Apple
PRESS

OpenDoc Cookbook

For the Mac OS

OpenDoc

Addison-Wesley Publishing Company

Reading, Massachusetts Menlo Park, California New York
Don Mills, Ontario Wokingham, England Amsterdam Bonn
Sydney Singapore Tokyo Madrid San Juan
Paris Seoul Milan Mexico City Taipei

Apple Computer, Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

ISBN 0-201-47956-7
1 2 3 4 5 6 7 8 9-MA-0099989796
First Printing, January 1996

Library of Congress Cataloging-in-Publication Data

Apple Computer, Inc.
OpenDoc cookbook for the Mac OS.
p. cm.
Includes index.
ISBN 0-201-47956-7
1. Cross-platform software development. 2. Macintosh (Computer)-Programming. I. Title.
QA76.76.D47A64 1996
005.7--dc20 95-47803
CIP

Contents

Chapter 2 SamplePart Tutorial 27

Listings

Chapter 2 SamplePart Tutorial

Appendix B System Object Model

About This Book

This book, the *OpenDoc Cookbook for the Mac OS,* presents tutorial information that explains how to create an OpenDoc part editor.

To understand this book thoroughly, you should also read the *OpenDoc Programmer's Guide for the Mac OS* and the *OpenDoc Class Reference for the Mac OS.* The *Programmer's Guide* provides an architectural overview, synthesizes design concepts, and gives specific programming recommendations. The *Class Reference* provides complete reference information about the classes, methods, types, constants, and exceptions defined by OpenDoc.

Who Should Read This Book

This book is written for software developers who wish to write OpenDoc part editors for the Mac OS platform. It consists primarily of code samples with prose explanations presenting background information, explication of details, and cross-references. This book presents a starting point for part developers: its code base, SamplePart, is a non-embedding part editor that implements a complete but minimum set of features.

This book covers the basic protocols common to all part editors. It does not cover advanced features, including embedding, data interchange (through drag and drop and linking), and scripting. It does, however, describe other code samples that illustrate some of these features.

The code samples appearing in this book are distributed as text files on a CD-ROM disk included with the *OpenDoc Programmer's Guide for the Mac OS*.

Structure of This Book

Following this preface, this book includes a brief chapter describing the MPW-based development environment as configured to compile the OpenDoc code samples. The next chapter is a tutorial presentation of SamplePart, a sample part editor developed by the OpenDoc engineering team for the Mac OS. The SamplePart tutorial is the largest portion of the book. The next chapter contains descriptions of other code samples included with OpenDoc for the Mac OS, relating those samples to certain concepts of OpenDoc not covered in the SamplePart tutorial. Last, this book presents two appendixes: one describes a set of utility classes, functions, and macros which, although unsupported, are included with OpenDoc for the Mac OS; the other appendix presents an introduction to the System Object Model™ (SOM™) technology on which OpenDoc is built, described in terms of the SOM interface of SamplePart.

Most of the methods described in this book include an introductory paragraph or two, followed by a step-by-step presentation of the method's algorithm, followed by a listing of the implementation source code. Some descriptions, especially those for brief or obvious methods, omit the step-by-step presentation.

Typographic Conventions

This book uses various conventions to present certain types of information.

Special Font

All code listings, reserved words, and the names of data structures, classes constants, fields, parameters, methods, and functions are shown in Letter Gothic (`this is Letter Gothic`).

Types of Notes

There are two types of notes used in this book, which are formatted like the following two paragraphs.

Note
A note formatted like this contains information that is interesting but possibly not essential to an understanding of the main text. ◆

IMPORTANT

A note like this contains information that is especially important. ▲

Coding Conventions

Following are some conventions that apply to the code samples in this book.

Identifier Names

The listings that appear in this book embody certain naming conventions designed to indicate the type and usage of identifiers. These conventions and examples of each are as follows:

OpenDoc classes begin with `OD`	`ODFrame`
Locally defined classes begin with `C`	`CFocus`
Virtual base classes begin with `V`	`VMyVirtualClass`
Members begin with `f`	`fDisplayFrames`
Constants begin with `k`	`kODSmallIconSize`
Functions begin with a capital	`LoadIcons`
Getter and setter methods begin with `Set`, `Get`, or `Is`	`GetViewType`
Static variables begin with `g`	`gMenuBar`
Static data members begin with `fg` (includes class globals)	`fgGlobalVar`
Enumeration types begin with `E`	`EColorType`

Class Definitions

Class definitions appearing in header files contain only the data members and method declarations; they contain no implementation. Inline methods for getters and setters, however, appear in the header file.

Developer Products and Support

The organizations described in this section are sources of useful tools and information for OpenDoc part developers.

APDA

APDA is Apple Computer's worldwide source for hundreds of development tools, technical resources, training products, and information for anyone interested in developing applications for Apple Computer platforms. Customers receive the *Apple Developer Tools Catalog* featuring all current versions of Apple development tools and the most popular third-party development tools. APDA offers convenient payment and shipping options, including site licensing.

To order products or to request a complimentary copy of the *Apple Developer Tools Catalog*, contact

APDA
Apple Computer, Inc.
P.O. Box 319
Buffalo, NY 14207-0319

Telephone	1-800-282-2732 (United States) 1-800-637-0029 (Canada) 716-871-6555 (International)
Fax	716-871-6511
AppleLink	APDA
America Online	APDAorder
CompuServe	76666,2405
Internet	APDA@applelink.apple.com

If you provide commercial products and services, call 408-974-4897 for information on the developer support programs available from Apple.

CI Labs

OpenDoc is presented and maintained through an organization devoted to promoting cross-platform standards, architectures, and protocols in a vendor-independent fashion. This organization, Component Integration Laboratories (CI Labs), is composed of a number of platform and application vendors with a common interest in solving OpenDoc issues and promoting interoperability.

CI Labs supports several levels of participation through different membership categories. If you are interested in shaping the future direction of component software, or if you simply need to be kept abreast of the latest developments, you can become a member. For an information packet, send your mailing address to

Component Integration Laboratories
PO Box 61747
Sunnyvale, CA 94088-1747

Telephone	408-864-0300
Fax	408-864-0380
Internet	cilabs@cilabs.org

Development Environment

Contents

This chapter describes the OpenDoc development environment used in this book. The OpenDoc development release ships with build support for Macintosh Programmer's Workshop (MPW). The following sections describe how to set up and use MPW to compile SamplePart.

Setting Up

These instructions assume that you have installed MPW Pro 19 (or a later version) with System Object Model™ (SOM™) headers and libraries in their proper locations. MPW is available on both the E.T.O. (Essentials, Tools, and Objects) and MPW Pro CD series from APDA. Refer to the preface of this book for information about APDA.

Documentation provided with the OpenDoc software development kit gives specific recommendations about system requirements.

OpenDoc Build Support

OpenDoc sample code support for MPW, provided with the sample code, consists of two items:

- the UserStartup•OpenDoc file
- the Build Support folder

Both of these items should be placed at the root level of your MPW folder. The UserStartup•OpenDoc script sets up pathname variables for the OpenDoc interfaces and utilities, and it adds the Build Support folder to the MPW command path. The Build Support folder contains build scripts and makefiles for the MPW compilers with which you can build OpenDoc parts. The Build Support folder also contains files that set build variables correctly for the various compilers.

Building SamplePart

This section explains how to build an executable SamplePart shared library.

Using the Build Script

The Build Support folder includes a build script named `BuildOpenDocPart`. You execute the build script with two required arguments: the path to the makefile you wish to use, and a list of the compilers you want to run, in order, in a comma-separated list. You can also specify certain options. The build script command syntax appears as follows:

`BuildOpenDocPart -f` *makefile* `-b` *compiler(s)* [*options*]

makefile is an MPW pathname, absolute or relative to the current directory.

compiler(s) is a list of one or more compilers, specified by their MPW tool filenames. The compilers you can use to build OpenDoc parts are specified as follows:

`Idl`	To generate .xh, .xih, and .cpp files
`Rez`	To generate .rsrc files
`SCpp`	To build part with SCpp compiler
`SC`	To build part with SC compiler
`MrCpp`	To build part with MrCpp compiler
`MrC`	To build part with MrC compiler

options if any, can be one or more of the following:

`-fat`	Merge the 68K and PowerPC shared libraries
`-k`	Rebuild all source files
`-nopch`	Don’t compile code using precompiled headers
`-toco` *option*	Temporarily override current setting of compiler option, where *option* is the option to be overridden, with the setting

to use for this compile, specified between straight double quotation marks. For example,

```
-toco "-d OptimizationOption=speed"
```

Examples

This section shows some example invocations of the `BuildOpenDocPart` script.

The following command line performs a Rez build as needed. That is, the Rez resource compiler is invoked to process newly changed source files according to the dependency rules in the SamplePart.make makefile:

```
BuildOpenDocPart -b rez ∂
        -f '8100:OpenDoc:SampleCode:SamplePart:SamplePart.make'
```

(The character ∂ is the MPW script language continuation symbol; it causes the MPW Shell to execute the two example lines as one.)

The following command line performs a full Interface Definition Language (IDL) build, then a full Rez build, then a full SCpp build:

```
BuildOpenDocPart -b idl,rez,scpp -k ∂
        -f '8100:OpenDoc:SampleCode:SamplePart:SamplePart.make'
```

The following command line performs an SCpp build, as needed, then performs a MrCpp build, as needed, then creates a fat binary.

```
BuildOpenDocPart -b scpp,mrcpp -fat ∂
        -f '8100:OpenDoc:SampleCode:SamplePart:SamplePart.make'
```

The `-fat` option used in the preceding example merges 68K and PowerPC shared libraries into a fat binary file that will run in native mode on either 68K or PowerPC systems. This option does not drive the build itself but requires the targets to be previously built, as they are in this example, as specified by the `-b` argument.

Setting OpenDoc Flags

Building OpenDoc parts requires setting certain flags, compiler symbols that must be defined as specified in the file CompDefs.h. The following definition removes SOM-related debug statements from generated code:

```
#define _RETAIL
```

The `_RETAIL` setting in CompDefs.h controls the definition of the `ODDebug` symbol, required by the Exception Handling (Except) and Debugging (ODDebug) utilities. The `_RETAIL` setting also controls traceback symbol generation for the PowerPC version of Macsbug. The following logic controls these settings:

```
#ifdef _RETAIL
    #ifndef ODDebug
    #define ODDebug 0
    #endif
#else
    #ifndef ODDebug
    #define ODDebug 1
    #endif
    #pragma traceback on
#endif
```

The following definitions indicate that the source code does not use obsolete Mac OS routine names and data structures:

```
#define OLDROUTINENAMES 0
#define OLDROUTINELOCATIONS 0
```

The following definition enables the compiler to include in certain header files only structures useful to the Mac OS platform:

```
#define _PLATFORM_MACINTOSH_ 1
```

The following definition specifies the endian format of the Mac OS platform for the Standard Type I/O (StdTypIO) utility:

```
#define _PLATFORM_BIG_ENDIAN_ 1
```

Using Precompiled Headers

Using precompiled headers can significantly shorten compile time when there have been no changes to included files. The Build Support folder contains two header files from which a precompiled header can be generated: the file SCPCHeaders++.pch (for C++ compilers) and the file SCPCHeaders.pch (for the C compilers). These files include OpenDoc headers, OpenDoc utilities, and Mac OS Toolbox headers required to build SamplePart and the other OpenDoc official samples.

Installing OpenDoc

This section describes installation of OpenDoc on the Mac OS.

Installer

OpenDoc ships with an installer script and installer application to simplify the installation procedure. You need only drag the installer script onto the installer application for it to put all of OpenDoc's components in their correct locations on your hard disk.

Editors Folders

The OpenDoc installer application creates several folders when OpenDoc is being installed. You should put part editor and part viewer shared library files into the Editors folder, which the installer puts into the System folder. Editors folders may also be located at the root of any mounted volume. This allows you to install part editors on a volume other that the startup volume. It also allows editors to reside on shared volumes or even on floppy disks. OpenDoc also scans subfolders in any of these recognized Editors folders for editors.

Resource Cache

To speed up launching, OpenDoc stores the name-mapping (`'nmap'`) resources of all editors in an Editors folder in a single cache file. The cache file is invisible and is located in the Editors folder. The cache is invalidated and regenerated when the modification date of any folder that contains editors changes. When this happens OpenDoc displays a small dialog box reading `Updating OpenDoc editors database`.

Use of this cache has no effect on users, but it shaves several seconds from document launch times. While you are developing part editors, however, you must realize that simply modifying an editor library in the Editors folder (for example, editing its `'nmap'` resources or recompiling the editor or its resources) does not cause OpenDoc to rescan the editor and load the new `'nmap'` resource because modifying a file in a directory does not change the directory's modification date. To ensure that OpenDoc reads the changed `'nmap'` resource, move the editor out of the directory and back in.

Aliases

Aliases to files and folders are permitted in the Editors folder. However, aliases to files or folders on other volumes are not permitted. In fact, OpenDoc moves these illegal aliases to the trash. Such aliases should be put in the Editors folder on the aliases' destination volume. All editors logically contained in a single Editors folder must be on a single volume.

Apple Guide Help Files

Providing Apple Guide support for an editor requires implementing a help file to be installed along with the editor shared library file. Apple Guide help files must be installed in the same folder as the editor itself. In addition, the editor shared library must include an `'nmap'` resource that specifies the name of the help file, linking it to the class identifier of the part editor.

The Stationery Folder

The Stationery folder is created by the OpenDoc installer at the root level of the startup volume. When you create stationery (by dropping an editor shared library on the OpenDoc application) OpenDoc places the stationery file in the Stationery folder.

Installing and Running Part Editors

This section explains how to install and run OpenDoc part editors.

Installing Part Editors

The result of building your part (that is, compiling your source code and linking it with the OpenDoc and Mac OS system libraries) is a shared library file. You should place this file in the Editors folder as described in the previous section.

Creating Stationery

You create stationery for your part editor by dropping the shared library file resulting from your build process on the OpenDoc application. The OpenDoc installer places the OpenDoc application in a folder named OpenDoc Libraries, which is inside the Extensions folder in the Mac OS System folder. OpenDoc places the stationery file in the Stationery folder at the root level of the startup volume.

Creating Documents

On the Mac OS platform, users create documents in three ways: by launching a part editor's stationery file from the Finder (double-clicking or selecting and opening), by choosing the New command from the Document menu when an OpenDoc document is running (the new document is the same kind as the root part of the frontmost window), or by dragging a selected content object or embedded part to the desktop in the Finder (the new document has the dragged part or content as its root part).

The new document is created in the same folder as the previous document or, if that is not possible, on the desktop. The new document is named by the category of the root part or the name of the stationery from which it is created. If more than one document by this name would exist, the new document name has an integer appended indicating its place.

Running Parts

Mac OS users launch documents by double-clicking them or selecting and opening them in the Finder. OpenDoc locates and launches each part editor required to display and manipulate the root part of the document and each part, if any, embedded within it.

SamplePart Tutorial

Contents

This chapter presents a tutorial that shows how to implement SamplePart, a part editor with the basic feature set common to all OpenDoc part editors.

Features of SamplePart

SamplePart implements a complete but minimum set of features. Although it's possible to write an executable part editor with even fewer features, it would not be very useful.

From the user's point of view, SamplePart's primary capability is simply to display a text string. It can also display itself in small icon, large icon, and thumbnail views. SamplePart also supports the Save command and, when embedded in a container part, the View as Window command.

In order to support its feature set and interact properly with other parts, SamplePart performs the following tasks, which are described in this chapter:

- initialization
- opening the part into a window
- handling part layout
- drawing the part's content
- handling basic events
- activation
- writing the part to persistent storage

In addition, this chapter shows how to set up your part editor's resources so that OpenDoc can match it with its parts.

SamplePart Structure

SamplePart is implemented primarily in a single C++ class, which is described in this chapter. It also uses a set of its own utility functions, collection classes, and various utilities provided with the Mac OS implementation of OpenDoc.

SamplePart System Object Model Interface

The System Object Model (SOM) is a standard object infrastructure upon which the OpenDoc component software architecture is built. All OpenDoc part editors are represented to OpenDoc by a subclass of `ODPart`, which is a SOM class. The interface to a SOM class is written in the SOM Interface Definition Language (IDL) and adheres to certain protocols specific to SOM.

SamplePart incorporates a scheme by which the part's SOM interface is largely hidden from the programmer. SamplePart has only one SOM class, which is a subclass of `ODPart`, referred to as *the SOM wrapper class.* This SOM class overrides all `ODPart` methods, although SamplePart implements only some of them. For those methods that SamplePart implements, the SOM wrapper class methods delegate the implementation to a C++ class that provides the capabilities of SamplePart.

The SOM wrapper class is named `som_SamplePart`, and it is defined in IDL. The SOM class methods merely call corresponding methods in the C++ class, which is named `SamplePart`. For `ODPart` methods that the `SamplePart` class does not implement, the SOM class override method bodies are empty. They are provided so that you can extend SamplePart simply by adding a call to a method in a C++ class—you do not need to use the SOM compiler or revise the SOM class IDL interfaces.

Generally, you can use SamplePart's SOM interface as provided in the sample code base—you don't need to understand SOM in order to understand SamplePart. For an introduction to IDL that describes SOM artifacts found in the definition of the `som_SamplePart` class, refer to Appendix B.

Calling Inherited Methods

For `ODPart` override methods that require calling the parent class implementation, the call is made in the SOM class implementation. To know whether you need to call the parent class, see the code for the `som_SamplePart` wrapper class (in which the inherited method calls are made) and the *OpenDoc Class Reference* (which explains for each method how its inherited method should be called).

SOM Wrapper Class and Part Wrapper Object

The SOM wrapper class is not the part wrapper object described in the *OpenDoc Programmer's Guide for the Mac OS.* The part wrapper object is a private object that OpenDoc instantiates and uses to represent the part editor.

OpenDoc passes a reference to the part wrapper object to the part editor in its `InitPart` or `InitPartFromStorage` method, as described in "The InitPart Method" on page 40.

SamplePart File Structure

The primary source files composing the SamplePart program are the following:

SamplePart.h	`SamplePart` class definition
SamplePart.cpp	`SamplePart` method implementations
SamplePartDef.h	Constant definitions
SamplePartUtils.h	Utility class definitions
SamplePartUtils.cpp	Utility method implementations
SamplePartGlobals.h	Global variables structure definition
SamplePartGlobals.cpp	External global variables initialization
SamplePartVers.h	Version definitions
SampleCollections.h	Collection class definitions
SampleCollections.cpp	Collection method implementations
SamplePart.r	Resource definitions
SamplePartOtherResources.rsrc	Other resources used by SamplePart
CompDefs.h	Defines for compiling SamplePart

The source files for SamplePart's SOM interface are the following:

som_SamplePart.idl	SOM wrapper class definition
som_SamplePart.xh	SOM-emitted public headers
som_SamplePart.xih	SOM-emitted private headers
som_SamplePart.cpp	SOM wrapper method implementations
som_SamplePartInit.cpp	CFM initialization function
som_SamplePart.exp	SOM-emitted class export symbols

SamplePart Class Definition

Most of the SamplePart implementation is contained within a single C++ class called `SamplePart`. The public methods declared in this class correspond exactly to methods of the same name in `som_SamplePart`, all of which override methods of `ODPart`. The protected methods are subroutines internal to SamplePart called in the implementation of the public methods. The private members of `SamplePart` are its data members.

Listing 2-1 shows the complete class definition of `SamplePart`.

Listing 2-1 `SamplePart` class definition

```
class SamplePart {

    public:

    SamplePart();
    virtual ~SamplePart();

    // -- Initialization --
    void    InitPart(Environment* ev, ODStorageUnit* storageUnit,
                                       ODPart* partWrapper);
    void    InitPartFromStorage(Environment* ev, ODStorageUnit* storageUnit,
                                       ODPart* partWrapper);

    // -- Storage --
    void    Release(Environment* ev);
    void    ReleaseAll(Environment* ev);
    ODSize  Purge(Environment* ev, ODSize size);
    void    Externalize(Environment* ev);
    void    ExternalizeKinds(Environment* ev, ODTypeList* kindset);
    void    ChangeKind(Environment* ev, ODType kind);
    void    CloneInto(Environment* ev, ODDraftKey key,
                                       ODStorageUnit* destinationSU,
                                       ODFrame* initiatingFrame);
    void    WritePartInfo(Environment* ev, ODInfoType partInfo,
                                       ODStorageUnitView* storageUnitView);
    ODInfoType ReadPartInfo(Environment* ev, ODFrame* frame,
                                       ODStorageUnitView* storageUnitView);
    void    ClonePartInfo(Environment *ev, ODDraftKey key, ODInfoType partInfo,
                                       ODStorageUnitView* storageUnitView,
                                       ODFrame* scopeFrame);

    // -- Layout --
    void    DisplayFrameAdded(Environment* ev, ODFrame* frame);
    void    DisplayFrameRemoved(Environment* ev, ODFrame* frame);
    void    DisplayFrameClosed(Environment* ev, ODFrame* frame);
    void    DisplayFrameConnected(Environment* ev, ODFrame* frame);
```

```
void    AttachSourceFrame(Environment* ev, ODFrame* frame,
                                    ODFrame* sourceFrame);
void    ViewTypeChanged(Environment* ev, ODFrame* frame);
void    FrameShapeChanged(Environment* ev, ODFrame* frame);
ODID    Open(Environment* ev, ODFrame* frame);

// -- Imaging --
void    Draw(Environment* ev, ODFacet* facet, ODShape* invalidShape);
void    GeometryChanged(Environment* ev, ODFacet* facet,
                                    ODBoolean clipShapeChanged,
                                    ODBoolean externalTransformChanged);
void    HighlightChanged(Environment* ev, ODFacet* facet);
void    FacetAdded(Environment* ev, ODFacet* facet);
void    FacetRemoved(Environment* ev, ODFacet* facet);

// -- Activation --
ODBoolean BeginRelinquishFocus(Environment* ev, ODTypeToken focus,
                                    ODFrame* ownerFrame,
                                    ODFrame* proposedFrame);
void    CommitRelinquishFocus(Environment* ev, ODTypeToken focus,
                                    ODFrame* ownerFrame,
                                    ODFrame* proposedFrame);
void    AbortRelinquishFocus(Environment* ev, ODTypeToken focus,
                                    ODFrame* ownerFrame,
                                    ODFrame* proposedFrame);
void    FocusAcquired(Environment* ev,ODTypeToken focus,
                                    ODFrame* ownerFrame);
void    FocusLost(Environment* ev,ODTypeToken focus,
                                    ODFrame* ownerFrame);

// -- Event handling --
ODBoolean HandleEvent(Environment* ev, ODEventData* event,
                                    ODFrame* frame, ODFacet* facet,
                                    ODEventInfo* eventInfo);
void    AdjustMenus(Environment* ev, ODFrame* frame);

protected:

// -- Initialization --
void    Initialize(Environment* ev);
```

```
// -- Storage --
void    CheckAndAddProperties(Environment* ev,
                              ODStorageUnit* storageUnit);
void    CleanseContentProperty(Environment* ev,
                              ODStorageUnit* storageUnit);
void    InternalizeStateInfo(Environment* ev,
                              ODStorageUnit* storageUnit);
void    InternalizeContent(Environment* ev,
                              ODStorageUnit* storageUnit);
void    ExternalizeStateInfo(Environment* ev,
                              ODStorageUnit* storageUnit,
                              ODDraftKey key, ODFrame* scopeFrame);
void    ExternalizeContent(Environment* ev, ODStorageUnit* storageUnit,
                              ODDraftKey key, ODFrame* scopeFrame);
void    SetDirty(Environment* ev);

// -- Event Handling --
ODBoolean HandleMenuEvent(Environment* ev, ODEventData* event,
                              ODFrame* frame);
ODBoolean HandleMouseEvent(Environment* ev, ODEventData* event,
                              ODFacet* facet, ODEventInfo* eventInfo);
void    DoMouseEvent(Environment* ev, ODFacet* facet, Point* where);
void    DoDialogBox(Environment* ev, ODFrame* frame,
                              ODSShort dialogID, ODUShort errorNumber = 0);

// -- Imaging --
void    DrawFrameView(Environment* ev, ODFacet* facet);
void    DrawIconView(Environment* ev, ODFacet* facet);
void    DrawThumbnailView(Environment* ev, ODFacet* facet);
void    GenerateThumbnail( Environment* ev, ODFrame* frame );

// -- Activation --
void    PartActivated(Environment* ev, ODFrame* frame);
ODBoolean ActivateFrame(Environment* ev, ODFrame* frame);
void    WindowActivating(Environment* ev, ODFrame* frame,
                              ODBoolean activating);
void    RelinquishAllFoci(Environment* ev, ODFrame* frame);

// -- Layout --
ODWindow* AcquireFramesWindow(Environment* ev, ODFrame* frame);
ODWindow* CreateWindow(Environment* ev, ODFrame* frame, ODType frameType,
```

```
                                WindowProperties* windowProperties);
    void    CleanupWindow(Environment* ev, ODFrame* frame);
    WindowProperties* GetDefaultWindowProperties(Environment* ev, ODFrame* frame,
                                Rect* windowRect);
    WindowProperties* GetSavedWindowProperties(Environment* ev, ODFrame* frame);
    Rect    CalcPartWindowSize(Environment* ev, ODFrame* sourceFrame);
    Rect    CalcPartWindowPosition(Environment* ev, ODFrame* frame,
                                Rect* partWindowBounds);
    ODFacet* GetActiveFacetForFrame(Environment* ev, ODFrame* frame);
    ODShape* CalcNewUsedShape(Environment* ev, ODFrame* frame);
    void    UpdateFrame(Environment* ev, ODFrame* frame, ODTypeToken view,
                                ODShape* usedShape);
    void    CleanupDisplayFrame(Environment* ev, ODFrame* frame,
                                ODBoolean frameRemoved);


    private:

    CList*      fDisplayFrames;
    ODBoolean   fDirty;
    ODPart*     fSelf;
    ODBoolean   fReadOnlyStorage;
};
```


Shared Global Variables

In addition to the method and instance variables declared in Listing 2-1, SamplePart uses a set of global variables, declared as members of a C++ structure. These variables are shared among all the currently running instances of the `SamplePart` object in a single document. In addition, SamplePart maintains two separate global variables to provide access the shared globals: a pointer to the global variables structure, and a count of the number of instances of the `SamplePart` class currently using the global variables.

The global variables are defined and initialized in the files SamplePartGlobals.h and SamplePartGlobals.cpp. The global variables structure is allocated in temporary memory by the `Initialize` method (see Listing 2-6 on page 45).

The global variables structure definition is shown in Listing 2-2.

Listing 2-2 SamplePart global variables

```
struct SamplePartGlobals; // forward

extern ODUShort             gGlobalsUsageCount;
extern SamplePartGlobals*   gGlobals;


struct SamplePartGlobals {
    public:
    SamplePartGlobals();
    ~SamplePartGlobals() {}

    ODMenuBar*          fMenuBar;
    ODFocusSet*         fUIFocusSet;
    Handle              fThumbnail;

    ODTypeToken         fSelectionFocus;
    ODTypeToken         fMenuFocus;
    ODTypeToken         fModalFocus;
    ODTypeToken         fFrameView;
    ODTypeToken         fLargeIconView;
    ODTypeToken         fSmallIconView;
    ODTypeToken         fThumbnailView;
    ODTypeToken         fMainPresentation;

    ODScriptCode        fEditorsScript;
    ODLangCode          fEditorsLanguage;
};

inline SamplePartGlobals::SamplePartGlobals()
{
    fMenuBar        = kODNULL;
    fUIFocusSet     = kODNULL;
    fThumbnail      = kODNULL;

    fSelectionFocus         = kODNullTypeToken;
    fMenuFocus              = kODNullTypeToken;
    fModalFocus             = kODNullTypeToken;
    fFrameView              = kODNullTypeToken;
    fLargeIconView          = kODNullTypeToken;
```

```
    fSmallIconView          = kODNullTypeToken;
    fThumbnailView          = kODNullTypeToken;
    fMainPresentation       = kODNullTypeToken;

    fEditorsScript      = 0;
    fEditorsLanguage    = 0;
}

ODUShort            gGlobalsUsageCount  = 0;
SamplePartGlobals*  gGlobals            = kODNULL;
```


Initialization

The first responsibility of a part editor is initialization. When the user launches a document, either preexisting or newly created from stationery, OpenDoc instantiates the part object belonging to each currently visible part in the document. In SamplePart, the part object is an instance of the `som_SamplePart` class, which is a subclass of `ODPart`. At that time, the SOM runtime system calls the part object's `somInit` method.

The `SamplePart` object's `somInit` method, belonging to `som_SamplePart`, does nothing. The SOM runtime system automatically calls the inherited `somInit` methods, in the manner of a C++ constructor. SOM automatically zeroes the instance variables of a newly constructed SOM object, so there is no need to do so in the `somInit` method.

Next, OpenDoc calls one of the part object's initialization methods. If the part is creating stationery, OpenDoc calls the `InitPart` method of the part object; if the part was previously created, either as the root part or embedded in a document, OpenDoc calls the part's `InitPartFromStorage` method. In SamplePart, these methods instantiate the `SamplePart` C++ class, call their inherited methods, and call the `SamplePart` object's methods of the same name. When the `SamplePart` class is instantiated, the C++ runtime system calls its constructor, which does set instance variables to zero, as shown in Listing 2-3.

In SamplePart, initialization code resides in four methods: the `SamplePart` constructor, the `InitPart` method, the `InitPartFromStorage` method, and the internal `Initialize` method. The `Initialize` method contains the code that is common to both initialization situations: initializing a part when creating stationery (when OpenDoc calls `InitPart`) and initializing a part previously created and written to persistent storage (when OpenDoc calls `InitPartFromStorage`). Both of those methods call `Initialize`. The following sections discuss the implementation of these methods.

The Constructor

In SamplePart, the constructor performs only one action: it sets initial values for the `SamplePart` object's private data fields. You should not do anything in the constructor that can fail, such as allocating memory. The `SOM_Trace` macro call indicates the name of the method currently executing for debugging purposes.

Listing 2-3 shows the `SamplePart` object's constructor.

Listing 2-3 `SamplePart` constructor

```
SamplePart::SamplePart()
{
    SOM_Trace("SamplePart","Constructor");

    fDisplayFrames  = kODNULL;
    fDirty          = kODFalse;
    fSelf           = kODNULL;
    fReadOnlyStorage = kODFalse;
}
```

The InitPart Method

If the part is and has no stored data, OpenDoc calls the `InitPart` method after it instantiates the part object. Every part must implement this method. You can do things that might fail in this method, such as allocating extra storage, setting up your storage unit, and getting resources if you need them.

As with all methods in SamplePart, the implementation is delegated. That is, OpenDoc calls the `InitPart` method belonging to the `ODPart` subclass, which in turn calls the `InitPart` method of the `SamplePart` object, which contains the method's implementation. For more information, refer to "SamplePart System Object Model Interface" on page 32.

The implementation of the `InitPart` method is contained within an exception handler, a block of code delimited by the macro calls `TRY` and `ENDTRY`. When the body of the method executes, the statements following the `TRY` macro execute; if any of them causes an exception to be thrown, the statements following the `CATCH_ALL` macro execute. The `RERAISE` macro causes the exception to be thrown again to the caller of `InitPart`. If no exception is thrown, control passes to the statement following the `ENDTRY` macro call (the end of the method body in this case). For more information about the OpenDoc exception-handling utility, see Appendix A, "OpenDoc Utilities."

The `SamplePart` implementation of the `InitPart` method performs the following actions:

1. **Initializes the part-wrapper field.**

   OpenDoc passes a pointer to its internal representation for the part editor, its part wrapper, when it calls the `InitPart` method, and the `SamplePart` object stores the pointer in its `fSelf` data member.

   OpenDoc uses the part wrapper in place of a pointer to the actual part object to enable swapping part editors at runtime for part translation. Wherever OpenDoc requires a reference to the part editor, such as when registering for idle time, you must pass the part wrapper pointer, rather than passing `this` (from the `SamplePart` C++ object) or `somSelf` (from the `som_SamplePart` object).

2. **Ensures that the part's destination storage is writable.**

   OpenDoc calls the method when a part is first instantiated, so we must be able to write part status and content information to its storage unit.

3. **Calls the common initialization code.**

   Initialization code common to `InitPart` and `InitPartFromStorage` resides in the internal `Initialize` method. The `Initialize` method is described in "The Initialize Method" on page 44.

4. **Sets the dirty flag.**

   Setting the dirty flag to `kODTrue` enables SamplePart to write out its state and content information at the next opportunity.

If any of the called methods throws an exception, the `CATCH_ALL` method puts the error code in SOM's `Environment` structure. Cleanup occurs in the `SamplePart` destructor.

Listing 2-4 shows the implementation of the `InitPart` method.

Listing 2-4 `InitPart` method

```
void SamplePart::InitPart( Environment*     ev,
                           ODStorageUnit*   storageUnit,
                           ODPart*          partWrapper )
{
    SOM_Trace("SamplePart","InitPart");

    TRY
        fSelf = partWrapper;
        fReadOnlyStorage = kODFalse;
        this->Initialize(ev);
        this->SetDirty(ev);
    CATCH_ALL
        RERAISE;
    ENDTRY
}
```

The InitPartFromStorage Method

If a part has previously been stored persistently, OpenDoc calls the `InitPartFromStorage` method, instead of `InitPart`, after it instantiates the part object. This situation occurs when a document or stationery is opened or when the part is embedded and its containing part reads it into memory. So, every part must also implement this method, which should do many of the same things as `InitPart`, but which must also handle reading content and status information from the storage unit into memory.

The part must not alter its storage unit in this method; if it does so, the document's Save menu item becomes enabled without the user having changed the document.

The `SamplePart` object's implementation of the `InitPartFromStorage` method performs the following actions:

1. **Initializes the part-wrapper field.**

   The method puts the part-wrapper pointer passed in from OpenDoc into the private `fSelf` data member.

2. **Determines if the draft from which the part is being opened is read only.**

   If the draft permissions are read only, the part must not write any data back to its storage unit. The method sets the part's private `fReadOnlyStorage` Boolean flag accordingly, to be checked before writing data in the `Externalize` method.

3. **Calls the common initialization code.**

   Initialization code common to `InitPart` and `InitPartFromStorage` resides in the internal `Initialize` method. The `Initialize` method is described in "The Initialize Method" on page 44.

4. **Reads the part's status information.**

   Because the part was previously written to its storage unit, `InitPartFromStorage` reads in the part's status information by calling the internal `InternalizeStateInfo` method, which is described in "The InternalizeStateInfo Method" on page 106.

5. **Reads the part's content value from the storage unit.**

   In SamplePart, the internal method that would read in the part's content value, `InternalizeContent`, does nothing, because SamplePart has no intrinsic content. A brief discussion of the method appears in "The InternalizeContent Method" on page 105.

Listing 2-5 shows the implementation of the `InitPartFromStorage` method.

Listing 2-5 `InitPartFromStorage` method

```
void SamplePart::InitPartFromStorage( Environment*      ev,
                                      ODStorageUnit*    storageUnit,
                                      ODPart*           partWrapper )
{
    SOM_Trace("SamplePart","InitPartFromStorage");

    TRY
        fSelf = partWrapper;
        fReadOnlyStorage = ( ODGetDraft(ev,storageUnit)->
                                GetPermissions(ev) < kODDPSharedWrite );
        this->Initialize(ev);
        this->InternalizeStateInfo(ev, storageUnit);
        this->InternalizeContent(ev, storageUnit);
```

```
    CATCH_ALL
        RERAISE;
    ENDTRY
}
```

The Initialize Method

The `Initialize` method is internal to the `SamplePart` class. OpenDoc doesn't call `Initialize`; both `InitPart` and `InitPartFromStorage` call it. The `Initialize` method contains the initialization code that is common to both situations, whether the part is newly created or is to be read in from persistent storage.

The `Initialize` method performs the following actions:

1. **Creates a frame list collection object.**

   The frame list collection object (`CList`) is necessary to keep track of the multiple display frames in which the part displays its content. The class is defined in the SamplePart utilities file SampleCollections.h.

2. **Checks the usage count of the SamplePart global variables.**

   If the usage count is not equal to zero, another instance of this part object is running. In that case, the following initialization steps have already been done and can be skipped. Otherwise, the method performs the following steps and sets the global variables usage count to 1.

3. **Stores a reference to the OpenDoc session object.**

   This is a convenience, because the session object provides access to session-wide global objects and services such as the window-state object and unique name tokenization. Note that the self-reference passed with the `ODGetSession` call is the part-wrapper object passed in by OpenDoc to `InitPart` or `InitPartFromStorage`.

4. **Creates the global variables structure.**

   The global variables structure is described in "Shared Global Variables" on page 37.

5. **Instantiates the part's menu bar.**

   The part editor instantiates its menu bar by copying OpenDoc's session-wide menu bar, a base menu bar object maintained by the window-state object. That action maintains consistency in the arrangement of default menu items. Also, because its menu bar is a copy, this part editor can add and subtract menus and items without affecting the menu bars of other parts.

Note that the menu bar object is shared among all the currently running instances of `SamplePart` in this document by virtue of its declaration in the shared global variables structure shown in Listing 2-2 on page 38.

6. **Tokenizes and stores values for the foci the part needs.**

   The tokens are used for equivalence tests in the part activation methods and for requesting foci from the arbitrator. The method also packages into a set the three user-interface foci required by the part editor when it is activated, so it can request them all at once. The tokenized foci values are stored in the part's global variables.

7. **Tokenizes view types and presentation type.**

   The method tokenizes the four view types that all part editors must support and the part editor's main presentation type. The method tokenizes these strings for convenience, because tokens are faster to handle than strings.

8. **Determines the script and language to which the part is localized.**

   The `GetEditorScriptLanguage` utility function is defined in the SamplePart utilities file SamplePartUtils.cpp.

The final logic of the `Initialize` method manages SamplePart's global variables usage count, which was mentioned in step 2. If the globals usage count was not equal to zero at that step, then another instance of the part is already running, and this instance can use the same tokens, focus set, and menu bar object. In that case, the method merely increments the global variables usage count.

Listing 2-6 shows the implementation of the `Initialize` method.

Listing 2-6 `Initialize` method

```
void SamplePart::Initialize( Environment* ev )
{
    SOM_Trace("SamplePart","Initialize");

    fDisplayFrames = new CList;

    if ( gGlobalsUsageCount == 0 )
```

```
    {
        ODSession* session = ODGetSession(ev,fSelf);
        gGlobals = new SamplePartGlobals;
        gGlobals->fMenuBar = session->GetWindowState(ev)->CopyBaseMenuBar(ev);

        gGlobals->fSelectionFocus = session->Tokenize(ev, kODSelectionFocus);
        gGlobals->fMenuFocus = session->Tokenize(ev, kODMenuFocus);
        gGlobals->fModalFocus = session->Tokenize(ev, kODModalFocus);

        gGlobals->fMainPresentation = session->Tokenize(ev, kMainPresentation);

        gGlobals->fFrameView = session->Tokenize(ev, kODViewAsFrame);
        gGlobals->fLargeIconView = session->Tokenize(ev, kODViewAsLargeIcon);
        gGlobals->fSmallIconView = session->Tokenize(ev, kODViewAsSmallIcon);
        gGlobals->fThumbnailView = session->Tokenize(ev, kODViewAsThumbnail);

        gGlobals->fUIFocusSet = session->GetArbitrator(ev)->CreateFocusSet(ev);
        gGlobals->fUIFocusSet->Add(ev, gGlobals->fMenuFocus);
        gGlobals->fUIFocusSet->Add(ev, gGlobals->fSelectionFocus);

        GetEditorScriptLanguage(ev, &gGlobals->fEditorsScript,
                                        &gGlobals->fEditorsLanguage);
        gGlobalsUsageCount = 1;
    }
    else
    {
        gGlobalsUsageCount++;
    }
}
```

After these initialization methods have executed, the SamplePart part editor is in a consistent state, ready to become active.

Opening the Part Into a Window

OpenDoc calls the `Open` method of a part editor in three cases: when the part is initially created, when the part is the root part of a document being opened, and when the part is embedded and the user opens it into a separate part window.

If the `frame` parameter has a value of `kODNULL`, then the part is being created for the first time. If the `frame` parameter points to a root frame, then an existing document is being opened. If the `frame` parameter points to a frame that is not a root frame, then an embedded frame is being opened into a part window.

The basic steps in the process of opening the part into a window are as follows:

1. If a frame pointer was passed into the `Open` call, check for an existing part window. If there is one, skip to step 4.
2. If no frame pointer was passed or the part window no longer exists, create a new window (to add the root frame).
3. Open the window (to add the root facet).
4. Show the window (to make it visible).
5. Select the window (to bring it to the front).
6. Return the window ID number.

The order of the sequence—opening, then showing, then selecting the window—is very important. In SamplePart, these steps are accomplished by the `Open` and `CreateWindow` methods, with some help from utility methods.

The Open Method

The `SamplePart` object's implementation of the `Open` method performs the following actions:

1. **Creates pointer variables for a window object and a window properties structure.**

   The `ODWindow` object is a wrapper for a platform-specific window. The `WindowProperties` object is a C structure (defined in the file WinUtils.h) to contain the attributes of a Mac OS–specific window, such as bounding rectangle, title string, and so forth.

   The method uses the macro `ODVolatile`, which is defined in the OpenDoc exception-handling utility file Except.h. This macro ensures that the variable will remain valid in the `CATCH_ALL` block after having been modified in the `TRY` block. The `ODVolatile` macro is documented in Appendix A, in the section "Make Variables That You Modify Volatile" on page 152.

2. **Handles the new document case.**

   If the `frame` parameter is null, the part must create a window for a new document. In this case, there are no saved window properties, so the method calls the `SamplePart` internal method `GetDefaultWindowProperties` to create a default set.

   Having filled in the window properties structure, the method then calls the `SamplePart` internal method `CreateWindow` to create the platform window and OpenDoc window wrapper. The `CreateWindow` method is described in "The CreateWindow Method" on page 50.

3. **Handles the existing document case.**

   If the `frame` parameter points to a root frame, the part must create a window to display the root frame of an existing document. In this case, the window properties were previously saved in a separate storage unit, to which a strong reference exists in the root frame's storage unit. The `SamplePart` internal method `GetSavedWindowProperties` retrieves the information using the `BeginGetWindowProperties` utility method defined in the file WinUtils.cpp.

   Having obtained the window properties, the method calls the `SamplePart` internal method `CreateWindow`. In this block, the method also uses the `ODReleaseObject` utility method to decrement the reference count of the frame object because it was incremented in `GetSavedWindowProperties`.

4. **Handles the embedded frame case.**

   If the `frame` parameter is not null, and it's not a root frame, then it's an embedded frame being opened into a part window. In this case, the method first tries to retrieve an existing window for the frame using the `SamplePart` internal method `AcquireFramesWindow`. A window can exist for the frame if it was previously opened into a part window.

   Otherwise, the method proceeds as in the new document case, except that it uses the frame to determine window size and property values. Finally, the method saves the part-window pointer in the frame's `CFrameInfo` object.

5. **Calls the window activation methods.**

   When it has created the window, the `Open` method calls three methods belonging to the OpenDoc window-wrapper object: `Open`, `Show`, and `Select`. The window's `Open` method creates the root facet for the window and notifies the part editor. The `Show` method makes the window visible. The `Select` method activates and selects the new window, bringing it to the front.

6. **Cleans up and returns the window's ID number to OpenDoc.**

Listing 2-7 shows the implementation of the `Open` method.

Listing 2-7 `Open` method

```
ODID SamplePart::Open( Environment*ev, ODFrame* frame )
{
    SOM_Trace("SamplePart","Open");

    ODID windowID;
    TempODWindow window(kODNULL);

    WindowProperties* windowProperties = kODNULL;
    ODVolatile(windowProperties);

    TRY
        if ( frame == kODNULL )
        {
            Rect windowRect = this->CalcPartWindowSize(ev, kODNULL);
            windowProperties = this->GetDefaultWindowProperties(ev,
                                                                kODNULL, &windowRect);
            window = this->CreateWindow(ev, kODNULL, kODFrameObject, windowProperties);
        }
        else if ( frame->IsRoot(ev) )
        {
            windowProperties = this->GetSavedWindowProperties(ev, frame);

            if ( windowProperties == kODNULL )
            {
                Rect windowRect = this->CalcPartWindowSize(ev, frame);
                windowProperties = this->GetDefaultWindowProperties(ev,
                                                                kODNULL, &windowRect);
            }

            window = this->CreateWindow(ev, frame, kODFrameObject, windowProperties);

            ODReleaseObject(ev, windowProperties->sourceFrame);
        }
        else // frame is a source frame
        {
            window = this->AcquireFramesWindow(ev, frame);
```

```
        if ( window == kODNULL )
        {
            Rect windowRect = this->CalcPartWindowSize(ev, frame);
            windowProperties = this->GetDefaultWindowProperties(ev,
                                                    frame, &windowRect);
            // Create a Mac Window and register it with OpenDoc.
            window = this->CreateWindow(ev, kODNULL,
                                    kODFrameObject, windowProperties);
            CFrameInfo* frameInfo = (CFrameInfo*) frame->GetPartInfo(ev);
            frameInfo->SetPartWindow(ev, window);
        }
    }

    window->Open(ev);
    window->Show(ev);
    window->Select(ev);

    ODDeleteObject(windowProperties);
    windowID = (window ? window->GetID(ev) : kODNULLID);

CATCH_ALL
    if ( windowProperties )
        ODSafeReleaseObject(windowProperties->sourceFrame);
    ODDeleteObject(windowProperties);
    windowID = kODNULLID;
    RERAISE;
ENDTRY

return windowID;
}
```

The CreateWindow Method

The `SamplePart` object's internal `CreateWindow` method is called by the part's `Open` method to create a window for a frame that is being opened. The method uses information passed in its `windowProperties` parameter to set the window attributes—the size of the new window, the string for its title bar, and so forth. The method then creates a Mac OS–specific window structure, and integrates the window into the OpenDoc environment by registering it in an OpenDoc window-state object, thereby creating an OpenDoc window pointer which is returned from the method.

The `CreateWindow` method performs the following actions:

1. **Creates a platform-specific window structure.**

   The method creates the window structure using the Mac OS toolbox routine `NewCWindow` and the OpenDoc memory manager utility. The `ODNewPtr` function allocates space for the window structure in temporary memory rather than the application heap.

2. **Creates an OpenDoc window object.**

   The window-state object, available through the session object, instantiates an OpenDoc window objects, which are wrappers for the platform-specific windows. OpenDoc uses the window-state object and window objects to keep track of each of the windows it handles in a platform-independent manner.

   If the `Open` method is opening a new document, `CreateWindow` calls the `ODWindowState` method `RegisterWindow` to create the OpenDoc window object and register it as a new window. To create and register the window for an existing document, the method calls the `ODWindowState` method `RegisterWindowForFrame`.

If the method fails to create the window successfully, it generates a dialog box to notify the user, using the SamplePart utility method `DoDialogBox`. It also uses the exception-handling utility `SetErrorCode` to let OpenDoc know the user was already notified of the error.

Listing 2-8 shows the implementation of the `CreateWindow` method.

Listing 2-8 `CreateWindow` method

```
ODWindow* SamplePart::CreateWindow( Environment*        ev,
                                    ODFrame*            frame,
                                    ODType              frameType,
                                    WindowProperties*   windowProperties)
{
    SOM_Trace("SamplePart","CreateWindow");

    ODPlatformWindow    platformWindow  = kODNULL;
    ODWindow*           window          = kODNULL;
```

```
platformWindow = NewCWindow((Ptr)ODNewPtr(sizeof(WindowRecord)),
                            &(windowProperties->boundsRect),
                            windowProperties->title,
                            kODFalse, /* visible */
                            windowProperties->procID,
                            (WindowPtr)-1L,
                            windowProperties->hasCloseBox,
                            windowProperties->refCon);

if ( platformWindow )
{
    TRY
        ODWindowState* windowState = ODGetSession(ev,fSelf)->GetWindowState(ev);
        ODBoolean saveWindow = (ODISOStrCompare(frameType,kODFrameObject) == 0);
        ODBoolean shouldDispose = kODFalse;

        if ( frame == kODNULL )
        {
            window = windowState->
                        RegisterWindow(ev,
                            platformWindow,                 // Mac OS WindowPtr
                            frameType,                      // Frame persistent?
                            windowProperties->isRootWindow, // Document window?
                            windowProperties->isResizable,  // Resizeable?
                            windowProperties->isFloating,   // Floating?
                            saveWindow,                     // Window persistent?
                            shouldDispose,                  // Dispose when done?
                            fSelf,                          // Self reference
                            gGlobals->fFrameView,           // What view?
                            gGlobals->fMainPresentation,    // What presentation?
                            windowProperties->sourceFrame); // Source frame, if any
        }
        else
        {
            window = windowState->
                        RegisterWindowForFrame(ev,
                            platformWindow,
                            frame,
                            windowProperties->isRootWindow,
                            windowProperties->isResizable,
```

```
                              windowProperties->isFloating,
                              saveWindow,
                              shouldDispose,
                              windowProperties->sourceFrame);
            }
        CATCH_ALL
            CloseWindow(platformWindow);
            ODDisposePtr(platformWindow);
            ODSShort errMsgNum = (!frame && windowProperties->sourceFrame)
                                 ? kErrCantOpenPartWindow : kErrCantOpenDocWindow;
            this->DoDialogBox(ev, frame, kErrorBoxID, errMsgNum);
            SetErrorCode(kODErrAlreadyNotified);
            RERAISE;
        ENDTRY
    }

    return window;
}
```


Handling Frame Layout

Parts lay themselves out for display in a document according to the demands of their content and in negotiation with their containing parts. This process takes place through the mechanism of the part's display frames. Methods illustrating how SamplePart handles its display frames are included in this section.

SamplePart is a noncontainer part—it does not support embedding of other parts in it—so its frame layout considerations are somewhat simpler than those of container parts. Nonetheless, SamplePart participates in the layout negotiations of its containing part when it is itself embedded in another part. In addition, SamplePart supports multiple display frames, displaying itself in multiple frames and synchronizing those displays as necessary.

The DisplayFrameAdded Method

OpenDoc calls the `DisplayFrameAdded` method when a new display frame is created for the part, for example, after the containing part calls the draft's

`CreateFrame` method. Generally, in response to the `DisplayFrameAdded` call, a part should set itself up to manage the new frame and ensure that it can handle the frame's display requirements.

The `SamplePart` object's implementation of the `DisplayFrameAdded` method performs the following actions:

1. **Sets up the presentation and view type correctly.**

   The method checks the new frame's presentation and view type. If the frame's presentation is not the SamplePart main presentation type, the method sets it to be so. If the view type is null, the method sets it to frame view, the most typical preferred type.

2. **Stores part info data for the new frame.**

   SamplePart creates a `CFrameInfo` object for this purpose and stores a reference to it in the frame's part info field.

3. **Sets the frame's window disposal flag.**

   If the frame being added is a root frame, then it has a window associated with it, and the window must be disposed of when the frame is removed. The window disposal flag is checked in SamplePart's internal `CleanupWindow` method.

4. **Updates the part's frame list.**

   Finally, the method creates a proxy for the new frame and adds a reference to the proxy to its internal display frame list.

Listing 2-9 shows the implementation of the `DisplayFrameAdded` method.

Listing 2-9 `DisplayFrameAdded` method

```
void SamplePart::DisplayFrameAdded( Environment*    ev,
                                    ODFrame*        frame )
{
    SOM_Trace("SamplePart","DisplayFrameAdded");

    if ( frame->GetPresentation(ev) != gGlobals->fMainPresentation )
        frame->SetPresentation(ev, gGlobals->fMainPresentation);

    if ( frame->GetViewType(ev) == kODNullTypeToken )
        frame->SetViewType(ev, gGlobals->fFrameView);
```

```
    CFrameInfo* frameInfo = new CFrameInfo;
    frame->SetPartInfo(ev, (ODInfoType)frameInfo);

    if ( frame->IsRoot(ev) )
        frameInfo->SetShouldDisposeWindow(kODTrue);
    CFrameProxy* proxy = new CFrameProxy;
    proxy->InitFrameProxy(ev,frame);
    fDisplayFrames->Add(proxy);

    this->SetDirty(ev);
}
```


The DisplayFrameConnected Method

OpenDoc calls the `DisplayFrameConnected` method if the part is embedded and the containing part reads the display frame into memory, having previously written it to storage. This occurs when the frame becomes visible through scrolling or other actions. OpenDoc calls this method instead of `DisplayFrameAdded` because a new frame is not being created; an existing one is being reconnected to the part.

The `SamplePart` object's implementation of the `DisplayFrameConnected` method performs the following actions:

1. **Updates the part's frame list.**

   The method iterates over SamplePart's list of display frames, attempting to match the frame's ID number with the ID numbers of the frame proxies in the list. If there is no match, the method adds the frame. If there is a match, the method updates the proxy's internal fields with information obtained from the frame.

2. **Ensures that the presentation is meaningful.**

   The part editor must be able to display the frame, so it must recognize the presentation. In SamplePart's case, the method compares the frame's presentation to the main presentation stored in the globals structure. If it differs, the method sets it to be the main presentation.

3. **Handles the root frame case.**

   If the frame is a root frame, the method does two things: it sets the window disposal flag to `kODTrue`, and it sets the view type to frame view.

Listing 2-10 shows the implementation of the `DisplayFrameConnected` method.

Listing 2-10 `DisplayFrameConnected` method

```
void SamplePart::DisplayFrameConnected( Environment*    ev,
                                        ODFrame*        frame )
{
    SOM_Trace("SamplePart","DisplayFrameConnected");

    ODBoolean found = kODFalse;
    CListIterator fiter(fDisplayFrames);
    for ( CFrameProxy* proxy = (CFrameProxy*) fiter.First();
            fiter.IsNotComplete(); proxy = (CFrameProxy*) fiter.Next() )
    {
        if ( proxy->GetID() == frame->GetID(ev) )
        {
            proxy->SetFrame(ev,frame);
            found = kODTrue;
        }
    }
    if ( found )
    {
        if ( frame->GetPresentation(ev) != gGlobals->fMainPresentation )
            frame->SetPresentation(ev, gGlobals->fMainPresentation);

        if ( frame->IsRoot(ev) )
        {
            CFrameInfo* frameInfo = (CFrameInfo*) frame->GetPartInfo(ev);
            frameInfo->SetShouldDisposeWindow(kODTrue);

            if ( frame->GetViewType(ev) != gGlobals->fFrameView )
                frame->SetViewType(ev, gGlobals->fFrameView);
        }
    }
    else
    {
        this->DisplayFrameAdded(ev, frame);
    }
}
```

The DisplayFrameRemoved Method

OpenDoc calls a part's `DisplayFrameRemoved` method when its containing part has permanently removed one of the part's display frames. Generally, implementations of the `DisplayFrameRemoved` method perform any actions required to remove the frame, including removing frames embedded within the removed frame, relinquishing foci, and updating the part's internal frame list.

The `SamplePart` object's implementation of the `DisplayFrameRemoved` method performs the following actions:

1. **Relinquishes any foci owned by the frame.**

   The method calls the `SamplePart` object's internal `RelinquishAllFoci` method, which instantiates a temporary frame object to wrap the reference returned by the arbitrator for each of the foci a SamplePart frame could own: the selection focus and the menu focus. The `RelinquishAllFoci` method compares the focus owner with the frame to be removed, and, if they are equal, relinquishes the focus through the arbitrator and notifies the part that the focus is lost.

   The `RelinquishAllFoci` method uses the `TempODFrame` class, a C++ template class declared in the file TempObj.h, and the `ODObjectsAreEqual` function, defined in the file ODUtils.h. They are described in Appendix A, "OpenDoc Utilities."

2. **Cleans up the display frame references.**

   The method calls the `SamplePart` object's internal `CleanupDisplayFrame` method. If this frame (that is, the frame to be removed) has a source frame, the `CleanupDisplayFrame` method gets a reference to the source frame and to its frame info object. It invalidates the source frame to force it to redraw without any possible effects of having been synchronized with this frame. The method notifies the source frame that it is going away and releases this frame's reference to the source frame (decrementing the source frame's reference count).

   If this frame is a root frame, then it is in a part window which is being closed, so the `CleanupDisplayFrame` method notifies the source frame that it no longer has a part window. Conversely, if the frame has a part window, the method closes and removes it.

   If the frame being removed has a dependent frame, the `CleanupDisplayFrame` method notifies it that its source frame is being removed and releases its own reference to the dependent frame.

3. **Cleans up any window associated with the frame.**

   The method calls the `SamplePart` object's internal `CleanupWindow` method, which checks this frame's `ShouldDisposeWindow` flag. If the flag is true, the method retrieves references to the frame's OpenDoc window object and its Mac OS platform window structure. It releases the OpenDoc window object, then closes and disposes of the platform window.

4. **Cleans up the frame and removes it from the part's internal frame list.**

   The method sets to null its pointer to its frame info object, then deletes the object using the `ODDeleteObject` utility macro (which is defined in the ODUtils.h file). Finally, the method removes this frame from its internal display frame list and sets the part's dirty flag.

If any of the preceding actions causes an exception to be thrown, the method catches it in its `CATCH_ALL` handler, which displays an error dialog box to the user and propagates the error.

Listing 2-11 shows the implementation of the `DisplayFrameRemoved` method.

Listing 2-11 `DisplayFrameRemoved` method

```
void SamplePart::DisplayFrameRemoved( Environment*  ev,
                                      ODFrame*      frame )
{
    SOM_Trace("SamplePart","DisplayFrameRemoved");

    TRY
        CFrameInfo* frameInfo = (CFrameInfo*) frame->GetPartInfo(ev);
        this->RelinquishAllFoci(ev, frame);
        this->CleanupDisplayFrame(ev, frame, kFrameRemoved);
        this->CleanupWindow(ev, frame);
        frame->SetPartInfo(ev, (ODInfoType) kODNULL);
        ODDeleteObject(frameInfo);

        CListIterator fiter(fDisplayFrames);
        for ( CFrameProxy* proxy = (CFrameProxy*) fiter.First();
                fiter.IsNotComplete(); proxy = (CFrameProxy*) fiter.Next() )
        {
            if ( ODObjectsAreEqual(ev, proxy->GetFrame(ev), frame) )
            {
```

```
            fiter.RemoveCurrent();
            delete proxy;
        }
    }
    this->SetDirty(ev);

CATCH_ALL
    this->DoDialogBox(ev, frame, kErrorBoxID, kErrRemoveFrame);
    SetErrorCode(kODErrAlreadyNotified);
    RERAISE;
ENDTRY
}
```


The DisplayFrameClosed Method

OpenDoc calls the `DisplayFrameClosed` method when a frame is closed as a result of the user closing its document. The SamplePart implementation of the `DisplayFrameClosed` method is virtually identical to that of its `DisplayFrameRemoved` method except it does not cache runtime information, so it does not set the part's dirty flag. Also, the `DisplayFrameClosed` method does not delete the frame proxy object because closed frames may be reconnected before the document is finally closed.

Listing 2-12 shows the implementation of the `DisplayFrameClosed` method.

Listing 2-12 `DisplayFrameClosed` method

```
void SamplePart::DisplayFrameClosed( Environment*   ev,
                                     ODFrame*       frame )
{
    SOM_Trace("SamplePart","DisplayFrameClosed");

    TRY
        CFrameInfo* frameInfo = (CFrameInfo*) frame->GetPartInfo(ev);
        this->RelinquishAllFoci(ev, frame);
        this->CleanupDisplayFrame(ev, frame, kFrameClosed);
        this->CleanupWindow(ev, frame);
        frame->SetPartInfo(ev, (ODInfoType) kODNULL);
        ODDeleteObject(frameInfo);
```

```
        CListIterator fiter(fDisplayFrames);
        for ( CFrameProxy* proxy = (CFrameProxy*) fiter.First();
                fiter.IsNotComplete(); proxy = (CFrameProxy*) fiter.Next() )
        {
            if ( proxy->GetID() == frame->GetID(ev) )
            {
                proxy->Purge(ev);
            }
        }

    CATCH_ALL
        this->DoDialogBox(ev, frame, kErrorBoxID, kErrRemoveFrame);
        SetErrorCode(kODErrAlreadyNotified);
        RERAISE;
    ENDTRY
}
```

The AttachSourceFrame Method

OpenDoc calls a part's `AttachSourceFrame` method during creation of a part window from a containing part. That is, if SamplePart is embedded in a frame of another part, and that frame is opened into a part window, the containing part iterates over its embedded frames and adds new corresponding frames in the part window. After each new embedded frame is created, the containing part calls the `AttachSourceFrame` method.

Listing 2-13 shows the implementation of the `AttachSourceFrame` method.

Listing 2-13 `AttachSourceFrame` method

```
void SamplePart::AttachSourceFrame( Environment*    ev,
                                    ODFrame*        frame,
                                    ODFrame*        sourceFrame )
{
    SOM_Trace("SamplePart","AttachSourceFrame");

    CFrameInfo* frameInfo = (CFrameInfo*) frame->GetPartInfo(ev);
    frameInfo->SetSourceFrame(ev, sourceFrame);
```

```
    frameInfo = (CFrameInfo*) sourceFrame->GetPartInfo(ev);
    frameInfo->SetDependentFrame(ev, frame);
}
```

The FrameShapeChanged Method

OpenDoc calls a part's `FrameShapeChanged` method whenever the part's display frame's shape has been changed, either by the user or by the containing part (if this part is embedded). OpenDoc passes a pointer to the frame whose shape has changed with the method call. The basic responsibility of this method is to update all synchronized frames by propagating the new frame shape to them. To do so, the method finds all the synchronized frames, pointers to which are stored in this frame's `CFrameInfo` object, and calls each frame's `RequestFrameShape` method.

Listing 2-14 shows the implementation of the `FrameShapeChanged` method.

Listing 2-14 `FrameShapeChanged` method

```
void SamplePart::FrameShapeChanged( Environment*    ev,
                                    ODFrame*        frame )
{
    SOM_Trace("SamplePart","FrameShapeChanged");

    if ( !frame->IsRoot(ev) )
    {
        TempODShape     frameShape = frame->AcquireFrameShape(ev, kODNULL);
        CFrameInfo*     frameInfo = (CFrameInfo*) frame->GetPartInfo(ev);
        ODFrame*        displayFrame;

        if ( frameInfo->HasSourceFrame() )
        {
            displayFrame = frameInfo->GetSourceFrame(ev);

            TempODShape frameShapeCopy = frameShape->Copy(ev);
            TempODShape returnShape = displayFrame->
                                RequestFrameShape(ev, frameShapeCopy, kODNULL);

            displayFrame->Invalidate(ev, kODNULL, kODNULL);
        }
```

```
        if ( frameInfo->HasDependentFrame() )
        {
            displayFrame = frameInfo->GetDependentFrame(ev);

            TempODShape frameShapeCopy = frameShape->Copy(ev);
            TempODShape returnShape = displayFrame->
                                    RequestFrameShape(ev, frameShapeCopy, kODNULL);

            displayFrame->Invalidate(ev, kODNULL, kODNULL);
        }
    }
}
```

Drawing the Part

Every part editor must implement its `Draw` method so it can display the visible portion of its content on demand, in response to the `Draw` call. Most part editors perform drawing of their content synchronously; that is, they allow OpenDoc to call the `Draw` method of their part editor object (`ODPart` subclass). OpenDoc calls the `Draw` method whenever portions of the document are marked invalid, as when the user scrolls a part's content into view. However, part editors can also draw asynchronously by calling their own `Draw` method. For example, a part that represents a clock would need to update and redraw its display every second.

To display its content, a part must have at least one frame, and it may have more than one frame, even in a single window. Part editors can display their content in different frames simultaneously, and they can display them differently in the same frame at different times.

During drawing, the part editor is responsible for examining the frame and displaying the correct information in the frame, properly transformed and clipped. If additional information is needed to perform rendering properly, the part editor may store it in the part info field of the frame or the facet.

The proper way to render a part on a particular display device may also vary depending on whether the device is static or dynamic. A part editor can use the `isDynamic` flag of the canvas object to determine the nature of the interaction style and draw its part accordingly. For example, it may draw scroll bars on a dynamic canvas but omit them for a static one.

The basic steps to perform in drawing are as follows:

1. Prepare the platform graphics environment for drawing.
2. Get the view type, the presentation if required, and any other information needed to determine the proper display method, such as selection state, highlight state, and the state of the stationery flag.
3. Render the content appropriately.
4. Restore the old graphics environment.

In SamplePart, these steps are accomplished by the `Draw` method and three subroutine methods to handle the four standard view types: frame, large icon, small icon, and thumbnail. In addition, `SamplePart` has methods that prepare for drawing and handle various other situations affecting imaging behavior. The following sections discuss these methods.

The Draw Method

OpenDoc calls the part object's `Draw` method whenever a facet of a part's display frame intersects the invalidated portion of an OpenDoc window. Parts may call their own `Draw` method whenever their content needs to be rendered onto a canvas.

The `SamplePart` object's implementation of the `Draw` method performs the following actions:

1. **Focuses the Mac OS drawing environment.**

   The `SamplePart` object's `Draw` method focuses the Mac OS QuickDraw port, origin, and clip shape for drawing in the facet passed into the `Draw` method. SamplePart accomplishes this by instantiating a stack-based object (here named `initiateDrawing`) of class `CFocus`, which is defined in the OpenDoc utility file FocusLib.cpp. The `CFocus` constructor saves the old port, origin, and clip shape and sets the new ones properly. At the end of the `Draw` method, when control passes out of its scope, the `CFocus` object is automatically deleted, and its destructor restores the port, origin, and clip shape previously in force.

2. **Gets the view type of the frame to which the current facet belongs.**

   The method gets a pointer to the frame from the facet, a pointer to which is passed in from OpenDoc with the `Draw` call. The frame is queried for its view type.

3. **Draws the part's content appropriately for the view type.**

   SamplePart has a separate method for each view type that can draw its content properly, and it branches to the appropriate one.

Listing 2-15 shows the implementation of the `Draw` method.

Listing 2-15 `Draw` method

```
void SamplePart::Draw( Environment* ev,
                       ODFacet*     facet,
                       ODShape*     invalidShape )
{
    SOM_Trace("SamplePart","Draw");

    CFocus initiateDrawing(ev, facet, invalidShape);

    ODTypeToken view = facet->GetFrame(ev)->GetViewType(ev);

    if ( view == gGlobals->fLargeIconView || view == gGlobals->fSmallIconView )
        this->DrawIconView(ev, facet);
    else if ( view == gGlobals->fThumbnailView )
        this->DrawThumbnailView(ev, facet);
    else
        this->DrawFrameView(ev, facet);
}
```

The DrawIconView Method

The `SamplePart` object's internal `DrawIconView` method draws an appropriate version of the frame's icon.

The `DrawIconView` method performs the following actions:

1. **Sets the icon transform type.**

   The method checks the facet's highlight state. If the facet is highlighted, the method will display the selected version of the icon. If a part window exists, it will display the version of the icon indicating that it is also open.

2. **Draws the icon.**

   The method sets the size of the rectangle in which to display the icon correctly and calls the Mac OS Toolbox routine `PlotIconID` to draw the correct version of the icon according to its icon transform type. Large icons are drawn in a 32-by-32-pixel rectangle; small icons are drawn in a 16-by-16-pixel rectangle.

Listing 2-16 shows the implementation of the `DrawIconView` method.

Listing 2-16 `DrawIconView` method

```
void SamplePart::DrawIconView( Environment* ev,
                               ODFacet*     facet )
{
    SOM_Trace("SamplePart","DrawIconView");

    Rect                iconRect;
    IconTransformType   transformType = ttNone;
    CFrameInfo*         frameInfo;
    ODFrame*            frame;
    ODTypeToken         viewType;

    frame   = facet->GetFrame(ev);
    viewType= frame->GetViewType(ev);
    frameInfo = (CFrameInfo*) frame->GetPartInfo(ev);

    if ( facet->GetHighlight(ev) == kODFullHighlight )
        transformType = ttSelected;

    if ( frameInfo->HasPartWindow() &&
         frameInfo->GetPartWindow()->IsShown(ev) )
        transformType |= ttOpen;

    if ( viewType == gGlobals->fLargeIconView )
        SetRect(&iconRect, 0, 0, kODLargeIconSize, kODLargeIconSize);
```

```
    else // ( viewType == gGlobals->fSmallIconView )
        SetRect(&iconRect, 0, 0, kODSmallIconSize, kODSmallIconSize);

    CUsingLibraryResources res;
    PlotIconID(&iconRect, atAbsoluteCenter, transformType, kBaseResourceID);
}
```

The DrawThumbnailView Method

Normally, a thumbnail view of a frame is a 64-by-64-pixel representation of its actual content. However, SamplePart has no intrinsic content, so its `DrawThumbnailView` method simply displays a `'PICT'` resource, a handle to which was previously stored in the `fThumbnail` field of the part's global variable structure. The same strategy is appropriate for parts that have been newly created from stationery and have no content yet. When the user has added content a "real" thumbnail can be created.

The `DrawThumbnailView` method performs the following actions:

1. **Sets the bounding rectangle of the thumbnail.**

   The method retrieves the bounding rectangle of the thumbnail picture resource by dereferencing its handle and sets the drawing offset accordingly.

2. **Draws the picture.**

   The method calls the QuickDraw routine `DrawPicture` to draw the picture resource at the proper position.

Listing 2-17 shows the implementation of the `DrawThumbnailView` method.

Listing 2-17 `DrawThumbnailView` method

```
void SamplePart::DrawThumbnailView( Environment*ev,
                                    ODFacet*/*facet*/ )
{
    SOM_Trace("SamplePart","DrawThumbnailView");

    LoadThumbnail(ev, &gGlobals->fThumbnail);

    Rect bounds = (**(PicHandle) gGlobals->fThumbnail).picFrame;
```

```
    OffsetRect(&bounds, -bounds.left, -bounds.top);
    DrawPicture((PicHandle) gGlobals->fThumbnail, &bounds);
}
```

The DrawFrameView Method

The implementation of the `SamplePart` object's internal `DrawFrameView` method (called by the part's `Draw` method) renders the full content view of the part when the view type is frame. SamplePart has no intrinsic content; the frame view simply draws two text strings with stylistic variations.

The `DrawFrameView` method performs the following actions:

1. **Gets the facet's frame and canvas, the frame shape, and the QuickDraw region of the frame area.**

   The method uses the facet passed as a parameter to get the frame. If the frame has a source frame, the method uses the source frame instead. The method gets a reference to the facet's canvas, which represents its underlying platform-specific drawing system (QuickDraw), so that the frame shape is returned in the correct coordinate system. The method then gets the shape and QuickDraw region of the frame.

2. **Sets the font characteristics.**

   The method calculates the height of the frame and sets the font size to 80% of the frame height. Then it sets the font to be the default application font for the current script system and sets its variation to be bold and condensed.

3. **Gets the text string to be drawn.**

   Before acquiring the string resource to draw, you must set up the resource chain so the resources contained in your dynamic library are available. This is handled in the `DrawFrameView` method by the OpenDoc utility routine `BeginUsingLibraryResources`, which is defined in the file UseRsrcM.cpp. At this point the method saves the QuickDraw pen state and resets it to normal, acquires the individual string from the resource, and moves the pen to an appropriate baseline position in preparation for drawing the text.

4. **Draws the text.**

   The method calls the QuickDraw routine `DrawString` to render a text string acquired from its string resource onto the screen, using the font characteristics calculated previously. If the facet's highlight state is `kODFullHighlight`, indicating that the part is selected, the method fills in the background of the drawing port with the highlight color. The method then draws another text string acquired from its string resource, this time at the fixed point size of 24 points, centered in the frame, and in color reversed from its background.

5. **Restores the resource chain and port characteristics.**

   The method calls `EndUsingLibraryResources` to restore the resource chain as configured prior to calling `BeginUsingLibraryResources`. You must call the ending routine if you have called the beginning routine, so you must not throw an exception between the two calls. If an exception is likely, therefore, you should save it and throw it after calling `EndUsingLibraryResources`. Last, the method restores the QuickDraw graphics port and resets its text font, size, and variation.

Listing 2-18 shows the implementation of the `DrawFrameView` method.

Listing 2-18 `DrawFrameView` method

```
void SamplePart::DrawFrameView( Environment*    ev,
                                ODFacet*        facet )
{
    SOM_Trace("SamplePart","DrawFrameView");

    ODFrame*    frame;
    ODUShort    frameHeight = 0;
    ODUShort    frameWidth = 0;
    RgnHandle   frameRgn;
    FontInfo    finfo;
    Str63       defaultString;
    CFrameInfo* frameInfo;
    GrafPtr     port;


    GetPort(&port);
    EraseRect(&port->portRect);

    frameInfo = (CFrameInfo*) facet->GetFrame(ev)->GetPartInfo(ev);
    if ( frameInfo->HasSourceFrame() )
        frame = frameInfo->GetSourceFrame(ev);
    else
```

```
    frame = facet->GetFrame(ev);

ODCanvas* biasCanvas = facet->GetCanvas(ev);

TempODShape frameShape = frame->AcquireFrameShape(ev, biasCanvas);
frameRgn = frameShape->GetQDRegion(ev);
frameHeight = (**frameRgn).rgnBBox.bottom - (**frameRgn).rgnBBox.top;
frameWidth = (**frameRgn).rgnBBox.right - (**frameRgn).rgnBBox.left;

ODUShort size = port->txSize;
ODUShort font = port->txFont;
Style face = port->txFace;

TextSize((ODUShort)(frameHeight * 0.8));
TextFont(1);
TextFace(bold + condense);

GetFontInfo(&finfo);

ODSLong rfRef;
rfRef = BeginUsingLibraryResources();
{
    PenState penState;
    GetPenState(&penState);

    PenNormal();
    GetIndString(defaultString, kMenuStringResID, kDefaultContent1ID);
    MoveTo((frameWidth / 2) - (StringWidth(defaultString) / 2),
                frameHeight - (finfo.descent - 2));
    DrawString(defaultString);

    if ( facet->GetHighlight(ev) == kODFullHighlight )
    {
        UInt8 mode = LMGetHiliteMode();
        BitClr(&mode,pHiliteBit);
        LMSetHiliteMode(mode);
        InvertRect(&port->portRect);
    }

    TextMode(srcXor);
    TextSize(24);
```

```
        TextFace(bold + extend);

        GetIndString(defaultString, kMenuStringResID, kDefaultContent2ID);
        MoveTo((frameWidth / 2) - (StringWidth(defaultString) / 2),
                    (frameHeight / 2) + 6);
        DrawString(defaultString);

        SetPenState(&penState);
    }
    EndUsingLibraryResources(rfRef);

    // Restore port chararcteristics.
    SetPort(port);
    port->txSize = size;
    port->txFont = font;
    port->txFace = face;
}
```

The ViewTypeChanged Method

OpenDoc calls a part's `ViewTypeChanged` method when the view type of one of the part's display frames has been modified, such as when the user changes the view type in the Part Info dialog box. In general, the `ViewTypeChanged` method should set up the facilities needed for a part to display itself with a particular view type.

The `SamplePart` object's implementation of `ViewTypeChanged` method performs the following actions:

1. **Gets the view type of the frame.**

   For this purpose, the `ODFrame` class provides a `GetViewType` method, which returns a tokenized string representing the view type.

2. **If thumbnail is the view type, prepares the thumbnail view.**

   In this case, `ViewTypeChanged` calls the `SamplePart` object's internal method `GenerateThumbnail`. The `GenerateThumbnail` method creates a 64-by-64-pixel representation of the current display frame. In SamplePart, this method calls the SamplePart utility method `LoadThumbnail`, which simply returns a handle to a preexisting `'PICT'` resource. The method puts a pointer to the thumbnail into the global variables structure. If the `GenerateThumbnail` method is unable to load the resource for some reason, it defaults to the regular frame view and throws the error returned by the Resource Manager as an exception (or, if there is no Resource Manager error, the method throws the `resNotFound` error).

3. **Changes the frame's used shape to match the new view type.**

   The method calls the `SamplePart` object's internal method `CalcNewUsedShape` to calculate the appropriate used shape for the new view type. If the view type is frame view, the `CalcNewUsedShape` method intentionally returns a null used shape, which resets the used shape to exactly the frame shape. Otherwise, the `CalcNewUsedShape` method returns a used shape equal to the appropriate icon or thumbnail view.

4. **Invalidates the frame.**

   The `ViewTypeChanged` method invalidates the frame, calls the frame's `ChangeUsedShape` method with the new used shape, and then invalidates the frame again.

Listing 2-19 shows the implementation of the `ViewTypeChanged` method, Listing 2-20 shows the `GenerateThumbnail` method, Listing 2-21 shows the `LoadThumbnail` method, and Listing 2-22 shows the `CalcNewUsedShape` method.

Listing 2-19 `ViewTypeChanged` method

```
void SamplePart::ViewTypeChanged( Environment*  ev,
                                  ODFrame*      frame )
{
    SOM_Trace("SamplePart","ViewTypeChanged");

    ODTypeTokenview = frame->GetViewType(ev);

    if ( view == gGlobals->fThumbnailView )
        this->GenerateThumbnail(ev, frame);

    TempODShape newUsedShape = this->CalcNewUsedShape(ev, frame);

    frame->Invalidate(ev, kODNULL, kODNULL);
    frame->ChangeUsedShape(ev, newUsedShape, kODNULL);
    frame->Invalidate(ev, kODNULL, kODNULL);
}
```

Listing 2-20 `GenerateThumbnail` method

```
void SamplePart::GenerateThumbnail( Environment*    ev,
                                    ODFrame*        frame )
{
    SOM_Trace("SamplePart","GenerateThumbnail");

    LoadThumbnail(ev, &gGlobals->fThumbnail);

    if ( gGlobals->fThumbnail == kODNULL )
    {
        frame->ChangeViewType(ev, gGlobals->fFrameView);
        THROW_IF_ERROR((ODError)ResError());
        THROW(resNotFound);
    }
}
```

Listing 2-21 `LoadThumbnail` method

```
void LoadThumbnail(Environment* ev, Handle* thumbnail)
{
    if ( *thumbnail ) return;

    ODSLong rfRef;
    rfRef = BeginUsingLibraryResources();
    {
        *thumbnail = (Handle) GetPicture(kThumbnailPicture);
    }
    EndUsingLibraryResources(rfRef);
}
```

Listing 2-22 `CalcNewUsedShape` method

```
ODShape* SamplePart::CalcNewUsedShape( Environment* ev,
                                       ODFrame*     frame )
{
    SOM_Trace("SamplePart","CalcNewUsedShape");
```

```
ODShape* usedShape = kODNULL;   ODVolatile(usedShape);
RgnHandle usedRgn;              ODVolatile(usedRgn);

ODTypeToken view = frame->GetViewType(ev);

if ( view == gGlobals->fLargeIconView ||
        view == gGlobals->fSmallIconView ||
        view == gGlobals->fThumbnailView )
{
    TRY
        Rect bounds;
        usedRgn = ODNewRgn();

        if ( view == gGlobals->fLargeIconView || view == gGlobals->fSmallIconView )
        {
            CUsingLibraryResources res;

            SetRect(&bounds, 0, 0,
                    (view == gGlobals->fLargeIconView) ?
                                    kODLargeIconSize : kODSmallIconSize,
                    (view == gGlobals->fLargeIconView) ?
                                    kODLargeIconSize : kODSmallIconSize);

            THROW_IF_ERROR( IconIDToRgn(usedRgn, &bounds,
                                        atAbsoluteCenter, kBaseResourceID) );
        }
        else if ( view == gGlobals->fThumbnailView )
        {
            bounds = (**(PicHandle)gGlobals->fThumbnail).picFrame;
            RectRgn(usedRgn,&bounds);
        }
        usedShape = frame->CreateShape(ev);
        usedShape->SetQDRegion(ev, usedRgn);

    CATCH_ALL
        ODSafeReleaseObject(usedShape);
        ODDisposeHandle((ODHandle)usedRgn);
        usedShape = kODNULL;
    ENDTRY
```

```
    }
    return usedShape;
}
```

The GeometryChanged Method

OpenDoc calls the `GeometryChanged` method when the external transform or clip shape of a facet belonging to the part's display frame changes. The only action of the `SamplePart` object's implementation of the method is to invalidate the clip shape of the facet, causing it to be redrawn.

Listing 2-23 shows the implementation of the `GeometryChanged` method.

Listing 2-23 `GeometryChanged` method

```
void SamplePart::GeometryChanged( Environment*  ev,
                                  ODFacet*      facet,
                                  ODBoolean     clipShapeChanged,
                                  ODBoolean     /* externalTransformChanged */ )
{
    SOM_Trace("SamplePart","GeometryChanged");

    if ( clipShapeChanged )
        facet->Invalidate(ev, kODNULL, kODNULL);
}
```

The HighlightChanged Method

OpenDoc calls a part's `HighlightChanged` method when the highlight state of one of its display frame's facets changes. The method is responsible for redrawing the facet's content with highlighting that is consistent with that of the containing part in which this part is embedded.

The `SamplePart` object's implementation of the `HighlightChanged` method gets a reference to the facet's frame, then (if its view type is not a frame view) simply invalidates the frame, causing its content to be redrawn. If the frame's view type is a frame view, the method does nothing.

Listing 2-24 shows the implementation of the `HighlightChanged` method.

Listing 2-24 `HighlightChanged` method

```
void SamplePart::HighlightChanged(Environment* ev, ODFacet* facet)
{
    ODFrame* frame = facet->GetFrame(ev);

    if ( frame->GetViewType(ev) != gGlobals->fFrameView )
        frame->Invalidate(ev, kODNULL, kODNULL);
}
```

The FacetAdded Method

OpenDoc calls the `FacetAdded` method when the containing part (or OpenDoc) adds a facet to one of the part's display frames. The part's basic responsibility in response to the `FacetAdded` method call is to prepare to draw the content visible in the new facet.

The `SamplePart` object's implementation of `FacetAdded` retrieves the facet's frame and the frame's info object. If the facet's frame is the root frame of a window, the method marks the frame for activation whenever the window is selected.

Finally, the method handles the possibility of the frame having a hidden part window. If the frame had become invisible previously, it would have hidden any part window it had. Therefore, the method checks to see if this is the first facet added to the frame, indicating that it is just becoming visible; if so, and if the frame has a part window, the method shows the part window.

Listing 2-25 shows the implementation of the `FacetAdded` method.

Listing 2-25 `FacetAdded` method

```
void SamplePart::FacetAdded( Environment*   ev,
                             ODFacet*       facet )
{
    SOM_Trace("SamplePart","FacetAdded");

    ODFrame* frame = facet->GetFrame(ev);
    CFrameInfo* frameInfo = (CFrameInfo*) frame->GetPartInfo(ev);
```

```
    if ( frame->IsRoot(ev) )
    {
        frameInfo->SetActiveFacet(facet);
        frameInfo->SetFrameReactivate(kODTrue);
    }
    if ( (CountFramesFacets(ev, frame) == 1) )
    {
        TempODWindow window = frameInfo->AcquirePartWindow(ev);
        if ( window ) window->Show(ev);
    }
}
```

The FacetRemoved Method

OpenDoc calls a part's `FacetRemoved` method when the containing part or OpenDoc removes a facet from one of this part's display frames.

The `SamplePart` object's implementation of `FacetRemoved` retrieves the facet's frame, and, if the frame indicates that this is the active facet, the method marks that indication false. Finally, if the facet being removed is the last facet belonging to its frame, and if its containing frame is null, the method hides the frame's part window, if it has one.

Listing 2-26 shows the implementation of the `FacetRemoved` method.

Listing 2-26 `FacetRemoved` method

```
void SamplePart::FacetRemoved( Environment* ev,
                               ODFacet*     facet )
{
    SOM_Trace("SamplePart","FacetRemoved");

    ODFrame*    frame = facet->GetFrame(ev);
    TempODFrame containingFrame = frame->AcquireContainingFrame(ev);
    CFrameInfo* frameInfo = (CFrameInfo*) frame->GetPartInfo(ev);

    if ( ODObjectsAreEqual(ev, frameInfo->GetActiveFacet(), facet) )
        frameInfo->SetActiveFacet(kODNULL);

    if ( (CountFramesFacets(ev, frame) == 0) &&
```

```
            (containingFrame == kODNULL) )
    {
        TempODWindow window = frameInfo->AcquirePartWindow(ev);
        if ( window ) window->Hide(ev);
    }
}
```


Handling Events

OpenDoc calls a part's `HandleEvent` method when a user event occurs within the purview of a focus currently owned by the part. For example, keystroke events are dispatched to the part that owns the keystroke focus. Geometry-based events, such as mouse clicks, are generally dispatched to the part within whose frames they occur, regardless of which part is currently active.

If the part editor handles the event, it should return a value of `kODTrue`. It can return a value of `kODFalse` if it does not handle the event. If the frame's `DoesPropagateEvents` method returns `kODTrue`, then the event is sent to the containing frame. If all containing frames fail to handle the event, and they propagate it, the OpenDoc shell attempts to handle the event itself.

For a given event, the dispatcher locates a dispatch module, and the dispatch module calls the part's `HandleEvent` method. The `facet` parameter of the `HandleEvent` method may be null (`kODNULL`), depending on the kind of event. The `frame` parameter is always valid, except in the case of some null (idle) events.

Event Constants

OpenDoc expects parts to handle the user events that are standard on the Mac OS, as represented by the following list of constants:

```
kODEvtNull
kODEvtMouseDown
kODEvtMouseUp
kODEvtKeyDown
kODEvtKeyUp
```

```
kODEvtAutoKey
kODEvtUpdate
kODEvtActivate
kODEvtOS
```

In addition to the standard Mac OS user events, parts should expect to receive OpenDoc-defined events. All parts may receive the events represented by the following constants:

```
kODEvtMenu
kODEvtWindow
kODEvtMouseEnter
kODEvtMouseWithin
kODEvtMouseLeave
kODEvtBGMouseDown
```

Container parts (those that can embed other parts) may also receive the events represented by the following constants:

```
kODEvtMouseDownEmbedded
kODEvtMouseUpEmbedded
kODEvtMouseDownBorder
kODEvtMouseUpBorder
kODEvtBGMouseDownEmbedded
```

The constant names representing the events differ slightly from the standard Mac OS event names for cross-platform compatibility. Part editors handle these events differently according to their own requirements. Refer to the *OpenDoc Programmer's Guide for the Mac OS* for detailed information about handling these types of events.

The HandleEvent Method

Generally, the implementation of a part editor's `HandleEvent` method works in much the same way as event-handling code in a standard Mac OS application. That is, the implementation acquires the event record, then branches to the appropriate event-handling routine based on the type of event. Unlike the standard Mac OS application, you don't need to poll for events by calling `WaitNextEvent`; in the case of standard events, the event record is passed as a parameter to the `HandleEvent` method.

The `SamplePart` object's implementation of the `HandleEvent` method performs the following actions:

1. **Performs a case switch on expected events.**

   An event is represented by an OpenDoc constant compared to the `what` field of an `ODEventData` structure, a pointer to which is passed in the `event` parameter of the `HandleEvent` call.

2. **Branches to the appropriate subroutine method.**

   The `HandleEvent` method handles simple events without branching.

3. **Returns a Boolean value indicating whether or not the event was handled.**

Listing 2-27 shows the implementation of the `HandleEvent` method.

Listing 2-27 `HandleEvent` method

```
ODBoolean SamplePart::HandleEvent( Environment*     ev,
                                   ODEventData*     event,
                                   ODFrame*         frame,
                                   ODFacet*         facet,
                                   ODEventInfo*     eventInfo )
{
    SOM_Trace("SamplePart","HandleEvent");

    ODBoolean   eventHandled = kODFalse;

    switch ( event->what )
    {
        case kODEvtMouseDown:
        case kODEvtMouseUp:
            eventHandled = this->HandleMouseEvent(ev, event, facet, eventInfo);
            break;
        case kODEvtMenu:
            eventHandled = this->HandleMenuEvent(ev, event, frame);
            break;
        case kODEvtActivate:
            this->WindowActivating(ev, frame, (event->modifiers & activeFlag));
            eventHandled = kODTrue;
            break;
        case kODEvtMouseEnter:
```

```
        case kODEvtMouseLeave:
            SetCursor(&ODQDGlobals.arrow);
            eventHandled = kODTrue;
            break;
        case kODEvtMouseWithin:
            eventHandled = kODTrue;
            break;
        case kODEvtNull:
        case kODEvtMouseDownEmbedded:
        case kODEvtMouseUpEmbedded:
        case kODEvtMouseDownBorder:
        case kODEvtMouseUpBorder:
        case kODEvtWindow:
        case kODEvtKeyDown:
        case kODEvtKeyUp:
        case kODEvtAutoKey:
        case kODEvtOS:
        case kODEvtDisk:
        default:
            break;
    }
    return eventHandled;
}
```

The `SamplePart` object's `HandleEvent` method illustrates a minimal set of event handlers that every part editor should implement. Naturally, you must also prepare to handle other events to which your part must respond to behave correctly.

The HandleMouseEvent Method

SamplePart calls its own internal `HandleMouseEvent` method from its `HandleEvent` method when it receives a mouse event of type `kODEvtMouseUp` or `kODEvtMouseDown`. OpenDoc passes the mouse event as a parameter to the part's `HandleEvent` method when the user clicks the mouse button within the bounds of one of the part's facets.

When a frame is inactive, the first mouse-up event (`kODEvtMouseUp`) it receives should activate it. Inactive frames do not receive mouse-down events (`kODEvtMouseDown`).

The `HandleMouseEvent` method performs the following actions:

1. **Ensures that the facet in which the mouse event occurred is valid.**

   If the `facet` parameter is null, the mouse event occurred outside the bounds of a modal window, in which case the implementation causes the Mac OS to sound a single system beep.

2. **Handles a mouse-up event.**

   After determining that the event occurred inside a valid facet, the method tests the event type against the `kODEvtMouseUp` constant.

3. **Handles the window's activation state.**

   If the event is a mouse-up, `HandleMouseEvent` checks the facet's window. If the window is not active, the method selects it and returns a value of `kODTrue`, which indicates that the method handled the mouse-up event. If the facet's window is already active, the method continues.

4. **Handles the frame's activation state.**

   `HandleMouseEvent` retrieves the facet's frame and the frame's `CFrameInfo` part info object. Using this information, the method determines if this is the active frame; if not, it calls its `ActivateFrame` method, which activates the frame by requesting the selection and menu foci.

   The method stores the active facet in its frame's `CFrameInfo` object, so the part editor will be able to position a part window properly if the user later chooses the View as Window command. If the `ActivateFrame` method call returned successfully, `HandleMouseEvent` returns `kODTrue`; otherwise it returns `kODFalse`.

5. **Handles a mouse-down event.**

   If the event was not a mouse-up event, `HandleMouseEvent` tests if it was of type `kODEvtMouseDown`. If so, the method localizes the coordinates of the mouse-down event to the facet's coordinates and calls the `SamplePart` object's internal `DoMouseEvent` method.

   The `SamplePart` object's `DoMouseEvent` method is empty. A part editor with real work to do in response to a mouse-down event would do it at this point. For example, if your part supports selection of its content by dragging the mouse, as with a marquee or lasso tool, you would handle those events at this point. Similarly, you would handle buttons or other controls here if they were managed directly by your part.

Listing 2-28 shows the `HandleMouseEvent` method. The `ActivateFrame` method is included in the "Activation" section as Listing 2-38 on page 95.

Listing 2-28 `HandleMouseEvent` method

```
ODBoolean SamplePart::HandleMouseEvent( Environment*    ev,
                                        ODEventData*    event,
                                        ODFacet*        facet,
                                        ODEventInfo*    eventInfo )
{
    SOM_Trace("SamplePart","HandleMouseEvent");

    if ( facet != kODNULL )
    {
        if ( event->what == kODEvtMouseUp )
        {
            ODWindow* window = facet->GetWindow(ev);
            TRY
                if ( !window->IsActive(ev) )
                    window->Select(ev);
                else
                {
                    ODFrame* frame = facet->GetFrame(ev);

                    CFrameInfo* frameInfo = (CFrameInfo*) frame->GetPartInfo(ev);
                    if ( !frameInfo->IsFrameActive() )
                    {
                        if ( this->ActivateFrame(ev, frame) )
                            frameInfo->SetActiveFacet(facet);
                        else
                            return kODFalse;
                    }
                }
            CATCH_ALL
            ENDTRY
        }
        else if ( event->what == kODEvtMouseDown )
        {
            Point where;
            where.h = FixedToInt(eventInfo->where.x);
```

```
            where.v = FixedToInt(eventInfo->where.y);
            this->DoMouseEvent(ev, facet, &where);
        }
    }
    else
    {
        SysBeep(1);
    }
    return kODTrue;
}
```

The HandleMenuEvent Method

SamplePart calls its own internal `HandleMenuEvent` method when it receives a menu event (type `kODEvtMenu`). OpenDoc converts a mouse-down event that occurs in the menu bar, or its keyboard equivalent, into a menu event. On receiving an event of this type, the `SamplePart` object's `HandleEvent` method calls `HandleMenuEvent`, passing the event record and a pointer to the active frame.

The `HandleMenuEvent` method performs the following actions:

1. **Retrieves the message field of the event record.**

   The method uses the message field to determine the number of the menu (contained in the high-order word) and the number of the menu item (contained in the low-order word), for the menu selection made by the user.

2. **Retrieves the position-independent number of the command.**

   With the menu and item numbers, the method calls the menu bar object's `GetCommand` method, which returns the command number of the user's menu selection.

3. **Branches to the appropriate command handler method.**

   Comparing the command number to constants representing the commands SamplePart can handle, the `HandleMenuEvent` method proceeds into a switch statement. SamplePart implements only two commands: About and View As Window. These cases call their appropriate subroutine method and return `kODTrue`. The remaining unimplemented command numbers return `kODFalse` by way of the default clause.

Listing 2-29 shows the implementation of the `HandleMenuEvent` method.

Listing 2-29 `HandleMenuEvent` method

```
ODBoolean SamplePart::HandleMenuEvent( Environment* ev,
                                       ODEventData* event,
                                       ODFrame*     frame )
{
    SOM_Trace("SamplePart","HandleMenuEvent");

    ODULong     menuResult  = event->message;
    ODUShort    menu        = HiWord(menuResult);
    ODUShort    item        = LoWord(menuResult);

    switch ( gGlobals->fMenuBar->GetCommand(ev, menu, item) )
    {
        case kODCommandAbout:
            this->DoDialogBox(ev, frame, kAboutBoxID);
            break;

        case kODCommandViewAsWin:
            this->Open(ev, frame);
            break;

        case kODCommandOpen:
        case kODCommandInsert:
        case kODCommandPageSetup:
        case kODCommandPrint:
        case kODCommandUndo:
        case kODCommandRedo:
        case kODCommandCut:
        case kODCommandCopy:
        case kODCommandPaste:
        case kODCommandPasteAs:
        case kODCommandClear:
        case kODCommandSelectAll:
        case kODCommandGetPartInfo:
        case kODCommandPreferences:
        default:
            return kODFalse;
    }
    return kODTrue;
}
```

The AdjustMenus Method

OpenDoc calls a part's `AdjustMenus` method when a user event of type `kODEvtMouseDown` occurs in the menu bar and the same part owns the menu focus. `AdjustMenus` is a general-purpose menu-handling method. Its purpose is to ensure that the visible state of the part's menus accurately reflect the state of the part. Accordingly, the `AdjustMenus` method enables and disables menu items, depending on whether or not their commands are available, and it changes the menu item text as necessary to describe accurately the actions ensuing from choosing those items.

The `SamplePart` object's implementation of the `AdjustMenus` method performs the following actions:

1. **Validates the menu bar if this part is the root part.**

   The menu bar object always calls the root part's `AdjustMenus` method before calling the menu focus owner's `AdjustMenus` method. Any other part can swap out the base menu bar at any time. Therefore, if the menu bar object has changed since it was previously copied, the method recopies the base menu bar from the window-state object. After copying the menu bar, you must also reinstall your part's menus.

2. **Enables or disables the menu commands, depending on conditions.**

   The method enables the View As Window command, but only if the frame that owns the menu focus (a pointer to which is passed into the method as it is called) is not the root frame of the window. (The frame that owns the menu focus is usually the active frame.)

3. **Sets the text of the About menu item correctly.**

   The method puts a reference to the focus owner's frame into a temporary frame object and tests it against the frame reference passed into this method call. If this frame owns the menu focus, the method gets the About menu item text from the SamplePart menu string resource, creates a temporary international text structure for the text, and sets the menu item. The temporary object automatically disposes of the memory allocated for the international text.

Listing 2-30 shows the implementation of the `AdjustMenus` method.

Listing 2-30 `AdjustMenus` method

```
void SamplePart::AdjustMenus( Environment*  ev,
                              ODFrame*      frame )
{
    SOM_Trace("SamplePart","AdjustMenus");

    if ( frame->IsRoot(ev) )
    {
        if ( gGlobals->fMenuBar->IsValid(ev) == kODFalse )
        {
            ODReleaseObject(ev, gGlobals->fMenuBar);
            gGlobals->fMenuBar =
                    ODGetSession(ev,fSelf)->GetWindowState(ev)->CopyBaseMenuBar(ev);
        }
    }

    gGlobals->fMenuBar->EnableCommand(ev, kODCommandViewAsWin, !frame->IsRoot(ev));

    TRY
        ODArbitrator* arbitrator = ODGetSession(ev,fSelf)->GetArbitrator(ev);
        TempODFrame menuOwner =
                            arbitrator->AcquireFocusOwner(ev, gGlobals->fMenuFocus);

        if ( ODObjectsAreEqual(ev, frame, menuOwner) )
        {
            Str63 text;
            ODGetIndString(text, kMenuStringResID, kAboutTextID);
            TempODIText menuItem(CreateIText(gGlobals->fEditorsScript,
                                    gGlobals->fEditorsLanguage, (StringPtr)&text));
            gGlobals->fMenuBar->SetItemString(ev, kODCommandAbout, menuItem);
        }
    CATCH_ALL
        // Consume exception
    ENDTRY
}
```

The DoDialogBox Method

SamplePart calls its own internal `DoDialogBox` method from its `HandleMenuEvent` method when the user chooses the About command. SamplePart also calls `DoDialogBox` from other methods to display error messages to the user. The method illustrates how parts can display a modal dialog box properly.

The `DoDialogBox` method performs the following actions:

1. **Gets access to the session object.**

   Access to the session object is provided by the `ODGetSession` utility function. The session object, in turn, provides needed access to the arbitrator and window-state objects.

2. **Gets a valid frame.**

   Only frames own foci. If the calling method does not pass in a valid frame reference, the `DoDialogBox` method gets one from SamplePart's internal list of display frames. This frame requests the modal focus needed to keep other parts from displaying a modal dialog box simultaneously.

3. **Requests the modal focus from the arbitrator.**

   If its focus request is not satisfied, the method causes the Mac OS to sound its system beep. Being unable to acquire the modal focus indicates that another modal dialog box is already being displayed.

4. **Deactivates the frontmost document window.**

   If its focus request is satisfied, the method calls the window-state object's `DeactivateFrontWindows` method.

5. **Displays the About box.**

   The method uses the OpenDoc utility routine `BeginUsingLibraryResources` to make the resources in its shared library available and uses the Mac OS Toolbox routine `GetNewDialog` to retrieve the dialog resource.

   If an error number greater than 0 was passed into this method, it sets up an error dialog box to display.

   If the dialog box resource has loaded properly, the `DoDialogBox` method ensures that the cursor is an arrow, shows the dialog box window, and calls the Mac OS Toolbox routine `ModalDialog` to display and handle the user's interaction with the dialog box.

6. **Cleans up after itself.**

   The method disposes of the dialog resource returned from the previous `GetNewDialog` routine. Finally, it restores the resource chain by calling `EndUsingLibraryResources`, relinquishes the modal focus to the arbitrator, and reactivates the frontmost document window.

Listing 2-31 shows the implementation of the `DoDialogBox` method.

Listing 2-31 `DoDialogBox` method

```
void SamplePart::DoDialogBox( Environment*  ev,
                              ODFrame*      frame,
                              ODSShort      dialogID,
                              ODUShort      errorNumber )
{
    SOM_Trace("SamplePart","DoDialogBox");

    ODFrame* focusFrame = frame;
    ODSession*session = ODGetSession(ev,fSelf);

    if ( focusFrame == kODNULL )
    {
        CListIterator fiter(fDisplayFrames);
        for ( CFrameProxy* proxy = (CFrameProxy*) fiter.First();
                fiter.IsNotComplete(); proxy = (CFrameProxy*) fiter.Next() )
        {
            if ( proxy->FrameIsLoaded() )
                focusFrame = proxy->GetFrame(ev);
            if ( focusFrame ) break;
        }
    }
    if ( session->GetArbitrator(ev)->RequestFocus(ev, gGlobals->fModalFocus,
                                                             focusFrame) )
    {
        DialogPtr   dialog;
        ODSShort    itemHit;
        session->GetWindowState(ev)->DeactivateFrontWindows(ev);

        ODSLong rfRef;
        rfRef = BeginUsingLibraryResources();
```

```
        {
            dialog = GetNewDialog(dialogID, kODNULL, (WindowPtr) -1L);
            if ( dialog )
            {
                if ( errorNumber > 0 )
                {
                    Handle  itemHandle;
                    Rect    itemRect;
                    short   itemType;
                    Str255  errStr;

                    GetIndString(errStr, kErrorStringResID, errorNumber);
                    GetDialogItem(dialog, kErrStrFieldID, &itemType,
                                                    &itemHandle, &itemRect);
                    SetDialogItemText(itemHandle, errStr);
                    HideDialogItem(dialog, cancel);
                    SetDialogDefaultItem(dialog, ok);
                }
                SetCursor(&ODQDGlobals.arrow);
                ShowWindow(dialog);
                ModalDialog(kODNULL, &itemHit);
                DisposeDialog(dialog);
            }
            else
            {
                SysBeep(2);
            }
        }
        EndUsingLibraryResources(rfRef);
        session->GetArbitrator(ev)->RelinquishFocus(ev, gGlobals->fModalFocus,
                                                                focusFrame);
        session->GetWindowState(ev)->ActivateFrontWindows(ev);
    }
    else
        SysBeep(2);
}
```

The View As Window Command

If the user chooses the View As Window command, the `HandleMenuEvent` method calls the `SamplePart` object's `Open` method, which is described in "Opening the Part Into a Window" on page 46.

Activation

When the user clicks within the used shape of any frame belonging to the part, the frame should activate itself. The frame should also activate itself when its window opens or becomes active if the part has stored information specifying that the frame should become active in those situations. In addition, a frame should activate itself when the user drags and drops data on it. The frame activates itself by acquiring the selection focus.

Methods illustrating the standard OpenDoc activation protocol are included in this section. These method implementations are short, so their descriptions are not shown as numbered steps.

The BeginRelinquishFocus Method

OpenDoc calls the `BeginRelinquishFocus` method when another frame requests ownership of a focus of which the specified frame is the current owner. Generally, a part's response to the `BeginRelinquishFocus` method call is to determine if it can safely relinquish the focus, in which case it returns `kODTrue`. A part does not actually relinquish the focus in response to the `BeginRelinquishFocus` call.

The `SamplePart` object's implementation of `BeginRelinquishFocus` first determines if the focus in question is its modal focus. If so, unless the frame requesting the focus belongs to SamplePart itself, the method returns `kODFalse`. That is, if another part wants to display a modal dialog box while SamplePart is displaying its own, SamplePart denies the request. Otherwise, the method returns `kODTrue`.

Listing 2-32 shows the implementation of the `BeginRelinquishFocus` method.

Listing 2-32 `BeginRelinquishFocus` method

```
ODBoolean SamplePart::BeginRelinquishFocus( Environment*    ev,
                                            ODTypeToken     focus,
                                            ODFrame*        /* ownerFrame */,
                                            ODFrame*        proposedFrame )
{
    SOM_Trace("SamplePart","BeginRelinquishFocus");

    ODBoolean willRelinquish = kODTrue;
    if ( focus == gGlobals->fModalFocus )
    {
        TempODPart proposedPart = ODAcquirePart(ev,proposedFrame);
        if ( ODObjectsAreEqual(ev, proposedPart, fSelf) == kODFalse )
            willRelinquish = kODFalse;
    }
    return willRelinquish;
}
```


The CommitRelinquishFocus Method

OpenDoc calls the `CommitRelinquishFocus` method when it is time for a frame to actually relinquish ownership of the specified focus, completing the process begun in response to a previous `BeginRelinquishFocus` method call. Generally, a part's response to the `CommitRelinquishFocus` method call is to remove any indications of the specified frame owning the focus; for example, the method could remove highlighting. If the focus is being transferred to a frame belonging to a different part, the part could do further actions, such as disabling menu items or removing a palette.

The `SamplePart` object's implementation of `CommitRelinquishFocus` calls the `FocusLost` method to do the actual work. The `FocusLost` method is shown in Listing 2-34.

Listing 2-33 shows the implementation of the `CommitRelinquishFocus` method.

Listing 2-33 `CommitRelinquishFocus` method

```
void SamplePart::CommitRelinquishFocus( Environment*    ev,
                                        ODTypeToken     focus,
                                        ODFrame*        ownerFrame,
                                        ODFrame*        /* proposedFrame */ )
{
    SOM_Trace("SamplePart","CommitRelinquishFocus");

    this->FocusLost(ev, focus, ownerFrame);
}
```

The FocusLost Method

The `SamplePart` object calls its own `FocusLost` method to do the actual work of relinquishing a focus specified by the `CommitRelinquishFocus` method call. In addition, OpenDoc may call the `FocusLost` method directly when the arbitrator has transferred ownership of a specified focus from the specified frame to another due to events, without benefit of the `BeginRelinquishFocus` and `CommitRelinquishFocus` method calls.

The `SamplePart` object's implementation of `FocusLost` acts only if the lost focus is the selection focus. In that case, the specified frame is being deactivated, so the method removes the indication that the frame is active, which is stored in the frame's `CFrameInfo` object.

Listing 2-34 shows the implementation of the `FocusLost` method.

Listing 2-34 `FocusLost` method

```
void SamplePart::FocusLost( Environment*    ev,
                            ODTypeToken     focus,
                            ODFrame*        ownerFrame )
{
    SOM_Trace("SamplePart","FocusLost");

    if ( focus == gGlobals->fSelectionFocus )
    {
        CFrameInfo* frameInfo = (CFrameInfo*) ownerFrame->GetPartInfo(ev);
```

```
        frameInfo->SetFrameActive(kODFalse);
    }
}
```

The AbortRelinquishFocus Method

OpenDoc calls the `AbortRelinquishFocus` method when it rescinds a previous request (made with a `BeginRelinquishFocus` call) to relinquish ownership of a focus. Generally, a part's response to the `AbortRelinquishFocus` method call is to back out of any changes it initiated in response to the previous `BeginRelinquishFocus` call.

The `SamplePart` objects's implementation of `AbortRelinquishFocus` does nothing.

Listing 2-35 shows the implementation of the `AbortRelinquishFocus` method.

Listing 2-35 `AbortRelinquishFocus` method

```
void SamplePart::AbortRelinquishFocus( Environment* ev,
                                       ODTypeToken  /*focus*/,
                                       ODFrame*     /*ownerFrame*/,
                                       ODFrame*     /*proposedFrame*/ )
{
    SOM_Trace("SamplePart","AbortRelinquishFocus");

    // Some parts may have suspended some events in the BeginRelinquishFocus
    // method. If so, they would resume those events here.
}
```

The FocusAcquired Method

OpenDoc calls the `FocusAcquired` method when the arbitrator has transferred ownership of the specified focus to the specified frame without benefit of the `BeginRelinquishFocus` and `CommitRelinquishFocus` method calls. Generally, a part's response to the `FocusAcquired` method call is to perform any actions needed to indicate that the specified frame now owns the focus. For example, if a frame acquired the selection focus, a part would highlight any selection within the frame.

The SamplePart object's implementation of FocusAcquired calls the arbitrator's RequestFocusSet method to request the complete focus set it needs to be active. If that action succeeds, the method calls the SamplePart object's internal method PartActivated, which puts the part into an active state, as shown in Listing 2-37.

Listing 2-36 shows the SamplePart object's implementation of the FocusAcquired method.

Listing 2-36 FocusAcquired method

```
void SamplePart::FocusAcquired( Environment*    ev,
                                ODTypeToken     focus,
                                ODFrame*        ownerFrame )
{
    SOM_Trace("SamplePart","FocusAcquired");

    ODArbitrator* arbitrator =  ODGetSession(ev,fSelf)->GetArbitrator(ev);

    if ( arbitrator->RequestFocusSet(ev, gGlobals->fUIFocusSet, ownerFrame) )
    {
        this->PartActivated(ev, ownerFrame);
    }
}
```

The PartActivated Method

The SamplePart object calls its own internal method PartActivated to display the part's menu bar and set the active flag in the specified frame's CFrameInfo object to true. Before displaying the menu bar, however, the method revalidates it, as described in "The AdjustMenus Method" on page 85.

Listing 2-37 shows the implementation of the PartActivated method.

Listing 2-37 PartActivated method

```
void SamplePart::PartActivated( Environment*    ev,
                                ODFrame*        frame )
{
    SOM_Trace("SamplePart","PartActivated");
```

```
    if ( gGlobals->fMenuBar->IsValid(ev) == kODFalse )
    {
        ODReleaseObject(ev, gGlobals->fMenuBar);
        gGlobals->fMenuBar =
                    ODGetSession(ev,fSelf)->GetWindowState(ev)->CopyBaseMenuBar(ev);
    }
    gGlobals->fMenuBar->Display(ev);
    CFrameInfo* frameInfo = (CFrameInfo*) frame->GetPartInfo(ev);
    frameInfo->SetFrameActive(kODTrue);
}
```


The ActivateFrame Method

The `SamplePart` object calls its own internal `ActivateFrame` method when a mouse-up event occurs in an inactive frame in an active window or when the window in which the frame is displayed is activated by the Mac OS.

The method requests the user-interface focus set (defined in the `Initialize` method) and, if that request is granted, calls the `PartActivated` method to display the menu bar and set the specified frame's active flag. If the method executes successfully, it returns `kODTrue` as a signal to the caller that the specified frame is now active; otherwise, it returns `kODFalse`.

Listing 2-38 shows the implementation of the `ActivateFrame` method.

Listing 2-38 `ActivateFrame` method

```
ODBoolean SamplePart::ActivateFrame( Environment*   ev,
                                     ODFrame*       frame )
{
    SOM_Trace("SamplePart","ActivateFrame");

    ODBoolean activated = kODFalse;

    if ( ODGetSession(ev,fSelf)->GetArbitrator(ev)
            ->RequestFocusSet(ev, gGlobals->fUIFocusSet, frame) )
    {
        this->PartActivated(ev, frame);
        activated = kODTrue;
```

```
    }
    return activated;
}
```

The WindowActivating Method

The `SamplePart` object calls its own internal `WindowActivating` method from its `HandleEvent` method when it receives an activate event from the Mac OS. The activate event (`kODEvtActivate`) indicates that a window displaying the specified frame is being either activated or deactivated, as indicated by the Boolean parameter `activating`.

If the window is being activated, and if the specified frame had the selection focus when it was deactivated, the method calls the `SamplePart` object's internal `ActivateFrame` method. If the window is being deactivated and the specified frame is active, the method marks the frame's reactivation flag true. By setting the flag in this way, the frame can reactivate itself if the window becomes active again later, using the previous block of this same method.

Listing 2-39 shows the implementation of the `WindowActivating` method.

Listing 2-39 `WindowActivating` method

```
void SamplePart::WindowActivating( Environment* ev,
                                   ODFrame*     frame,
                                   ODBoolean    activating )
{
    SOM_Trace("SamplePart","WindowActivating");

    CFrameInfo* frameInfo = (CFrameInfo*) frame->GetPartInfo(ev);
    if ( activating && frameInfo->FrameNeedsReactivating() )
    {
        this->ActivateFrame(ev, frame);
        frameInfo->SetFrameReactivate(kODFalse);
    }
    else if ( !activating && frameInfo->IsFrameActive() )
    {
        frameInfo->SetFrameReactivate(kODTrue);
    }
}
```

Persistent Storage

Persistent storage is a way to retain data on a long-term basis, supported by a nonvolatile device such as a hard disk. Persistent data remains stable between computing sessions. All persistent storage in OpenDoc is represented by storage units (`ODStorageUnit`), which provide a standard, cross-platform interface for all persistent objects. Every object in OpenDoc that needs to maintain its state between sessions is a persistent object, and each has a storage unit. Part objects must handle their storage units in a particularly disciplined manner because they need to satisfy many more requirements than other persistent objects.

Storage units have any number of properties, which are like separate forks of files, and properties have any number of values, which are separate streams of each fork. Each value in the same property holds a different representation of the same data; it should not hold different data. For example, every part has a contents property (`kODPropContents`), and multiple representations of the content can be stored in different values, but only content data should be stored in the contents property.

If parts have multiple representations of their content, they must write them to storage in order of fidelity. For example, a part's most faithful representation of text may be styled text, while a lower fidelity representation of the same content would be plain ASCII text, a separate value for which would be added later to the same property. The highest fidelity representation of part content is its native format, specific to and usually proprietary to the part editor that created it. Lower fidelity representations enable the part to be viewed in documents without a full complement of part editors, to maintain portability of documents.

To load your part's content into memory from persistent storage, you should basically reverse the process of writing your part. However, your part must be able to work from an empty storage unit as well as one with stored content. Refer to the section "Initialization" on page 39 for a description of this process.

SamplePart implements its persistent storage protocol in its `Externalize`, `ExternalizeStateInfo`, and `ExternalizeContent` methods. Other methods dealing with storage are `WritePartInfo`, `ReadPartInfo`, `ClonePartInfo`, `CloneInto`, and `Purge`. In addition, the utility method `SetDirty` manipulates the dirty flag, which is simply a Boolean value SamplePart uses to avoid redundancy: it writes the part content and notifies the draft only if the part has been altered.

The sections that follow show the implementations of these methods, except for the `CloneInto` method, which OpenDoc calls to perform data interchange. The `CloneInto` method also uses the storage unit API.

The Externalize Method

OpenDoc calls the `Externalize` method whenever it is necessary to write the part to persistent storage. Your part can also call its own `Externalize` method whenever it wants to. Before returning from this method, you must write all data that you need to accurately recreate the content and state of your part.

This method must call its parent class behavior (inherited class), because one of its parent class methods contains implementation. This is done in the SOM class implementation, which otherwise delegates all implementation to this method. Refer to the `som_SamplePart__Externalize` method of the `som_SamplePart` class in the file som_SamplePart.cpp.

The `SamplePart` object's implementation of the `Externalize` method performs the following actions:

1. **Checks the part's dirty flag and storage unit privileges.**

   If the part's dirty flag is set to `kODTrue`, meaning that the part has been changed since it was last written, and if the part's storage unit is not read-only, the method proceeds.

2. **Retrieves a pointer to the part's storage unit.**

   The method calls the `GetStorageUnit` method inherited from the `ODPart` superclass `ODPersistentObject`, using the `fSelf` field to refer to the part editor.

3. **Ensures that the storage unit properties are appropriate.**

   The method calls SamplePart subroutines, internal methods `CheckAndAddProperties` and `CleanseContentProperty`, to verify that the properties and values are correct.

4. **Writes out the part's status information.**

   The method accomplishes this step by calling an internal method, `ExternalizeStateInfo`.

5. **Writes out the part's content data.**

   The method writes out its content data by calling another internal method, `ExternalizeContent`.

6. **Sets the part's dirty flag to false.**

Listing 2-40 shows the implementation of the `Externalize` method. The other methods that the `Externalize` method calls are described in the next few sections.

Listing 2-40 `Externalize` method

```
void SamplePart::Externalize( Environment* ev )
{
    SOM_Trace("SamplePart","Externalize");

    TRY
        if ( fDirty && !fReadOnlyStorage )
        {
            ODStorageUnit* storageUnit = fSelf->GetStorageUnit(ev);
            this->CheckAndAddProperties(ev, storageUnit);
            this->CleanseContentProperty(ev, storageUnit);
            this->ExternalizeStateInfo(ev, storageUnit, kODNULLKey, kODNULL);
            this->ExternalizeContent(ev, storageUnit, kODNULLKey, kODNULL);
            fDirty = kODFalse;
        }
    CATCH_ALL
        this->DoDialogBox(ev, kODNULL, kErrorBoxID, kErrExternalizeFailed);
        SetErrorCode(kODErrAlreadyNotified);
        RERAISE;
    ENDTRY
}
```

The CheckAndAddProperties Method

SamplePart calls its own internal `CheckAndAddProperties` method to verify that the part's storage unit has the properties it needs to run. If such properties are not present, `CheckAndAddProperties` adds them.

The `CheckAndAddProperties` method performs the following actions:

1. **Sets up the contents property if it is not present.**

   After ensuring that the contents property exists, the method checks for, and if necessary adds, the part's kind value to the contents property. These actions are necessary in case the storage unit is new and the part has not been previously written to storage.

2. **Sets up the preferred kind property if it is not present.**

   The method writes out the default part kind for the editor. The user's chosen kind is written out in the `ExternalizeStateInfo` method.

3. **Sets up the part's display frame list if it is not present.**

   The method checks for and, if necessary, adds the display frames property and value.

Listing 2-41 shows the implementation of the `CheckAndAddProperties` method.

Listing 2-41 `CheckAndAddProperties` method

```
void SamplePart::CheckAndAddProperties( Environment*    ev,
                                        ODStorageUnit*  storageUnit )
{
    SOM_Trace("SamplePart","CheckAndAddProperties");

    if ( !storageUnit->Exists(ev, kODPropContents, kODNULL, 0) )
        storageUnit->AddProperty(ev, kODPropContents);
    if ( !storageUnit->Exists(ev, kODPropContents, kSamplePartKind, 0) )
    {
        storageUnit->Focus(ev, kODPropContents, kODPosUndefined,
                                                        kODNULL, 0, kODPosAll);
        storageUnit->AddValue(ev, kSamplePartKind);
    }
    if ( !storageUnit->Exists(ev, kODPropPreferredKind, kODISOStr, 0) )
    {
        TRY
            ODSetISOStrProp(ev, storageUnit, kODPropPreferredKind,
                                                        kODISOStr, kSamplePartKind);
        CATCH_ALL
            ODSURemoveProperty(ev, storageUnit, kODPropPreferredKind);
        ENDTRY
    }
```

```
    if ( !storageUnit->Exists(ev, kODPropDisplayFrames, kODNULL, 0) )
        storageUnit->AddProperty(ev, kODPropDisplayFrames);
    if ( !storageUnit->Exists(ev, kODPropDisplayFrames, kODWeakStorageUnitRefs, 0) )
    {
        storageUnit->Focus(ev, kODPropDisplayFrames, kODPosUndefined,
                                                    kODNULL, 0, kODPosAll);
        storageUnit->AddValue(ev, kODWeakStorageUnitRefs);
    }
}
```


The CleanseContentProperty Method

The `SamplePart` object calls its own internal `CleanseContentProperty` method from its `Externalize` method. The purpose of this method is to remove any value in the contents property that the part cannot write out accurately, such as values added to the contents property during drag-and-drop operations.

The `CleanseContentProperty` method performs the following actions:

1. **Focuses the storage unit to its contents property.**
2. **Retrieves the type of each value in the contents property.**

   The method uses the count of the number of values in the contents property to iterate through all of them. It focuses the storage unit on each value and gets its type.
3. **Removes any unsupported values.**

   The method uses the OpenDoc utility method `ODISOStrCompare` to identify unsupported values by comparing their types to the `kSamplePartKind` data type. The method then deletes unsupported values using the `ODStorageUnit` method `Remove` on the previously focused storage unit.

Listing 2-42 shows the implementation of the `CleanseContentProperty` method.

Listing 2-42 `CleanseContentProperty` method

```
void SamplePart::CleanseContentProperty( Environment*   ev,
                                         ODStorageUnit* storageUnit )
{
    SOM_Trace("SamplePart","CleanseContentProperty");
```

```
    ODULong numValues;
    ODULong index;

    storageUnit->Focus(ev, kODPropContents, kODPosUndefined,
                        kODNULL, 0, kODPosAll);
    numValues = storageUnit->CountValues(ev);

    for (index = numValues; index >= 1; index--)
    {
        storageUnit->Focus(ev, kODPropContents, kODPosUndefined,
                            kODNULL, index, kODPosUndefined);
        TempODValueType value = storageUnit->GetType(ev);
        if ( ODISOStrCompare(value, kSamplePartKind) != 0 )
            storageUnit->Remove(ev);
    }
}
```

The ExternalizeStateInfo Method

The `SamplePart` object calls its internal `ExternalizeStateInfo` method from its `Externalize` method when it writes the part to storage. This method writes out state information—any information pertaining to the working of the part editor—rather than the content. Such state information may be lost during data interchange operations, so the part must be able to recover gracefully if the state information is incomplete or missing.

The `ExternalizeStateInfo` method performs the following actions:

1. **Deletes weak references to the part's display frames.**

   First the method focuses on the display frames property of the part's storage unit, then removes and adds back the weak storage unit references associated with that property. This action deletes previously written persistent object references, which are not deleted by simply deleting the content of the value.

2. **Gets ID numbers for each display frame in the part's display frame list.**

   The method creates a `CListIterator` object to visit each of the part's display frames, retrieving the frame ID number for each.

   If, however, a draft key is passed in the `key` parameter, it indicates that the part is being cloned to another draft, in which case the method creates a weak clone of the display frame and uses the frame ID of the cloned frame instead. A draft key is a unique number that identifies a cloning operation on a draft; because cloning is a multistep process, the key is needed to preserve the integrity of each operation.

3. **Writes out weak references for each of the part's display frames.**

   Still within the iteration loop of the `CListIterator`, the method gets the weak reference to the storage unit of each of the part's display frames. Finally, using a macro named `StorageUnitSetValue`, the method writes that value into the display frames property of the part's storage unit.

   The `StorageUnitSetValue` macro, defined in the file StorUtil.h, simplifies handling of the `ODByteArray` structure required by the `SetValue` method of `ODStorageUnit`, which the macro calls.

Listing 2-43 shows the implementation of the `ExternalizeStateInfo` method.

Listing 2-43 `ExternalizeStateInfo` method

```
void SamplePart::ExternalizeStateInfo( Environment*     ev,
                                       ODStorageUnit*   storageUnit,
                                       ODDraftKey       key,
                                       ODFrame*         scopeFrame )
{
    SOM_Trace("SamplePart","ExternalizeStateInfo");

    ODStorageUnitRef    weakRef;
    ODID                frameID;
    ODID                scopeFrameID =
                                ( scopeFrame ? scopeFrame->GetID(ev) : kODNULLID );
    ODDraft*            fromDraft = ODGetDraft(ev,fSelf);

    storageUnit->Focus(ev, kODPropDisplayFrames, kODPosUndefined,
                                kODWeakStorageUnitRefs, 0, kODPosUndefined);
    storageUnit->Remove(ev);
    storageUnit->AddValue(ev, kODWeakStorageUnitRefs);

    CListIterator fiter(fDisplayFrames);
    for ( CFrameProxy* proxy = (CFrameProxy*) fiter.First();
            fiter.IsNotComplete(); proxy = (CFrameProxy*) fiter.Next() )
    {
        frameID = proxy->GetID();
```

```
        if ( key )
            frameID = fromDraft->WeakClone(ev, key, frameID, kODNULLID, scopeFrameID);
        storageUnit->GetWeakStorageUnitRef(ev, frameID, weakRef);
        TRY
            StorageUnitSetValue(storageUnit, ev, kODStorageUnitRefSize,
                                                        (ODPtr)&weakRef);
        CATCH_ALL
            // Consume the exception
        ENDTRY
    }
}
```

The ExternalizeContent Method

The `SamplePart` object's `ExternalizeContent` method is empty, although any implementation would contain a statement to focus the part's storage unit on its contents property (`kODPropContents`). Every part must have a property of this type in which to store its content data. OpenDoc uses the contents property to match parts to their correct part editors. Finally, the method would also write the part's content data out to its storage unit in an appropriate manner. SamplePart has no intrinsic content so `ExternalizeContent` does nothing.

The CloneInto Method

OpenDoc calls the `CloneInto` method during data interchange operations, that is, when a part is copied to the Clipboard, to a drag-and-drop object, or to a link-source object. The `CloneInto` method is inherited from the `ODPersistentObject` class. Generally, a part should respond to the `CloneInto` method call by writing its own data to the specified destination storage unit and cloning any objects to which it has strong persistent references and which are within the scope of the frame passed in the `initiatingFrame` parameter.

Note

The scope of a frame includes the content of frames embedded within it but excludes other content of the parts belonging to those embedded frames. Scope and other concepts of cloning are explained in the *OpenDoc Programmer's Guide for the Mac OS*. ◆

The `SamplePart` object's implementation of the `CloneInto` method writes only its own data, state information, and content, to the destination storage unit. Because SamplePart does not support embedding of other parts within itself, it has no need to clone any other objects.

SamplePart does the actual work of externalizing its data in the internal methods `CheckAndAddProperties` (Listing 2-41 on page 100), `ExternalizeStateInfo` (Listing 2-43 on page 103), and `ExternalizeContent` ("The ExternalizeContent Method" on page 104).

Listing 2-44 shows the implementation of the `CloneInto` method.

Listing 2-44 `CloneInto` method

```
void SamplePart::CloneInto( Environment*    ev,
                            ODDraftKey      key,
                            ODStorageUnit*  destinationSU,
                            ODFrame*        initiatingFrame )
{
    SOM_Trace("SamplePart","CloneInto");

    if ( destinationSU->Exists(ev, kODPropContents, kSamplePartKind, 0) == kODFalse )
    {
        this->CheckAndAddProperties(ev, destinationSU);
        this->ExternalizeStateInfo(ev, destinationSU, key, initiatingFrame);
        this->ExternalizeContent(ev, destinationSU, key, initiatingFrame);
    }
}
```

The InternalizeContent Method

The `SamplePart` object's internal `InternalizeContent` method does nothing, because SamplePart has no intrinsic content.

Generally speaking, for parts having content, a method such as this would focus the part's storage unit on the `kODPropContents` property, then read the stored data values. A reference to the storage unit is passed by OpenDoc to the part's `InitPartFromStorage` method (which in turn calls this method).

The InternalizeStateInfo Method

The `SamplePart` object calls its own internal `InternalizeStateInfo` method from its `InitPartFromStorage` method when it reads the part in from its persistent storage unit. This method reads in state information—any information pertaining to the working of the part editor—rather than the content. Generally, state information enables a part to present the same setup or configuration to the user as it had when last written out to storage.

The `InternalizeStateInfo` method reads from storage a list of weak references to its display frames, previously written out by the `ExternalizeStateInfo` method. The method validates each reference; if the reference is valid, the method adds it to its display frame list using lazy internalization. That is, the method uses a frame proxy object, adding the proxy pointer to its display frame list. The part reads in the actual display frame object only when it is actually needed.

Listing 2-45 shows the implementation of the `InternalizeStateInfo` method.

Listing 2-45 `InternalizeStateInfo` method

```
void SamplePart::InternalizeStateInfo( Environment*     ev,
                                       ODStorageUnit*   storageUnit )
{
    SOM_Trace("SamplePart","InternalizeStateInfo");

    ODStorageUnitRef    weakRef;
    ODULong             size;

    if ( storageUnit->Exists(ev, kODPropDisplayFrames, kODWeakStorageUnitRefs, 0) )
    {
        storageUnit->Focus(ev, kODPropDisplayFrames, kODPosUndefined,
                               kODWeakStorageUnitRefs, 0, kODPosUndefined);
        size = storageUnit->GetSize(ev);
        storageUnit->SetOffset(ev, 0);

        for ( ODULong offset = 0; offset < size; offset += kODStorageUnitRefSize )
        {
            TRY
                StorageUnitGetValue(storageUnit, ev, kODStorageUnitRefSize,
                                                            (ODPtr)&weakRef);
```

```
                if ( storageUnit->IsValidStorageUnitRef(ev, weakRef) )
                {
                    ODID frameID = storageUnit->GetIDFromStorageUnitRef(ev, weakRef);
                    CFrameProxy* proxy = new CFrameProxy;
                    proxy->InitFrameProxy(frameID, ODGetDraft(ev,storageUnit));
                    fDisplayFrames->Add(proxy);
                }
            CATCH_ALL
                // Consume exception
            ENDTRY
        }
    }
}
```


The ReadPartInfo Method

Every part is displayed in at least one frame represented by an object of class `ODFrame`. Frame objects have a part info field in which a part editor can store information describing how it should display its part's data in that frame. When you write your part to storage, OpenDoc calls your part's `WritePartInfo` method, and when you load your part into memory, OpenDoc calls its `ReadPartInfo` method. Generally, a part should respond to the `WritePartInfo` method call by writing enough information to persistent storage to be able to reconstruct each frame's part info field, and it should perform that reconstruction in its `ReadPartInfo` implementation. The `WritePartInfo` method is described in the section following this one.

The `SamplePart` object stores a pointer to an object of a C++ helper class named `CFrameInfo` in its part info field.

The `ReadPartInfo` method performs the following actions:

1. **Instantiates a frame info object.**

   The `CFrameInfo` constructor initializes the object's internal data fields.

2. **Reads the frame info object into memory.**

   The `InitFromStorage` method reads the `CFrameInfo` object, containing the frame's status information, from its storage unit.

3. **Returns a pointer to the frame info object.**

If the `CFrameInfo` object's `InitFromStorage` method fails, the method deletes the object and propagates the exception to the calling method.

Listing 2-46 shows the implementation of the `SamplePart` object's `ReadPartInfo` method, the `CFrameInfo` constructor (defined inline), and the `CFrameInfo` object's `InitFromStorage` method.

Listing 2-46 `ReadPartInfo`, `CFrameInfo` constructor, and `CFrameInfo::InitFromStorage` methods

```
ODInfoType SamplePart::ReadPartInfo( Environment*       ev,
                                     ODFrame*           frame,
                                     ODStorageUnitView* storageUnitView )
{
    SOM_Trace("SamplePart","ReadPartInfo");

    CFrameInfo* frameInfo = new CFrameInfo;

    TRY
        frameInfo->InitFromStorage(ev, storageUnitView);
    CATCH_ALL
        ODDeleteObject(frameInfo);
        RERAISE;
    ENDTRY

    return (ODInfoType)frameInfo;
}

CFrameInfo::CFrameInfo()
{
    fFrameActive = kODFalse;
    fFrameReactivate  = kODFalse;
    fShouldDisposeWindow = kODFalse;
    fActiveFacet = kODNULL;
    fSourceFrame = kODNULL;
    fDependentFrame = kODNULL;
    fPartWindow = kODNULL;
}
```

```
void CFrameInfo::InitFromStorage(Environment* ev, ODStorageUnitView* storageUnitView)
{
    ODStorageUnit* storageUnit = storageUnitView->GetStorageUnit(ev);

    if ( storageUnit->Exists(ev, kODNULL, kSamplePartInfo, 0) )
    {
        TRY
            storageUnit->Focus(ev, kODNULL, kODPosSame,
                                    kSamplePartInfo, 0 , kODPosUndefined);
            ODStorageUnitRef weakRef = {0,0,0,0};
            StorageUnitGetValue(storageUnit, ev, sizeof(ODStorageUnitRef),
                                (ODPtr)&weakRef);
            if ( storageUnit->IsValidStorageUnitRef(ev, weakRef) )
            {
                ODID frameID = storageUnit->GetIDFromStorageUnitRef(ev, weakRef);
                CFrameProxy* proxy = new CFrameProxy;
                proxy->InitFrameProxy(frameID, ODGetDraft(ev,storageUnit));
                fSourceFrame = proxy;
            }
        CATCH_ALL
            ODDeleteObject(fSourceFrame);
            fSourceFrame = kODNULL;
        ENDTRY

        TRY
            ODStorageUnitRef weakRef = {0,0,0,0};
            StorageUnitGetValue(storageUnit, ev, sizeof(ODStorageUnitRef),
                                (ODPtr)&weakRef);

            if ( storageUnit->IsValidStorageUnitRef(ev, weakRef) )
            {
                ODID frameID = storageUnit->GetIDFromStorageUnitRef(ev, weakRef);
                CFrameProxy* proxy = new CFrameProxy;
                proxy->InitFrameProxy(frameID, ODGetDraft(ev,storageUnit));
                fDependentFrame = proxy;
            }
        CATCH_ALL
            ODDeleteObject(fDependentFrame);
            fDependentFrame = kODNULL;
```

```
        ENDTRY
    }
}
```

The WritePartInfo Method

OpenDoc calls a part's `WritePartInfo` method for each of its display frames whenever the document is saved.

The `SamplePart` object's implementation of the `WritePartInfo` method calls the `CFrameInfo` object's `Externalize` method, which first gets a reference to the storage unit of the storage-unit view object passed with the call to `WritePartInfo`. The `Externalize` method then calls the `CFrameInfo` object's `CleanseFrameInfoProperty` method, which iterates through the value types in the storage unit and removes any that are not supported by SamplePart. Finally, `Externalize` calls the `CFrameInfo` object's `ExternalizeFrameInfo` method to actually write out the frame's part info data.

The `CFrameInfo` object's `ExternalizeFrameInfo` method works much the same as the `SamplePart` object's `ExternalizeStateInfo` method. That is, the method removes, then adds back, the value containing weak references in the storage unit. Then, the method writes weak references to its source frame, if any, and its dependent frame, if any. In both cases, if a draft key exists, the method creates a weak clone of the display frame and writes out the weak reference to the storage unit. The `SamplePart` object's `ExternalizeStateInfo` method is described in "The ExternalizeStateInfo Method" on page 102.

Listing 2-47 shows the implementation of the `SamplePart` object's `WritePartInfo` method, the `CFrameInfo` object's `Externalize` method, and the `CFrameInfo` object's `ExternalizeFrameInfo` method.

Listing 2-47 `WritePartInfo`, `CFrameInfo::Externalize`, and `CFrameInfo::ExternalizeFrameInfo` methods

```
void SamplePart::WritePartInfo( Environment*        ev,
                                ODInfoType          partInfo,
                                ODStorageUnitView*  storageUnitView )
{
    SOM_Trace("SamplePart","WritePartInfo");
```

```
    ((CFrameInfo*) partInfo)->Externalize(ev, storageUnitView);
}

void CFrameInfo::Externalize(Environment* ev, ODStorageUnitView* storageUnitView)
{
    ODStorageUnit* storageUnit = storageUnitView->GetStorageUnit(ev);
    this->CleanseFrameInfoProperty(ev, storageUnit);
    this->ExternalizeFrameInfo(ev, storageUnit, kODNULLKey, kODNULL);
}

void CFrameInfo::ExternalizeFrameInfo(Environment* ev, ODStorageUnit* storageUnit,
                                        ODDraftKey key, ODFrame* scopeFrame)
{
    if ( storageUnit->Exists(ev, kODNULL, kSamplePartInfo, 0) )
    {
        storageUnit->Focus(ev, kODNULL, kODPosSame, kSamplePartInfo, 0,
                                                        kODPosUndefined);
        storageUnit->Remove(ev);
    }
    storageUnit->AddValue(ev, kSamplePartInfo);

    {
        ODStorageUnitRef weakRef = {0,0,0,0};

        if ( fSourceFrame )
        {
            ODID     frameID = fSourceFrame->GetID();
            ODID     scopeFrameID = ( scopeFrame ? scopeFrame->GetID(ev) : kODNULLID );
            ODDraft* fromDraft = fSourceFrame->GetDraft();

            if ( key )
                frameID = fromDraft->WeakClone(ev, key, frameID, kODNULLID,
                                                        scopeFrameID);
            storageUnit->GetWeakStorageUnitRef(ev, frameID, weakRef);
        }
        StorageUnitSetValue(storageUnit, ev, sizeof(ODStorageUnitRef),
                                                        (ODPtr)&weakRef);
    }
    {
        ODStorageUnitRef weakRef = {0,0,0,0};
```

```
        if ( fDependentFrame )
        {
            ODID     frameID = fDependentFrame->GetID();
            ODID     scopeFrameID = ( scopeFrame ? scopeFrame->GetID(ev) : kODNULLID );
            ODDraft* fromDraft = fDependentFrame->GetDraft();

            if ( key )
                frameID = fromDraft->WeakClone(ev, key, frameID, kODNULLID,
                                                                    scopeFrameID);
            storageUnit->GetWeakStorageUnitRef(ev, frameID, weakRef);
        }
        StorageUnitSetValue(storageUnit, ev, sizeof(ODStorageUnitRef),
                                                                    (ODPtr)&weakRef);
    }
}
```

The ClonePartInfo Method

OpenDoc calls a part's `ClonePartInfo` method when any of its display frames is cloned during data transfer. Generally, a part editor should respond to the `ClonePartInfo` method call by writing out the frame's part info data, including any additional objects to which the part has strong persistent references and which are within the scope of the specified frame.

The `SamplePart` object's implementation of `ClonePartInfo` calls the `CloneInto` method of the `CFrameInfo` helper object holding the specified frame's part info data. The `CFrameInfo` implementation of `CloneInto` gets the storage unit, prefocused to a property but not to a value, and writes out the frame's part info data by calling the `CFrameInfo` object's `ExternalizeFrameInfo` method, which is shown in Listing 2-47 on page 110.

Listing 2-48 shows the implementation of the `SamplePart` object's `ClonePartInfo` method and the `CFrameInfo` object's `CloneInto` method.

Listing 2-48 `ClonePartInfo` and `CFrameInfo::CloneInto` methods

```
void SamplePart::ClonePartInfo( Environment*        ev,
                                ODDraftKey          key,
                                ODInfoType          partInfo,
```

```
                                ODStorageUnitView*  storageUnitView,
                                ODFrame*            scopeFrame )
{
    SOM_Trace("SamplePart","ClonePartInfo");

    ((CFrameInfo*) partInfo)->CloneInto(ev, key, storageUnitView, scopeFrame);
}

void CFrameInfo::CloneInto(Environment *ev, ODDraftKey key,
                            ODStorageUnitView* storageUnitView, ODFrame* scopeFrame)
{
    ODStorageUnit* storageUnit = storageUnitView->GetStorageUnit(ev);

    if ( storageUnit->Exists(ev, kODNULL, kSamplePartInfo, 0) == kODFalse )
    {
        this->ExternalizeFrameInfo(ev, storageUnit, key, scopeFrame);
    }
}
```


The Release Method

A part's `Release` method is called by an object, such as another part editor, whenever it releases a reference to this part. The `Release` method is inherited from the `ODRefCntObject` class, and the inherited implementation does the actual reference-count management. The `som_SamplePart` object's `Release` method calls the inherited method before it calls the `SamplePart` object's `Release` method described in this section (see also "SamplePart System Object Model Interface" on page 32).

The `SamplePart` object's implementation of the `Release` method releases the part-wrapper object to which the `fSelf` field points, if its reference count falls to 0.

Listing 2-49 shows the implementation of the `Release` method.

Listing 2-49 The `Release` method

```
void SamplePart::Release( Environment* ev )
{
    SOM_Trace("SamplePart","Release");
```

```
    if ( fSelf->GetRefCount(ev) == 0 )
        ODGetDraft(ev,fSelf)->ReleasePart(ev,fSelf);
}
```

The ReleaseAll Method

OpenDoc calls a part's `ReleaseAll` method when the part object is about to be deleted by its draft. At this point, the part must release all the references it has acquired to other reference-counted objects. Otherwise, it will cause an invalid reference count error at some later time. This method is inherited from the `ODPersistentObject` class. The `som_SamplePart` object's `ReleaseAll` method calls the inherited method after it calls the `SamplePart` object's `ReleaseAll` method described in this section (see also "SamplePart System Object Model Interface" on page 32).

The `SamplePart` object's implementation of the `ReleaseAll` method performs the following actions:

1. **Cleans up the SamplePart global variables.**

   The `ReleaseAll` method first ensures that the global variables are no longer needed. The global variables are shared among all instances of the `SamplePart` class that are currently running, and each instance increments a usage count accordingly. The method decrements the usage count. If the usage count reaches 0, the method releases the menu bar object, deletes the user interface focus set object, and deletes the global variables structure.

2. **Cleans up the part's display frame list.**

   The `ReleaseAll` method first ensures that the part's display frame list is not null. SamplePart maintains proxy display frame objects in its list to support lazy internalization: the actual frames are not read into memory until they are needed. The method iterates through the display frame list, removing the pointer for each proxy from the list and deleting the proxy object. Then the method deletes the frame list object itself.

The `ODDeleteObject` and `ODReleaseObject` utility macros, used in the `ReleaseAll` method to delete objects and release reference-counted objects, are defined in the ODUtils.h file.

Listing 2-50 shows the implementation of the `ReleaseAll` method.

Listing 2-50 The `ReleaseAll` method

```
void SamplePart::ReleaseAll( Environment* ev )
{
    SOM_Trace("SamplePart","ReleaseAll");

    TRY
        if ( --gGlobalsUsageCount == 0 )
        {
            ODReleaseObject(ev, gGlobals->fMenuBar);
            ODDeleteObject(gGlobals->fUIFocusSet);
            ODDeleteObject(gGlobals);
        }

        if ( fDisplayFrames )
        {
            CListIterator fiter(fDisplayFrames);
            for ( CFrameProxy* proxy = (CFrameProxy*) fiter.First();
                    fiter.IsNotComplete(); proxy = (CFrameProxy*) fiter.Next() )
            {
                fiter.RemoveCurrent();
                delete proxy;
            }
            ODDeleteObject(fDisplayFrames);
        }
    CATCH_ALL
        RERAISE;
    ENDTRY
}
```

The Purge Method

When OpenDoc detects a possible shortage of memory, it may call a part's `Purge` method. The part should free as much memory as possible. OpenDoc passes in the requested number of bytes to free with the method call. Obviously, parts should not free any resources they need to keep running.

The `SamplePart` object's implementation of the `Purge` method performs the following actions:

1. **Checks the view type of each of its display frames.**

   The method checks first to see if its internal list of display frames has been created. If not, there is no storage to free, so the method returns. Otherwise, the method iterates through all the frame proxy objects in its display frames list. If the frame associated with the proxy has not been loaded into memory. the method ignores it.

2. **Releases the unused thumbnail resource.**

   The method ensures that no frame has a view type of thumbnail, as determined in its previous iteration of its frame list, and that the thumbnail resource was previously read into memory. If these conditions prevail, the method increments its count of freed bytes by the size of the resource, releases the resource, and sets to null its global variable that points to the resource.

3. **Returns the cumulative count of the number of bytes freed.**

Listing 2-51 shows the implementation of the `Purge` method.

Listing 2-51 `Purge` method

```
ODSize SamplePart::Purge( Environment*  ev,
                          ODSize        /*size*/ )
{
    SOM_Trace("SamplePart","Purge");

    if ( fDisplayFrames == kODNULL ) return 0;

    ODSize      bytesFreed      = 0;
    ODBoolean   usingThumbnail  = kODFalse;

    CListIterator fiter(fDisplayFrames);
    for ( CFrameProxy* proxy = (CFrameProxy*) fiter.First();
            fiter.IsNotComplete(); proxy = (CFrameProxy*) fiter.Next() )
    {
        if ( proxy->FrameIsLoaded() )
        {
            ODTypeTokenframeView = proxy->GetFrame(ev)->GetViewType(ev);
```

```
            if ( frameView == gGlobals->fThumbnailView )
                usingThumbnail = kODTrue;
            proxy->Purge(ev);
        }
    }
    if ( !usingThumbnail && (gGlobals->fThumbnail != kODNULL) )
    {
        bytesFreed += (ODSize) ODGetHandleSize(gGlobals->fThumbnail);
        ReleaseResource(gGlobals->fThumbnail);
        gGlobals->fThumbnail = kODNULL;
    }
    return bytesFreed;
}
```


The SetDirty Method

The `SamplePart` object's internal `SetDirty` method sets its dirty flag to true, indicating that part's data has been changed by the user. The part editor calls its own `SetDirty` method whenever it changes its content.

The `SetDirty` method performs the following actions:

1. **Checks the dirty flag and write status.**

   The implementation is protected by its own flag, the internal variable `fDirty`. If the flag is already true, or if the part's draft is read-only, the method body doesn't execute. Otherwise the method performs the subsequent steps.

2. **Sets the dirty flag to true.**

3. **Notifies the draft that its content has changed from its previous version.**

   The method gets access to the draft through the `ODGetDraft` utility method and calls its `SetChangedFromPrev` method.

Listing 2-52 shows the implementation of the `SetDirty` method.

Listing 2-52 `SetDirty` method

```
void SamplePart::SetDirty( Environment* ev )
{
    SOM_Trace("SamplePart","SetDirty");
```

```
    if ( !fDirty && !fReadOnlyStorage )
    {
        fDirty = kODTrue;
        ODGetDraft(ev,fSelf)->SetChangedFromPrev(ev);
    }
}
```

Defining Types and Constants

SamplePart uses several files containing definitions of types and constants, which are included in the file SamplePart.r. The files Types.r and SysTypes.r contain resource type and constant definitions for the Mac OS. The file CodeFragmentTypes.r contains the code fragment resource (`'cfrg'`) definition, which enables the Code Fragment Manager (on which the Mac OS implementation of SOM is built) to find the shared libraries in the part editor file. The file ODTypes.r contains OpenDoc's `'nmap'` type definition, the name-mapping resource template. SamplePart resources are described in "Defining Resources" on page 122.

The file StdDefs.h contains constant definitions for OpenDoc standard part kinds and categories, and various other resource types, icon sizes, and so forth. The file SamplePartDef.h defines constants specifically for SamplePart, as shown in Listing 2-54. The file SamplePartVers.h defines SamplePart's version resources, as described in the section "Version Numbers" on page 124.

Listing 2-53 shows the compiler include statements for these definition files.

Listing 2-53 SamplePart types and constant definitions includes

```
#define SystemSevenOrBetter 1   // so have the extended types
#define SystemSevenOrLater  1   // Types.r uses this variable

// -- MPW Rez includes --

#include "Types.r"
#include "SysTypes.r"
#include "CodeFragmentTypes.r"
```

```
// -- OpenDoc includes --

#ifndef __ODTYPES_R__
#include "ODTypes.r"
#endif

#ifndef SOM_Module_OpenDoc_StdDefs_defined
#include "StdDefs.r"
#endif

// -- SamplePart includes --

#ifndef _SAMPLEPARTDEF_
#include "SamplePartDef.h"
#endif

#ifndef _SAMPLEPARTVERS_
#include "SamplePartVers.h"
#endif
```

Listing 2-54 shows constants defined specifically for SamplePart in the file SamplePartDef.h. The use of most of these constants is explained in subsequent sections of this chapter, and some of the definitions are repeated there to make the explanations clearer. In the following listing, the designation `(CH)` in comments indicates symbols that probably need redefinition for your own part editor.

Listing 2-54 SamplePart constant definitions

```
// Class / editor ID (CH)
#define kPartClassName                  "som_SamplePart"
#define kSamplePartID                   "SampleCode::"kPartClassName

// Editor user string (CH)
#define kSamplePartEditorUserString     "SamplePart 1.0"

// Kind (CH)
#define kSamplePartKind                 kODISOPrefix "Apple:Kind:SamplePart"

// Kind user string (CH)
```

```
#define kSamplePartKindUserString       "SamplePart"

// Category (CH)
#define kSamplePartCategory             kODISOPrefix "Apple:Category:Sample Code"

// Category user string (CH)
#define kSamplePartCategoryUserString   Sample Code"

// SamplePart OSTypes (CH)
#define kSamplePartEditorOSType         'SPED'
#define kSamplePartViewerOSType         'SPVW'
#define kSamplePartDocumentOSType       'SPDC'
#define kSamplePartStationeryOSType     'sPDC'

// ISO strings (CH)
#define kMainPresentation               kODISOPrefix "SamplePart:Presentation:Main"
#define kSamplePartInfo                 kODISOPrefix "SamplePart:Display Frame Info"

// SamplePart defines
#define kBaseResourceID                 20001

// NMAP Resource IDs
#define kKindCategoryMapId              kBaseResourceID+1
#define kEditorKindMapId                kBaseResourceID+2
#define kEditorUserStringMapId          kBaseResourceID+3
#define kKindUserStringMapId            kBaseResourceID+4
#define kCategoryUserStringMapId        kBaseResourceID+5
#define kOldMacOSTypeMapId              kBaseResourceID+6

// Text items
#define kMenuStringResID                kBaseResourceID
#define kAboutTextID                    1
#define kDefaultContent1ID              2
#define kDefaultContent2ID              3

// Error messages
#define kErrorStringResID               kMenuStringResID+1
#define kErrStrFieldID                  3
#define kErrCantInitializePart          1
#define kErrCantOpenDocWindow           2
#define kErrCantOpenPartWindow          3
```

```
#define kErrRemoveFrame                4
#define kErrWindowGone                 5
#define kErrExternalizeFailed          6

// Bundles/FREFs
#define kDocumentBundle                kBaseResourceID
#define kEditorBundle                  kBaseResourceID+1
#define kViewerBundle                  kBaseResourceID+2
#define kDocumentFREF                  kBaseResourceID
#define kStationeryFREF                kBaseResourceID+1
#define kEditorFREF                    kBaseResourceID+2
#define kViewerFREF                    kBaseResourceID+3

// Icons
#define kLargeIcons                    1
#define kSmallIcons                    2
#define kDocumentIcons                 kBaseResourceID
#define kStationeryIcons               kBaseResourceID+1
#define kEditorIcons                   kBaseResourceID+2
#define kViewerIcons                   kBaseResourceID+3

// Pictures
#define kEditorIconPicture             kBaseResourceID
#define kThumbnailPicture              kBaseResourceID+1

// Dialog boxes & windows
#define kAboutBoxID                    kBaseResourceID
#define kErrorBoxID                    kBaseResourceID+1
#define kMacWindowTitleBarHeight       20
#define kALittleNudge                  4
#define kMinVertVisPortion             10
#define kMinHorzVisPortion             16

// Display frames
#define kFrameRemoved                  1
#define kFrameClosed                   0

// Geometry
#define kMaxImagingResolution          72 // dpi
```

Defining Resources

SamplePart uses Mac OS resources to define and store various types of structured data, as described in this section. Most of SamplePart's resources are defined in the file SamplePart.r. Other resource data used in SamplePart, such as `'PICT'` data, is contained in the file SamplePartOtherResources.rsrc. SamplePart.r and SamplePartOtherResources.rsrc are compiled to produce the resource objects files SamplePart.PPC.rsrc (for the PowerPC version) and SamplePart.68k.rsrc (for the 68K version).

OpenDoc-OLE Interoperability

OpenDoc provides interoperability with OLE (Object Linking and Embedding) technology, enabling OLE servers to be embedded in OpenDoc container parts and OpenDoc parts to be embedded in OLE applications. For your part to be embedded directly in an OLE application, it must have its own OLE class identifier (CLSID), an alphanumeric string which uniquely identifies your part to the OLE runtime system. On the Mac OS platform, parts maintain their CLSID in a resource of type `'olcr'`.

To obtain an OLE class identifier, you must contact Microsoft Corporation. To do so, log on to CompuServe, enter `GO "WINOBJ"` and post a public or private message requesting the number of class identifiers you need. Include your company's name, address, and telephone number. If you don't specify a number, Microsoft allocates a block of 250 OLE class identifiers for each request.

In your `'olcr'` resource, you must represent your CLSID as a string delimited by braces and double quotation marks. Listing 2-55 shows SamplePart's OLE interoperability resource definition.

Listing 2-55 SamplePart OLE interoperability resource

```
resource 'olcr' (0, "OLE Class ID")
{
"{80C11F40-7503-8576-00D01113F11}";
};
```

Menu IDs

Menu IDs on the Mac OS are positive short integers, and hierarchical menu IDs must be in the 1-byte range. (Negative values are reserved.) Menu IDs must be the same as the menu resource ID, if the menu is resource based.

Because all menus installed in the Mac OS menu bar must have unique IDs, there is potential conflict among the OpenDoc document shell and container applications, shell plug-ins and services, and the currently active part. Therefore, you should use the following ranges for regular menus:

Document shell and container applications	255–16383
Plug-ins and services	16384–20000
Part editors	20001–32767

You should use the following ranges for hierarchical menus:

Document shell and container applications	0–127
Plug-ins and services	128–193
Part editors	194–254

Since there may be multiple plug-ins or services, they must assign their IDs dynamically. That is, a plug-in should choose an ID within the specified range and look for a menu with that ID. If one is found, the plug-in should add 1 to the ID and try again.

Bundle Resources

SamplePart includes certain resource definitions for its own purposes, such as icons for its icon views. The Finder uses bundle resources (type `'BNDL'`) to associate conventional applications and documents with each other and with the icons it displays to the user for them. The bundle resource contains the application's four-letter signature and the resource ID numbers of its signature, icon list, and file reference resources. Refer to *Inside Macintosh: Macintosh Toolbox Essentials* for more information about bundle resources and the Finder interface.

In OpenDoc, a part editor has two or more bundle resources; SamplePart has three: editor, viewer, and document. Bundle resources associate part editor and viewer icon families (specified according to type) with the shared library file containing their executable code (specified according to signature). Whereas conventional Mac OS applications have a type of `'APPL'`, part editors and viewers have the type `'shlb'`. SamplePart's editor signature is `'SPED'` and its viewer signature is `'SPVW'`.

OpenDoc parts are not owned by their part editors in the way conventional Mac OS documents are owned by their creator applications. Rather, all OpenDoc documents have a creator type of `'odtm'`. They are associated with their Finder icons by means of their type, which can be unique because it does not need to indicate the type of data contained by the document. You could use the same four-letter code to represent your part editor and documents. SamplePart documents, however, use the type `'SPDC'`.

Stationery documents are very important in OpenDoc because users typically see and manipulate stationery—double-clicking or dragging a stationery icon in the Finder—to create an instance of a part. Stationery documents differ from regular documents only in the Finder Info bit that is set for stationery; they share the same file type and creator. You must include a stationery icon family in your document bundle resource with a file type identical to your document file type except that its first letter is a lowercase `s` (regardless of the first letter of the document file type). The Finder then makes the proper association between the stationery icons and documents with their stationery bit set. So, SamplePart's stationery icons are associated with a type of `'sPDC'`.

Editor, viewer, and document signatures should be registered with Apple Developer Technical Support. SamplePart maintains these definitions in the file SamplePartOtherResources.rsrc. You can view the resources themselves using a resource editor such as ResEdit.

Version Numbers

Parts need to maintain three separate sets of version numbers that are necessary to the part's correct operation: CFM (Code Fragment Manager) version numbers, Finder file version numbers, and SOM class version numbers. These version numbers should be synchronized.

All of these types of version numbers include one major and one minor number, separated by a decimal point. In addition, CFM and Finder version numbers include a second minor version number called the *fix version.* For example, in the version number 2.3.1, the major portion is 2, and the minor portion is 3.1. Major version numbers have a range of 0–99; both minor version numbers have a range of 0–9. (The Code Fragment Manager and the Finder also provide for development stage designations. You can ignore these designations, but be aware that the Code Fragment Manager uses them if provided.)

Version numbers are maintained in binary-coded-decimal format. For example, using hexadecimal notation to represent the binary, the version number 2.3.1 is represented as 0x0231. Note that the 3 and the 1 are both in the same byte.

CFM version numbers enable the Code Fragment Manager of the Mac OS to find and load the correct version of the part editor's shared library. CFM version numbers are defined in the part's `'cfrg'` resource, shown in Listing 2-58 on page 128. The Finder version numbers are displayed by the Finder in response to the Get Info menu command. SOM version numbers are used in the SOM Interface Definition Language (.idl) file; they ensure compatibility between the definition and implementation of SOM classes.

SamplePart includes a file, SamplePartVers.h, that contains a set of compiler definitions to generate all three version numbers correctly. This file is included in SamplePart.r and som_SamplePart.idl. In addition, you must specify the correct version numbers to the linker in your makefile (for MPW) or project preferences (for integrated environments).

CFM version numbers are explained in *Inside Macintosh: PowerPC System Software*. Finder version numbers are explained in *Inside Macintosh: Macintosh Toolbox Essentials*. SOM version numbers are discussed in the *SOMobjects Developer Toolkit Users Guide* from IBM.

Listing 2-56 shows SamplePart's version number constant definitions. These constants are used in the file SamplePart.r, as shown in Listing 2-57, which shows SamplePart's Finder version resource definitions, and Listing 2-58, which shows SamplePart's code fragment resource definition. The version number constants are used again in the file som_SamplePart.idl, which is described in Appendix B, "System Object Model."

Listing 2-56 SamplePart version number definitions

```
// Development stages
#define dsUndefined     0x00
#define dsPreAlpha      0x20
#define dsAlpha         0x40
#define dsBeta          0x60
#define dsFinal         0x80
#define dsReleased      dsFinal
#define dsGoldenMaster  dsFinal
```

```
// • Change often •

// Current major version (version = major.minor.fix)
#define currentMajorVersion 1

// Current minor version (version = major.minor.fix)
#define currentMinorVersion 0

// Current fix version (version = major.minor.fix)
#define currentFixVersion   0

// Development stage
#define developmentStage    dsFinal

// Prerelease number
#define preReleaseNumber    0

// Short version string
#define shortVersionStr     "1.0"



// • Change seldom •

// Old compatibility definition major version (for CFM only)
#define oldCompDefnMajorVersion 0

// Old compatibility definition minor version (for CFM only)
#define oldCompDefnMinorVersion 0

// Old compatibility definition fix version (for CFM only)
#define oldCompDefnFixVersion   0

// Prerelease number
#define oldCompDefnPreRelNumber 0

// Development stage
#define oldCompDefnDevStage     dsUndefined
```

```
// • Generated version numbers •
//       (Don't change!!)

#define currentVersion (currentMajorVersion<<24)+(currentMinorVersion<<20) \
                        +(currentFixVersion<<16)+(developmentStage<<8)+preReleaseNumber
#define compatibleVersion (oldCompDefnMajorVersion<<24)+(oldCompDefnMinorVersion<<20) \
                            +(oldCompDefnFixVersion<<16)+(oldCompDefnDevStage<<8) \
                            +oldCompDefnPreRelNumber
#define finderMinorVersion (currentMinorVersion<<4)+(currentFixVersion<<0)
```

Listing 2-57 SamplePart finder version resources

```
resource 'vers' (1) {
    currentMajorVersion,
    finderMinorVersion,
    developmentStage,
    preReleaseNumber,
    verUS,
    shortVersionStr,
    shortVersionStr", © Apple Computer, Inc. 1994-1995"
};

resource 'vers' (2) {
    currentMajorVersion,
    finderMinorVersion,
    developmentStage,
    preReleaseNumber,
    verUS,
    shortVersionStr,
    "OpenDoc™ Sample Code"
};
```

Code Fragment Resources

Because SOM on the Mac OS depends on the Code Fragment Manager, your part editor's shared library needs to include a code fragment (`'cfrg'`) resource. It is important that the name for the fragment description be the editor ID; if your development environment automatically assigns the name of the library file to the fragment description, you need to change it.

Listing 2-58 shows SamplePart's code fragment resource definition, from the file SamplePart.r.

Listing 2-58 SamplePart code fragment resource

```
resource 'cfrg' (0) {
    {   /* [1] */
#ifdef _68KBUILD_
        kMotorola,
#else
        kPowerPC,
#endif
        kFullLib,
        currentVersion,
        compatibleVersion,
        kDefaultStackSize,
        kNoAppSubFolder,
        kIsLib,
        kOnDiskFlat,
        kZeroOffset,
        kWholeFork,
        kSamplePartID,  /* This must be the class ID */
        /* [2] */
#ifdef _68KBUILD_
        kMotorola,
#else
        kPowerPC,
#endif
        kFullLib,
        currentVersion,
        compatibleVersion,
        kDefaultStackSize,
        kNoAppSubFolder,
        kIsLib,
        kOnDiskFlat,
        kZeroOffset,
        kWholeFork,
        kPartClassName      /* This must be the SOM class name */
```

```
                    /* for this part */
    }
};
```

Name-Mapping Resources

Dynamic binding is the process by which OpenDoc matches a part editor at runtime to a part appearing in a document. On the Mac OS, dynamic binding is implemented through a set of six name-mapping resources (of type `'nmap'`) in the shared library files of part editors. These resources describe various aspects of the relationships between content and part editors. OpenDoc uses these resources to construct name spaces in the name-space object, maintained by the session object, when the user opens a document.

The name mappings that SamplePart defines are described in the following sections. SamplePart's `'nmap'` resources are defined in the file SamplePart.r. The constant definitions strings on which they depend are defined in the file SamplePartDef.h. The listings in the following sections combine fragments of these files to illustrate the name mappings.

Mapping Kind to Category

A part stores its content in its contents property as one or more part kind, and part kinds belong to one or more part category. OpenDoc requires part editors to map their part kinds to their part categories.

A part kind identifies its data format uniquely. A kind designation is an ISO string (7-bit ASCII) identifying the part kind, usually in a company-specific way, for proprietary and standard data types. For example, the following could be kind designations: `SurfCorp:SurfText`, `SurfCorp:Picture:BlackAndWhite`, and `SurfCorp:Picture:Color`.

The part category of a part's content defines a generic classification of its data format. For example, OpenDoc recognizes part categories of plain text, styled text, object-based graphics, 3D object-based graphics, and many others. For a list of OpenDoc's standard part categories with explanations, see the *OpenDoc Programmer's Guide for the Mac OS*.

A part's kind-to-category mapping must specify the category (or categories) for each kind the editor supports. A kind can belong to more than one category, in which case the categories are unordered. OpenDoc uses the information from this mapping to help the user define a default editor for each category.

Listing 2-59 shows SamplePart's very simple kind-to-category mapping.

Listing 2-59 Kind-to-category mapping

```
#define kSamplePartKind     kODISOPrefix "Apple:Kind:SamplePart"

#define kSamplePartCategory kODISOPrefix "Apple:Category:Sample Code"

#define kBaseResourceID     20001
#define kKindCategoryMapId  kBaseResourceID+1

resource kODNameMappings (kKindCategoryMapId) {
    kODKind,
    {   /* array Types: 1 elements */
        /* [1] */
        kSamplePartKind,
        kODIsAnISOStringList
        {
            {   /* array ClassIDs: 1 elements */
                /* [1] */
                kSamplePartCategory
            }
        }
    }
};
```

Mapping Editor to Kind

OpenDoc requires each part editor to map its editor identifier to its part kinds. An editor identifier represents a part editor. It comprises the editor's SOM module name and `ODPart` subclass name, separated by a double colon. A part kind designation represents the unique data format of a part editor, as described in the previous section.

A class identifier is associated with one or more part kind. The part kinds must be listed in decreasing order of fidelity.

Listing 2-60 shows SamplePart's editor-to-kind mapping.

Listing 2-60 Editor-to-kind mapping

```
#define kPartClassName      "som_SamplePart"
#define kSamplePartID       "SampleCode::"kPartClassName
#define kSamplePartKind     kODISOPrefix "Apple:Kind:SamplePart"
#define kEditorKindMapId    kBaseResourceID+2

resource kODNameMappings (kEditorKindMapId) {
    kODEditorKinds,
    {   /* array Types: 1 elements */
        /* [1] */
        kSamplePartID,
        kODIsAnISOStringList
        {
            {   /* array ClassIDs: 1 elements */
                /* [1] */
                kSamplePartKind
            }
        }
    }
};
```

Mapping ISO Strings to User-Readable Names

OpenDoc requires each part editor to map its editor identifier, part kind, and part category to user-readable strings. OpenDoc manipulates editor identifiers, part kinds, and part categories as ISO strings, which are not appropriate for display to the user. OpenDoc requires that user-readable text be in the form of international strings that can be in any script or language, so you must provide a name mapping resource to associate these three ISO string designations with user-readable names.

Listing 2-61 shows SamplePart's editor-to-string mapping.

Listing 2-61 Editor-to-string mapping

```
#define kPartClassName              "som_SamplePart"
#define kSamplePartID               "SampleCode::"kPartClassName
#define kSamplePartEditorUserString "SamplePart 1.0"
#define kEditorUserStringMapId      kBaseResourceID+3
```

```
resource kODNameMappings (kEditorUserStringMapId) {
    kODEditorUserString,
    {   /* array Types: 1 elements */
        /* [1] */
        kSamplePartID,
        kODIsINTLText
        {
            smRoman,
            langEnglish,
            kSamplePartEditorUserString
        }
    }
};
```

Listing 2-62 shows SamplePart's kind-to-string mapping.

Listing 2-62 Kind-to-string mapping

```
#define kSamplePartKind             kODISOPrefix "Apple:Kind:SamplePart"

#define kSamplePartKindUserString   "SamplePart"

#define kKindUserStringMapId        kBaseResourceID+4

resource kODNameMappings (kKindUserStringMapId) {
    kODKindUserString,
    {   /* array Types: 1 elements */
        /* [1] */
        kSamplePartKind,
        kODIsINTLText
        {
            smRoman,
            langEnglish,
            kSamplePartKindUserString
        }
    }
};
```

It is not necessary to provide user-readable names for OpenDoc's standard part categories because these names are already defined by OpenDoc. You should use these standard categories if at all possible. SamplePart is a bad example in this case—because SamplePart has no content, it cannot use any of the standard categories. OpenDoc's standard categories are listed in the *OpenDoc Programmer's Guide for the Mac OS*.

Listing 2-63 shows SamplePart's category-to-string mapping.

Listing 2-63 Category-to-string mapping

```
#define kSamplePartCategory           kODISOPrefix "Apple:Category:Sample Code"
#define kSamplePartCategoryUserString   "Sample Code"
#define kCategoryUserStringMapId        kBaseResourceID+5

resource kODNameMappings (kCategoryUserStringMapId) {
    kODCategoryUserString,
    {   /* array KeyList: 1 elements */
        /* [1] */
        kSamplePartCategory,
        kODIsINTLText {
            smRoman,
            langEnglish,
            kSamplePartCategoryUserString
        }
    }
};
```

Mapping Kind to Mac OS Type

OpenDoc requires part editors to provide a one-to-one mapping of their part kinds to platform-specific file types. On the Mac OS, this table maps part kinds to old-style Mac OS four-letter file types. When OpenDoc creates documents from stationery or dragging content to the Finder, it uses this resource to figure out the `OSType` file type of the resulting file. The Finder uses this information to associate the proper icon with its kind.

Listing 2-64 shows SamplePart's kind–to–Mac-OS-type mapping.

Listing 2-64 Kind–to–Mac-OS-type mapping

```
#define kSamplePartKind             kODISOPrefix "Apple:Kind:SamplePart"
#define kSamplePartDocumentOSType   'SPDC'
#define kOldMacOSTypeMapId          kBaseResourceID+6

resource kODNameMappings (kOldMacOSTypeMapId) {
    kODKindOldMacOSType,
    {   /* array KeyList: 1 elements */
        /* [1] */
        kSamplePartKind,
        kODIsMacOSType {
            kSamplePartDocumentOSType
        }
    }
};
```

Where To Go From Here

Contents

This chapter presents brief descriptions of code samples that illustrate part editor features not supported by SamplePart, such as embedding of other parts, data interchange via the Clipboard, linking, and drag and drop. These code samples are well commented and designed specifically to illustrate proper implementation of these features.

Information on these subjects is available in the *OpenDoc Programmer's Guide for the Mac OS*, in technical notes and engineering recipes included with OpenDoc releases, and via World Wide Web pages linked to the OpenDoc home page at the following universal resource locator (URL) address:

```
http://www.opendoc.apple.com
```

The following sections describe official sample part editors that ship with OpenDoc for the Mac OS.

SoundEditor

SoundEditor enables users to record, play back, save, and rerecord sounds from the current audio input device. It supports data stored in Mac OS `'snd '` format. SoundEditor uses the SOM-wrapper-object architecture used in SamplePart, which encapsulates its SOM interface in a class with almost no implementation.

SoundEditor implements the following features:

- displaying in the four standard view types (frame, large icon, small icon, and thumbnail)
- the View as Window command
- handling of its own menus
- Record, Pause, Stop, and Play commands
- Clipboard operations for sound data
- binding with existing Mac OS `'snd '` files
- Save command

PictureViewer

PictureViewer allows users to drop arbitrary `'PICT'` data into OpenDoc documents. Pictures can be cropped or scaled to the frame (the default mode is cropped). PictureViewer is a part viewer, not a part editor, so its content is not editable. PictureViewer is implemented as a straight SOM class; that is, it provides its own implementation and does not delegate to a separate C++ class in the manner of SamplePart.

PictureViewer implements the following features:

- displaying in the four standard view types (frame, large icon, small icon, and thumbnail)
- the View as Window command
- Clipboard operations for `'PICT'` data
- binding of Mac OS `'PICT'` files
- handling of its own Display menu for cropping or scaling
- printing
- Save command

TextEditor

TextEditor implements a feature set is similar to the Mac OS utility application SimpleText. TextEditor is implemented as a straight SOM class; that is, its `ODPart` subclass contains implementation.

TextEditor implements the following features:

- support of Mac OS text data as well as native data formats
- displaying in the four standard view types (frame, large icon, small icon, and thumbnail)
- Clipboard operations for text data
- handling of text style menus (Font, Size, Style)

- support of multiple languages
- Text Services Manager support (inline input for two-byte systems)
- drag and drop of any text
- scrolling when root part
- translation of foreign text types
- text ruler (with Show and Hide commands and as part preference)
- preferences for setting document margins, default font, and so on

DrawEditor

DrawEditor is based on QuickDraw and provides tools for editing and creating shapes. DrawEditor uses the SOM-wrapper-object architecture used in SamplePart, which encapsulates its SOM interface in a class with almost no implementation.

DrawEditor implements the following features:

- displaying in the four standard view types (frame, large icon, small icon, and thumbnail)
- Clipboard operations
- creation of shapes (rectangles, ovals, triangles, and lines)
- editing of shapes by resizing and z-ordering
- styling objects (pen color, pen pattern, pen width, fill color, and fill pattern)
- embedding
- drag and drop of any content
- linking of any content
- undo of all user actions
- floating tool and color palettes

ScriptRunner

ScriptRunner is an OSA (Open Scripting Architecture) scripting palette that works in conjunction with the TextEditor sample.

ScriptRunner implements the following features:

- an OpenDoc shell plug-in
- nonpersistent palette and result windows
- an `ODExtension` subclass for simple data transfer
- an `ODExtension` subclass for clients to control the palette window

Appendixes

OpenDoc Utilities

This appendix describes some of the unsupported utilities provided with the Mac OS implementation of OpenDoc for the convenience of developers. The utilities described in this appendix are those used by SamplePart. They comprise classes, types, macros, and functions implemented in files with the following names:

Except.cpp
FlipEnd.cpp
FocusLib.cpp
IText.cpp
ODMemory.cpp
ODUtils.cpp
StdTypIO.cpp
StorUtil.cpp
TempObj.cpp
UseRsrcM.cpp
WinUtils.cpp

These filenames also apply to header files containing class definitions, method declarations, and function, type, and macro definitions. The header files have the same filenames with the extension .h. To use these utilities, link the .cpp files into your library, and include the .h header files in your source files.

Note
All of these utilities are supplied with the Mac OS implementation of OpenDoc on an "as-is" basis. They have not received the rigorous quality-assurance testing given to OpenDoc itself, and they are not part of the official OpenDoc API. Source code is provided, and you are welcome to modify the routines as necessary. ◆

Exception Handling (Except)

This section describes the OpenDoc exception-handling utility, which resides in the files Except. h and Except.cpp. You can use the exception-handling utility to generate and handle your own exceptions as well as respond to any exceptions generated as a result of calls to OpenDoc. The exception-handling utility implements a simple throw-and-catch exception-handling scheme, similar to those found in some application frameworks and development environments.

Using the Exception-Handling Utility

The use of the exception-handling utility is optional. However, if you don't use it you must check the environment variable (`ev`) after every SOM call you make and handle it appropriately. The exception-handling utility facilitates this requirement, as described in "Handling SOM Exceptions" on page 149.

To use the exception-handling utility, you add the file Except.cpp to your project (if you use a project-based system) or makefile (if you use MPW) and include Except.h in your source files.

IMPORTANT

You must include the header Except.h in your source files *before* including the headers of any SOM classes (.xh files in C++). If you precompile any SOM headers, you should also precompile Except.h and put it first among the headers that you precompile. ▲

If you're building your project in debug mode (the symbol `ODDebug` is defined as `1`), then Except.cpp will call functions from Crawl.cpp, and you'll need to add that source file to your project or makefile. Crawl.cpp in turn depends on ToolLibs.o (68K) or PPCToolsLib.o (PowerPC), which are part of your development system.

The Exception-Handling Scheme

In the kind of exception system implemented by the exception-handling utility, an error is signaled by being thrown, or made known to the system, by using the macro call `THROW` or one of its variants. The stack then unwinds back to the point where a calling function has set up an error handler for this exception. The handler then catches, or responds to, the exception; it performs whatever recovery or cleanup is necessary. The error handler can then allow execution to continue or—more commonly—it can reraise the exception, throwing it back to the next exception handler on the stack.

An exception handler in this scheme is defined as a block of code, delimited by the macro calls `TRY`, `CATCH_ALL`, and `ENDTRY` (or their variants). Only exceptions that are thrown within the scope of the `TRY`/`ENDTRY` pair of a handler can be handled by that handler.

The following is an example of the most basic use of this kind of exception system. It shows the exception handling involved with the fail-safe allocation of a pair of handles:

```
ODHandle MyNewHandle(ODSize size)
{
    OSErr err;
    ODHandle h = NewTempHandle(size,&err);
    THROW_IF_ERROR(err);
    return h;
}
```

This function (`MyNewHandle`) throws an exception if the `NewTempHandle` function returns a nonzero error. `MyNewHandle` does not itself catch any exceptions. The next function (`TwoHandles`) uses `MyNewHandle` to allocate the pair of handles:

```
void TwoHandles( )
{
    Handle h1, h2;
    h1 = MyNewHandle(10000);
    TRY{
        h2 = MyNewHandle(10000);
    }CATCH_ALL{
        ODDisposeHandle(h1);
        RERAISE;
    }ENDTRY
    ...
}
```

In `TwoHandles`, if the first call to `MyNewHandle` fails, it throws an error. Since `TwoHandles` has not set up an exception handler around that call, the exception is thrown out of `TwoHandles` and into its caller, and so on up the stack until an exception handler is found. None of the code shown here handles that exception.

If the first call succeeds, however, execution passes to the second call to `MyNewHandle`, which is inside an exception handler. If this call fails and throws an exception, the exception is caught by the exception handler, and the block after `CATCH_ALL` executes. This block cleans up by disposing of the first allocated handle (`h1`), preventing a memory leak. Thus, either both handles are allocated or neither is.

After `h1` is disposed of, the exception handler calls `RERAISE`, which re-throws the same exception up the stack until the next enclosing exception handler is found. (If the handler hadn't called `RERAISE`, execution would have fallen out of the exception handler to the statement following `ENDTRY`.)

If the second call to `MyNewHandle` succeeds, execution falls out of the entire exception handler, skipping the `CATCH_ALL` block entirely (since there was no exception) and ending up at the statement immediately following `ENDTRY`.

Throwing Exceptions

In the OpenDoc exception-handling utility, you throw an exception by calling the `THROW` macro or one of its variants. This causes execution to jump immediately to the closest exception handler below it on the stack. These are the variants of `THROW`:

THROW

`THROW` throws an exception with the error number that is supplied to it. The error can be a standard OpenDoc error or a platform-specific error. The error code must be a nonzero value; it doesn't make sense to throw an exception whose value is `kODNoError`.

THROW_IF_NULL

`THROW_IF_NULL` throws the exception `kODErrOutOfMemory` if a null pointer is supplied to it. Call this macro after you call a memory-allocation function (such as `SOMNew` or `MMNewPtr`) that returns null when there is insufficient memory.

Do not use `THROW_IF_NULL` with functions that can return null for other reasons. For example, the Mac OS Resource Manager routine `GetResource` returns null if the resource cannot be found; in that case, you should first call ResError to find the actual error code, and then call THROW.

THROW_IF_ERROR

THROW_IF_ERROR throws an exception if the error supplied to it is nonzero. If the error value is kODNoError, nothing happens. This is a useful call to use following a function call whose return value is an error code.

For example, the Mac OS File Manager function FSpOpenDF returns zero if it succeeds, and otherwise a nonzero OSErr code. Passing the result to THROW_IF_ERROR ensures that the right exception is thrown if the call to FSpOpenDF fails.

Exception Handlers

An exception handler consists of a TRY block, zero or more CATCH_ALL blocks, and an ENDTRY:

```
TRY{
    // statements
}CATCH_ALL{
    // statements
}ENDTRY
```

It's perfectly legal lexically (and not uncommon) to nest exception handlers in a single function. Any error caught and reraised by the inner handler will be caught by the outer one.

The rest of this section describes what actions each of the macro statements and its associated code block perform.

TRY

Following a TRY macro, the immediately subsequent statements are executed. If one of the statements, or any function one of the statements calls, throws an exception that reaches this exception handler, then one of the following CATCH_ALL blocks may be executed. Otherwise, after the last statement in the TRY block finishes, control passes to the statement following the ENDTRY.

CATCH_ALL

If an exception is thrown to this handler, the statements following the `CATCH_ALL` macro are executed. To tell what error code was thrown, use the `ErrorCode` function.

The flow of control for the `CATCH_ALL` is the same as for `CATCH`. If no exception is raised or re-raised, control passes from the last statement in this `CATCH_ALL` block to the statement following the `ENDTRY`.

ENDTRY

The `ENDTRY` macro statement indicates the end of the exception handler. After a `TRY` or `CATCH` block finishes without throwing or re-raising an exception, the exception handler removes itself from the stack and control passes to the statement following `ENDTRY`.

RERAISE

The `RERAISE` macro statement is called within a `CATCH_ALL` block. It causes the exception to be thrown again, to the next active exception handler on the stack. This is the normal behavior for an exception handler —most of the time you don't want to hide the error, you want to propagate it so a higher level handler can deal with it.

The SOM Environment Parameter

OpenDoc objects are SOM objects, which means that they follow the CORBA rules for handling exceptions. Every method call made to an OpenDoc object (including your part, as a subclass of `ODPart`) must therefore include an environment parameter (`ev`), a pointer to a value that can describe an error. For example, the `CreateLinkSource` method of `ODDraft` has the following prototype (in IDL):

```
ODLinkSource CreateLinkSource(in ODPart part);
```

The method takes a single parameter, of type `ODPart`. To use this method, however, a caller in C++ must supply two parameters:

```
MyLinkSource = MyDraft->CreateLinkSource(ev, somSelf);
```

If execution of the method results in an error condition, the receiver of the call (the draft object in this case) must place an exception code in the value pointed to by `ev` and return. The caller must therefore examine the `ev` parameter after every call to a SOM object, to see if an exception has been raised.

All OpenDoc methods that you call, as well as all public methods of your part editor that you write, must return errors this way. What this means for your exception handling is that

- you must supply an environment variable with all method calls to OpenDoc objects
- you must check the environment variable after the call returns

The environment variable is passed along through a sequence of calls and can be used in calls to both SOM and C++ objects. For example, the environment variable is passed in these situations:

- If your C++ method (that does not itself receive an environment parameter) calls a SOM method, in which case it must use a SOM utility method to retrieve the environment variable.
- If your SOM method calls another SOM method, it can simply pass on the environment parameter it receives.
- If your SOM method calls a C++ method that may in turn call a SOM method, your SOM method can pass the environment parameter on to the C++ method (if the C++ method was designed to accept it; see next bullet).
- If your C++ method is called by a SOM method and in turn makes calls to SOM methods, it is best to design it to accept an environment parameter that it can then pass on.

For more information on the environment parameter and exceptions, see *SOMobjects Developer Toolkit Users Guid*e and *SOMobjects Developer Toolkit Programmers Reference Manual* from IBM.

Any exception-handling scheme that you use must support this method of passing exceptions. The OpenDoc utility described in this section helps you check the environment variable after each method call.

Handling SOM Exceptions

The exception-handling utility has some special features that simplify working with SOM. There are two reasons why these features are necessary:

- SOM has its own way of returning error codes, based on an environment variable, a pointer to which is passed into every method of a SOM object.
- You cannot throw an exception, or allow one to be thrown, out of a SOM method. SOM requires that a method return normally, and throwing an exception that is caught by a handler in some other function farther up the stack would violate this.

This implies that the `ev` parameter must be checked for an error value after every call to a SOM method and that an exception raised in a SOM method or any function it calls must be caught and its error code stored in the `ev` parameter. The exception-handling utility includes functions to simplify these tasks, which are variants of the exception handling macros previously introduced:

```
SOM_TRY
SOM_CATCH_ALL
SOM_ENDTRY
```

These macros are identical to `TRY`, `CATCH_ALL`, and `ENDTRY`, except that when they catch an exception, they store the exception value in the method's `ev` parameter where the caller can see it.

Because you cannot throw an exception out of a SOM method, it is illegal to `RERAISE` in the `SOM_CATCH_ALL` block. You should exit the function normally by falling off the end or calling `return` (in C++ the former generates slightly better code).

IMPORTANT

`SOM_ENDTRY` works differently than `ENDTRY` in that its default behavior is to reraise the exception by storing the error information in the `Environment` variable so it is propagated to the caller. (With `ENDTRY` you must explicitly reraise in your `CATCH_ALL` block or the exception will disappear.) If you don't want to return the exception to the caller, you must call `SetErrorCode(kODNoError)` in the `SOM_CATCH` block. ▲

Automatic Environment Checking

If you include the header Except.h in your source files, it defines a special preprocessor symbol that modifies the way SOM messages are sent. Any SOM headers (.xh files for development in C++) included after Except.h has been included are modified so that, after the message is sent and control returns to the caller, the environment variable (ev) is checked and an exception raised if the variable contains an error.

For example, the following code fragment does not use automatic environment checking:

```
#include <ODWingDing.xh>
...
long AFunction (Environment *ev, ODWingDing *wingDing)
{
    long result = wingDing->Spin(ev);
    if(ev->_major) {            // ODWingDing::Spin returns error
        result = 0;
        goto handle_error;
    }
    ...
handle_error:
    return result;
}
```

In this example, Except.h is not included, so environment checking is not automatic, and the caller (AFunction) can and must check the environment variable after every SOM method call.

Here is the same example with automatic environment checking:

```
#include <Except.h>             // Enables automatic ev checking
#include <ODWingDing.xh>
...
long AFunction (Environment *ev, ODWingDing *wingDing)
{
    long result;
    SOM_TRY
        long result = wingDing->Spin(ev);
        ...
    SOM_CATCH_ALL
        result = kODNULL;
    SOM_ENDTRY
    return result;
}
```

Since Except.h is included before ODWingDing.h, environment-checking code is added to the call to the `Spin` method of `ODWingDing`. If `Spin` encounters an error and returns error status in `ev`, an exception with that same error code is thrown, which will be caught by the `SOM_CATCH_ALL` exception handler, and in its turn returned in the `ev` parameter of `AFunction`.

There are two important precautions to keep in mind:

- You must include Except.h *before* including any headers that declare SOM classes if you want to use automatic environment checking for them.
- When using automatic environment checking, any SOM method call may throw an exception, so any SOM method that calls other SOM methods must be prepared to handle exceptions.

Coding Precautions

To achieve its results as a C++ library, the OpenDoc exception-handling utility relies on complex macros and sophisticated library functions. To some extent, it "fools" the compiler. Because of this, there are some precautions you have to take to avoid causing the compiler to generate incorrect code.

Very few C++ compilers have intrinsic support for exceptions, so the OpenDoc exception-handling utility is based on the ANSI `setjmp` and `longjmp` calls. Because of this basis, the compiler cannot always track the possible flow of control when exceptions are thrown and caught. The compiler can generate code that improperly fails to pop an exception handler off the stack or that makes incorrect assumptions about flow of execution.

This section discusses the precautions you must follow to make sure that the complier makes no mistakes, even when you have nested exception handlers.

Make Variables That You Modify Volatile

The compiler's register allocator and optimizer can make incorrect assumptions and generate bad code unless you take this precaution: always declare as volatile any variable or parameter that you modify in a `TRY` block and then use in a `CATCH` or `CATCH_ALL` block.

The reason for this is that the compiler doesn't understand that the `TRY` block can be executed on the way to a `CATCH` block, and therefore that the variable may be modified before the `CATCH` block is reached. It may therefore end up using an obsolete value for the variable while in the `CATCH` block. To work around this, you have to tell the compiler not to store the variable's value in a register (which may be out of date) but always to look it up from the stack frame.

The C++ `volatile` keyword in the variable declaration does this (tells the compiler not to store the variable's value in a register). Unfortunately, some compilers don't implement it properly, and it can be confusing to use properly with pointer variables. For this reason the exception system defines a macro `ODVolatile` that declares a variable to be volatile. All you have to do is put this after the variable declaration. Here's an example:

```
void *p = kODNULL; ODVolatile(p);
TRY{
    Zog1();
    p = ODNewPtr(10000);
    Zog2();
}CATCH{
    ODDisposePtr(p);
    RERAISE;
}ENDTRY
```

The purpose of the exception handler is to make sure that `p` is disposed of on the way out in case it was allocated by `ODNewPtr`. Because `p` is modified inside the `TRY` block, it has to be marked as volatile. (Note that when the `CATCH` block is called, `p` might still be `NULL`—if `Zog1` or `ODNewPtr` threw the exception—or it might be a valid pointer, if it was `Zog2` that threw the exception. Fortunately, we pre-initialized `p` to `kODNULL`, and `ODDisposePtr` can safely be passed a null pointer. If we hadn't initialized `p`, this code might crash.)

Data Value Manipulation (FlipEnd)

This section describes the utilities defined in the files FlipEnd.h and FlipEnd.cpp. These routines are used to convert between big-endian (most-significant byte first) and little-endian (least-significant byte first) data values, which may be required for cross-platform data storage.

The utility assumes that big-endian platforms define the compiler switch `_PLATFORM_BIG_ENDIAN`. The standard format for the functions and macros defined in the utility is little endian. Therefore, using the utility to coerce data into standard format means you are writing data in little-endian format.

Conversion Functions

The following functions convert the indicated types of values to the opposite endian format in memory. Clients typically do not use these functions directly, because they always swap bytes, whether it is appropriate to do so or not on the current platform. Instead, clients normally use the macros (described in the following section) which convert to and from standard format.

ODFlipShort

The `ODFlipShort` function takes as a parameter a single 2-byte integer and returns the value with the opposite endian format. The prototype of this function appears as follows:

```
ODUShort ODFlipShort(ODUShort n);
```

ODFlipShortArray

The `ODFlipShortArray` function takes as parameters a pointer to an array of 2-byte integers and a count of the number of integers in the array. The function converts the endian format of each integer in the array. The prototype of this function appears as follows:

```
void ODFlipShortArray(ODUShort* a, unsigned long count);
```

ODFlipLong

The `ODFlipLong` function takes as a parameter a single 4-byte integer and returns the value with opposite endian format. The prototype of this function appears as follows:

```
ODULong ODFlipLong(ODULong n);
```

ODFlipLongArray

The `ODFlipLongArray` function takes as parameters a pointer to an array of 4-byte integers and a count of the number of integers in the array. The function converts the endian format of each integer in the array. The prototype of this function appears as follows:

```
void ODFlipLongArray(ODULong* a, unsigned long count);
```

ODFlipStruct

The `ODFlipStruct` function takes as parameters a pointer to a C++ structure and a pointer to a zero-terminated array of short integers. The prototype of this function appears as follows:

```
void ODFlipStruct(void* structure, const short* groups);
```

This function inverts the endian format of the contents of memory in the `structure` parameter, according to the layout described by the `groups` parameter. The `groups` parameter points to a zero-terminated array of shorts, where each short describes the size of the next chunk of memory in the structure to be processed. A negative value `-n` in the groups array indicates a block of endian-neutral memory, like a string, and causes `n` bytes of memory to be skipped over. A positive value `n` in the groups array indicates a block of `n` bytes of memory that should have its bytes flipped end for end. Only positive values in the set `{ 2, 4, 8 }` are handled. (Other positive values are handled like negative values: blocks of memory are skipped).

Here is an example of a structure and the groups array indicating how it should be converted:

```
struct snod {
        long  beta;
        char  gamma[8];
        long  delta;
        short alpha;
};

const short snodGroups[] = {
        4, // beta
        -8, // gamma
        4, // delta
        2, // alpha
        0, // zero-termination
};
```

Conversion Macros

The following macros convert to and from standard (little-endian) format. The macro definitions show what the macros do in terms of the functions described in the previous section. The `#define` statements following the `#ifdef` conditional define the macros for big-endian platforms; the statements following the `#else` conditional define the same macros correctly for little-endian platforms.

```
#ifdef _PLATFORM_BIG_ENDIAN_

#define ConvertODUShortToStd(n)        ODFlipShort(n)
#define ConvertODUShortFromStd(n)      ODFlipShort(n)

#define ConvertODSShortToStd(n)        \
                        ((ODSShort) ODFlipShort((ODUShort) n))
#define ConvertODSShortFromStd(n)        \
                        ((ODSShort) ODFlipShort((ODUShort) n))

#define ConvertODULongToStd(n)         ODFlipLong(n)
#define ConvertODULongFromStd(n)       ODFlipLong(n)

#define ConvertODSLongToStd(n)         \
                        ((ODSLong) ODFlipLong((ODULong) n))
#define ConvertODSLongFromStd(n)       \
                        ((ODSLong) ODFlipLong((ODULong) n))

#define ConvertODStructToStd(s, g)     ODFlipStruct((s),(g))
#define ConvertODStructFromStd(s, g)   ODFlipStruct((s),(g))

#define ConvertODUShortArrayToStd(a,c)    ODFlipShortArray((a),(c))
#define ConvertODUShortArrayFromStd(a,c)  ODFlipShortArray((a),(c))

#define ConvertODSShortArrayToStd(a,c)   \
                        ODFlipShortArray((ODUShort*)(a),(c))
#define ConvertODSShortArrayFromStd(a,c) \
                        ODFlipShortArray((ODUShort*)(a),(c))

#define ConvertODULongArrayToStd(a,c)     ODFlipLongArray((a),(c))
#define ConvertODULongArrayFromStd(a,c)   ODFlipLongArray((a),(c))
```

```
#define ConvertODSLongArrayToStd(a,c)    \
                            ODFlipLongArray((ODULong*)(a),(c))
#define ConvertODSLongArrayFromStd(a,c)  \
                            ODFlipLongArray((ODULong*)(a),(c))

#else

#define ConvertODUShortToStd(n)        (n)
#define ConvertODUShortFromStd(n)      (n)

#define ConvertODSShortToStd(n)        (n)
#define ConvertODSShortFromStd(n)      (n)

#define ConvertODULongToStd(n)         (n)
#define ConvertODULongFromStd(n)       (n)

#define ConvertODSLongToStd(n)         (n)
#define ConvertODSLongFromStd(n)       (n)

#define ConvertODStructToStd(s, g)     /* do nothing */
#define ConvertODStructFromStd(s, g)   /* do nothing */

#define ConvertODUShortArrayToStd(a,c)    /* do nothing */
#define ConvertODUShortArrayFromStd(a,c)  /* do nothing */

#define ConvertODSShortArrayToStd(a,c)    /* do nothing */
#define ConvertODSShortArrayFromStd(a,c)  /* do nothing */

#define ConvertODULongArrayToStd(a,c)     /* do nothing */
#define ConvertODULongArrayFromStd(a,c)   /* do nothing */

#define ConvertODSLongArrayToStd(a,c)     /* do nothing */
#define ConvertODSLongArrayFromStd(a,c)   /* do nothing */

#endif /* _PLATFORM_BIG_ENDIAN_ */
```

QuickDraw Focus Library (FocusLib)

This section describes the utilities defined in the files FocusLib.h and FocusLib.cpp. These utilities are useful for setting up the drawing environment to render into a facet for Mac OS part editors using the classic QuickDraw imaging environment. Using QuickDraw GX is discussed in the recipe *QuickDraw GX and OpenDoc*.

What the Focus Library Does

The term *focus*, as used in the focus library, is only dimly related to the regular OpenDoc concept of a focus. The term comes from the `Focus` method in MacApp, which sets up QuickDraw to draw into a view.

Focusing does the following things:

- Makes the facet's canvas the current graphics port (`GrafPort`).
- Moves the origin of the graphics port to the origin of the frame's coordinate system, based on the internal and external transformations. In other words, (0,0) to QuickDraw is now the same place as (0,0) in your frame's coordinates.
- Sets the clip region to the facet's clip shape, to prevent you from drawing outside of the facet.

Once your drawing environment is focused, you can start issuing QuickDraw commands (or doing higher level things that use QuickDraw) using your frame's coordinate system.

What the Focus Library Does Not Do

The focus library sets up the QuickDraw environment, so it cannot set up any kind of drawing state or transformations that QuickDraw does not understand. In particular, it does not handle any type of transformations other than offsets. If your facet ends up scaled, rotated, or skewed, the focus library helps you only with the offset portion of the transformation. You can do the rest of the transformation manually by transforming the coordinates of all points before you draw them.

Transformations other than scaling are particularly hard to handle in QuickDraw, which provides no native facilities for rotating text, bitmaps, or ellipses. QuickDraw GX handles all kinds of transformations automatically.

Using the Focus Library From C++

Using the focus library is easy. The usual way, for C++ clients, is to declare a `CFocus` object on the stack. When the object is constructed, the focusing takes place. When the object goes out of scope and is destroyed, the previous state of QuickDraw is restored. For example:

```
void DrawMyStuff( Environment *ev, ODFacet *facet ) {
    CFocus foc(ev,facet);
    MoveTo(0,0);
    LineTo(100,100);
}
```

There are three variants of `CFocus`, described in the following sections.

CFocusWindow

The `CFocusWindow` class sets the window, not the facet's canvas, as the current graphics port. There is no difference, unless your facet is on an offscreen canvas. In that case, a regular `CFocus` would not cause the drawing to appear immediately on screen since it would first go into the offscreen canvas until the next update event. For interactive use such as rubber-banding a line or object while the mouse is down, use `CFocusWindow` to ensure that things are drawn immediately to the screen.

CFocusFrame

The `CFocusFrame` class does not take into account the frame's internal transformation. This means that (0,0) will be the top-left corner of the facet. This is useful when drawing frame adornments such as borders or scroll bars instead of the actual contents.

CFocusWindowFrame

The `CFocusWindowFrame` class is a combination of `CFocusWindow` and `CFocusFrame`.

The constructors of any of the CFocus classes take an optional extra parameter, which is a pointer to an ODShape object. If it is supplied, drawing is further clipped to the intersection of that shape and the facet's clip shape. This is useful when drawing into only part of the facet (as when handling a Draw method call).

Using the Focus Library From C

If you are using C, or do not use C++ features like constructors, you can explicitly call the BeginFocus and EndFocus functions. For example:

```
void DrawMyStuff( Environment *ev, ODFacet *facet ) {
    FocusState state;
    BeginFocus(ev,&state,facet,kODTrue,kODFalse,kODNULL);
    MoveTo(0,0);
    LineTo(100,100);
    EndFocus(&state);        // Must explicitly end focusing!
}
```

You must declare a FocusState variable and then call BeginFocus, whose parameters look like this:

```
void BeginFocus( Environment *ev, FocusState*, ODFacet*,
    ODBoolean toContent, ODBoolean toWindow, ODShape *clipTo );
```

The toContent parameter determines whether to clip to the frame's content (as in CFocus) or to the frame border (as in CFocusFrame).

The toWindow parameter determines whether to draw directly into the window (as in CFocus) or into the facet's canvas (as in CFocusWindow).

The clipTo parameter, if not kODNULL, is an ODShape to which drawing is clipped.

IMPORTANT

It is important that you always call EndFocus after BeginFocus. If you don't, the drawing state is not restored and you will leak some memory. If you use exceptions, and anything between BeginFocus and EndFocus could throw an exception, you need to catch the exception and call EndFocus before re-raising it. (The C++ classes are based on the Destructo class, so they always clean up automatically.) ▲

PostScript Printing

The focus library takes care of some tricky situations in PostScript printing. The LaserWriter driver does not handle QuickDraw Regions, so any attempt to clip to a nonrectangular area is ignored in the PostScript output. Not being able to clip to nonrectangular areas is a problem, since facets are often clipped to nonrectangular areas.

To work around this, the focus library includes two utility functions that emit some fancy PostScript code to set the clipping properly. If you are using the focus library calls described previously, these functions are called automatically and you don't need to worry about them. You need to know about these calls only if you do not want to use the rest of the focus library.

`ODBeginPostScriptClip` emits PostScript code to clip to the `ODShape` object passed in (in the coordinate system of the current graphics port). `ODEndPostScriptClip` ends the clipping. These functions will have no effect unless the current graphics port is in fact a printing port that is printing via the LaserWriter driver.

International Text (IText)

This section describes the utilities defined in the files IText.h and IText.cpp. These utilities create, destroy, and manipulate international text (`ODIText`) structures, which contain a variable-size text buffer as well as Mac OS script and language codes.

Creation in default heap

The following functions create an `ODIText` structure using a C string. On the Mac OS, the `ODScriptCode` and the `ODLangCode` parameters correspond to the platform script code and language code. The `CreateIText` function is overloaded to use different types of input parameters, as shown throughout this section.

```
ODIText* CreateITextCString(ODScriptCode script, ODLangCode lang,
                                                    char* text);
```

```
ODIText* CreateIText(ODScriptCode script, ODLangCode lang, char* text);
```

The following functions create an `ODIText` structure using a Pascal string.

```
ODIText* CreateITextPString(ODScriptCode script, ODLangCode lang,
                                                     StringPtr text);

ODIText* CreateIText(ODScriptCode script, ODLangCode lang,
                                                     StringPtr text);
```

The following functions create an `ODIText` structure with an empty string of specified length.

```
ODIText* CreateITextClear(ODScriptCode script, ODLangCode lang,
                                                 ODSize stringLength );

ODIText* CreateIText(ODScriptCode script, ODLangCode lang,
                                                 ODSize stringLength );
```

The following functions create an `ODIText` structure with a buffer of characters of specified length.

```
ODIText* CreateITextWLen(ODScriptCode script, ODLangCode lang,
                                     ODUByte* text, ODSize textLength );

ODIText* CreateIText(ODScriptCode script, ODLangCode lang,
                                     ODUByte* text, ODSize textLength)
```

The following function sets the buffer size of the `ODIText` structure. If the input `ODIText` pointer is `kODNULL`, this function is equivalent to the `CreateITextClear` function.

```
ODIText* SetITextBufferSize(ODIText* text, ODSize bufferSize,
                                         ODBoolean preserveContents );
```

Destruction

The following function disposes of an `ODIText` structure and any memory associated with it.

```
void  DisposeIText(ODIText* text);
```

The following function is the same as the `DisposeIText` function except that it works on `ODIText` structure allocated on the stack.

```
void  DisposeITextStruct(ODIText text);
```

Duplication

The following function allocates and returns an exact copy of the `ODIText` structure passed in.

```
ODIText* CopyIText(ODIText* original);
```

The following function is the same as the `CopyIText` function except that the returned `ODIText` structure is allocated on the stack.

```
ODIText  CopyITextStruct(ODIText* original);
```

Accessing attributes

The following functions set and get the script code of the `ODIText` structure passed in.

```
void  SetITextScriptCode(ODIText* text, ODScriptCode script);

ODScriptCode GetITextScriptCode(ODIText* text);
```

The following functions set and get the language code of the `ODIText` structure passed in.

```
void  SetITextLangCode(ODIText* text, ODLangCode lang);

ODLangCode GetITextLangCode(ODIText* text);
```

The following function sets the length of the `ODIText` structure's string length field. If `kODNULL` is passed in as the input `ODIText`, the function is equivalent to the `CreateITextClear` function.

```
ODIText* SetITextStringLength( ODIText* text, ODSize length,
                                            ODBoolean preserveText );
```

```
ODIText* CreateIText(ODSize length);
```

The following function returns the string length of the `ODIText` structure passed in.

```
ODULong  GetITextStringLength(ODIText* text);
```

Accessing the string

The following function returns a pointer to the raw text without allocating any memory.

IMPORTANT
This function should be used with extreme caution because the pointer returned belongs to the `ODIText` structure. ▲

```
char* GetITextPtr(ODIText* text);
```

The following functions set the string of the `ODIText` structure with a C string. Note that the `SetITextString` function is overloaded and can also take a Pascal string.

```
void SetITextCString(ODIText* iText, char* cString);

void SetITextString(ODIText* iText, char* cString);
```

The following functions set the string of the `ODIText` structure with a Pascal string.

```
void SetITextPString(ODIText* iText, StringPtr pString);

void SetITextString(ODIText* iText, StringPtr pString);
```

The following function sets the string of the `ODIText` structure with a buffer of the specified length.

```
void SetITextText(ODIText* text, ODUByte* text, ODSize textLength);
```

The following functions return a pointer to a C string which corresponds to the string in the `ODIText` structure. If a string is passed in, the same string is used to return the result. Otherwise, this function allocates memory for the returned string. Note that the `GetITextString` function is overloaded and can also take and return a Pascal string.

```
char*  GetITextCString(ODIText* iText, char* cString);

char* GetITextString(ODIText* iText, char* cString);

char* GetCStringFromIText(ODIText* iText);
```

The following functions work like the `GetITextCString` function except that they return a Pascal string.

```
StringPtr GetITextPString(ODIText*, Str255 pString);

StringPtr GetITextString(ODIText* i, StringPtr pString);

StringPtr GetPStringFromIText(ODIText* iText);
```

Memory Management (ODMemory)

This section describes the OpenDoc memory manager utility, the Mac OS implementation of which resides in the files ODMemory. h and ODMemory.cpp.

OpenDoc includes a memory management utility that you can use for allocating and manipulating memory as needed by your parts. On each platform, the OpenDoc memory manager supplements the capabilities of the platform's own memory manager; you can use platform memory manager calls alone, you can use the OpenDoc memory manager alone, or you can use both as needed.

The OpenDoc memory manager is a fast and very space-efficient memory allocator. It is a utility library used by OpenDoc but independent of it. The OpenDoc memory manager's only requirement is that its clients use a procedural shared library mechanism, such as CFM on the Mac OS or DLL on Windows.

The OpenDoc memory manager works with but is not dependent upon SOM. When both are installed, the OpenDoc memory manager takes over the SOM memory management routines; in that case, calls to functions such as `SOMMalloc` and `SOMFree` use the OpenDoc memory manager.

For a brief introduction to SOM, refer to Appendix B, "System Object Model." For information about the SOM memory manager, see *SOMobjects Developer Toolkit Users Guide* and *SOMobjects Developer Toolkit Programmers Reference Manual* from IBM.

Allocating Heaps

A heap is a space in which blocks of memory of arbitrary size can be allocated. All blocks allocated by the OpenDoc memory manager (other than handles) are in one of its heaps. When the OpenDoc memory manager initializes itself, it creates a heap for you; you can create additional heaps if you want to. You can also delete heaps, and deleting a heap with blocks still in it is both legal and faster than deleting all the blocks individually.

All storage used by the OpenDoc memory manager originally comes from the operating system's platform-specific memory manager. The OpenDoc memory manager gets memory for its heaps from the platform memory manager in large chunks (typically 32 KB or greater), and then subdivides these chunks as needed to allocate blocks. When a heap runs out of room, the OpenDoc memory manager asks the platform memory manager for another chunk, and when the OpenDoc memory manager frees all blocks in a chunk, it returns the entire chunk to the platform memory manager.

The data type that represents a heap (`MemHeap`) is opaque; no internal structure is visible to you. You refer to heaps using pointers, and you can operate on them only with the OpenDoc memory manager functions.

Memory for a heap can come from one of three places:

- system memory (shared among all processes on the system)
- application memory (local to the current process or OpenDoc document)
- temporary memory (from a shared pool available to all applications)

On non–Mac OS platforms, application and temporary memory may be identical; on the Mac OS, however, temporary memory is important because very little application memory may be available in the fixed-size partition available to an OpenDoc process. For cross-platform code, therefore, it is better to specify temporary memory.

These are the principal heap-manipulation functions provided by the OpenDoc memory manager:

- `MMNewHeap` creates a new heap with a given location and initial size (and optionally a name). Whenever the heap runs out of space, it will request more bytes from the platform memory manager.
- `MMDisposeHeap` disposes of a heap, returning to the operating system all the memory it has allocated. As a result, all blocks in the heap, and pointers to those blocks, become invalid.
- `MMGetDefaultHeap` returns a pointer to the current default heap. There is always a default heap; memory allocation calls that don't explicitly specify a heap use the current default heap. (If you use SOM, this includes SOM's memory management calls.)
- `MMSetDefaultHeap` makes a specified heap the default heap. In this way you can change the default heap at any time.

Using multiple heaps can be convenient for your part editor, although it is not quite as efficient as storing everything in one heap (because free memory in one heap is not available to another). However, allocating a heap for temporary use and then deleting it when done can help reduce memory fragmentation, since deleting the heap leaves a small number of large free blocks, rather than a large number of small ones.

Allocating Nonrelocatable Blocks

Memory within a heap is allocated nonrelocatable blocks. The interface for creating and operating on these blocks is similar to the ANSI C memory API, which is similar to that used by SOM. In fact, the SOM memory calls are rerouted to these routines, so that calling `SOMMalloc`, for example, is identical to calling `MMAllocate`.

These are the principal block-allocation functions provided by the OpenDoc memory manager:

- `MMAllocate` allocates a new block of a specified size from the default heap and returns a pointer to it. (The largest block that can be allocated is 0xFFFFFF, or 16 megabytes.) `MMAllocateClear` similarly allocates a block but also fills it with zeros. `MMAllocateIn` and `MMAllocateClearIn` also allocate blocks but let you specify the heap.

- `MMReallocate` changes the size of an already-allocated block and returns a pointer to the new location of the block.
- `MMFree` frees (disposes of) a previously allocated block.
- `MMBlockSize` returns the size of a block.
- `MMGetHeap` returns a pointer to the heap that owns a block.
- `MMSetIsObject` sets the is-object flag of a block; `MMIsObject` queries the is-object flag. If the flag is set, the OpenDoc memory manager assumes that the block contains a valid SOM object. Some of the memory debugging calls (see “Memory Debugging” on page 169) make use of this flag; you can also use it for your own purposes.

You should clear the is-object flag before freeing any block because the debugging configuration of the OpenDoc memory manager warns you if you free a block containing an object. SOM objects that inherit from `ODObject` (the OpenDoc root object class) automatically set the is-object flag when created and clear it when deleted.

Allocating Relocatable Blocks (Handles)

For convenience, the OpenDoc memory manager also provides operations for allocating relocatable blocks, referenced via handles. These blocks are allocated directly by the platform’s memory manager, not by the OpenDoc memory manager, and they don’t reside in heaps managed by the OpenDoc memory manager. However, you can still specify the same types of locations.

Because relocatable blocks are allocated by the platform’s memory manager, an OpenDoc handle (type `MMHandle`) is the same as a platform-specific handle and can be passed to operating system routines that take handles; likewise, a handle allocated by an operating system routine can be passed to any of the OpenDoc memory manager routines that take a parameter of type `MMHandle`.

These are the principal handle-allocation functions provided by the OpenDoc memory manager:

- `MMAllocateHandle` allocates a new relocatable block from the default heap and returns a handle to it; `MMAllocateHandleIn` allocates a new relocatable block from a specified heap. The source of the block (system, application, or temporary memory) is that of the indicated heap, even though the block is not actually allocated inside that heap.

- `MMFreeHandle` frees (disposes of) a previously allocated relocatable block.
- `MMCopyHandle` makes an exact copy of a relocatable block and returns a handle to the copy.
- `MMGetHandleSize` returns the size of a relocatable block.
- `MMSetHandleSize` changes the size of a relocatable block.
- `MMLockHandle` locks a relocatable block, which prevents it from being relocated by the operating system in response to other memory requests. `MMLockHandle` returns a direct pointer to the contents of the block; you can dereference this pointer to access the block's contents as long as the block is locked.

  On the Mac OS platform a handle is just a pointer to a pointer to a block, and the data in the block can be accessed at any time by doubly dereferencing the handle. Other platforms, such as Windows, have a more opaque notion of a handle. Therefore, to make your code cross-platform, you should always use the pointer returned by the `MMLockHandle` function instead of dereferencing your handles.
- `MMUnlockHandle` and `MMUnlockPtr` unlock a relocatable block, given either a handle to the block or a pointer to its contents.

IMPORTANT

Calls to `MMLockHandle` and `MMUnlockHandle` do not nest. The first call to `MMUnlockHandle` unlocks the block (and invalidates any pointers to its contents) no matter how many times `MMLockHandle` has been called. ▲

Memory Debugging

There are two configurations of the OpenDoc memory manager utility: regular and debugging. During development, if you link with the debugging configuration you can use its extra functions to help debug your code's memory management. These functions allow you to detect whether you are passing illegal values to the OpenDoc memory manager or overwriting heap data outside of blocks. You can also determine whether a given block is valid, and you can collect statistics on a heap as a whole or on all blocks in a heap.

Besides providing these extra routines, the debugging configuration also performs more internal checking of function parameters and data structures; this makes it slower but better able to detect problems.

These are the principal debugging functions provided by the debugging configuration of the OpenDoc memory manager:

- `MMBeginMemValidation` and `MMEndMemValidation` turn memory validation on and off. When validation is on, newly allocated blocks are filled with 0xBB ("Born"), and freed blocks are filled with 0xDD ("Dead"). Calls that take a block pointer as a parameter verify that the block is valid; if it isn't, they warn you (typically via a low-level debugger) and the operation fails.

  Memory validation can also be turned on and off via the ODDebug menu in debugging builds of OpenDoc. (In the Mac OS implementation, this is a submenu of the Apple menu.)

  Note that you can nest memory-validation calls.

- `MMBeginHeapChecking` and `MMEndHeapChecking` turn heap-checking on and off. Heap-checking includes memory validation, but in addition most OpenDoc memory manager calls scan through their heap to verify that its internal structure is intact and valid. if it isn't, they warn you (typically via a low-level debugger) and the operation fails.

  Heap checking can be very slow, especially if the heap contains a large number of blocks. Memory-intensive operations like opening or closing a storage object can take tens of times longer than normal. Nevertheless, heap checking can be the best way to track down obscure bugs that destroy heaps.

  Heap checking can also be turned on and off via the ODDebug menu in debugging builds of OpenDoc. (In the Mac OS implementation, this is a submenu of the Apple menu.)

  Note that you can nest heap-checking calls.

- `MMDoesHeapExist` determines whether the given heap is known to the OpenDoc memory manager. If the heap was allocated by another process, you may still be able to manipulate it with the platform-specific memory manager, although that may be bad practice.

- `MMValidatePtr` and `MMValidateHandle` validate a single block (normal or relocatable.) If the pointer or handle passed in does not reference a valid block, or if the block's heap is corrupted, you are warned via a low-level debugger.

- MMValidateObject verifies a block in the same manner as MMValidatePtr but also verifies that the block's is-object flag is set. It also verifies that the block looks like a valid object to the SOM runtime system.
- MMValidateHeap and MMValidateAllHeaps check either a single heap, or all known heaps, for consistency. This can be a slow operation if there are large numbers of blocks. If a heap is corrupted, you are warned via a low-level debugger.
- MMGetHeapInfo returns useful information about a heap, such as its name, its size, the number of free bytes it contains, the number of blocks it contains, and the number of objects (blocks with the is-object flag set) it contains.
- MMWalkHeap Lets you examine every allocated block in a heap, using a pointer you provide to a callback function that is called once for every block in the heap.

Object Handling (ODUtils)

This section describes the object-handling utilities defined in the files ODUtils.h and ODUtils.cpp. These utilities are useful for handling OpenDoc objects, especially reference-counted objects.

ODDeleteObject

The ODDeleteObject macro deletes an object, which can be a SOM object or another type of object (such as a C++ object), and sets the variable pointing to it to kODNULL. This macro takes one parameter, which points to the object, as follows:

```
ODDeleteObject(object)
```

ODReleaseObject

The ODReleaseObject macro releases (instead of deleting) a reference-counted SOM object and sets the variable pointing to it to kODNULL. This macro takes parameters pointing to the SOM environment structure and the object, as follows:

```
ODReleaseObject(ev, object)
```

ODFinalReleaseObject

The `ODFinalReleaseObject` macro is similar to `ODReleaseObject`, but it is meant to be used to release the last reference to a reference-counted object. It asserts that the object's reference count is equal to 1 before calling its `Release` method. This macro takes parameters pointing to the SOM environment structure and the object, as follows:

```
ODFinalReleaseObject(ev, object)
```

ODAcquireObject

The `ODAcquireObject` function increments the reference count of an object by 1 unless the object pointer passed into the function is `kODNULL`. The prototype of this function appears as follows:

```
void ODAcquireObject(Environment* ev, ODRefCntObject* object);
```

ODSafeReleaseObject

The `ODSafeReleaseObject` function releases a reference-counted object but requires no environment parameter. This function will not throw an exception. It is designed to be used in destructors, `CATCH_ALL` exception handling blocks, and `somUninit` methods where no pointer to the environment structure is available. The prototype of this function appears as follows:

```
void ODSafeReleaseObject(ODRefCntObject* object);
```

ODTransferReference

The `ODTransferReference` function decrements one object's reference count while incrementing another object's reference count. It is designed to be used in situations such as when using setter methods where you need to acquire a reference to one object while simultaneously releasing another.

This function ensures that the parameters do not point to the same object and that neither is null. It is possible for this function to throw an exception in the unlikely case that the `Acquire` or `Release` method call fails. The prototype of this function appears as follows:

```
void ODTransferReference( Environment*, ODRefCntObject* oldObj,
                                         ODRefCntObject* newObj );
```

ODCopyAndRelease

The `ODCopyAndRelease` function returns a pointer to a copy of an object and releases the original object. This function is overloaded: one form takes and returns pointers to `ODShape` objects; the other takes and returns pointers to `ODTransform` objects. This function is designed to transfer control of an object to the caller, giving the caller permission to modify the object.

If the reference count of the original object is 1, the function is optimized simply to return a pointer to the original object, thereby avoiding the unnecessary expense of copying it. It is possible for this function to throw an exception in the unlikely case that the `GetRefCount` or `Release` method call fails. The prototypes of this function appear as follows:

```
ODShape* ODCopyAndRelease(Environment* ev, ODShape* shape);

ODTransform* ODCopyAndRelease(Environment* ev, ODTransform* transform);
```

ODObjectsAreEqual

The `ODObjectsAreEqual` function returns `kODTrue` if both `ODObject` pointers passed as parameters are not null and point to the same object.

IMPORTANT

Simply comparing the values of the pointers (as in the expression `a==b`) is not sufficient in the presence of distributed objects. Two pointers to the same remote object may have different numeric values. ▲

The prototype of the `ODObjectsAreEqual` function appears as follows:

```
ODBoolean ODObjectsAreEqual(Environment* ev, ODObject* a, ODObject* b);
```

Standard Type Input and Output (StdTypIO)

This section describes the utilities defined in the files StdTypIO.h and StdTypIO.cpp. These utilities allow you to manipulate OpenDoc storage units more simply and enable you to read and write various commonly used data types (such as integers, ISO strings, time values, storage unit references, and so forth) in a standard storage format to facilitate document exchange.

The standard type input and output functions are designed to be used independent of property and value type, and in many cases can even be used to manipulate data in the middle of values. To do so, pass in a prefocused storage unit with the offset set correctly, and pass in `kODNULL` for the `ODPropertyName` and the `ODValueType` parameters.

Boolean Values

The following function returns a Boolean value from a storage unit.

```
ODBoolean ODGetBooleanProp(Environment* ev,
                ODStorageUnit* su, ODPropertyName prop, ODValueType val)
```

Short Values

The following functions get and set unsigned and signed 16-bit values.

```
ODUShort ODGetUShortProp(Environment* ev,
                ODStorageUnit* su, ODPropertyName prop, ODValueType val)

void    ODSetUShortProp(Environment* ev,
                ODStorageUnit* su, ODPropertyName prop, ODValueType val,
                ODUShort value)

ODSShort ODGetSShortProp(Environment* ev,
                ODStorageUnit* su, ODPropertyName prop, ODValueType val)

void    ODSetSShortProp(Environment* ev,
                ODStorageUnit* su, ODPropertyName prop, ODValueType val,
                ODSShort value)
```

Long Values

The following functions get and set unsigned and signed 32-bit values.

```
ODULong  ODGetULongProp(Environment* ev,
                ODStorageUnit* su, ODPropertyName prop, ODValueType val)
```

```
void    ODSetULongProp(Environment* ev,
                ODStorageUnit* su, ODPropertyName prop, ODValueType val,
                ODULong value)

ODSLong ODGetSLongProp(Environment* ev,
                ODStorageUnit* su, ODPropertyName prop, ODValueType val)

void    ODSetSLongProp(Environment* ev,
                ODStorageUnit* su, ODPropertyName prop, ODValueType val,
                ODSLong value)
```

ISO String Values

An ISO string (`ODISOStr`) is a string of 7-bit ASCII characters terminated by a zero byte. The functions in this section get and set ISO string values in storage units.

```
ODISOStr ODGetISOStrProp(Environment* ev,
                ODStorageUnit* su, ODPropertyName prop, ODValueType val,
                ODISOStr value, ODULong* size)

void    ODSetISOStrProp(Environment* ev,
                ODStorageUnit* su, ODPropertyName prop, ODValueType val,
                ODISOStr value)
```

Type List Values

The functions in this section get and set values of type `ODTypeList`. An `ODTypeList` property value containing `n` elements begins with `(n+1)` offsets, followed by `n` ISO strings with their null termination. The first `n` offsets identify the starting positions of the corresponding ISO string. The last offset is always equal to the size of the value and is immediately before the first character of the first ISO string. For example, a property value representing an empty `ODTypeList` object is four bytes long and contains offset four, signifying that there are no ISO strings present.

```
void ODGetTypeListProp(Environment* ev,
                ODStorageUnit* su, ODPropertyName prop, ODValueType val,
                ODTypeList* typeList)
```

```
void ODSetTypeListProp(Environment* ev,
                ODStorageUnit* su, ODPropertyName prop, ODValueType val,
                ODTypeList* typeList)
```

Text Values

The following functions manipulate international text, Unicode text, and traditional Mac OS text values.

```
void TradMacTextToUnicode(ODUByte* macText, ODULong macTextLength,
                ODUShort** unicodeText, ODULong* uniCodeBufferLength);

void UnicodeToTradMacText(ODUShort* unicodeText,
                ODULong uniCodeTextLength,
                ODUByte** macText, ODULong* macTextLength);

ODIText* UnicodeToIText(ODIText* iText, ODUShort* unicodeText,
                        ODULong unicodeTextLength);
```

In the `ODGetITextProp` function, if the `iText` parameter is `kODNULL`, a variable of type `ODIText` is allocated and passed back. If not, the `_buffer` field of the text within the `iText` structure is deallocated and a new `_buffer` is allocated and filled. If no value is passed in the `ODPropertyName` and `ODValueType` parameters, the function returns `kODNULL`.

```
ODIText* ODGetITextProp(Environment* ev,
                ODStorageUnit* su, ODPropertyName prop, ODValueType val,
                ODIText* iText)
```

In the `ITextToUnicode` function, storage passed back must be deallocated with the `ODDisposePtr` function.

```
void ITextToUnicode(ODIText* iText, ODUShort** unicodeText,
                        ODULong* unicodeTextLength);

void ODSetITextProp(Environment* ev,
            ODStorageUnit* su, ODPropertyName prop, ODValueType val,
            ODIText* iText)
```

Time Values

The following functions get and set values of type `ODTime`.

```
ODTime   ODGetTime_TProp(Environment* ev,
                ODStorageUnit* su, ODPropertyName prop, ODValueType val)

void     ODSetTime_TProp(Environment* ev,
                ODStorageUnit* su, ODPropertyName prop, ODValueType val,
                ODTime value)
```

Geometric Values

The following functions get and set values of type `ODPoint`, `ODRect`, `ODPolygon`, and `ODMatrix`.

```
ODPoint* ODGetPointProp(Environment* ev,
                ODStorageUnit* su, ODPropertyName prop, ODValueType val,
                ODPoint* value)

void     ODSetPointProp(Environment* ev,
                ODStorageUnit* su, ODPropertyName prop, ODValueType val,
                ODPoint* value)

ODRect*  ODGetRectProp(Environment* ev,
                ODStorageUnit* su, ODPropertyName prop, ODValueType val,
                ODRect* value)

void     ODSetRectProp(Environment* ev,
                ODStorageUnit* su, ODPropertyName prop, ODValueType val,
                ODRect* value)

ODPolygon* ODGetPolygonProp(Environment* ev,
                ODStorageUnit* su, ODPropertyName prop, ODValueType val,
                ODPolygon* value)

void     ODSetPolygonProp(Environment* ev,
                ODStorageUnit* su, ODPropertyName prop, ODValueType val,
                const ODPolygon* value)
```

```
ODMatrix* ODGetMatrixProp(Environment* ev,
                ODStorageUnit* su, ODPropertyName prop, ODValueType val,
                ODMatrix* value)

void ODSetMatrixProp(Environment* ev,
                ODStorageUnit* su, ODPropertyName prop, ODValueType val,
                ODMatrix* value)
```

Storage Unit Reference Values

The following functions get and set strong and weak storage unit references.

```
ODID    ODGetStrongSURefProp(Environment* ev,
                ODStorageUnit* su, ODPropertyName prop, ODValueType val)

void    ODSetStrongSURefProp(Environment* ev,
                ODStorageUnit* su, ODPropertyName prop, ODValueType val,
                ODID id)

ODID    ODGetWeakSURefProp(Environment* ev,
                ODStorageUnit* su, ODPropertyName prop, ODValueType val)

void    ODSetWeakSURefProp(Environment* ev,
                ODStorageUnit* su, ODPropertyName prop, ODValueType val,
                ODID id)
```

Icon Family Values

The functions in this section get and set values of type `ODIconFamily`. The `ODIconFamily` type is platform-specific, so the functions have platform-specific implementations. The following values are used for the `iconMask` parameter.

```
enum {
    kAllIconsMask = 0xFFFFFFFF, // All icons usable on this platform
    kBWIconsMask  = 0x0421      // 1 bit deep, 16,32,64 pixels wide
};
```

Expected values for the `ODValueType` parameter begin with `OpenDoc:Type:IconFamily:` followed by the name of a platform. The platform names are defined in StdTypes.idl as `kODIconFamilyMac`, `kODIconFamilyWin`, kODIconFamilyOS2, and kODIconFamilyAIX. If you specify only kODIconFamily, the type of the current platform is used.

```
ODIconFamily ODGetIconFamilyProp(Environment* ev,
                        ODStorageUnit* su, ODPropertyName prop,
                        ODValueType val, ODULong iconMask);

void         ODSetIconFamilyProp(Environment* ev,
                        ODStorageUnit* su, ODPropertyName prop,
                        ODValueType val, ODIconFamily iconFamily,
                        ODBoolean deleteOtherPlatformIcons);
```


Storage (StorUtil)

This section describes the utilities defined in the files StorUtil.h and StorUtil.cpp. These utilities wrap the GetData methods of the class ODStorageUnit, letting you pass in direct pointers to data buffers instead of an ODByteArray wrapper.

Storage Utility Functions

The storage utility defines the functions described in the following sections.

StorageUnitGetValue

The StorageUnitGetValue function takes as a parameter a pointer to an OpenDoc storage unit, the SOM Environment variable, the buffer size, and a pointer to the buffer. The function returns the number of bytes actually read. The prototype of this function appears as follows:

```
ODULong        StorageUnitGetValue(ODStorageUnit* su, Environment* ev,
                        ODULong size, ODPtr buffer);
```

StorageUnitViewGetValue

The StorageUnitViewGetValue function takes as a parameter a pointer to an OpenDoc storage unit view object, the SOM Environment variable, the buffer size, and a pointer to the buffer. The function returns the number of bytes actually read. The prototype of this function appears as follows:

```
ODULong         StorageUnitViewGetValue(ODStorageUnitView* suv,
                        Environment* ev, ODULong size, ODPtr buffer);
```

Temporary Objects (TempObj)

This section describes the template utilities defined in the files TempObj.h and TempObj.cpp. These utilities provide exception-safe temporary object references, and they handle reference counting automatically.

These are simple template classes that act as a transparent wrapper around an OpenDoc object pointer. The temporary object can be used wherever a pointer to the OpenDoc object would be used. When the temporary object goes out of scope, the object it wraps is either deleted or released (depending on the temporary object class used).

Need for Temporary Objects

When writing OpenDoc-based code, you often need to create temporary objects that later need to be freed or to acquire temporary references to reference-counted objects that later need to be released. In these situations, it is easy to forget to free the object or release the reference. It is also possible for an exception to be thrown while the temporary is active, in which case you can't free the object or release the reference unless you include more complicated exception-handling in your code. In these cases, it is easy to create a memory leak or reference-counting error.

The temporary objects defined in the temporary-objects utility are implemented as stack-based C++ objects whose destructors are called automatically whenever they go out of scope, either by exiting a block or via an exception. This scheme is implemented in the exception-handling utility as the `Destructo` class, from which the temporary object template classes inherit.

Using Temporary Objects

To use this utility, just include the file TempObj.h in your source files and link TempObj.cpp into your libraries. This gives you access to the following classes:

```
TempODFrame
TempODPart
TempODShape
TempODStorageUnit
TempODTransform
TempODWindow

TempODFocusSetIterator
TempODFrameFacetIterator
```

Note
Iterators are not reference-counted, so the classes `TempODFocusSetIterator` and `TempODFrameFacetIterator` delete the iterator object at the end instead of releasing it. ◆

If your compiler supports C++ templates, you can define a symbol `_USE_TEMPLATES_` before including TempObj.h. This will ensure that the header uses templates to implement these classes. This might make the implementation more efficient, and it also makes it much easier to extend the mechanism to new classes. If you can't or don't want to use templates, just don't define this symbol; the default is that the classes are implemented without using templates.

Pitfalls

The biggest mistake you can make in using this utility is forgetting that the object is always released. This can cause a problem if you need to use the object as the return value of a function:

```
ODShape *snod( ODFrame *frame ) {
    TempODShape s = frame->GetFrameShape(ev,kODNULL);
    DoSomething(s);
    return s;
}
```

The `ODShape` is released before it's returned, when the destructor of `s` is called. This is bad news, since the function will return either a pointer to a deleted object or to an object whose reference count is one too low. Either case is likely to cause a crash.

It's still nice to use a `TempODShape` in this function, for safety in case `DoSomething` throws an exception. You just want to tell `s` not to release itself when it's being returned. You can do this by setting the shape to `kODNULL` before returning it:

```
ODShape *snod( ODFrame *frame ) {
    TempODShape s = frame->GetFrameShape(ev,kODNULL);
    DoSomething(s);
    ODShape *temp = s;
    s = kODNULL;          // s will not be released by the destructor now
    return temp;
}
```

Of course, this is a kludge in that you have to store the value of `s` in a temporary variable to keep it from being lost. Instead, you can use a convenience method called `DontRelease` that will set the reference to null but return its previous value:

```
ODShape *snod( ODFrame *frame ) {
    TempODShape s = frame->GetFrameShape(ev,kODNULL);
    DoSomething(s);
    return s.DontRelease();      // Note that "." is used, not "->"
}
```

Using Temporary Iterators

The temporary-objects utility contains some extra classes that are temporary objects for OpenDoc iterator classes. In addition to managing the automatic deletion of the iterator object itself, they also simplify the process of using the iterator and shrink the resulting code. The following example illustrates use of these temporary iterator classes:

```
    extern void DoSomethingWith( ODSnod* );
    extern void OrSomethingWith( ODSnod* );
    ...
    ODBazz *bazz;
    ...
    for( TempODSnodIterator iter(ev,bazz); iter.Current(); iter.Next() )
    {
```

```
        DoSomethingWith(iter);
        OrSomethingWith(iter.Current());
    }
```

Within the loop you can use `iter.Current()` or just `iter` to refer to the current object to which the iterator is pointing. You can also use the following syntax to control the iteration loop because the iterator itself can be used as a synonym for its current object, and the ++ operator is the same as calling `Next`:

```
for( TempODSnodIterator iter(ev,bazz); iter; iter++ )
```

Adding New Temporary Classes

There are other OpenDoc classes for which you might want to have temporary objects available. You can define your own temporary object classes using the temporary-objects utility. This is especially easy if your compiler supports templates.

Adding New Classes Using Templates

If you're using templates (by defining `_USE_TEMPLATES_` before including TempObj.h), you can declare a temporary reference to any type of reference-counted object by using the class `TempRef<className>` in the following manner:

```
TempRef<ODDraft> su = doc->AcquireDraft(ev);
```

You can also use temporary instances of objects that are not reference counted by using `TempObj<class>` in the following manner:

```
TempObj<ODPeanutIterator> iter = peanut->GetIterator(ev);
```

Adding New Classes Without Using Templates

If you're not using templates, you'll need to do some more work, adding several weird looking `#define` and `#include` statements. You can add these to the existing TempObj files, or put them in your own files. To add your own temporary class to the TempObj files, perform the following steps:

1. Open TempObj.h and find the correct location.

Scroll down to the comment that reads `// Instantiations of TempObj and TempRef. Add your own if necessary`. Under the line reading `#else /* not _USE_TEMPLATES_*/`, find a series of groups of lines, each group of which looks like this:

```
#define _T_     ODFrame
#define _C_     TempODFrame
#include "TempRef.th"
```

2. **Add another one of these groups.**

   You can do this in TempObj.h, or in a separate header of your own. Change `_T_` to the OpenDoc class for which you want to make a temporary class. Change `_C_` to the name of the temporary-reference class. If the OpenDoc class is not reference counted, include TempObj.th instead of TempRef.th.

3. **Open TempObj.cpp and find the correct location.**

   Scroll down to the comment that reads `// Define the non-inline methods of the various template classes`. Below find another list of `#define` and `#include` statements like the ones shown above.

4. **Add another of these groups.**

   You add another group in the same manner as in TempObj.h.

5. **Recompile TempObj.cpp.**

   If you put the declarations in a separate utility library and not directly in your project, you'll need to build the library first. If TempObj.h is precompiled, the first thing you must do is rebuild the precompiled header.

Type-Checking Errors

If, after adding a new class, you get a type-mismatch error in TempObj.h (probably at line 139) or in TempObj.th or TempRef.th, this indicates that you are trying to use `TempObj` with a class that is not a subclass of `ODObject`, or `TempRef` with a class that is not a subclass of `ODRefCntObject`. This mismatch can happen even if the class is correct if the compiler hasn't seen the declaration of the class before the declaration of the temporary. In other words, the following is wrong:

```
class ODSnod;
TempRef<ODSnod> ref = ......;
#include "ODSnod.h"
```

At the time that the compiler instantiates the template for `ODSnod`, it does not know anything about the class, such as whether it is a subclass of `ODRefCntObject`, so it will therefore report type-checking errors. You can avoid this problem by including the header for `ODSnod` before using the `TempRef` class.

Resource Handling (UseRsrcM)

This section describes the utilities defined in the files UseRsrcM.h and UseRsrcM.cpp. These utilities enable you to access resources from your part editor's resource fork.

Using Mac OS resources from an OpenDoc part handler is a little more difficult than from a regular Mac OS application. Part handlers are implemented as shared libraries, and the Code Fragment Manager does not automatically open the resource fork of a shared library when the library is in use. Leaving the resource fork open all the time would cause resource conflicts among libraries and their host applications, but opening it every time a library is called would have too much overhead. Instead, a code fragment is responsible for remembering where its file lives, for opening the resource fork when it needs to access resources, and for closing it when it's done.

Setting Up the Build System

To use these resource utilities, you need to add the utility file UseRsrcM.cpp to your project if you use a project-file development system, or add it to a makefile if you use MPW.

You also need to tell the build system you have a CFM initialization routine (described in the next section). In MPW, you use the `-init name` command line flag of the ILink or PPCLink tool. The routine can be called anything you like, but typically you append `CFMInit` to the part editor name, for example, `SamplePartCFMInit`.

Initializing Your Library

If your part editor needs to access its resources (as almost any part handler does) you need to provide a CFM initialization routine. This routine is called by the Code Fragment Manager when your library is instantiated—that is, whenever the first connection is made to your library by a process. For a part editor, this occurs the first time an instance of your part is created. The initialization routine is the very first piece of your code to be called.

The initialization routine is passed a pointer to an initialization block. This block contains an `FSSpec` field that gives the location on disk of the part handler library. In the implementation of your initialization routine you need to pass a pointer to the initialization block to the function `InitLibraryResources` so it can open the library's resource fork and keep it around for when you need to access it.

You should also provide a termination routine, which the Code Fragment Manager calls when your part editor library is unloaded. (Typically this happens when no instances of classes defined in your library are in memory and OpenDoc decides to purge memory to free up space.) The termination routine should call `CloseLibraryResources` to close your library's resource fork and free up the memory occupied by its resource map (and any resources from it that haven't been purged or released.)

Bare-bones initialization and termination routines look like this:

```
#ifndef __USERSRCM__
#include "UseRsrcM.h"
#endif

#ifndef __FRAGLOAD__
#include <FragLoad.h>
#endif

extern "C" pascal OSErr MyPartCFMInit( InitBlockPtr );


OSErr MyPartCFMInit (InitBlockPtr initBlkPtr)
{
    return InitLibraryResources(initBlkPtr);
}

void MyPartCFMTerminate( )
{
    CloseLibraryResources();
}
```

The call to `InitLibraryResources` opens your library's resource fork but does not put it in the resource chain. This effectively makes it invisible to the Resource Manager, but allows it to be activated at a moment's notice. The termination routine closes the resource fork and releases any memory it may have been using.

Note
The initialization routine is also a good place to do other one-time initializations, such as setting up global variables. A common thing to do is to call the Mac OS Toolbox routine `Gestalt` to determine whether various system services are available, and to store the results in global Boolean variables for later use. Keep in mind, though, that the initialization routine is not called every time a part is instantiated—it is called only when the library is first linked into a process. ◆

Of course, simply declaring these routines is not enough. You need to tell the linker that these are special CFM routines. See the above section "Setting Up the Build System" on page 185, as well as your development tools' documentation, for full details.

For more information on initialization and termination routines, see *Inside Macintosh: PowerPC System Software*.

Accessing Your Library's Resources

Before accessing your library's resources (directly or indirectly), call `BeginUsingLibraryResources`. This routine activates your library's resource fork and makes it the current resource file. (It also returns a magic 32-bit value that you should save for later.) You may then safely call Resource Manager routines like `Get1Resource` or `Count1Resources`, or Toolbox routines that indirectly call the Resource Manager, such as `GetMenu` or `GetNewWindow`.

As soon as possible, deactivate your library's resource fork by calling `EndUsingLibraryResources`, passing in the magic 32-bit value you received from `BeginUsingLibraryResources`. Leaving the resource fork active for too long can cause conflicts with other part editors (or OpenDoc subsystems) that need to use resources. In particular, you should not make any OpenDoc API calls while your resource fork is active, or (even worse) return from a call to your part without deactivating it.

Activating and deactivating your resource fork is a very quick process, without much overhead. Don't worry about it slowing down the system.

Consider the following code fragment:

```
ODSLong x = BeginUsingLibraryResources();
fMenu = GetMenu(kMyMenuID);
EndUsingLibraryResources(x);
fMenuBar->AddMenuLast(ev,kMyMenuID, fMenu, fPart);
```

After calling `EndUsingLibraryResources`, any resources that have been loaded into memory are still there. However, since your resource file is not active and is not in the resource chain, you can't perform any resource operations on resources in it, such as `LoadResource`, `GetResInfo` or `ReleaseResource`. Before you can call any Resource Manager routines on a resource you've loaded, you need to call `BeginUsingLibraryResources` again.

In particular, you must activate the resource file before releasing the resource, as this example shows:

```
ODSLong x = BeginUsingLibraryResources();
ReleaseResource((Handle)fMenu);
EndUsingLibraryResources(x);
```

You can't call `ReleaseResource` when your resource fork is inactive. And you can't just call `DisposeHandle` on the resource, or the Resource Manager will encounter problems.

For C++ Users

If you use C++, there is an alternative to using these calls, based on the standard C++ idiom of a lightweight stack-based class whose constructor sets up a state and whose destructor removes it. The class is called `CUsingLibraryResources`. Declaring an instance of `CUsingLibraryResources` activates your resource fork, as in this example:

```
{
    CUsingLibraryResources using;
    fMenu = GetMenu(kMyMenuID);
}

fMenuBar->AddMenuLast(ev,kMyMenuID, fMenu, fPart);
```

When the object goes out of scope (when the flow of control leaves the enclosing block) the resource fork is deactivated.

One nice aspect of the object-oriented model is that, unlike in the procedural model, you can return or break from a block containing a `CUsingLibraryResources`. The return or break statement will cause the object to go out of scope, and the compiler automatically calls the destructor.

A `CUsingLibraryResources` object is a `Destructo` (defined in the file Except.h) and so will automatically be destroyed if it goes out of scope as a result of an exception. This means that your resource fork automatically is deactivated if an exception is thrown out of the block: a very desirable thing to have happen. For this reason, if you use C++, it's preferable to use this form instead of using `BeginUsingLibraryResources` and `EndUsingLibraryResources`.

Note

Remember, the Resource Manager only loads one copy of a resource into any single process. However, any number of instances of your part may be active in a single document process. This means that, unless you explicitly use the `ODReadResource` utilities described in the next section, all instances of your part in a single document have to share the resources. ◆

A common error is for a part to load a resource and then later release it, perhaps in its destructor or `somUninit` method. The problem is that other instances of the part might still exist in the document, and they might also have loaded the same resource. After the first part releases the resource, the other parts have invalid dangling handles and will probably end up reading garbage or corrupting the heap if they try to use the resource thereafter.

A good solution is to treat resources as global variables. Note that they have the same scope (per process) as your part handler library's global variables. This means that you can safely load a resource and assign the handle to a global variable, which can then be shared by all active instances of your part. If you release the resource, perhaps in your `Purge` method, set the global variable to `kODNULL` so that other instances of your part know it's been disposed of. They can then load the resource again the next time they read it. A more advanced variation on this is to keep a reference count on a resource and release the resource when the reference count goes to zero.

Resource-Loading Utilities

There are some resource-loading utilities you might want to use. These have the advantage that they don't load the resources into the application heap (which has very little free space in an OpenDoc environment) and that the resource data isn't shared between all instances of your part. They also take care of activating and deactivating your resource file automatically.

`ODReadResource` and `ODReadNamedResource` are comparable to `GetResource` and `GetNamedResource`, except for the following differences:

- They explicitly load the resource out of your part editor.
- The result is a detached handle, which means you get a new copy every time you call these routines. It also means you can dispose of the handle normally using `ODDisposeHandle`.
- They put the handle in temporary memory. (It was allocated via `ODNewHandle`.)
- They throw exceptions if any errors occur. In particular, they throw `resNotFound` if the resource is not found.

`ODReadResourceToPtr` and `ODReadNamedResourceToPtr` are similar, except that they load the resource data into a nonrelocatable block and return a pointer to it. (The block is allocated via `ODNewPtr` and should be disposed of via `ODDisposePtr` or `MMFree`.) These functions are obviously not appropriate for Toolbox-defined resource types like `'PICT'` that have to be referenced via handles, but for your own types it can be preferable since the memory allocation is more efficient and access to the data requires only single indirection.

`ODGetString` reads the contents of a `'STR '` resource from your part editor into a `Str255` variable that you pass in. It throws an exception (usually `resNotFound`) if the resource can't be read.

`ODGetIndString` reads a string from a `'STR#'` resource from your part editor into a `Str255` variable that you pass in. It is like `GetIndString` except that it automatically activates and deactivates your resource fork and throws an exception if the resource can't be found.

Window Utilities (WinUtils)

This section describes the utilities defined in the files WinUtils.h and WinUtils.cpp. When you're reopening a window at document-launch time, you can use these utilities to retrieve the window properties stored with the root frame of any persistently stored window.

Retrieving Window Properties

When a saved document is opened, OpenDoc retrieves the root frame of each saved window and calls the `Open` method of the part belonging to that frame, passing the frame. The part is responsible for recreating the platform window and creating an `ODWindow` object using the `RegisterWindowForFrame` method.

The properties of the window are saved in a storage unit referenced from the root frame. The utility functions `BeginGetWindowProperties` and `EndGetWindowProperties` can be used to retrieve these properties without using the storage system API directly.

Using the Window Utilities

The window utilities functions allow a part editor to obtain the properties necessary to create a window from the storage annotation on a root frame. The `BeginGetWindowProperties` function returns `kODTrue` if the annotation exists and fills in the `properties` structure. The `EndGetWindProperties` function releases the frame specified in the `sourceFrame` field of the structure.

The following code fragment illustrates use of the window utilities:

```
WindowProperties props;
ODWindow* window = kODNULL;

if (BeginGetWindowProperties(ev, frame, &props))
{

    ODPlatformWindow platformWindow =
                NewCWindow(kODNULL,
                    &(props.boundsRect),
```

```
                    props.title,
                    kODFalse,
                    props.procID,
                    (WindowPtr)-1L,
                    props.hasCloseBox,
                    props.refCon);

    window =  fSession->GetWindowState(ev)->
                RegisterWindowForFrame(ev, platformWindow,
                    frame,
                    props.isRootWindow,// keeps draft open
                    kODTrue,// is resizable
                    kODFalse,// is floating
                    kODTrue,// shouldSave
                    props.sourceFrame);
    EndGetWindowProperties(ev, &props); // release source frame
}
```

System Object Model

This appendix presents an introduction to the System Object Model (SOM), the standard object infrastructure upon which the OpenDoc component software architecture is built. Developed by IBM Corporation, SOM is a programming technology for building, packaging, and manipulating object-oriented class libraries.

For complete documentation of SOM, see the *SOMobjects Developer Toolkit Users Guide* and *SOMobjects Developer Toolkit Reference Manual* from IBM.

Features of the System Object Model

SOMobjects™ for the Mac OS is the Mac OS implementation of the System Object Model (SOM). It underlies the Mac OS implementation of OpenDoc. SOMobjects for the Mac OS comprises several components, the most important of which are

- a kernel, which implements the basic SOM runtime environment
- SOM class libraries, which augment the runtime environment
- the SOM compiler, which translates SOM's Interface Definition Language (IDL) object specifications into a target language such as C++

SOM is not a complete implementation language or programming system. Instead, SOM complements such languages, providing a number of advantages, such as

- language neutrality, so that objects can be implemented in different programming languages yet work together
- binary compatibility, solving the "fragile base-class problem," which requires client programs to be recompiled whenever the class library on which they depend is modified
- cross-platform compatibility, because SOM is an emerging industry standard implemented on most major platforms

By virtue of these features, SOM enables greater code reuse for object libraries such as OpenDoc and greater flexibility for application programmers.

Development Process

To have the advantages of SOM, you must define objects with well-defined interfaces separated from their implementations. The SOM compiler enables you to do this. At runtime, the SOM kernel supports execution of such objects.

You define the interface to a SOM object in the SOM Interface Definition Language (IDL) described in the next section. However, you implement the methods of a SOM object and write client programs of the object in a full-featured programming language such as C++, the language used for SamplePart part editor.

After you define a SOM class in IDL, you run the SOM compiler on the IDL file. The SOM compiler produces three files in a target language, for which the SOM compiler must have a language-specific emitter. SamplePart uses the C++ emitter. The compiler output files are a usage binding, an implementation binding, and an implementation template file. The usage binding file (extension .xh) is similar to a regular C++ header file; client programs that use the SOM class include the usage binding file. The implementation binding file (extension .xih) is private to the SOM class and included in the class implementation; it contains macro definitions enabling the class implementation to have access to its instance variable and to call superclass methods. The implementation template file (extension .cpp) is similar to a regular C++ implementation file; as emitted by the SOM compiler, it contains stub function definitions for each method declared in the IDL file. Writing in C++, you must fill in the function bodies for each new and overridden method in the class.

Interface Definition Language

The SOM interface definition language (IDL) describes the interface of a SOM object in a set of declarations. Generally, these declarations can specify constants, type of the object, attributes (instance variables), operations (methods), exceptions, and module (which scopes the object).

In SOM, the runtime entities that provide services to clients are always objects, which contain methods and instance variables (also called *fields* or *attributes*). Client programs can call the methods to request whatever services the object provides, and the object uses its instance variables to store its state information. The interface to the object, which is expressed in IDL, describes what clients must know to use the object's services. Every SOM object is an instance of a single SOM class, but the implementation language of the object need not support the class concept.

The SOM Interface of SamplePart

The OpenDoc class that represents a part editor is named `ODPart`. It is a SOM class, as are all the classes in the OpenDoc class library. All part editors are built around a subclass of `ODPart`. The SamplePart part editor, however, incorporates a scheme by which the part's SOM interface is largely hidden from the programmer.

SamplePart has only one SOM class, a subclass of `ODPart` named `som_SamplePart`, referred to as the *SOM wrapper class*. This SOM class overrides all `ODPart` methods, although SamplePart implements only some of them. For those methods that SamplePart implements, the SOM wrapper class methods delegate the implementation to a C++ class named `som_SamplePart`.

The SOM class `som_SamplePart` is defined in IDL. The SOM class methods merely call corresponding methods in the C++ class, which is named `SamplePart`. For `ODPart` methods that SamplePart does not implement, the SOM class override method bodies are empty. They are provided so that you can extend SamplePart simply by adding a call to a method in a C++ class. Therefore, you do not need to revise the SOM class IDL interfaces and use the SOM compiler to extend SamplePart.

The remainder of this appendix describes the artifacts of IDL that appear in the definition of `som_SamplePart` class in the file som_SamplePart.idl.

The Class Definition

SOM provides a scoping mechanism to group objects into modules; the definition of the `SamplePart` class declares it to belong to the `SampleCode` module. The interface statement of the `som_SamplePart` object shows that it inherits from `ODPart`. Listing B-1 shows the interface statement.

Listing B-1 Interface statement

```
module SampleCode
{
    interface som_SamplePart : ODPart
    {
```

The next part of the interface definition is the implementation section, which is protected by an `#ifdef __SOMIDL__` compiler directive to maintain compatibility with pre-SOM versions of IDL. The `majorversion` and `minorversion` statements specify a combined version number which the SOM compiler can use to ensure compatibility among different versions of the `som_SamplePart` class. The `functionprefix` identifier customizes the names of the implementation functions in the .cpp file. Next, the definition lists all of the methods that `som_SamplePart` overrides.

Listing B-2 shows the beginning of the implementation section.

Listing B-2 Implementation section

```
            majorversion = 1; minorversion = 0;

            functionprefix = som_SamplePart__;
            override:
        //# ODObject methods
                somInit,
                somUninit,
                AcquireExtension,
                HasExtension,
                Purge,
                ReleaseExtension,
        //# ODRefCountedObject methods
                Release,
        //# ODPersistentObject methods.
                CloneInto,
                Externalize,
                ReleaseAll,

        //# ODPart methods
                AbortRelinquishFocus,
```

```
                AcquireContainingPartProperties,
                AdjustBorderShape,
            ...
```

The final portion of the implementation section, which is private to the som_SamplePart object, contains two parts: a passthru statement and declarations for the som_SamplePart object's instance variable. The passthru statement directs the SOM compiler to write specified information directly into a specified output file. In this case, the information is a forward declaration for the class type SamplePart, which is required by the C++ compiler that will process the output file. The passthru statement specifies the output file to be the implementation binding file with extension .xih. The declaration of the som_SamplePart object's instance variable follows, specifying the variable's data type and identifier.

Listing B-3 shows the final portion of the som_SamplePart class interface definition.

Listing B-3 Last section of the som_SamplePart class definition

```
#ifdef __PRIVATE__
            passthru C_xih =
                "class SamplePart;";

            SamplePart*     fPart;

#endif //__PRIVATE__
        };

#endif //__SOMIDL__
    };
};
```

SOM class definitions can also include a releaseorder statement to maintain binary compatibility for the SOM class, although som_SamplePart does not need or use the feature. The releaseorder statement specifies the order in which the SOM compiler must incorporate the methods in the class's data structure. The releaseorder specification appears in a private form, protected by an #ifdef __PRIVATE__ compiler directive, and a public form for clients, which reserves space for the correct number of methods without naming them. Listing B-4 shows an example of a `releaseorder` statement.

Listing B-4 `releaseorder` statement

```
    releaseorder:
#ifdef __PRIVATE__
        method1, method2, method3;
#else
        reserved1, reserved2, reserved3;
```

Implementation Template

The SOM compiler generates an implementation template for each method declared in a SOM class. You must fill in the complete implementation for each method in your class. The SOM compiler puts certain macro invocations and other artifacts into these stub method definitions, which you can see by examining the emitted implementation template file (extension .cpp).

Define and Include Directives

Because the implementation template file is the primary source file for the SOM object declared in the corresponding IDL file, the SOM compiler generates a compiler symbol specifying the module name (if any is declared in the IDL specification), the class name, and the words `Class_Source`, all separated by underscore characters. This directive forces a one-to-one correspondence between the IDL class specification and its implementation.

Listing B-5 shows the `som_SamplePart` class source define directive.

Listing B-5 Class source define directive

```
#define SampleCode_som_SamplePart_Class_Source
```

The include directives in the implementation template file include the implementation binding or private header file (extension .xih) only for the same class whose implementation file this is. The private implementation file is generated by the SOM compiler. It contains macros that give access to instance variables and invoke superclass methods.

Include directives for other SOM classes used in the implementation code include the usage binding or public header file (extension .xh) generated for those classes. For non-SOM classes defined in C++, such as `SamplePart`, the implementation template file includes the regular C++ header file (extension .h).

Function Prototype

The prototype of each stub method definition generated by the SOM compiler includes several symbols defined in the implementation binding file. Generally, you do not need to worry about these symbols because the SOM compiler simply does the right thing.

Listing B-6 shows a typical SOM-generated function prototype with parameter list that appears in the `som_SamplePart` implementation template file.

Listing B-6 Typical SOM function prototype

```
SOM_Scope       void
SOMLINK         som_SamplePart__InitPart
            (
                SampleCode_som_SamplePart*  somSelf,
                Environment*                ev,
                ODStorageUnit*              storageUnit,
                ODPartWrapper*              partWrapper
                    )
```

The symbol `SOM_Scope` is defined in the implementation binding file as `extern C` to generate correct language bindings with parameters in the proper order. The term `void` is the return value of the method. The symbol `SOMLINK` is defined by SOM; it is a preprocessor directive to help the linker, and its value is system specific. The method name appears next appended to its function prefix value, which is defined in the IDL file, as `som_SamplePart__`. The parameter list is described in the following section.

Parameter List

The stub function implementations include two standard parameters in every signature: the self-pointing parameter and the environment parameter. The IDL descriptions of some SOM classes also include a context specification, causing a third standard parameter to be generated, but it does not appear in `som_SamplePart`. Other parameters are specific to the individual method.

The self-pointing parameter is a pointer to the object that responds to the method call. This parameter is required for implementation languages having no concept of objects, such as C. To call a SOM object's method from C, you must pass the object pointer as the first argument of the calling syntax. From C++, however, you specify the object with the method call in the standard C++ manner (such as `myPart->Externalize`). The name for this parameter is always `somSelf`, a convention upon which the macros in the implementation binding file rely.

The environment parameter is a pointer to the environment data structure defined by CORBA. (CORBA stands for *Common Object Request Broker Architecture,* an interface standard promulgated by the Object Management Group industry consortium.) The environment structure passes exception information between the caller and the called method.

Default Method Calls

By default, every stub method includes three statements. Listing B-7 shows the default statements that appear in the `som_SamplePart` object's `InitPart` method definition.

Listing B-7 Stub method default statements

```
SampleCode_som_SamplePartData *somThis =
                         SampleCode_som_SamplePartGetData(somSelf);
SampleCode_som_SamplePartMethodDebug("SampleCode_som_SamplePart",
                         "som_SamplePart__InitPart");
SampleCode_som_SamplePart_parent_ODPart_InitPart(somSelf,ev,
                         storageUnit,partWrapper);
```

The first statement initializes a local pointer variable named `somThis` that provides access to the instance variables (or attributes) of the class. The `somThis` variable points to a SOM-generated data structure representing the instance variables, which has a type created by appending the word `Data` to the class name. Macros in the implementation binding file depend on the `somThis` variable to create getter and setter methods for each instance variable.

The second of the three default statements aids debugging. It depends on the `SampleCode_som_SamplePartMethodDebug` macro defined in the implementation binding file. If the SOM global variable `SOM_TraceLevel` is set to 1 in the client program, this macro produces a debugging message each time the method executes.

The third default statement is a macro that invokes the inherited superclass method of the same name. This statement is generated only for overridden methods. As you fill in the function body of each method, you should delete this statement or place it appropriately in your code: before or after the actions you take in your override.

Index

U

V

W

This Apple manual was written, edited, and composed on a desktop publishing system using Apple Macintosh computers and FrameMaker software. Proof pages were created on an Apple LaserWriter Pro printer. Final page negatives were output directly from text files on an Agfa Large-Format Imagesetter. Line art was created using Adobe Illustrator™ and Adobe Photoshop™. PostScript™, the page-description language for the LaserWriter, was developed by Adobe Systems Incorporated.

Text type is Palatino® and display type is Helvetica®. Bullets are ITC Zapf Dingbats®. Some elements, such as program listings, are set in Adobe Letter Gothic.

WRITER
Alan Spragens

DEVELOPMENTAL EDITOR
Laurel Rezeau

ILLUSTRATOR
Deb Dennis

PRODUCTION EDITOR
Alexandra Solinski

LEAD WRITER
Dave Bice

WRITING MANAGER
Trish Eastman

Special thanks to Steve Smith, for writing and explaining the sample code, and to Jens Alfke, for writing much of the original utilities documentation.

Acknowledgments to Tantek Çelik, Sue Dumont, Troy Gaul, Vincent Lo, David McCusker, Nick Pilch, Richard Rodseth, and Dave Stafford.