SetCursorColumn(
VarTbEditPtr *edit*, **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 VARTB_GetMods(VarTbPtr *varTb***);**

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 DS interface. In particular, opening a edition for this object is done through DS_StartEdit.

Column Interaction

The largest part of the interface for this object is in fact in the DS interface. In particular, opening a edition for this object is done through DS_StartEdit.

Cell Interaction

The largest part of the interface for this object is in fact in the DS interface. In particular, opening a edition for this object is done through DS_StartEdit..

Virtual Interface Implementation

Variant List Implementation

extern "C" RClasPtr VarTbGetClass(void);

extern "C" void VarTbConstruct(ResPtr *re***s, RClasCPtr** *rclas***, RClasCreateCPtr** *rCreate***);**

extern "C" void VarTbDestruct(ResPtr *res***);**

Variant List Row Implementation

extern "C" RClasPtr VarTbRowGetClass(void);

extern "C" void VarTbRowConstruct(ResPtr *re***s, RClasCPtr** *rclas***,**
RClasCreateCPtr *rCreate);*

extern "C" void VarTbRowDestruct(ResPtr *res***);**

Variant List Row Implementation

extern "C" RClasPtr VarTbColGetClass(void);

extern "C" void VarTbColConstruct(ResPtr *re***s, RClasCPtr** *rclas***, RClasCreateCPtr** *rCreate***);**

extern "C" void VarTbColDestruct(ResPt*r res***);**

Variant List Cell Implementation

extern "C" RClasPtr VarTbCellGetClass(void);

extern "C" void VarTbCellConstruct(ResPtr *re***s, RClasCPt***r rclas***, RClasCreateCPtr** *rCreate***);**

extern "C" void VarTbCellDestruct(ResPtr *res***);**

Chapter

49 VarTr

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

RClasPtr VARTR_Class(void);

Returns a pointer to the resource class for VarTr.

Class

RClasPtr VARTRNODEACCESSOR_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 VARTR_GetTitle(VarTrPtr *varTr***);**

BoolEnum VARTR_SetTitle(VarTrPtr *varTr***, CStr** *str***);**

Gets the title for the tree datasource, if any.

SetTitle

void VARTREDIT_SetTitle(VarTrEditPtr *edit***, 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 Accessors

Use accessors to traverse the tree datasource for nodes.

The node accessor is created and manipulated completely independently of the tree datasource. It is a pointer to a node.

Create

VarTrNodeAccessorPtr VARTRNODEACCESSOR_Create(void);

Allocates and constructs a node accessor for traversing nodes in a tree datasource.

Creates and constructs a variant-tree-datasource node accessor.

Clone

VarTrNodeAccessorPtr VARTRNODEACCESSOR_Clone(VarTrNodeAccessorCPtr *src***);**

Creates a node accessor using “src” as a template.

Destruct

void VARTRNODEACCESSOR_Destruct(
VarTrNodeAccessorPtr *accessor***);**

Dispose

void VARTRNODEACCESSOR_Dispose(
VarTrNodeAccessorPtr *accessor***);**

Destroys and deallocates a variant-tree-datasource node accessor.

Dispose0

void VARTRNODEACCESSOR_Dispose0(
VarTrNodeAccessorPtr *accessor***);**

If given a null pointer, does nothing; otherwise, destroys and deallocates the specified variant- tree-datasource node accessor.

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 VARTRNODEACCESSOR_GoFirstRoot(
VarTrNodeAccessorPtr *accessor***);**

Positions the node accessor on the first root node.

GoFirstChild

void VARTRNODEACCESSOR_GoFirstChild(
VarTrNodeAccessorPtr *accessor***);**

Positions the node accessor on the first child node of the node where it is currently positioned.

GoFirstSibling

void VARTRNODEACCESSOR_GoFirstSibling(
VarTrNodeAccessorPtr *accessor***);**

Positions the node accessor on the first sibling node of the node where it is currently positioned.

GoNext

void VARTRNODEACCESSOR_GoNext(
VarTrNodeAccessorPtr *accessor***);**

Positions the node accessor on the next node.

GoPrev

void VARTRNODEACCESSOR_GoPrev(
VarTrNodeAccessorPtr *accessor***);**

Positions the node accessor on the previous node.

GoParent

void VARTRNODEACCESSOR_GoParent(
VarTrNodeAccessorPtr *accessor***);**

Positions the node accessor on the parent node of the node where it is currently positioned.

Convenient Navigation

GoNthRoot

void VARTRNODEACCESSOR_GoNthRoot(
VarTrNodeAccessorPtr *accessor***, VarTrIndexVal** *index***);**

Positions the node accessor on the *n*th root node.

GoNthChild

void VARTRNODEACCESSOR_GoNthChild(VarTrNodeAccessorPtr *accessor***,**
VarTrIndexVal *index***);**

Positions the node accessor on the *n*th child node of the node where it is currently positioned.

GoNthSibling

void VARTRNODEACCESSOR_GoNthSibling(
VarTrNodeAccessorPtr *accessor*, **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 VARTR_AddNode(
VarTrPtr *varTr*, **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 VARTREDIT_AddNode(
VarTrEditPtr *edit*, **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 VARTR_RemoveNode(
VarTrPtr *varTr*, **VarTrNodeAccessorPtr** *accessor***);**

Removes a node at the position identified by the node accessor "node." An edit object is internally used for the operation.

RemoveNode

void VARTREDIT_RemoveNode(
VarTrEditPtr *edit*, **VarTrNodeAccessorPtr** *accessor***);**

Removes a node at the position identified by the node accessor "node."

RemoveTree

BoolEnum VARTR_RemoveTree(
VarTrPtr *varTr*, **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 VARTREDIT_RemoveTree(
VarTrEditPtr *edit*, **VarTrNodeAccessorPtr** *accessor***);**

Removes a tree starting from the position identified by the node accessor "accessor."

Class Operations

Create

VarTrPtr VARTR_Create(void);

Creates and constructs an instance of the VarTr class.

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 VARTR_GetNumRoots(VarTrPtr *varTr***);**

Returns the number of root nodes in the tree datasource.

GetNumChildren

VarTrIndexVal VARTR_GetNumChildren(
VarTrPtr *varTr***, VarTrNodeAccessorCPtr** *accessor***);**

Returns the number of child nodes relative to the current “accessor” location.

GetNumSiblings

VarTrIndexVal VARTR_GetNumSiblings(
VarTrPtr *varTr***, VarTrNodeAccessorCPtr** *accessor***);**

Returns the number of sibling nodes relative to the current “accessor” location.

IsNodeValid

BoolEnum VARTR_IsNodeValid(
VarTrPtr *varTr***, VarTrNodeAccessorCPtr** *accessor***);**

Returns BOOL_TRUE if a node exists at the current “accessor” location.

AreNodesEqual

BoolEnum VARTR_AreNodesEqual(VarTrPtr *varTr***,**
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 VARTR_GetCursor(VarTrPtr *varTr***);**

SetCursor

BoolEnum VARTR_SetCursor(
VarTrPtr *varTr***, 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 VARTR_DisposeCursor(VarTrPtr *varTr***);**

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 VARTREDIT_SetCursor(
VarTrEditPtr *edit***, VarTrNodeAccessorPtr** *accessor***);**

Modifying the Tree Datasource

Tree-Node Values

QueryNodeValue

void VARTR_QueryNodeValue(
VarTrPtr *varTr***, VarTrNodeAccessorCPtr** *accessor***, VarPtr** *value***);**

Returns the value of the node referenced by node accessor “node” into the variant “value.”

GetNodeValue

VarPtr VARTR_GetNodeValue(
VarTrPtr *varTr***, VarTrNodeAccessorCPtr** *accessor***);**

SetNodeValue

BoolEnum VARTR_SetNodeValue(
VarTrPtr *varTr***, 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 VARTR_QueryNodeID(
VarTrPtr *varTr*, **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 VARTR_GetNodeID(
VarTrPtr *varTr*, **VarTrNodeAccessorCPtr** *accessor***);**

SetNodeID

BoolEnum VARTR_SetNodeID(VarTrPtr *varTr*, **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 VARTR_StartEdit(VarTrPtr *varTr***);**

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 VARTR_StartNodeEdit(
VarTrPtr *varTr*, **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 VARTREDIT_SetNodeValue(
VarTrEditPtr *edit*, **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 VARTRNODEEDIT_SetValue(
VarTrNodeEditPtr *edit*, **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 VARTREDIT_SetNodeID(
VarTrEditPtr *edit*, **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 VARTRNODEEDIT_SetID(VarTrNodeEditPtr *edit*, **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 VARTR_GetMods(VarTrPtr *varTr*);

Get a description of the last modifications made on the tree datasource through an edit object.

Chapter

50 VStr Class

he 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 VSTR_SetStr(VStrPtr *vstr*, **CStr** *str***);**

VSTR_SetStr replaces the contents of a variable string with a copy of a source string. Unlike in VSTR_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 VSTR_SetNatStr(VStrPtr *vstr***, NatCStr** *str***);**

VSTR_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 VSTR_SetCtStr(VStrPtr *vstr***, CtCPtr** *ct***, NatCStr** *str***);**

VSTR_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 VSTR_SetStrSub(VStrPtr *vstr***, CStr** *str***, StrIVal** *slen***);**

VSTR_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 VSTR_SetNatStrSub(VStrPtr *vstr***, NatCStr** *str***, StrIVal** *slen***);**

VSTR_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 VSTR_SetCtStrSub(VStrPtr *vstr***, CtCPtr** *ct***, NatCStr** *str***, StrIVal** *slen***);**

Set

Copies one variable string to another.

void VSTR_Set(VStrPtr *vstr***, VStrCPtr** *vstr2***);**

VSTR_Set copies the contents of vstr2 into a variable string.

Copy

Copies one variable string to another.

void VSTR_Copy(VStrPtr *vstr***, VStrCPtr** *vstr2***);**

VSTR_Copy copies the contents of vstr2 into a variable string.

Queries

GetLen

Returns the length of a variable string.

StrIVal VSTR_GetLen(VStrCPtr *vstr***);**

VSTR_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 VSTR_GetStr(VStrCPtr *vstr***);**

VSTR_GetStr retrieves the string equivalent of the variable string passed and returns it.

QueryStrSub

Finds a substring within a variable string.

void VSTR_QueryStrSub(VStrCPtr *vstr***, CStrPtr** *strp***, StrIValPtr** *lenp***);**

VSTR_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 VSTR_AppendStr(VStrPtr *vstr***, CStr** *str***);**

VSTR_AppendStr appends a string to a variable string.

AppendStrSub

Appends a substring to a variable string.

void VSTR_AppendStrSub(VStrPtr *vstr***, CStr** *str***, StrIVal** *slen***);**

VSTR_AppendStrSub appends a substring to a variable string.

Append

Appends one variable string to another.

void VSTR_Append(VStrPtr *vstr***, VStrCPtr** *vstr2***);**

VSTR_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 VSTR_AppendChar(VStrPtr *vstr*, **ChCode** *ch***);**

VSTR_AppendChar appends a character to a variable string.

TruncAt

Truncates a variable string exactly to the length specified.

void VSTR_TruncAt(VStrPtr *vstr*, **StrIVal** *pos***);**

VSTR_TruncAt truncates exactly to length.

Truncate

Truncates a variable string at or before the length specified.

void VSTR_Truncate(VStrPtr *vstr*, **StrIVal** *pos***);**

VSTR_Truncate truncates at or before length.

Clear

Resets a variable string.

void VSTR_Clear(VStrPtr *vstr***);**

VSTR_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 VSTR_CmpStr(VStrCPtr *vstr*, **CStr** *str2***);**

CmpEnum VSTR_ICmpStr(VStrCPtr *vstr*, **CStr** *str2***);**

VSTR_CmpStr and VSTR_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.

VSTR_ICmpStr is the same at VSTR_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 VSTR_Cmp(VStrCPtr *vstr***, VStrCPtr** *vstr2***);**

CmpEnum VSTR_ICmp(VStrCPtr *vstr***, VStrCPtr** *vstr2***);**

VSTR_Cmp compares the strings by comparing the characters in each string by code value.

VSTR_ICmp is the same as VSTR_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 VSTR_SetRes (VStrPtr *vstr***, CStr** *mod***, CStr** *res***);**

VSTR_SetRes sets the given string resource as the contents of a variable string.

Arrays Of Strings

Constructor

VStr array construction.

VStrArrayPtr VSTR_ArrayAlloc (void);

Allocates a VStr array.

void VSTR_ArrayConstructVStrArray(VStrArrayPtr *va***, 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.

void VSTR_ArrayDestruct(VStrArrayPtr *va***);**

Index