Microsoft

, Software Design Engineer
Systems Developer Relations

Version 1.01

Your feedback is a very important part in providing documents such as these to the developer community for Microsoft Windows. Please let me know how you used this document, how you used the sample code, what aspects you found helpful, and what you didn't like. A work like this document is always open to improvement, so please report any problems, errors, or general criticisms you might have. Reach me through mail, fax (dial (206)93MSFAX), or electronic mail at the following addresses:

Internet: kraigb@microsoft.com
Compuserve: 70750,2344

At the very least, please tell me what you think. With your help, future documents and samples covering technologies in Microsoft Windows will be even better!

Kraig Brockschmidt
12 February, 1992
Redmond, Washington USA

For technical support in implementing OLE into your application, contact Microsoft Product Support Services using Microsoft OnLine or through the WINEXT forum on Compuserve. *Please, do not ask the author for such technical support as any requests for such will simply be referred to the appropriate support service.*

Updates and error lists to the document and sample code will be posted on both OnLine and Compuserve as necessary.

Object Linking and Embedding Client Implementation Guide

Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

Table of Contents

1 Introduction

This *Object Linking and Embedding (OLE) Client Implementation Guide* is intended to help you, as an applications programmer, add OLE client capabilities to a new or existing application. This guide provides OLE technical background information, suggestions to prepare an application for becoming an OLE client, and step-by-step details about where to add code, what OLE functions to call, and what specific actions to perform.

A classic problem in implementing OLE, which I encountered in writing the sample client, is that you must write considerable code before testing anything. The step-by-step implementation section provides various points where you may compile and possibly test the OLE code you just added. This incremental approach gave me a clearer picture of what the code was actually doing and allowed me to focus on debugging a small piece of code.

OLE is a protocol that complements, not replaces, DDE and standard clipboard data exchange. It is also a protocol that easily sits on top of an existing application. If you are planning to write a server application and have not yet done so, write the non-OLE application first then follow the steps in this guide to implament OLE. "Integrating OLE" into an application is simply not necessary, because OLE only affects a few specific parts.

This document will not teach you the concepts and architecture of OLE. For background information, consult the Windows 3.1 Software Development Kit.

With this guide you should be able to add basic OLE support to a suitable client application within a week, give or take some days depending on the complexity of your application. The sample client demonstrates the steps described in this guide and contains many pieces that you can immediately transplant to your application. Documentation for these pieces is given in the context of the step-by-step implementation section where they apply.

2 Required Windows Programming Knowledge

This document assumes a working knowledge of those areas of Windows listed below. All areas except atoms and DDE you will need to understand–if you are unfamiliar with an area, please consult one of the listed references.

Area	***Reason for Understanding the Area***
Atoms	Atoms are a convenient method to store variable length strings in a single integer, especially for structures.
Callback functions	MakeProcInstance required for initializing function tables.
Clipboard I/O	OLE clients need to be able to open the clipboard and paste OLE objects, as well as possibly look for other formats such as metafiles and bitmaps.
DDE	Since OLE 1.x works off the DDE protocol, a knowledge of DDE may help you understand how the OLE protocol works.

Dialog Boxes Many of the user interface requirements for an OLE client require dialog boxes with lists of object or object classes. A typical OLE client will add at least two new dialog boxes, one of which is quite complicated.

Dynamic Menu Changes Part of an OLE menu items to reflect the verbs available for embedded objects.

File I/O Any client application that benefits from OLE needs to save objects to a file, albeit file I/O for an OLE client is simple.

Mapping modes The OLE libraries express all dimensions in MM_HIMETRIC units; your application may need to convert such units to another mapping mode.

Message Loops The OLE 1.x libraries depend on DDE messages, so the application must process messages to allow OLE to function, possibly impacting background processing.

References

Petzold, Charles	*Programming Windows 2nd Edition*	Microsoft Press 1990
Richter, Jeffrey	*Windows 3: A Developer's Guide* 1991	M&T Publishing
Wilton, Richard	*Windows Developer's Workshop*	Microsoft Press 1991
Yao, Paul and Norton, Peter	*Programming Techniques*	*Windows 3.0 Power* Bantam Books 1990

3 Conventions

1. In-line code, taken from the sample server included with this guide, is presented in small fixed-pitch fonts:

```
os=OleUpdate(pObj);

if (OLE_WAIT_FOR_RELEASE==os)
   FOLEReleaseWait(FALSE, pDoc, pObj);
```

2. Special information of importance is offset in gray boxes.

3. Definitions of terms used in this guide, like "client" and "Native," are given in Appendix A.

4 Sample Client: PATRON

Accompanying this implementation guide is a sample OLE client called Patron, which simply stands as a loose synonym for 'client.' Perhaps it's personal revenge on my part for everything being called 'demo' or 'client.'

Patron is a single-document application that simply allows you to save and load files composed of embedded and linked objects. Each object is contained within a separate child window, since that method is most convenient for demonstrating how to use the OLE API. Your application most likely has other significant data structures for items like pictures or tables as well as methods for dealing with their display and positioning. Patron does not get that complicated because techniques to move objects in a document have no bearing on implementing an OLE client.

In writing Patron I have made an effort to provide a considerable amount of reusable code to greatly reduce your implementation time. Appendix B contains a guide to the reusable functions in Patron's OLE code. However, many of the operations in an OLE client require enumerating objects in a document and retrieving application-specific data about each object. So in order to use Patron's code verbatim you need to use the functions and techniques to allocate and manage various data structures. Of course, since you have the source you can always modify the code to fit better into your application.

5 Source Code Structure

File	Contents
Files Dealing with OLE	
blackbox.c	BlackBox creation function and BlackBox window procedure. BlackBox is the window class that holds an object and simply provides a rectangle in which to draw an object.
blackbox.h	Prototypes and definitions for blackbox.c.
clip.c	Functions to handle Cut, Copy, Paste, and Paste Link commands.
exit.c	Application cleanup function.
file.c	Function to handle File New, Open, Save, Save As, and Exit as well as maintenance of the dirty flag.
fileio.c	File I./O functions, to read and write .PTN files.
init.c	Initialization functions that call OLEDOC.C to initialize the application as an OLE Client.
insdrop.c	Handlers for the Edit Insert Object command and the WM_DROPFILES message.
patron.c	Main window procedure, object window cleanup functions, and the CallBack function required of an OLE client application.
patron.h	Prototypes and definitions for application-specific functions.
Registration Database Components	
register.c	Registration database helper functions to enumerate classes, verbs for a class, and to find a class or descriptive name from other information such as a file extension or a class name.
register.h	Function prototypes for REGISTER.C enabling an application to use REGISTER.C as a library.
OLE-Components	

oclient.h	Prototypes, definitions, and structures for OLE functions. OCLIENT.H acts as a header for a library composed of the OLE-specific functions below.
oclient.rc	Dialog boxes and string resources for OLE specific functions.
oledoc.c	Constructor and destructor for the DOCUMENT data structure which hold all document-related information such as clipboard formats and a list of objects contained in the document.
oleins.c	Function to display and handle the Insert Object dialog box. Also creates an object of the chosen class to return to the caller.
olelib.c	Miscellaneous OLE helper functions: wrapper for the Common Dialog File Open/Save As, parsing filenames and extensions from full pathnames, file read and write functions that handle >64K data, and mapping mode conversions.
olelink1.c	Functions to display and handle the complex Links dialog.
olelink2.c	Helper functions for the links dialog to create and manipulate listbox strings.
oleload.c	Functions to handle link updating on File Open.
olemenu.c	Functions to manipulate the Edit menu commands depending on clipboard data availability and the selected object.
oleobj.c	Constructor and destructor for the OBJECT data structure and functions to help manipulate them, such as searching for a particular object or enumerating them.
olestrea.c	Constructor and destructor for the STREAM data structure as well as standard OLESTREAM methods.
olevtbl.c	Constructors and destructors for OLECLIENTVTBL and OLESTREAMVTBL structures.

6 Isolation of Global Data and Strings

Since the OLE protocol can quickly have you using global variables, I have isolated those not dealing with OLE and those dealing with OLE into two separate structures: GLOBALS and DOCUMENT. Two global variables are pointers to these structures: pGlob and pDoc. pGlob points to application globals unrelated to OLE except that OLE code makes use of them. It contains the currently loaded file, window handles, the application instance, etc. pDoc, allocated through the PDocumentAllocate function in OLEDOC.C, contains document-related variables such as OLE clipboard formats, headers to the list of OBJECT structures, and even temporary work memory. Patron only uses one DOCUMENT structure since it has only one document. An MDI client would use multiple DOCUMENT structures, each containing OLE information pertaining to a document.

I chose to write the code in this manner to separate these unrelated globals from each other and to provide an easy method to identify the use of such globals in code. Anytime a global is used it must be referenced off one of the pointers, as in pGlob->szFile or pDoc->pszData1. This clearly marks the use of a global as opposed to a local variable.

Two other global variables, **rgpsz** (PATRON.C) and **rgpszOLE** (OLEDOC.C), store near pointers to strings loaded from the string tables defined in PATRON.RC and OCLIENT.RC, respectively. At startup, the functions HLoadAppStrings (INIT.C) and HLoadOLEStrings (OLEDOC.C) load the string tables into local memory and initialize the arrays. All strings are subsequently referenced by an offset into **rgpsz** (as in rgpsz[IDS_CLASSPATRON], for application strings) or through the macro **PSZOLE** (defined in OCLIENT.H, for OLE-related strings). Note that the OLE string table starts at string 1024 and the PSZOLE macro uses the string index minus 1024 to select a pointer from rgpszOLE.

7 OLE Technical Background

Before dealing heavily with the implementation steps for an OLE client application, some background on the OLE components and concepts of an OLE client:

- OLECLI.DLL, OLE.H
- SHELL.DLL, SHELLAPI.H
- Library Redistribution and Installation
- OLE Communication Routes
- OLE Data Structures and Application-Specific Variations
- Handling Asynchronous Operations
- OLE Clipboard Formats
- Registration Database: OLE Keys and Values

Note that OLECLI.DLL and SHELL.DLL are redistributable libraries; any application that makes use of these libraries must ship the required DLLs for users who may be running under Windows 3.0 instead of Windows 3.1. During your application installation program, perform version checks on these DLLs before copying them to a user's hard drive. In addition, if you redistribute OLECLI.DLL also ship OLESVR.DLL to insure that both libraries are updated together. More information about redistributable libraries and version checking is available in the Windows 3.1 Software Development Kit.

8 OLECLI.DLL and OLECLI.LIB

The OLECLI dynamic link library contains functions through which a client application registers documents and manages objects within those documents. OLECLI.LIB is the import library to which you link your client application. In all, OLECLI.DLL exports 55 functions for use by client applications. For full documentation for these functions, consult a Windows 3.1 Programmer's Reference.

The following tables list the OLECLI functions organized into various functional groups. The column "When Used" lists the menu command or other operation that uses the function. Those functions marked <optional> are not used from any standard functional requirements of an OLE client.

Document Management: Documents are containers for objects.

Function	When Used	Description
OleRegisterClientDoc	File New, Open	Registers a client document.
OleRenameClientDoc	File Save As	Informs OLECLI that a registered document was renamed.
OleRevertClientDoc	File Reload[1]	Informs OLECLI that a registered document was reloaded.
OleRevokeClientDoc	File Close	Informs OLECLI that a registered document was closed.
OleSavedClientDoc	File Save, Save As	Informs OLECLI that a registered document was saved.

Object Creation and Destruction: Adding or removing object from a container document.

Function	When Used	Description
OleClone	<optional>	Creates an exact copy of another object.
OleCopyFromLink	<optional>	Creates an embedded object copy of a linked object.
OleCreate	Insert Object	Creates an embedded object of a specified class.
OleCreateFromClip	Edit Paste	Creates an embedded object from data on the clipboard.
OleCreateFromFile	WM_DROPFILES[2]	Creates an embedded object using the contents of a file.
OleCreateFromTemplate	<optional>	Creates an embedded object using a file as a template.
OleCreateInvisible	<optional>	Creates an embedded object with no data or display.
OleCreateLinkFromClip	Edit Paste Link	Creates a linked object from data on the clipboard.
OleCreateLinkFromFile	<optional>	Creates a linked object with a link to a specified file.
OleDelete	Edit Clear	Permanently deletes an object from a document.
OleLoadFromStream	File Open	Loads an object from a document being loaded.
OleObjectConvert	Edit Links	Creates a static object from an existing object.
OleRelease	File Close	Frees an object from memory.

1If an application supports such a function to reload a file, discarding changes.

3 2The WM_DROPFILES message is sent to a client application when the user drags files from File Manager and drops them on the client's document window. The client must call DragAcceptFiles to receive this message.

Object Management: Opening, closing, and drawing objects.

Function	When Used	Description
OleActivate	Editing an Object	Executes an object's verb which could start the object's server.
OleClose	<optional>	Breaks the connection between an object and a server.
OleCopyToClipboard	File Copy, Cut	Places a copy of an object on the clipboard.
OleDraw	Painting, printing	Draws an object onto any device context.
OleEnumObjects	<optional>	Enumerates the objects in a document.
OleEqual	<optional>	Compares two objects for equality.
OleExecute	<optional>	Sends DDE execute commands to an object's server.
OleReconnect	<optional>	Reconnects a linked object to a server after OleClose.
OleRename	<optional>	Informs OLECLI that an object name changed.
OleSaveToStream	File Save, Save As	Saves an object to a file or other storage.
OleUpdate	File Open, Edit Links	Updates an object's data and display.

Object Information Retrieval:

Function	When Used	Description
OleEnumFormats	<optional>	Enumerates available data formats for an object.
OleGetData	Anytime	Retrieves an object's data in a specified format.
OleGetLinkUpdateOptions	Edit Links	Determines if a linked object is automatic or manual.
OleQueryBounds	Anytime	Retrieves the bounding rectangle for object.
OleQueryCreateFromClip	WM_INITMENU[1]	Determines if OleCreateFromClip will succeed.
OleQueryLinkFromClip	WM_INITMENU	Determines if OleCreateLinkFromClip will succeed.
OleQueryName	Anytime	Retrieves the name of an object stored in OLECLI.
OleQueryOpen	Anytime	Determines if a server is currently editing an object.
OleQueryOutOfDate	<optional>	Determines whether an object is out-of-date
OleQueryProtocol	<optional>	Determines if an object supports a protocol
OleQueryReleaseError	Waiting for release	

1or WM_INITMENUPOPUP. The OleQueryCreateFromClip and OleQueryCreateLinkFromClip functions are used like IsClipboardFormatAvailable to determine if a linked or embedded object can be pasted.

		Determines if an error caused an object's release.
OleQueryReleaseMethod	Waiting for release	
		Determines which operation released an object.
OleQueryReleaseStatus	Waiting for release	
		Determines if an object is released or busy.
OleQuerySize	<optional>	Retrieves the size of an object.
OleQueryType	Anytime	Determines if object is linked, embedded, or static.
OleRequestData	<optional>	Retrieves data from a server in a specified format.

Object Information Updating: Informing OLECLI of changes made in the client document to an object.

Function	**When Used**	**Description**
OleSetBounds	Object resizing	Informs OLECLI of the new object rectangle.
OleSetData	Edit Links (etc.)	Changes the object's data for a specified format.
OleSetHostNames	Object creation names.	Provides OLECLI with the object and document
OleSetLinkUpdateOptions		Edit Links
		Changes a linked object between automatic and manual.
OleSetTargetDevice	Printing <optional>	
		Provides OLECLI with information about the output device.

Miscellaneous:

Function	**When Used**	**Description**
OleIsDcMeta	<optional>	Identifies a metafile device context.
OleLockServer	File Open	Keeps a server in memory for updating multiple objects.
OleQueryClientVersion	<optional>	Retrieves the version number of OLECLI.DLL.
OleSetColorScheme	<optional>	Recommends colors for documents and objects.
OleUnlockServer	File Open	Releases a server locked with OleLockServer.

9 OLE.H

OLE.H is the standard include file for all OLE applications, clients and servers alike, and defines structures like OLECLIENT and OLESTREAM. It also enumerates error codes for the OLESTATUS return type, which most OLE functions return.

10 SHELL.DLL, SHELL.LIB, and SHELLAPI.H

SHELL.DLL contains functions to manipulate the registration database and to support the Windows 3.1 Drag/Drop interface. SHELL.LIB is the import library to which you link your application. The include file SHELLAPI.H contains prototypes for the functions below with the exception of the WM_DROPFILES message that is defined in windows.h.:

Function/Message	Description
Drag/Drop	
DragAcceptFiles	Notifies SHELL.DLL that a window can or cannot accept dropped files.
WM_DROPFILES	Message sent to a window when files are dropped on it.
DragQueryFile	Retrieves the filename of a dropped file.
DragFinish	Instructs
Registration Database	
RegCloseKey	Closes a key given a key handle.
RegCreateKey	Creates a key given a name, generates a key handle.
RegDeleteKey	Deletes a key given a key handle and a subkey name.
RegEnumKey	Enumerates subkeys of specified key into a string.
RegOpenKey	Opens a key given a name, providing a key handle.
RegQueryValue	Retrieves text string for specified key.
RegSetValue	Sets the text string (value) for a specified key.

An OLE Client uses the RegEnumKey, RegOpenKey, RegQueryValue, and RegCloseKey functions to find what OLE servers exist in the system and to retrieve specific information about an object class. Specifically, an OLE client needs to retrieve an object's descriptive name and verbs it supports for various user interface purposes. The client will also need to enumerate all OLE object class names contained in the registration database.

Use of the Drag/Drop interface in an OLE client is optional, but without much work you allow users to create Packager objects within your document. This document will describe more about Packager and handling the WM_DROPFILES message later.

11 Library Redistribution and Installation

You may ship various components that your client application uses if you intend to target Windows 3.0 systems that may not have these libraries installed. Redistribution requires no royalties to Microsoft, but does require that your application version check each component before copying them to a user's hard drive, possibly replacing existing versions of the libraries. If the user currently has the same or newer library installed, do not copy the version shipped with your application. Version checking is not covered in this document, so consult a Windows 3.1 Programming Reference for more information on versioning API.

The obvious library you might ship is OLECLI.DLL, but if you ship that library you must also ship the matching OLESVR.DLL to insure compatibility between the two libraries. Since client applications must make use the registration database you must also ship SHELL.DLL. Finally, in order to provide version checking capabilities, ship VER.DLL that contains the versioning API.

12 OLE Communication Routes

Communication between a client application, a server application, an optional object handler, and the two OLE libraries, OLECLI.DLL and OLESVR.DLL, takes place on several different levels:

- The client calls API functions in OLECLI.DLL.
- OLECLI.DLL sends notifications to the client through the "Callback" method.
- If an object handler exists, OLECLI.DLL may call the object handler's exported functions to perform various operations for an object, eliminating the need to start the server application.
- The server calls API functions in OLESVR.DLL.
- OLESVR.DLL calls the exported server methods to request various actions in on the server, document, or object level.
- The server sends notifications to the client through a "CallBack" method pointer provided by OLESVR. OLESVR intercepts these calls and may not necessarily pass the notification on to OLECLI.DLL and the client.
- OLECLI.DLL and OLESVR.DLL communicate through DDE messages.

Although the OLE 1.0 client library, OLESVR.DLL, uses DDE commands to communicate with the server library, a client application should not depend on this fact. Future versions of the OLE libraries may not necessarily use the DDE mechanism. The OLE libraries hide the underlying mechanism beneath a set of function calls and allow the mechanism to change and improve without requiring changes to the application. Concentrate on the OLE protocol and avoid concerning yourself with DDE.

An OLE client application in this model makes function calls to OLECLI.DLL functions and OLECLI calls the CallBack method in the client application to notify it of changes. For example, when the a server changes a linked object it sends an OLE_CHANGED notification (through OLESVR) that eventually ends up in the client's CallBack method. In response to this notification, the client repaints the object by calling **OleDraw** in OLECLI, which in turn asks OLESVR for the updated data.

13 OLE Data Structures and Application-Specific Variations

There are four data structures defined in OLE.H of interest to an OLE client application:

Data Structure	***Contents as defined in OLE.H***
OLECLIENT	A single LPOLECLIENTVTBL
OLECLIENTVTBL	A single far pointer to the client's notification procedure: CallBack.
OLESTREAM	A single LPOLESTREAMVTBL
OLESTREAMVTBL	Far pointers to stream methods, Get and Put.

An OLE client uses LPOLEOBJECT as a type to declare variables, but does not allocate the structure.

These data structures are quite limited as defined in OLE.H: each structure only contains a single pointer to a VTBL that contains pointers to various callback functions:

```
typedef struct _OLECLIENT
   {
   LPOLECLIENTVTBL   lpvtbl;
   } OLECLIENT;

typedef struct  _OLESTREAM
   {
   LPOLESTREAMVTBL   lpstbl;
   } OLESTREAM;
```

To fully utilize these structures, define application-specific modifications to each structure in your own client, adding any additional fields that relate to the structure. Whenever you create an OLE object in a client you pass a pointer to or load a file one of these structures. When a method (CallBack, Get, or Put) is called you are given the same pointer. Since that pointer references the same structure you initially allocated, any information it originally had is still there. The key point to remember is that each structure must *always* have an LPOLE*VTBL type **first** in which the server stores a pointer to the appropriate VTBL.

The Patron sample defines three structures: DOCUMENT, OBJECT, and STREAM. DOCUMENT is NOT a replacement for OLECLIENT but rather is a structure containing document related information global to all objects within that document. The OBJECT structure contains an LPOLECLIENTVTBL as its first field, and is used where an OLE function call required a pointer to an OLECLIENT. Finally, the STREAM structure is used in place of OLESTREAM and contains a file handle as a single additional field.

14 OLECLIENT CallBack and OLESTREAM Methods

The CallBack function contained in the OLECLIENTVTBL is the single method through which OLECLI notifies the client of actions that affect an object. CallBack receives an LPOLECLIENT, a notification code, and an LPOLEOBJECT. The LPOLECLIENT pointer is whatever you passed to various create functions as the LPOLECLIENT parameter. In the case of the Patron sample, this is actually an LPOBJECT pointer, providing CallBack with application-defined data. The notification code contains one of the specifies which event occurred:

Value	Meaning
OLE_CHANGED	The object was changed in the server application. The client repaints the object to show the changes.
OLE_CLOSED	(Embedded objects only) The server that was editing the embedded object closed.
OLE_QUERY_PAINT	OLECLI is processing a lengthy draw operation on an object, so this notification allows the client application to stop drawing if desired.
OLE_QUERY_RETRY	An OLE function call in the client returned OLE_BUSY. This notification allows the application to attempt to retry the operation or terminate it.
OLE_RELEASE	An asynchronous operation has finished and other actions can be taken on the single object affected.
OLE_RENAMED	(Linked objects only) Informs the client that a linked object

was renamed allowing the client to update private data structures. All information in OLECLI is already updated.

OLE_SAVED — Informs the client that an object was saved (linked object) or updated (embedded objects). The client should update and repaint the object.

The CallBack function must be exported from the client application. The notification codes are described again in the **Step-by-Step OLE Client** section.

15 OLE Client Streams and Persistent Naming

An important concept to understand with OLE clients is that of a Stream, which simply means a storage location. OLE allows a client application to store objects anywhere–in memory, in application document files, in separate files, in a database, etc. When a client application calls OLECLI to save an object to a stream with the **OleSaveToStream** function, OLECLI calls the OLESTREAMVTBL Put method; when loading an object from a stream the client calls **OleLoadFromStream** that calls the OLESTREAMVTBL Get method. From within these methods the client determines where and how it stores and retrieves those objects. Note that these methods must handle data potentially greater than 64K.

When a client saves an object, it must store a *persistent name* for that object in its document file. This name uniquely identifies an object allowing the client to locate and reload it when requested to do so. The *persistent* modifier means that an object should retain this name until explicitly renamed with the **OleRename** function or until that object is deleted with **OleDelete**. The persistent name will become increasingly important in the future as object store becomes more integrated with the file system. Finally, since this name is used to locate the object, it must be stored separate from the object itself.

16 OLE Use of Pointers

All OLE structures in OLE function calls and in an application's methods are referenced through pointers, primarily so you can define application-specific structures to replace OLECLIENT and OLESTREAM. A direct result of pointer use is that ***OLE does not work in real mode Windows (3.0)***. If you have an application that currently operates under real mode, adding OLE will eliminate that capability.

The use of pointers necessitates that you allocate memory and keep it locked until freed, a cardinal sin in real mode. However, since far pointers in standard and enhanced mode Windows contain LDT (Local Descriptor Table) selectors, instead of physical segment values, memory can move without requiring the selector (or the pointer) to change. Therefore you can allocate and lock a structure to pass a pointer to OLECLI, and leave that memory locked until you free it.

Make the Best use of Local/Global Memory for OLE Structures

Local Memory: (LocalAlloc) Allocate as LMEM_FIXED or LPTR (windows.h defines LPTR as LMEM_FIXED | LMEM_ZEROINIT in windows.h). Do not use LMEM_MOVEABLE followed by a LocalLock since that creates a sandbar in the high area of the local heap. Allocating LMEM_FIXED allocates from the bottom of the stack, which is the best place for locked memory to reside.

Global Memory: (GlobalAlloc) Allocate as GMEM_MOVEABLE followed by a GlobalLock. The largest concern with global memory is how much of it resides in conventional memory below the 1MB line. Allocating GMEM_FIXED automatically places that memory as low as possible in the global heap whereas GMEM_MOVEABLE allocates from the top. Since the memory allocated GMEM_MOVEABLE can physically move after GlobalLock, you create no sandbars.

17 Handling Asynchronous Operations

The OLESVR and OLECLI libraries under OLE 1.x communicate through DDE messages; while you should never depend on this fact, it does have repercussions in your application. In particular, most of the OLE function calls may return the OLE_WAIT_FOR_RELEASE code, signifying that the OLE libraries started an asynchronous operation on a specific object. While an asynchronous operation is happening, the client cannot call any other OLE function that affects the same object since only one asynchronous operation per object is supported (this is especially important within the CallBack method–be sure to make no OLE calls from within that function). Synchronizing calls in this manner is called "waiting for release" on the object in question.

However, the application may continue operations on other objects during this time and does so by processing messages (allowing the libraries to process DDE messages) in a special message loop until the object is released. The client has two techniques to determine when an object is released. Either method works the same and the release will occur at the same time:

1. [Interrupt technique] Watch for the OLE_RELEASE notification in the CallBack method. When this notification occurs, set a flag that causes the message loop to exit.

2. [Polling technique] While processing messages, repeatedly call **OleQueryReleaseStatus** until it returns OLE_OK.

Since the original OLE function call that returned OLE_WAIT_FOR_RELEASE cannot return any other error code, a client must call **OleQueryReleaseError** to determine if the function was actually successful. If OleQueryReleaseError returns OLE_OK then everything is fine. Otherwise the return value provides the details of the error. For example, consider the code below to update an object:

```
OLESTATUS       os;

...

os=OleUpdate(pObj->pObj);

if (OLE_WAIT_FOR_RELEASE==os)
```

```
    {
    //FOLEReleaseWait processes messages while waiting for pObj.
    FOLEReleaseWait(FALSE, pDoc, pObj);
    os=OleQueryReleaseError(pObj->pObj);
    }

if (OLE_OK!=os)
    //Signal error condition
    ;
```

If **OleUpdate** initially returns anything except OLE_WAIT_FOR_RELEASE, then we immediately skip to checking the return value for an error code. Otherwise we need to wait for release using a message processing function like FOLEReleaseWait in Patron's OLEOBJ.C. Once the object is released, we reload our OLESTATUS variable with the outcome of the asynchronous operation returned from OleQueryReleaseError.

Finally, the **OleQueryReleaseMethod** function returns a code indicating which OLE call started the asynchronous operation that was last completed. For example, when the CallBack method receives the OLE_RELEASE notification for the OleUpdate call above, OleQueryReleaseMethod would return OLE_UPDATE. From CallBack as well, we could call OleQueryReleaseError to determine the cause of the release. Given the operation and the result of that operation, we can then take specific actions to terminate the sequence of OLE calls, notify the user, and so on.

In most cases, Patron passes the return value of an OLE function to an error handler, **OsError** (OLEOBJ.C), that waits for the object to be released if necessary and calls **OleQueryReleaseError** for the final outcome of the operation. It also handles the OLE_BUSY return value described below.

18 Waiting for All Objects

Some operations, such as closing a file, affect all objects in a document together, so waiting for each object in turn is slow. Instead of waiting one object at a time, you can wait for all objects together. Note, however, that in such a procedure you will find it much more difficult to articulate errors for individual objects once you exit the message loop. However, for operations like document close, you may simply not care about such errors.

To wait for all objects together, maintain a special counter to track how many objects are released and how many are still waiting:

- Before executing an operation on all object, set the counter to zero.
- For every OLE call that returns OLE_WAIT_FOR_RELEASE, increment the counter.
- For every OLE_RELEASE notification received in CallBack, decrement the counter.
- In your message processing function, watch for this counter to fall to zero, at which point you terminate the loop.

Waiting for all objects at once keeps your application somewhat asynchronous during OLE operations, and as mentioned, makes it harder to detect and recover from specific errors. If you wait for each object individually as soon as any call returns OLE_WAIT_FOR_RELEASE, you turn OLE into a more synchronous protocol. Waiting for all objects together is simply a possibility that you may be able to take advantage of. Take extra caution to insure that other OLE operations do not occur between the time you set your counter and the time you wait. Otherwise your counter may fall below zero.

19 The OLE_BUSY Return Code

If an object's server is locked in some modal operation or an asynchronous operation is not complete on the object, an OLE function call may return OLE_BUSY, indicating that the operation cannot be executed. You may either wait for the object to be released or terminate the operation. Whenever this busy condition occurs, your CallBack method will receive the OLE_QUERY_RETRY_BUSY notification. The return value of CallBack indicates whether or not to continue the operation in response to this notification. You may also want to allow the user to wait or cancel.

20 Debugging Asynchronous Operations

Multiple operations on multiple objects can, of course, become a mess to follow. During development of your client, stick with waiting for an object to be released as soon as any OLE function returns OLE_WAIT_FOR_RELEASE, blocking all operations on any other objects. While waiting for each object in turn will probably cause the application to run slower, it effectively makes OLE synchronous. Once you have debugged your operations on single objects you can reinstate asynchronous actions for performance reasons, if it's even necessary.

In short, don't let the asynchronous behavior of OLE get in your way of programming the right sequence of calls.

21 Clipboard Formats and Conventions

OLE clients are concerned with several standard clipboard formats:

1. "**Native**," a server's raw data structures.
2. "**OwnerLink**," information about a server, used for embedding an object.
3. "**ObjectLink**," information about a file a server has saved, used for linking an object.
4. **CF_METAFILEPICT**, a continuously scalable presentation displayed in the client.
5. **CF_BITMAP**, a roughly scalable presentation displayed in the client.

Native, **OwnerLink**, and **ObjectLink** are formats defined in the OLE protocol; all OLE applications, servers and clients, register these formats with RegisterClipboardFormat, which returns the same integer value in any applications.[1] These three formats describe a linked or embedded object within a client document, allowing the OLE libraries to launch the correct server application when the user activates an object in the client document. The CF_METAFILEPICT and CF_BITMAP formats provide OLECLI with a visual representation of an embedded or linked object.

Client applications generally deal with only the ObjectLink format when it displays information about a linked object (such as the filename of the link). Clients do not generally use Native data, probably will not use OwnerLink, and would only be concerned with metafiles and bitmaps if it can manipulate that data outside the context of OLE. In all actuality a client application may only need to register ObjectLink, but should register all the OLE formats for future use.

22 Native, OwnerLink, and ObjectLink Formats

Format Name	Contents
Native	Application-specific data structure, understood only by the server application that created it. It must enable the server to completely recreate the object. The OLE libraries and client applications treat Native data as a stream of raw bytes, that is, they do not assume anything about the contents of that data.
OwnerLink	Sequence of three null-terminated strings in memory, where the next string follows the preceding string's null-terminator and the sequence is terminated by two NULLs: String 1 Object class name String 2 Document name String 3 Object selection/name, assigned by the server application The OwnerLink format describes an embedded object.
ObjectLink	Identical to OwnerLink in OLE 1.x, but describes a linked object: String 1 Object class name String 2 Full path to the document file String 3 Object selection/name, assigned by the server application

1RegisterClipboardFormat simply uses AddAtom on the given string, and atoms are constant across the system for any given string.

In OwnerLink and ObjectLink the *object class name* is the registered class of objects that a server handles. The *document name* and *object name* in OwnerLink are strictly used to identify the object within whatever server or document it resides. The *document name* in ObjectLink contains a path name of the linked file, allowing OLECLI and the client application to determine the exact file to which an object is linked. The *object name* in ObjectLink specifies what part of the document applies to the object, such as a range of cells in a spreadsheet. The object name only has meaning to the server application and the user, as clients do not parse or manipulate the name.

23 Registration Database: OLE Keys and Values

The registration database is a system resource that contains keys and values, both of which are strings. All OLE-related keys start from a root key called HKEY_CLASSES_ROOT, as all objects are members of some class. The first *subkeys* from HKEY_CLASSES_ROOT are the object's classname and the application's file extension:

Key Name	**Required Value**	**Example**
HKEY_CLASSES_ROOT*classname*	Readable version of class name.	Schmoo Server 1.0
HKEY_CLASSES_ROOT\\.*ext*	Associated class name for the extension	Schmoo1.0

The HKEY_CLASSES_ROOT*classname* key has two standard extensions to which additional subkeys are attached:

HKEY_CLASSES_ROOT*classname*\protocol\StdFileEditing
HKEY_CLASSES_ROOT*classname*\protocol\StdExecute

Additional subkeys attached to **\protocol\StdFileEditing** describe more specific characteristics of the OLE protocol supported by the server:

Key Name ...\StdFileEditing	**Value**	**Example**
server	Full path to server executable	e:\win31\schmoo\ schmoo.exe
handler (optional)	Full path to object handler DLL	e:\win31\schmoo\schmooh.dll
verb\0	Primary verb	Edit
verb\1, **verb\2**, ... (optional)	Secondary, tertiary, etc., verbs	Open, etc.
SetDataFormats (optional)	CSV string of data formats	Native, CF_METAFILEPICT

RequestDataFormats (optional) CSV string of data formats Native, CF_METAFILEPICT

The **\protocol\StdExecute\server** is an optional key that has a value of the application path, just like the server subkey in **StdFileEditing**. Windows uses this entry to find the server if a client application attempts to send commands through the **OleExecute** function.

Verbs are the types of actions a user can perform on an object, such as "Play," "Edit," and "Open." For most graphical applications "Edit" is the only verb provided, since editing is the only thing to do with the data. An application like the Windows 3.1 Sound Recorder supports two verbs, "Play" and "Edit," where Play is the *primary* verb and Edit is the *secondary* verb. When a user double-clicks an object in a client, the client application invokes the primary verb for that object; for Sound Recorder that means play the sound. All other verbs are accessed through the client application's menu.

24 Preparing an Application to Become an OLE Client

Any application that manages some sort of document and can contain items such as pictures can become an OLE client application. However, OLE does intrude somewhat into the existing structure of the application and small changes to your application prior to implementing OLE can help you stay more focused on adding OLE support. Most of the suggestions below deal with isolating areas of your code that will be affected by OLE. When you have a single function to perform a specific operation, it will be much easier to add a few OLE calls to that one function. The sections below discuss various parts of your code to isolate, with the exception of the first that deals more with files:

- Decide How to Reference Objects in Files; Version Numbers
- Isolate Data
- Isolate Initialization and Cleanup Procedures
- Isolate Painting and Printing Code for Objects
- Isolate Menu Enabling and Disabling Functions
- Isolate Clipboard I/O
- Isolate Your Dirty Flag
- Isolate Background Processing Schedulers
- Isolate Mapping Mode Conversions

These suggestions are not a mandate–in no way are you required to isolate code in this fashion. Doing so may hurt your application's performance because of the increased overhead in function calls. However, isolating functions will speed your development as it saves you from having to find every case in which you must add an OLE call.

25 Decide How to Reference Objects in Files; Version Numbers

OLE gives your application complete control over where you store objects. The most convenient method is to simply store objects directly in your existing document files; however, you can also store them in their own files, in a database, etc. In any case, modify your existing file format to reference an object with a persistent name that your OLESTREAMVTBL methods can use to locate and load an object.

So as OLE affects your file format, isolate your file read and write functions such that adding an OLE call is trivial. When writing a document file, store a small data structure for every object in the document to identify that object and its storage location. When you load a document file and encounter one of these structures, you then have the name and location of that object allowing you to reload it.

Finally, Since OLE is an evolving technology, mark your files or object structures with some sort of version information related to the version of OLE under which you saved the file. This will insure that as OLE changes you can provide whatever conversions are necessary between old and new versions of your application.

26 Isolate Data

Consider reorganizing your existing global and static data with the objective of isolating non-OLE data from OLE-specific data that you will add later. For client applications, OLE becomes somewhat integrated with the application in that it affects almost all functions of file manipulation and requires the application to manage where objects reside within documents. OLE is not yet something that is exactly "integrated" a great deal with an application. In the future, OLE may change independent of upgrades in the Windows system, so be prepared to make a revision to the application's OLE code without making changes to the remainder of the application.

Your application will at least one new variable visible at the application level–a structure or structure pointer that contains OLE document-related information, such as OLE clipboard formats, an OLE document handle, and so forth. You can then pass a pointer to this structure to any OLE-specific functions you create to handle various operations.

27 Isolate Initialization and Cleanup Procedures

During application initialization you will need to perform additional steps necessary for OLE, such as registering new clipboard formats and calling **OleRegisterClientDoc**. Isolate your application initialization code to prepare a place for these additional steps. In addition, some of the OLE initialization steps have opposites to perform on application shutdown, so isolate your cleanup code as well.

28 Isolate Painting and Printing Code for Objects

Painting or printing an object requires at least one call to **OleDraw**. If you currently draw some sort of objects in your document, isolate your code to display or print these existing objects. Later, you can add a quick check for an OLE object in this procedure and call OleDraw as necessary. Isolating such code now will save you from tracking down every case where you draw or print an object in order to add the OLE call.

29 Isolate Menu Enabling/Disabling Functions

OLE will affect your existing Edit menu, or require you to create such a menu. Besides adding various new menu items, OLE adds an extra step in handling standard menu items such as Cut, Copy, and Paste–to determine whether or not to disable these items, OLE adds function calls to check for availability of OLE formats.

First, isolate your code that handles the WM_INITMENU or WM_INITMENUPOPUP message case in your main window procedure to enable or disable menu items. Second, centralize any other code to enable or disable the standard menu items. The new code to handle the Edit menu that you will add later is not exactly trivial.

30 Isolate Clipboard I/O

Isolate code to handle each clipboard operation such as Cut, Copy, and Paste. Where you normally cut and copy data you will need to determine if an object is selected and call **OleCopyToClipboard**. Where you normally paste you will need to determine if you want to paste an OLE object instead of other available data. OLE also adds the Paste Link and/or Paste Special commands that may make use of your existing Paste functionality.

31 Isolate Your Dirty Flag

All applications that load, modify, and save files track some sort of 'dirty' flag that signals when the user has made a modification but has not yet saved those changes in a file. Above what your application currently does to set or clear this flag, various OLE operations, such as creating or changing an object, will make the file dirty as well. Isolate code to set such a flag so that you can control the flag from any other function, such as the OLECLIENTVTBL CallBack method.

32 Isolate Background Processing Schedulers

Your application may use a modified message loop in which it detects idle time (no messages to process) and performs a step of background processing before checking for new messages. As describes in the section above on **Handling Asynchronous Operations**, OLE requires a client to enter a separate message loop during an asynchronous operation and wait until an object (or all objects) is released. The message loop is necessary to process DDE messages between OLESVR and OLECLI that perform the actual operation.

Like any other message loop, there may be idle time during this OLE wait loop during which you can again perform some background task. By isolating the code you execute to perform a step of this task, you can call it from any message loop anywhere in the application with the same results.

33 Isolate Mapping Mode Conversions

OLE expresses any rectangles or other dimensional quantities in MM_HIMETRIC units, such as a rectangle returned by the **OleGetBounds** function. If your application does not deal in MM_HIMETRIC already, create a function to convert between MM_HIMETRIC and the mapping mode you normally use, such as MM_TEXT. Your application can then continue to deal in its usual mapping mode, only converting units when exchanging dimensions of an object with OLECLI.

With that, let's start coding...

34 Step-By-Step OLE Client

This section describes the necessary code additions and changes to make an existing application an OLE client. The incremental approach in this implementation guide provides points at which you can compile and test your code, marked by a gear symbol. At these points your server may not be fully functional, but you can insure that certain elements do work perfectly. This is very important to making your life with OLE simpler, because later steps depend on the previous steps working correctly.

Implementing an OLE Client involves requires a considerable amount of work to just meet the user interface standards. Creating documents and objects is quite simple compared to the user interface. To that end, all the OLE-related code in Patron is readily usable in your application at least as a starting point, but requires that you use its structures (and functions to allocate those structures) and API.[1] Appendix B contains documentation for the functions contained in the API. Since the source to Patron is provided, you are, of course, free to make your own modifications.

This section is organized into the following steps:

Define OLE Data Structures	Modify your include files to contain the necessary OLE structures.
Create Registration Database Utility Functions	An OLE client application makes frequent use of the registration database. These utility functions greatly simplify the extraction of key information from the registration database.
Implement Basic Methods	The methods for an OLE client are quite trivial to implement, but are necessary to write before writing OLE initialization code.
Initialize the Application and VTBLs	Register the OLE clipboard formats, allocate and initialize OLE-related structures, and register initial documents.
Handle Simple Shutdown: File Close	Release objects and revoke documents.
Create an Object Manager	Before creating objects, your application needs a method to store and enumerate those objects.
Add OLE Menu Items	Add the basic user interface for an OLE client, including the Edit Paste Link/Special

1Be sure to test this code with your application as well to incude that it meets your standards for error handling and robustness.

command and the Insert Object command.

Create Objects and Other Object Operations Implement OLE-specific Paste, Paste Link, Paste Special, and Insert Object commands; place those objects back on the clipboard, convert those objects to static items, and release or delete an object.

Display and Print Objects; Resizing Draw the object or print it to a device context; handle resizing of the object.

Add the Object Verb Menu and Execute Verbs Execute an object's verbs with a quick double-click; attach verbs to your Edit menu to fulfill an important user interface requirement.

Implement File Menu Commands Close a document and register a new one; Save objects to a file and load them though the OleSaveToStream and OleLoadFromStream functions.

Update Links and Create the Links Dialog Update links when loading them and implement the links dialog, the most complex user interface requirement in OLE.

Additional OLE Client Functions Overview of other OLE functions not previously mentioned.

35 Define OLE Data Structures

OLE clients concern themselves with two basic data structures defined in OLE.H. However, the definitions of these structures include only a single pointer to a method callback table:

```
typedef struct _OLECLIENT
        {
        LPOLECLIENTVTBL         lpvtbl;
        } OLECLIENT;

typedef struct _OLESTREAM
        {
        LPOLESTREAMVTBL         lpstbl;
        } OLESTREAM;
```

As mentioned before, define your own application-specific versions of these structures, keeping the **lpvtbl** (or **lpstbl**) field at the beginning, then adding any additional data. Place object-related data in your OLECLIENT replacement and storage information (like a file handle or pathname) in your OLESTREAM replacement. You can also name these structures anything you like and typecast them to the appropriate OLE type when passing pointers to OLE function calls.

The Patron sample defines three structures: DOCUMENT, OBJECT, and STREAM, where OBJECT replaces OLECLIENT, STREAM replaces OLESTREAM, and DOCUMENT holds information global to all objects with in a document. Each of these structures is described in more detail below.

36 The DOCUMENT Structure

Warning:

The DOCUMENT structure in Patron ***is not*** used in place of the OLECLIENT structure. It acts as a document structure holding variables applicable to all objects in a document. Patron never passes a pointer to this structure to any OLE API calls, although a pointer is almost always passed to Patron's OLE-specific functions.

```
typedef struct
   {
   LPOLECLIENTVTBL     pvt;          //Stores the global VTBL for all objects
   LHCLIENTDOC         lh;           //Required for later OLE calls.
   ATOM                aCaption;     //Caption of the application.
   ATOM                aFile;        //Filename for the document
   LPFNMSGPROC         pfnMsgProc;   //Message translate/dispatch function.
   LPFNMSGPROC         pfnBackProc;  //Background processing function.
   LPSTREAM            pStream;      //Pointer to our document STREAM

   HWND                hWnd;         //HWND of document window.
   HANDLE              hMemStrings;  //Memory containing OLE strings.

   WORD                cObjects;     //Number of objects in the list.
   LPOBJECT            pObjFirst;    //Pointer to start of OBJECT list.
   LPOBJECT            pObjLast;     //Pointer to end of OBJECT list.

   WORD                cfNative;     //OLE Clipboard formats.
   WORD                cfOwnerLink;
   WORD                cfObjectLink;

   HWND                hList;        //Listbox handle for use in Links dialog
   HWND                cxList;       //Tab width of the links dialog.
   WORD                cLinks;       //Number of links we load from a file.
   WORD                cWait;        //Number of objects awaiting release.

   HANDLE              hData;        //Global handle to scratch area.
   LPSTR               pszData1;     //Pointers to blocks in hData
   LPSTR               pszData2;     //each containing CBSCRATCH
   LPSTR               pszData3;     //bytes.
   } DOCUMENT;
```

Of all the information in the DOCUMENT structure, the lh, cf*, and cWait fields are the most important. **lh** holds the client document handle returned from a call to **OleRegisterClientDoc** that is required in any OLE function call that creates an object. The **cf*** fields contain clipboard formats returned from RegisterClipboardFormat for the "Native," "OwnerLink," and "ObjectLink" formats; the ObjectLink format is used most to retrieve information such as the file to which an object is linked. Finally, the **cWait** field counts OLE_WAIT_FOR_RELEASE return codes for operations affecting all objects as described above in **Handling Asynchronous Operations**.

Of special note are the **hData** field and the three pointers **pszData1**, **pszData2**, and **pszData3**. Supporting the user interface standards for OLE requires a good deal of string manipulation. Instead of continually allocating temporary work buffers in specific functions, Patron allocates a single piece of global memory (storing the handle in hData), locks it down, and stores three pointers into that memory in pszData1, pszData2, and pszData3. In Patron's implementation each block referenced through these pointers is 1K bytes (defined as CBSCRATCH, OCLIENT.H).

Not only does this method relieve functions from making temporary allocations, but greatly reduces the number of possible error conditions in functions, simplifying their flow. The cost is added care to keep these buffers secure across function calls.

37 The OBJECT Structure

Patron's OLECLIENT replacement structure is called OBJECT simply because it contains information relevant to each *object* in a document. Patron uses this structure in place of any OLECLIENT required by the OLECLI library, so the first field is the LPOLECLIENTVTBL pointer:

```
typedef struct _OBJECT
   {
   LPOLECLIENTVTBL     pvt;          //Lets us use this as an OLECLIENT.
   LPOLEOBJECT         pObj;         //Identifies the object in OLECLI
   BOOL                fRelease;     //Released flag.
   BOOL                fOpen;        //Was this object activated?
   struct _OBJECT FAR *pPrev;        //Previous and next OBJECTs in
   struct _OBJECT FAR *pNext;        //the object list.
   LPDOCUMENT          pDoc;         //Parent document
   ATOM                aName;        //Name of object.
   ATOM                aClass;       //Classname of the object.
   ATOM                aLink;        //Path of linked document.
   ATOM                aSel;         //Selection information.
   DWORD               dwType;       //Object type from OleQueryType
   OLEOPT_UPDATE       dwLink;       //Type of link, auto, manual, or static
   BOOL                fNoMatch;     //Marks the object when updating links.
   LPOLEOBJECT         pObjUndo;     //Clone OLEOBJECT for undo usage.
   BOOL                fUndoOpen;    //Indicates if the cloned object is open.
   BOOL                fLinkChange;  //Indicates modification in Links dialog.
   HANDLE              hData;        //App-defined data.
   } OBJECT;
```

Additional information stored in OBJECT simplifies object management. First, it stores whatever OLEOBJECT was created for this particular OBJECT in **pObj**. It also stores two pointers, **pPrev** and **pNext**, to reference the previous and next objects in a linked list–such a list allows the client to quickly enumerate objects. The ATOMs **aName**, **aClass**, **aLink**, and **aSel** are used to store strings for the object's name, class, link file (for linked objects), and selection information (for linked objects). In structures, ATOMs are much more convenient than character arrays to store strings that may be arbitrary long. Note that OLE does not require use of ATOMs–character strings are plenty acceptable.

The **dwType** and **dwLink** fields specify the type of object (embedded, linked, or static) and the update option for a linked object (automatic, manual, unavailable, or static), respectively. Finally, **hData** field provides a HANDLE for an application to store any other data. Patron uses hData to store the window handle where it displays the object.

38 The STREAM Structure

```
typedef struct
   {
   LPOLESTREAMVTBL    pvt;          //Standard
   HANDLE             hFile;        //File handle we need in methods.
   } STREAM;
```

Since the OLESTREAMVTBL methods Get and Put are necessary to read and write data from a file, those methods require some information to locate the object, such as a file handle or file name–that information must be sufficient to relocate the object in some storage. Storing the information in the STREAM structure prevents you from having global variables to pass the same information. When you load or save an object, you first fill your structure and pass its pointer to **OleLoadFromStream** or **OleSaveToStream**. OLECLI then passes this same pointer to the Get and Put methods. Patron just passes a file handle when it uses the stream for file I/O.

Not all stream operations deal with saving objects to a file. If you include an object as part of a larger data structure for clipboard I/O (such as copying rich text format information), then you can use **OleSaveToStream** to save the object to a memory block. The Stream functions and methods simply give the application access to an object's native data.

39 Constructors, Initializers, and Destructors

The Patron sample references its DOCUMENT and OBJECT structures heavily. If you plan on using any of Patron's OLE specific code, you will need to use these data structures (or modify Patron's code to handle your changes). To that end, Patron borrows some C++ techniques to simplify management of these structures: Constructors and Destructors[1] for five separate structures:

Structure	Constructor	Destructor	File
DOCUMENT	PDocumentAllocate	PDocumentFree	OLEDOC.C
OBJECT	PObjectAllocate	PObjectFree	OLEOBJ.C
OLECLIENTVTBL	PVtblClientAllocate	PVtblClientFree	OLEVTBL.C
STREAM	PStreamAllocate	PStreamFree	OLESTREA.C
OLESTREAMVTBL	PVtblStreamAllocate	PVtblStreamFree	OLEVTBL.C

1A constructor allocates and initializes a structure; a destructor frees resources associated with the structure and frees the structure itself.

On startup, Patron calls **PDocumentAllocate** that not only allocates a DOCUMENT structure but also registers clipboard formats, initializes VTBLs, calls **PStreamAllocate**, and allocates the work strings stored in the pszData fields. In short, calling PDocumentAllocate handles almost all the OLE-specific initialization that we'll discuss later and initializes the OLECLIENTVTBL structure through **PVtblClientAllocate**.

PStreamAllocate simply allocates a STREAM structure and calls **PVtblStreamAllocate**, passing pointers to the OLESTREAM methods. By default, PStreamAllocate uses StreamGet and StreamPut in OLESTREA.C.

Before creating an object, Patron calls **PObjectAllocate** to just allocate an OBJECT structure, initialize the LPOLECLIENTVTBL field, and insert itself into the object list referenced in a DOCUMENT. After an OLE function successfully creates an object, Patron calls **PObjectInitialize** to fill the remaining fields. This separate initialize function is necessary because OLECLI allocates OLEOBJECTs, meaning that we cannot attach data to that native structure. Instead, we create the OBJECT structure to use in place of the LPOLECLIENT parameters to various create functions. We cache data in this OBJECT structure, but to initialize the data we need the OLEOBJECT. Since we provide the OBJECT pointer as the LPOLECLIENT parameter, we get the same pointer, and thus all the data in that structure, through the LPOLECLIENT parameter of the CallBack method.

Each constructor function takes a pointer to a BOOL and returns a pointer. If the BOOL is FALSE on return, then the function failed, but the pointer may be non-NULL. In this case, call the destructor to free the data. Each destructor function simply cleans up anything that exists in the structure, then frees the structure itself. Note that **PDocumentFree** calls **PVtblClientFree** and **PStreamFree** calls **PVtblStreamFree**.

While there is little to possibly test here, you can make sure your structures compile cleanly.

40 Create Registration Database Utility Functions

An OLE Client makes considerable use of information stored in the registration database through SHELL.DLL functions. However, the SHELL functions are only primitives to open keys and read values. To simplify OLE operations, create specific utility functions that provide associations between different values in the registration database. For definitions of terms (like descriptive name) please see Appendix A.

1. **Enumerate Class Descriptions** Fill a list with the descriptive names of available objects.
2. **Class Name from Descriptive Name** Find the class name associated with a descriptive name.
3. **Class Name from File Extension** Find the classname associated with a given file

extension.

4. **Enumerate Verbs for a Class** Generate a list of verbs supported by a given object class.
5. **Descriptive Name from Class Name** Find the descriptive name associated with a given class.

In the continuing effort to make client implementation easier, these functions are provided in Patron's REGISTER.C with prototypes in REGISTER.H.

41 Enumerate Class Descriptions: WFillClassList

OLE clients provide an Insert Object command that displays a listbox containing the descriptive names of all available OLE object classes. Obtaining this list of names requires a function to enumerate class names in the registration database, retrieve their descriptive names, and add that name to the list. The WFillClassList function (REGISTER.C) performs these steps to fill a given listbox:

1. If you are filling a listbox in this function, send it LB_RESETCONTENT to insure a clean list.
2. Open the HKEY_CLASSES_ROOT key with a NULL subkey.
3. Set an index counter to zero and enter a loop to find each subkey:
   a. Call RegEnumKey using the index in the counter.
   b. If RegEnumKey fails, then we've enumerated all subkeys; exit loop
   c. If RegEnumKey succeeds, call RegQueryValue on the subkey "<classname>\protocol\StdFileEditing\server" where <classname> is the string from RegEnumKey.
   d. If RegQueryValue fails then the subkey is not a valid OLE class name; continue loop.
   e. Call RegQueryValue on the key from step **a** to retrieve the descriptive name for the class.
   f. Add the string to the list and continue the loop.
4. Call RegCloseKey for the key obtained in **2** and return.

42 Find Class Name Given a Descriptive Name: WClassFromDescription

You may encounter a case where you need the class name for an object's descriptive name. For example, if the user selects a descriptive name from the Insert Object dialog box, then you need to retrieve a class name for that descriptive name before creating an object. This process, as implemented in WClassFromDescription (REGISTER.C), is much like enumerating class names except you search for a match between the given descriptive name and the value of enumerated names:

1. Open the HKEY_CLASSES_ROOT key with a NULL subkey.
2. Set an index counter to zero and enter a loop to find each subkey:

a. Call RegEnumKey using the index in the counter.
b. If RegEnumKey fails, then we've enumerated all subkeys; exit loop
c. If RegEnumKey succeeds, call RegQueryValue on the subkey for that classname.
d. Call **lstrcmp** to compare the value of that classname to the desired descriptive name.
e. If the names match, exit the loop and return the classname.
f. If the names do not match, continue the loop.

3. Call RegCloseKey for the key obtained in **2**.
4. If we found a matching descriptive name, return the classname. Otherwise return an error.

43 Find Class Name Given a File Extension: WClassFromExtension

At a later point you will allow the user to change the file to which an object is linked, using the extension of that file and the class name of that file as defaults in the GetOpenFileName common dialog. To prepare for that capability, implement this simple function to perform a quick lookup:

1. Open HKEY_CLASSES_ROOT key with a NULL subkey.
2. Call RegQueryValue using the key from **1** and the file extension as the subkey. This value is the class name.
3. Call RegCloseKey for the key from **1** and return.

44 Enumerate Verbs for a Class: CVerbEnum

Another user interface requirement for a client is an item on the Edit menu listing the verbs supported by the currently selected object (if there's only a single object selected). An object class' verbs are contained under its classname subkey in the registration database. CVerbEnum (REGISTER.C) generates a list of null-terminated strings where each string contains a verb and the list itself is double null-terminated:

1. Open HKEY_CLASSES_ROOT with the classname as subkey.
2. Call RegOpenKey to open the subkey "**protocol\StdFileEditing\verb**" of the key obtained in **1**.
3. Close the key obtained in **1**.
4. Set a counter to zero and enter a loop:
   a. Convert the counter to ASCII.
   b. Call RegQueryValue using the ASCII string of the counter as a subkey of the key from **2**.
   c. If RegQueryValue succeeds, the value is a verb string; add the string to the list.
   d. If RegQueryValue fails, exit the loop, otherwise continue.
5. Call RegCloseKey for the key obtained in **2** and return.

45 Find Descriptive Name Given a Class Name: WDescriptionFromClass

Along with a selected object's verbs, and OLE client must show the object's descriptive name in the Edit menu. Since an object's classname is readily available from the object itself, we only need to do a quick lookup in the registration database to find the descriptive name:

1. Open HKEY_CLASSES_ROOT key with a NULL subkey.
2. Call RegQueryValue using the key from **1** and the classname as the subkey. This value is the descriptive name.
3. Call RegCloseKey for the key from **1** and return.

First, verify that you can cleanly compile and link these new functions. Since we will not make use of these functions until we add user interface code, create a small test suite allowing you to walk through the functions in a debugger to verify their operation.

46 Implement Basic Methods

In order to initialize OLECLIENTVTBL and OLESTREAMVTBL structures, first create the methods referenced in those structures. At this point you can almost completely implement each of the three methods.

47 CallBack

A basic CallBack method needs to do very little for each notification code that were described in section 2.5.1:

Notification	Basic Action
OLE_CLOSED	SetFocus to the main application window, set a closed flag, and repaint.
OLE_SAVED	Resize your object to the new size from **OleQueryBounds** and repaint.
OLE_CHANGED	Same as OLE_SAVED.
OLE_RELEASE	Decrement the wait counter in your DOCUMENT data structure and possibly set another flag indicating the released status.
OLE_RENAMED	Reinitialize any stored information concerning a linked file.
OLE_QUERY_RETRY	Return TRUE to continue waiting for busy objects or FALSE to always terminate the operation on a busy object..
OLE_QUERY_PAINT	Return TRUE to always repaint the object completely.

Since repainting objects and setting focus to a window involves application-specific data, implement the CallBack function as part of the application; Patron's ClientCallback (PATRON.C) is little more than a template:

```
int FAR PASCAL ClientCallback(LPOBJECT pObj, OLE_NOTIFICATION wCode,
                    LPOLEOBJECT pOLEObj)
  {
  switch (wCode)
     {
     case OLE_CLOSED:    //Server closed for an embedded object.
        SetFocus(pGlob->hWnd);
        pObj->fOpen=FALSE;
        PostMessage((HWND)pObj->hData, BBM_OBJECTNOTIFY, wCode, (LONG)pObj);
        break;

     case OLE_SAVED:
     case OLE_CHANGED:
     case OLE_RENAMED:   //Server renamed a link file.
```

Reminder:

Call no OLE functions on the given object from within CallBack. OLE functions will usually return OLE_BUSY since OLECLI sends notifications to CallBack from within an asynchronous operation. Applications generally need to post a message that affects the desired operation. In addition, do not perform any action in CallBack that might display a message box or dialog or anything else that might enter another message loop.

```
        PostMessage((HWND)pObj->hData, BBM_OBJECTNOTIFY, wCode, (LONG)pObj);
        break;

     case OLE_RELEASE:
        pObj->fRelease=TRUE;
        pObj->pDoc->cWait--;
        break;

     case OLE_QUERY_RETRY:
        return FALSE;

     case OLE_QUERY_PAINT:
        return TRUE;

     default:
        break;
     }

  return FALSE;
  }
```

Note that instead of taking an LPOLECLIENT type as the first parameter, Patron immediately changes it to an LPOBJECT, which acts in place of an LPOLECLIENT. When Patron calls a function to create an object, it passes an OBJECT pointer which is passed back to this method.

In order to repaint the object, Patron sends a private message to the BlackBox window associated with the object. Also note the references to global variables in pGlob->hWnd (the main application window) and pObj->pDoc->cWait (the object wait counter).

48 StreamGet and StreamPut

The Get and Put methods contained in the OLESTREAMVTBL are very straightforward as they only need to read or write a given number of bytes, ***potentially larger than 64K***, from the application's object storage location. The Get method is called from **OleLoadFromStream** and the Put method is called from **OleSaveToStream**. In calling the OLE functions, provide a pointer to your application-specific OLESTREAM structure in which you store any information necessary to locate or store the object. This information can be as simple as a file handle (as in Patron) but is completely determined by your application.

For example, if you want to copy data fro myour application to the clipboard, and that data includes information other than OLE objects, then you must save all the object information as part of a larger structure. We'll cover this in more detail later, but such an operation will require different stream methods to save or load an object to and from a memory block, not to a file. Patron, however, does not handle such an operation, simply doing file I/O in its StreamGet and StreamPut methods:

```
DWORD FAR PASCAL StreamGet(LPSTREAM pStream, LPBYTE pb, DWORD cb)
   {
   DWORD      cbRead;

   /*
    * With a file handle, just read cb bytes of the data into pb from that file
    * handle.  This assumes that we are in the process of reading a file and
    * store objects directly in the file.
    */

   if (NULL==pStream->hFile)
      return 0L;

   cbRead=DwReadHuge(pStream->hFile, (LPVOID)pb, cb);

   //Return the number of bytes actually read.
   return cbRead;
   }




DWORD FAR PASCAL StreamPut(LPSTREAM pStream, LPBYTE pb, DWORD cb)
   {
   DWORD      cbWritten;

   /*
    * With a file handle, just write cb bytes of the data from pb to that file
    * handle.  This assumes that we are in the process of writing a file and
    * store objects directly in the file.
    */

   if (NULL==pStream->hFile)
      return 0L;

   cbWritten=DwWriteHuge(pStream->hFile, (LPVOID)pb, cb);

   //Return the number of bytes actually written.
   return cb;
   }
```

Both methods return a number of bytes read or written. If this value does not match the number of bytes requested (in the cb parameter) then OLECLI will signal an error for OleLoadFromStream or OleSaveToStream.

The reusable functions **DwReadHuge** and **DwWriteHuge** (OLELIB.C) use the Windows API calls **_lread** and **_lwrite** to read or write a data block in 32K chunks. Therefore each data block can be larger than 64K, the maximum size handled by _lread and _lwrite. Note also that Windows 3.1 (including Beta releases after build 68) contain the **_hread** and **_hwrite** functions that eliminate the need for functions like DwReadHuge..

49 Initialize the Application and VTBLs

This section describes what an OLE client must do during application (instance) initialization above its normal operations. Instance initialization takes place before the application creates its main window and enters its message processing loop.

1. Register clipboard formats for "Native," "OwnerLink," and "ObjectLink."
2. Allocate and initialize VTBLs for the OLECLIENTVTBL and OLESTREAMVTBL structures.
3. Allocate and initialize your application-specific OLESTREAM structure (such as STREAM).
4. Register the client application with **OleRegisterClientDoc**.
5. (optional) Register the application as able to accept files dropped from File Manager by calling **DragAcceptFiles**.

If any of these steps fails, except for registering "Native," registering "OwnerLink," and calling DragAcceptFiles, then terminate the application. A client application does not necessarily need to accept dropped files nor does it need the "Native" and "OwnerLink" clipboard formats for most operations. Any other error, however, is fatal.

As mentioned above, the function PDocumentAllocate in OLEDOC.C performs steps 1-3. Patron registers the client document and calls DragAcceptFiles in the WM_CREATE message case of the main window procedure in PATRON.C.

50 Register Clipboard Formats

Regardless of what clipboard I/O the client application does, it needs at least the "ObjectLink" clipboard format. It should also register formats for "Native" and "OwnerLink" as it may make use of them. Store these registered formats in variables visible to all objects (such as a DOCUMENT structure) as they are necessary to handle object data. Register the three standard formats with **RegisterClipboardFormat**:

```
pDoc->cfNative    =RegisterClipboardFormat("Native");
pDoc->cfOwnerLink =RegisterClipboardFormat("OwnerLink");
pDoc->cfObjectLink=RegisterClipboardFormat("ObjectLink");
```

51 Allocate and Initialize VTBLs and VTBL Pointers

Before calling any function in OLECLI, initialize all VTBLs with the **MakeProcInstance** call, setting each field in the OLECLIENTVTBL and OLESTREAMVTBL structures (OLESTREAMVTBL does not necessarily have to occur at this time, see below). You must allocate these structures (or use global variables) since the OLECLIENT and OLESTREAM structures (or your variants) simply contain a *pointer* to VTBLs. Note that if you have already implemented basic methods as described in the previous section, then you have already exported them in your .DEF file. Now is a great time to verify that again.

If any MakeProcInstance call fails, then fail initialization and terminate the application. An OLE client cannot function without the ability for OLECLI to call these methods. If you terminate the application on a failed MakeProcInstance, you could call FreeProcInstance for any instance thunk you created, but Windows automatically frees all thunks when the application terminates.

52 Allocate and Initialize Your OLESTREAM Structure

An OLE client only needs a single OLESTREAM (or application-specific modification) structure with an initialized VTBL pointer. During initialization, allocate your STREAM structure and initialize the LPOLESTREAMVTBL pointer within it. You can allocate and initialize OLESTREAM and OLESTREAMVTBL structures during file I/O, that is, it does not necessarily have to be done at initialization. However, allocations made at startup are generally more likely to succeed, reducing the chance that a user would not be able to save a file on some internal error condition.

53 Load and Register the Initial Document(s)

For any document you create or load on startup, call **OleRegisterClientDoc**. If the document is a real file specified on the client's command line, use that filename as the document name passed to OleRegisterClientDoc. If you simply create a new file on startup, then use '**(Untitled)**' or something else suitable as the document name. Note that OleRegisterClientDoc creates an LHCLIENTDOC handle that you must store where any other operation within the document can reference it, such as when you create objects. A DOCUMENT structure is a great place.

54 Register the Window for Drag/Drop

If your application wishes to accept files dropped from File Manager, call **DragAcceptFiles(hWnd, TRUE)** where hWnd is the main application window and TRUE enables that the application to accept dropped files (FALSE disables the capability). When the user drops files on a client's window, that client generally creates an embedded "Packager" object for each dropped file. The client may, however, do whatever it wishes with dropped files. The later section **Create, Copy, Delete, and Release Objects** discusses specific handling of dropped files and creating Packager objects. For initialization purposes, however, just call DragAcceptFiles.

55 Handle Simple Shutdown: File Close

Before compiling and testing your initialization code, provide for simple application shutdown. Note that this procedure includes steps to handle objects within a document, none of which we can even create at this point. For now, ignore steps 1-4. After completing section 4.8, you will be able to complete this procedure where you perform steps 1-6 for each *document* and steps 7-8 when you have closed all documents:

1. Set your 'release' counter to zero if you wait for all objects at once, otherwise skip this step.
2. Enumerate all objects in the document.
3. For each object, call **OleRelease** and if it returns OLE_WAIT_FOR_RELEASE either wait for the object or increment your release counter.
4. When all objects have been enumerated, wait for release on all objects if necessary.
5. Call **OleRevokeClientDoc** for all open documents, using the handles returned from **OleRegisterClientDoc**.
6. Free the DOCUMENT structure.
7. Call **DragAcceptFiles**(hWnd, FALSE) if the application previously called DragAcceptFiles(hWnd, TRUE).
8. Free the OLESTREAM structure and all VTBLs.

Call OleRevokeClientDoc before posting a WM_CLOSE message to your main window or otherwise destroying it. If OleRevokeClientDoc fails, you might still want to alert the user in which case you need a valid window handle. Call DragAcceptFiles from the WM_DESTROY case in the main window procedure, since by that time you know you are truly closing but the window is still valid. Free the data structures after you exit your message loop.

56 Create an Object Manager

Before creating any objects, your application will need some way to track those objects. OLECLI's **OleEnumObjects** function does provide some very basic and limited object management– limited mainly because the enumeration only provides pointers to the OLEOBJECTs within OLECLI. Since these pointers reference no application-supplied data, your application must then search for associated data. In addition, while you can retrieve any information about an object from the OLEOBJECT pointer, it's much more efficient to cache much of that information in something like the OBJECT structure and update it when necessary. For these reasons, Patron does not use OleEnumObjects and instead implements its own object manager to track its own structures.

In your object manager, provide four basic functions: allocate, initialize, enumerate, and free. The allocate function simply allocates memory for the structure and inserts it into whatever list you maintain. In addition, if you follow the recommendations in this document place an LPOLECLIENTVTBL field at the beginning of this structure and initialize it at this time. On return from this allocate function, you have a pointer to pass as the LPDOCUMENT parameter to various OLE object creation functions that create objects. Once an OLE create function provides an OLEOBJECT pointer, you can use your initialize function to retrieve and cache various pieces of information. Since you may later change the OLEOBJECT within a particular document object, code your initialization function so it may be called repeatedly for a single object, cleaning up any prior allocations as necessary.

Implement an enumeration function that walks the list of objects and calls a given callback function for each object. That callback function performs any action it so desires on that object, and since it is given an *application* object, not an OLEOBJECT, it has all the information it requires to carry out that action. The primary advantage using a callback function in an enumeration is that the action you perform on the list of objects is reduced to an action on a single object. The control structure to loop through those objects is hidden from the function, simplifying the flow-of-control.

Finally, implement the free function to simply take an application object, free any information inside it (such as ATOMs or GDI objects), and free the memory for that structure. This does *not* mean deleting the OLEOBJECT itself (with **OleDelete**) since creating and destroying OLE objects should happen outside the context of your object manager. By the time you call this free function, the client should have already cleaned up the OLEOBJECT.

57 Example: The OBJECT Structure and OLEOBJ.C

Patron provides an example of just such a manager for the OBJECT structures described above. Patron's DOCUMENT structure contains pointers to the first and last objects in the document. Patron allocates OBJECTs through PObjectAllocate, initializes them with PObjectInitialize, and frees them with PObjectFree. The allocate and free functions insert and remove the OBJECT from the object list in a DOCUMENT.

The OBJECT structure simply holds OLEOBJECT information available from various OLE API functions like OleGetData–allocating or freeing the structure does not affect the OLEOBJECT it refers to. To affect some operation on all the OLEOBJECTs, Patron uses its FObjectsEnumerate (OLEOBJ.C) that walks the list and calls a function for each object. An example of using FObjectsEnumerate can be seen in OLEMENU.C in MenuOLEClipboardEnable, which uses the enumeration function FEnumOLEPaste to simply see if at least one linked object exists.

58 Add OLE Menu Items

To give users a way to create objects, add the standard menu items listed in the table below if they do not already exist. The table shows each required command (unless marked as optional), the preferred menu on which it should appear, and an alternate menu on which it may appear (& precedes the mnemonic character for this item):

Command	**Menu**	**Optional**	**Menu/Comment**
[&Insert]Object...		Insert	
		Edit, as "Insert Object..." It may appear as just "Object..." on an Insert menu.	
&Copy	Edit	None–must be on the Edit menu.	
Cu&t	Edit	None	
&Paste	Edit	None	
Paste &Link	Edit	None. Paste Link is optional if you provide Paste Special.	
Paste &Special	Edit	None. Paste Special is optional if you provide Paste and Paste Link	
Con&vert to Static[1]		Edit	
		None	
Lin&ks...	Edit	None	
&Object	Edit	None. This is a placeholder for a more specific menu item that changes with the selected object. We'll cover this in detail in the section **Add the Object Verb Menu and Execute Verbs** below.	

59 Enabling and Disabling OLE Menu Items

Depending on available clipboard data and status of currently selected objects, enable or disable the menu items that OLE affects. The modifications described here only apply to OLE objects, not to other data that your application may already support. If you currently have code to enable or disable the Cut, Copy, and Paste commands, execute this OLE-specific code before it, so even if there's nothing OLE can deal with, your existing code will override the state of these menu items.

1This particular menu item is not a user interface standard, but do consider including it since a user might otherwise have no convenient method to remove the embedded status from an object.

To determine the status of the Copy, Cut, and Paste menu items, call the **OleQueryCreateFromClip** and **OleQueryCreateLinkFromClip** functions. The first parameter to these functions is a protocol string that is either "StdFileEditing" or "Static," and specific combinations of these functions and parameters indicate available objects on the clipboard:

- Embedded object available: OleQueryCreateFromClip("StdFileEditing" ...) returns OLE_OK
- Static embedded object available: OleQueryCreateFromClip("Static" ...) returns OLE_OK
- Linked object available: OleQueryCreateLinkFromClip("StdFileEditing" ...) returns OLE_OK

The table below summarizes conditions for which a you enable and disable OLE menu items:

Item	**Condition and Effect**
[Insert]Object...	Enable always.
Copy	Enable if *any* object is selected, disabled otherwise.
Cut	Enable if *any* object is selected, disabled otherwise.
Paste	Enable if an embedded or static object is available, disable otherwise.
Paste Link	(if supported) Enable if a linked object is available, disable otherwise.
Paste Special	(if supported) Enable if an embedded, static, or linked object is available, disable otherwise.
Convert to Static	Enable if *any* object is selected, disable otherwise.
Links...	Enable if any *linked* objects exist in the document, disable otherwise.
Object	Enable if *any* object is selected, disable otherwise.

Since only an application knows if an object is 'selected,' it must control enabling the Copy, Cut, and Convert to Static commands, as Patron does in MenuClipboardEnable (CLIP.C).

60 Example: MenuOLEClipboardEnable in OLEMENU.C

Patron has a reusable function called MenuOLEClipboardEnable (OLEMENU.C) that demonstrates how to call OleQueryCreate*FromClip to enable the various menu items. It also manipulates the Links... item by enumerating available objects and looking for any one linked object. The Links... item is only disabled if no linked objects exists.

For the most part, MenuOLEClipboardEnable calls **OleQueryCreateFromClip** and **OleQueryCreateLinkFromClip** to enable the various Paste items. It then uses FObjectsEnumerate (OLEOBJ.C) to search the list of objects passing each object to FEnumOLEPaste, which checks if the object is linked and stops the enumeration if it is. If FObjectsEnumerate returns without enumerating the entire list, then FEnumOLEPaste found a linked object and we enable the Links... item.

Note that MenuOLEClipboardEnable does not affect the Object item. This item is generally replaced with an object-specific string or popup menu in the MenuOLEVerbAppend, discussed later. If no object is selected, MenuOLEVerbAppend insures that a grayed "Object" item appears on the menu.

61 Create Objects and Other Object Operations

With a method to store and create objects in place, you can now begin to create objects and perform operations on them. This section contains information on a variety of object operations:

Waiting For Release Implement a function to process messages while waiting for an asynchronous operation to complete. Implement this step before creating any object.

Implement Paste Commands Implement the Paste, Paste Link, or Paste Special menu commands.

Implement the Insert Object Command Implement the Insert Object dialog to create an object of a specific class.

Handle WM_DROPFILES Process the WM_DROPFILES message to create Packager objects.

Copy and Cut Objects to the Clipboard Place existing objects back on the clipboard.

Convert Objects to Static Call OleObjectConvert to create a Static copy of an item that can no longer be activated through OLE

Close, Release, and Delete Objects Call OleClose to break an object's connection to a server, OleRelease to free an object's memory, and OleDelete to permanently delete an object from a document.

Any operation that creates an object follows four general steps, making use of your object manager:

1. Allocate your OBJECT structure and initialize the LPOLECLIENTVTBL field.

2. Create a unique name string for the new object. A good method is to append a number to an English name you use for the objects. The name must be unique within whatever storage device you use for objects.

3. Call an OleCreate* function, passing the OBJECT you allocated in **1** as the LPOLECLIENT parameter and the unique name from **2** as the object name parameter. *Note that all create functions require a client document handle from OleRegisterClientDoc.*

4. If the object creation succeeds, initialize your OBJECT structure with information from the new OLEOBJECT. If creation fails, free the allocated OBJECT and fail the operation.

   Patron applies this sequence of steps for each creation case described below. To handle step 4, Patron creates a BlackBox window that intializes the object, then sends that window a message to force it to update.

62 Wait For Release

OLE functions that create objects will often return the OLE_WAIT_FOR_RELEASE code, specifying that your application must not perform any other actions on the object until it has been released. Methods to detect the released state are discussed in **Handling Asynchronous Operations** (section 2.7) earlier in this document. So before creating any objects, implement a technique to wait for release.

63 Example: FOLEReleaseWait in OLEOBJ.C

An example of processing messages is the **FOLEReleaseWait** function in OLEOBJ.C. This function either waits for a single object by watching for the OBJECT's fRelease flag to go TRUE, or waits for all objects by watching the **cWait** counter in a DOCUMENT structure.

To make this function a reusable part of Patron, the DOCUMENT structure contains two message function pointers, **pfnMsgProc** and **pfnBackProc**, pointers to functions that take a single LPMSG parameter. FOLEReleaseWait calls the pfnMsgProc function whenever it retrieves a message to process. Calling an application-supplied function like MessageProcess in PATRON.C allows the application to do whatever it wants with a message, such as TranslateMessage/DispatchMessage or IsDialogMessage–FOLEReleaseWait assumes nothing about that process. In the same manner, when FOLEReleaseWait detects idle time, it calls the pfnBackProc function allowing the application to perform any slice of background processing. If no background function is given, or if the background function returns FALSE (indicating it has nothing to do), FOLEReleaseWait calls WaitMessage:

```
BOOL FAR PASCAL FOLEReleaseWait(BOOL fWaitForAll, LPDOCUMENT pDoc, LPOBJECT pObj)
  {
  BOOL      fRet=FALSE;
  MSG       msg;

  while (TRUE)
```

```
    {
    //Test terminating condition.
    if (fWaitForAll)
        {
        if (0==pDoc->cWait)
            break;
        }
    else
        {
        if (pObj->fRelease)
            break;
        }

    if (PeekMessage(&msg, NULL, NULL, NULL, PM_REMOVE))
        {
        if (NULL!=pDoc->pfnMsgProc)
            (*pDoc->pfnMsgProc)(&msg);
        }
    else
        {
        if (NULL==pDoc->pfnBackProc)
            WaitMessage();
        else
            {
            if (!(*pDoc->pfnBackProc)(&msg))
                WaitMessage();
            }

        fRet=TRUE;
        }
    }

return fRet;
}
```

64 Implement the Paste Commands

A Paste command is the simplest operation to create an OLE object, with three variations: Paste, Paste Link, and Paste Special. Before calling any OLE functions mentioned below call **OpenClipboard** to insure your application can access the clipboard. Immediately after pasting any information, call **CloseClipboard.** If the create function you call returns OLE_WAIT_FOR_RELEASE, process messages until that particular object is released. An example of pasting is found in Patron's **FEditPaste** function (CLIP.C).

On the **Paste** command, first attempt to paste any application-specific data created in your application. For example, if you are working in a word-processor and copy some text, you would expect to paste that text, not an embedded object containing that text. If no application data exists, call **OleCreateFromClip("StdFileEditing" ...)** to create an embedded object. If that fails, then call **OleCreateFromClip("Static" ...)** to create a static object. If that call fails as well, attempt to paste any other non-OLE information your client supports.

On the **Paste Link** command, attempt to create a linked object by calling **OleCreateLinkFromClip("StdFileEditing" ...)**. If that call fails then you simply cannot create a linked object.

On the **Paste Special** command, first display a Paste Special dialog that allows the user to choose which format to paste instead of pasting the default selection. Paste Special should only be provided if the application supports formats other than OLE:[1]

To generate the text shown in this dialog box, first retrieve the ObjectLink format from the clipboard (or OwnerLink if ObjectLink is unavailable). The first string in that data is the object class name for which you need to retrieve the descriptive name from the registration database (using the utility function you implemented earlier).

Form the string next to **Source**: by appending the second and third strings from the ObjectLink (or OwnerLink) data to the object's descriptive name. The **Data Type** listbox displays the available data formats that the client can paste. If (and only if) OwnerLink is available, then add a string formed by appending "Object" to the object's descriptive name. For other available clipboard formats that you can paste, add the appropriate string: "Picture" for CF_METAFILEPICT, "Bitmap" for CF_BITMAP, etc. Select the listbox item for your application's preferred Paste format as the default choice.

Whenever the "<classname> Object" string is selected, enable the Paste and/or Paste Link buttons if OwnerLink and/or ObjectLink data are available, respectively. Do not enable Paste Link for any other clipboard formats you support unless you perform DDE linking outside of OLE. When the user selects either Paste or Paste Link, perform that command using the selected data format as if the same command was chosen from the menu.

65 Implement the Insert Object Command

The Insert Object command is quite easy to implement and allows the user to create an object of a specified class, which the user chooses from the Insert Object dialog:

This dialog only serves to let the user select a specific object to create. Fill the listbox with the descriptive names of available classes in the registration database. Earlier in this document we implemented a function to fill such a listbox with these names, as does WFillClassList in REGISTER.C.

Once the user has selected a descriptive name, retrieve the class name for that description and call either **OleCreate** or **OleCreateInvisible**. OleCreate starts the server application for with a new object and allows the user to immediately edit that object. OleCreateInvisible eliminates the user interaction by creating a blank object, with or without starting the server (which OLECLI determines). OleCreateInvisible allows quick creation of an object and allows a client to immediately work with it instead of waiting for the server.

1Patron does not implement this dialog since Patron pastes nothing but objects.

OleCreateInvisible will not return OLE_WAIT_FOR_RELEASE, allowing the client to quickly create and work with an object (possibly calling other OLE functions on it). OleCreate, however, will *usually* return OLE_WAIT_FOR_RELEASE, in which case you must wait.

66 Example: FEditInsertObject (INSDROP.C), FOLEObjectInsert (OLEINS.C)

When Patron sees the Insert Object command it calls **FEditInsertObject** (INSDROP.C) which calls FOLEObjectInsert (OLEINS.C). FOLEObjectInsert is a reusable function to display the Insert Object dialog, allocate an OBJECT, call OleCreate on the selected class name, and wait for release. On return from FOLEObjectInsert, Patron initializes the OBJECT by creating a window in which it stores and displays objects.

67 Handle WM_DROPFILES

The most involved method to create a new object is to drop one or more files from File Manager on to the client application's document window. To accept dropped files, be sure to call **DragAcceptFiles** during application initialization as described earlier in this document. Note that Windows 3.0 does not support Drag/Drop, so if you target that version you can ignore this step completely.

When the user drops files on the document window, that window receives the WM_DROPFILES message. Process this message with the steps below:

1. Call **DragQueryFile** passing -1 for the index. DragQueryFile then returns the number of files dropped.

2. Enter a loop to process each file, starting a file index at 0 and counting up to the number of files from step 1:
   a. Call DragQueryFile with the current index to retrieve the path name.
   b. Allocate your OBJECT in which to store the OLEOBJECT.
   c. Call **OleCreateFromFile** using the class name "**Package.**" This creates an embedded object for the Packager application shipped with Windows 3.1.[1]
   d. Wait for release if OleCreateFromFile returns OLE_WAIT_FOR_RELEASE.
   e. Initialize your OBJECT. If desired, call DragQueryPoint to determine where the file(s) was dropped and show the object there.

3. Call **DragQueryFinish** to complete the operation.

If your application is based on the Multiple Document Interface (MDI) you may want to check if each file is one generated by your application and open it separately if so. How you wish to handle such files is your decision.

An implementation of the steps above is found in Patron's **FCreateFromDropFiles** (INSDROP.C).

1If you are running under Windows 3.0, you cannot receive WM_DROPFILES and therefore need not worry that Packager is unavailable.

68 Copy and Cut Objects to the Clipboard

A simple but necessary responsibility of an OLE client is copying a selected object to the clipboard (when the user chooses Edit Copy or Edit Cut) so that other clients may paste them:

1. Call **OpenClipboard**.
2. Call **OleCopyToClipboard** passing the pointer to the OLEOBJECT to copy.
3. Call **CloseClipboard**.
4. If you are cutting the object, call **OleDelete** for the OLEOBJECT (wait for release if necessary) and free your object structure.

See the FEditCut and FEditCopy functions in Patron's CLIP.C for examples. Note that to delete an object Patron uses a function called WindowDelete (PATRON.C) that also destroys the window in which it stores the object.

69 Selections that Include Objects and Other Data

As discussed before in this document, saving an object to a stream many not always mean saving to a storage device. If you have a selection that contains one or more objects or contains other data such as spreadsheet cells or text, then the objects must *not* be copied with **OleCopyToClipboard**. Instead, make the object part of the application-specific data structure (like Rich Text Format, RTF) copied to the clipboard:

1. Determine the size of the allocation:
   a. Include your application-specific formats.
   b. For each object, store an application-specific header.
   c. Call **OleQuerySize** to determine the amount of memory required for each object. This size should be stored in the object's header.

2. Allocate memory for the clipboard data.

3. Allocate and initialize a special OLESTREAM and OLESTREAMVTBL:
   a. The Get method is non-functional, but must exist in order for initialization to succeed.
   b. The Put method should receive a pointer at which to store the object's data. This requires a suitable global varaible or a special OLESTREAM structure containing the pointer.

4. Build the application-specific data structure. For each object, place a pointer in this memory where the Put method can see it and call **OleSaveToStream**. The special Put method simply copies data from OLECLI into this memory.

With this procedure, the objects become part of a larger selection, which may itself be an OLE object as the client may also act as a server.

70 Convert Objects to Static

For various reasons, a user may wish to cancel any OLE interaction for a particular object, converting it to a "static" object. OLECLI still maintains this object, but any OLE call made with this object will fail with the OLE_ERROR_OBJECT code. To convert an object:

1. If you provide an Undo feature, call **OleClone** to save a copy of the object prior to this change. Be sure to call **OleDelete** for this cloned object when you delete or replace the contents of your Undo buffer. With this clone, retrieve the object's name with **OleQueryName** and save it with the clone.
2. Call **OleObjectConvert**, passing "static" as the second parameter specifying the protocol. OleObjectConvert creates a new object
3. Call **OleDelete** on the original object and wait if necessary.
4. Assuming you still have the original object name, call **OleRename** on the new static object to change its name to that of the original.
5. Reinitialize any other object information, such as the type (which will now be OT_STATIC).

Patron handles this procedure in **FEditConvertToStatic** (CLIP.C).

71 Close, Release, and Delete Objects

The three OLE functions **OleClose**, **OleRelease**, and **OleDelete** are similar but perform different operations.

OleClose breaks the connection between any open object and the server in which that object is open. Future changes and updates to the object in the server will have no effect. If the object is linked, the client may later call **OleReconnect** to try re-establishing this connection. However, reconnection *only* works when the server containing that linked object is currently running. The client must otherwise reactivate the object to restart the server.

When closing a document, or when a client application no longer wishes to display or manipulate an object, call **OleRelease** to instruct OLECLI to free any memory allocated for that object. For example, a client may only load objects from a file that are currently displayed. When an object scrolls out of view, the client might release it; the client then loads new objects scrolling into view as necessary. OleRelease allows the application to selectively create and free objects that are still part of a client document, although the user may not manipulate those objects. The key point here is that the object's data still exists in storage somewhere, such as in the client's document file.

To permanently destroy an object, which implies deleting it from any storage device, call **OleDelete,** the last word for an object. All memory associated with that object is freed from OLECLI, and that object is assumed to no longer exist in any storage. The physical difference between OleDelete and OleRelease in OLE 1.0 is small–they both free memory–but the meaning of OleDelete is much stronger. In the future, when link tracking is part of the file system, the difference will be more pronounced; OleDelete might actually delete file records as well as memory allocations, whereas OleRelease would only free the memory.

72 Display and Print Objects; Resizing

Simply creating an object gives little feedback to the user until you display the object's image. To display or print the object on any device context, call **OleDraw**. However, be aware of a few issues when calling this function.

The rectangle in the *lprcBounds* parameter to OleDraw is *relative* to the *device context* regardless of what type of device context, screen or printer. If you paint the client area of a window that is dedicated to display an object, then that rectangle is the client rectangle of that window. If the device context is a metafile DC, then you must pass this same rectangle in the *lprcWBounds* parameter.

The *hdcFormat* parameter to OleDraw can be NULL when drawing to the screen. When printing, this device context should completely represent the target device. *hDCFormat* may contain a different mapping mode than the hDC on which the object is to be drawing, in which case OLECLI (or an object handler) may scale the image.

Whenever the client changes the target device, such as before printing, fill an OLETARGETDEVICE structure (in OLE.H) and call **OleSetTargetDevice**. Whenever the target device is the screen, pass NULL as the pointer to the OLETARGETDEVICE structure. OLESVR.DLL notifies the server application for an object and that server can then render an image of the object optimized for that target device, if it wants to implement that capability.

If OLECLI draws an object from a metafile, it will periodically send the OLE_QUERY_PAINT notification to your CallBack function, allowing you to terminate the painting or perform some other small operation. Note, however, that you cannot perform any other action on the affected object from within CallBack.

Note that you must also paint a special hatch pattern across an embedded object if the object is open. We'll cover that in the **Add the Object Verb Menu and Execute Verbs** section below, as we need to know when we activate the object before we paint a hatch pattern. That section also gives an example from Patron showing how to draw the pattern.

73 Handle Object Resizing

Whenever an object's size changes, on the client side or through OLECLI, the client application must keep the object's rectangle in sync between itself and OLECLI. Whenever CallBack receives the OLE_CHANGED or OLE_SAVED notifications, call **OleQueryBounds** to retrieve the new size of the object, resize your object to match, and repaint that object. If you use a mapping mode other than MM_HIMETRIC (in which the rectangle is given) be sure to convert the rectangle.

Whenever the object changes size in the client, call **OleSetBounds**. *The rectangle to OleSetBounds must not only be in MM_HIMETRIC but must also be coordinates on the* ***target*** *device.* This means the rectangle must be the coordinates of the object **on the screen**, not on the client area of the application.

To synchronize OLECLI with an object's rectangle, Patron calls the two functions FObjectRectSet and FObjectRectGet (OLEOBJ.C). FObjectRectSet calls **OleSetBounds** after converting the rectangle (which it expects to be in screen coordinates) into MM_HIMETRIC from another mapping mode. FObjectRectGet calls **OleQueryBounds** and converts that rectangle into another mapping mode from MM_HIMETRIC.

If you have added a simple call to OleDraw, then once you create an object you can immediately see its graphical representation within your object's boundaries. You can also resize that object to insure that the image is scaled appropriately.

74 Add the Object Verb Menu and Execute Verbs

Now that you have created and displayed objects, you can really begin to see OLE work by activating those objects with **OleActivate**. Activating really means to execute a verb that a particular object supports. In many cases the object will support a verb like "Edit," which means open the server application and allow the user to edit the object. Some objects have multiple verbs, such as the Windows 3.1 Sound Recorder that supports the verbs "Play" and "Edit." In addition, the Windows 3.1 Packager application supports the two verbs "Activate Contents" and "Edit Package." The object's server application defines these verbs and stores them in the registration database.

An OLE client provides a quick method to execute an object's primary verb: double-clicking the object or selecting it and pressing Enter[1]. For the Sound Recorder, the primary verb is "Play," so when a user activates the object it will play back the recorded sound. To allow the user to execute other verbs, an OLE client creates a special menu item listing all verbs. Only through this menu can a user execute verbs such as "Edit" for the Sound Recorder.

1The user interface guide for OLE specifies that Enter activates the primary verbs for the object when that object is selected. However, some applications like word processors may want to replace the object with a carriage return and line feed since Enter replaces any other selection in the document. In this case the Enter key does not activate the object, and mouseless users must activate the object through the verb menu.

75 Executing Verbs and Handling Notifications

To execute any verb, first check if the object is busy by calling **OleQueryReleaseStatus**. If the return value is OLE_BUSY, then either abort the operation (notifying the user) or wait for the object to be released. Otherwise, simply call **OleActivate** and wait for release if necessary. The second parameter to OleActivate is the zero-based index to the verb to execute where the primary verb is defined as zero. This index is passed directly to the server application requesting it to execute that verb.

After calling OleActivate and waiting for release on an *embedded* object, set a flag, such as the fOpen in OBJECT, to indicate that OleActivate succeeded and that the object is open. Immediately repaint the object to show a hatch pattern:

As an object is modified in the server application, your CallBack method will receive notifications. First, if the server saved a link file (for linked objects) or updates an embedded object, you will receive the OLE_SAVED notification. In that case, retrieve the new rectangle for the object with **OleQueryBounds** and repaint with **OleDraw**. If the open flag is still set, continue to paint the hatch pattern since the server is still open. When a server editing an embedded object closes, you will receive OLE_CLOSED, at which time you reset your open flag and repaint the object to remove the hatch pattern. Also update embedded objects on OLE_CHANGED.

When a server modifies an automatic[1] linked object, you will receive OLE_CHANGED in which case update the object's rectangle and repaint. A linked object will also receive OLE_SAVED when the server saves a linked file in which case update the rectangle and repaint as well. If the server saves the linked file under a new name, CallBack will receive OLE_RENAMED; in response, post a message on which you update any cached object data pertaining to the linked filename–the new filename already exists in OLECLI, so simply call **OleGetData** to retrieve the object's ObjectLink data and save the new filename. Note that a linked object will NOT receive OLE_CLOSED when the server closes.

76 Examples: FObjectPaint in OLEOBJ.C

When Patron needs to paint an object (that is, when a BlackBox window receives WM_PAINT), it calls FObjectPaint in OLEOBJ.C. This function calls **OleDraw** for any object, and paints a hatch pattern across any open embedded object:

```
BOOL FAR PASCAL FObjectPaint(HDC hDC, LPRECT pRect, LPOBJECT pObj)
  {
  OLESTATUS     os;
  HBRUSH        hBr, hBrT;

  //Draw the object
  OleDraw(pObj->pObj, hDC, pRect, NULL, NULL);
```

1The later section **Update Links and the Create the Links Dialog** defines 'automatic' with other update options. Automatic links are the only ones that receive OLE_CHANGED when modified in the server application.

3

```
//If this object is open, patch a hatch over the image.
if (OLE_OK==os && OT_EMBEDDED==pObj->dwType && TRUE==pObj->fOpen)
   {
   hBr=CreateHatchBrush(HS_BDIAGONAL, GetSysColor(COLOR_HIGHLIGHT));
   hBrT=SelectObject(hDC, hBr);

   /*
    * The 0x00A000C9L ROP code does an AND between the pattern and
    * the destination; there is no standard definition for this
    * ROP code, but it's exactly what we want to draw COLOR_HIGHLIGHT
    * lines across the object when it's open.
    */
   PatBlt(hDC, pRect->left, pRect->top,
        pRect->right-pRect->left, pRect->bottom-pRect->top, 0xA000C9L);

   SelectObject(hDC, hBrT);
   DeleteObject(hBr);
   }

return (OLE_OK==os);
}
```

77 Creating the Object Verb Menu

To provide access to all verbs, modify your Edit menu to reflect the available verbs for the currently selected object whenever your main window receives WM_INITMENU[POPUP]. In your resource file, define a single "Object" menu item, initially grayed. In your include file, define a constant (if possible) for that item's position in the Edit menu. This menu item may become a popup menu in itself so you will be required to reference it by position and not command.

The Object item in the Edit menu takes one of three forms:

- If no object is selected (or exists) the item appears as "Object" and is disabled and grayed.

- If an object supporting one verb is selected, the item appears as "<***verb***> <***name***> &Object" where <*verb*> is the primary verb, <*name*> is the descriptive name for the object class, and the "&" in "&Object" creates an underline on the "O."

- If an object supporting multiple verbs is selected, the item appears as "<***name***> &Object" as for items with one verb, but this menu item also has a submenu that lists each verb, one per line.

To handle these modifications, first define an ID value for verb commands on the menu. For example, Patron reserves the numbers 250 through 299 for verbs, where verb 0 is 250, verb 1 is 251, etc. When the main window procedure receives a WM_COMMAND message with an ID value in this range, subtract the low value of the range from the ID and you have a verb index to immediately pass to **OleActivate**. Note that you cannot depend on any verb, such as Edit, always using the same index.

One you have defined menu identifiers, create a function that you can call from the WM_INITMENU[POPUP] message case to create the menu items described above::

1. Delete the existing menu item in the position to modify with **DeleteMenu**.

2. If no object is selected or exists, call **InsertMenu** to add a disabled and grayed "Object" item and exit.

3. If an object does exist, retrieve its class name by calling **OleGetData**. If the object is embedded, request the OwnerLink format; if the object is linked, request ObjectLink. With the class name, retrieve the descriptive name from the registration database.

4. Enumerate verbs for the object class, using the enumeration function you implemented earlier (like CVerbEnum in REGISTER.C). If you find *no* verbs, use "Edit" as a default in **5**.

5. If there is one verb, create a string in the format "<***verb***> <***descriptive name***> &Object", insert it as the menu item in the position to modify with the ID value for verb 0. Exit the function.

6. If there are multiple verbs, call **CreatePopupMenu** to create the menu to hold the verb list.

7. Create a string in the format "<***descriptive name***> &Object" and insert it as the menu item in the position to modify. *Be sure to pass the menu created in **6** as the* idNewItem *parameter.*

8. For each verb, append a menu item to the menu from **6** using the string for the verb as the menu text and sequential ID values corresponding to verb 0, verb 1, and so on..

Patron implements this exact procedure in the MenuOLEVerbAppend function in OLEMENU.C. MenuOLEVerbAppend is reusable within your code provided you are using Patron's object management scheme or have modified the object manager (and this function) to suit your needs.

After completing the steps to execute verbs and add the Object Verb menu, you can run your client application and really see OLE in action. First, verify that your menu appears correct for various objects with no verbs, one verb, and multiple verbs. If you cannot locate a server with a specific number of verbs, start the Windows 3.1 Registration Database Editor (REGEDIT.EXE) with **-v** on the command line and modify some existing server's verb list. The **-v** parameter enabled RegEdit to modify the registration database instead of just viewing it.

The Sound Recorder and Packager are good applications to test with multiple verbs. If you handle the WM_DROPFILES message, then create a Packager object by dropping a file from File Manager into your document. Double-clicking the object should start the application associated with that file. Selecting the "Edit Contents" verb from your verb popup menu starts Packager and allows you to change the package itself.

78 File Menu Commands: Close, New, Open, and Save [As]

Saving and loading files is only moderately affected by OLE. In short, you only have to manage documents with **OleRegisterClientDoc** and **OleRevokeClientDoc** and call **OleSaveToStream** and **OleLoadFromStream** to save and load objects. This section lists the steps necessary to correctly manage OLE document and objects in files.

First, examine what operations in your application make the document 'dirty' in which case you would prompt the user to save changes before carrying out some operation like File New. Almost all OLE operations should set this flag: creating, destroying, updating, or resizing objects; whenever CallBack receives OLE_CHANGED, OLE_SAVED, or OLE_RENAMED; and canceling or changing links (see the next section). Note that loading objects from a file or releasing them **does not** affect the dirty flag since the file itself does not need to change.

Before modifying your file procedures to handle OLE, decide how you will store objects and how you will reference them in your application's files. Your application only needs a simple structure to reference an object, for example:

```
typedef struct
   {
   RECT       rc;           //Rectangle of object
   WORD       wID;          //ID value of object
   char       szName[40];   //Persistent name of object.
   DWORD      cbObject;     //Size of OLE object--just for demonstration
   } FILEOBJECT;
```

This is an example where the FILEOBJECT structure is stored before object data in a document file. If the objects were stored in a SQL database, for example, this structure might contain a server name, database name, and object identifier instead of just a character string name. Whatever you do use to identify the object, that identifier must be unique within the *object storage*. If you store objects in a file, then the name must be unique within the file; if you store in a database, the name must be unique within the entire database. In short, the header structure you create for your objects must enable your application to completely relocate the object when loading a document.

79 Closing a File: Prompt the User to Save Changes

Closing a file means to release each object contained in the document. The procedure below describes how to close a single document, which was already mentioned in section 4.5, **Handle Simple Shutdown**:

1. Set your 'release' counter to zero if you wait for all objects together.
2. Enumerate all objects in the document.
3. For each object, call **OleRelease** and if it returns OLE_WAIT_FOR_RELEASE, either wait for release or increment your release counter.
4. When all objects have been enumerated, wait for release on all objects if necessary.
5. Call **OleRevokeClientDoc** using the handle returned from the **OleRegisterClientDoc** call for this document.

Patron's FFileClose (FILE.C) uses the FObjectsEnumerate function (OLEOBJ.C) to handle the enumeration, and the enumeration callback (FEnumClose, FILE.C) calls **OleRelease** and increments the release counter as necessary. Patron uses FFileClose from other its other file management functions FFileNew, FFileOpen, and FFileExit. The latter function prompts the user to save changes, closes the file, and calls **PostQuitMessage** to close the application.

Note that there is no step in this procedure to prompt the user to save changes in a dirty file. The File new and File Open cases below perform this step since a case like must save the document before retrieving a new filename to open, and if that open fails you want to keep the same document in memory. Closing a file and saving changes is not necessarily the same operation.

In any case, applications normally prompt the user to save changes to which the user may reply Yes, No, or Cancel. In the Yes case, save the file, which should call OleSavedClientDoc. In the No case, call **OleRevertClientDoc** to inform OLECLI that the document has not changed since the last save, regardless of what operations we've done in the meantime. You can see this call in Patron's FCleanVerify (FILE.C). Cancel, of course, stops the file operation.

80 File New

1. MDI clients skip to **4**.
2. For **non-**MDI clients, check the dirty flag and prompt the user to save changes if necessary.
3. For **non-**MDI clients, close the existing file.
4. Create the new document and call **OleRegisterClientDoc** with the new name. If that new document is untitled, use '**(Untitled)**' as the name.

Patron handles File New in its FFileNew (FILE.C) function that performs steps 1 and 2 before calling FFileOpen (FILE.C) for step 3.

81 File Open

1. MDI clients skip to **2**. **Non-**MDI clients check the dirty flag and prompt the user to save changes if necessary.
2. Prompt the user for the new filename. Terminate the File Open operation if the user presses Cancel here. If desired, verify that file's existence before proceeding.
3. MDI clients skip to **4**. **Non-**MDI clients close the existing file.
4. Call **OleRegisterClientDoc** with the new filename. If this fails then the operation must fail.

5. Open and read the file. Whenever you encounter a structure that references an OLE object you wish to load, create a new object:
    a. Allocate your application's OBJECT structure.
    b. Call **OleLoadFromStream** to create the OLEOBJECT. This OLE function will call your StreamGet method to physically load that object into memory.
    c. Initialize the object.
    d. If the object is linked **and** the object is open (that is, there's a server application open with that linked file loaded), update the object. See the next section **Updating Links and the Links Dialog** for more information on updating links.

6. If loading the file fails, either close the document or leave it as a new document and rename it to "(Untitled)" with the **OleRenameClientDoc** function.

Patron handles steps 1-4 and step 6 in FFileOpen (FILE.C). It loads files (step 5) in FPtnFileRead (FILEIO.C).

82 File Save [As]

1. If the user chose Save As, or if the application has no filename to use with Save, prompt the user for the filename.
2. Write the file. Whenever you need to save an OLE object:
    a. Retrieve the object's unique identifier and name to save.
    b. Write an object header (like FILEOBJECT) to your application's document file that will identify the object when the file is reopened.
    c. Call **OleSaveToStream** which calls your StreamPut method.

3. If the command was Save As call **OleRenamedClientDoc** to inform OLECLI of the new name.
4. If the command was Save for an already existing filename, call **OleSavedClientDoc** to inform OLECLI of the condition of the document. If the command was Save but the user had to provide a filename, call **OleRenameClientDoc** to give the new name to OLECLI.

Patron handles steps 1, 3, and 4 in FFileSave/FFileSaveAs (FILE.C). FPtnFileWrite (FILEIO.C) handles step 2.

With document and file management function implemented, you can now save files containing linked and embedded objects then load them back. The single missing feature is updating links on loading a file and changing attributes of linked objects, which is the topic of the next section.

83 Update Links and Create the Links Dialog

Handling linked objects is a major portion of the code involved in making an application an OLE client. If you have little time in which to implement OLE support, you can elect to support only embedded objects by not offering a Paste Link or Paste Special menu command. You can also make maximal use of the code in Patron to support links and the links dialog which should save you considerable time.

Besides creating linked objects and saving them to files, an OLE client must be able to update those links when a file is loaded and be able to edit those links (as opposed to editing the contents of a *linked object*). While editing links, the user may update links, cancel links, or change links from one file to another (of the same class). The first part of this section deals with updating links after loading a document. The second part deals with the Links dialog, the most complex user interface requirement of an OLE client. But first, some definitions of the types of links an object may have.

The type of link is called an *update option* and is a value in the OLEOPT_UPDATE type defined in OLE.H, either **oleupdate_always** or **oleupdate_oncall**. Clients manipulate the update option through the OLE functions **OleGetLinkUpdateOptions** and **OleSetLinkUpdateOptions**. This document refers to these types of links as *Automatic* and *Manual*. A third type, *Unavailable*, has no representation in the OLEOPT_UPDATE type, but is used in the user interface of the Links dialog:

Name	Definition
Automatic	Also called a 'hot link' meaning that changes made to the linked object in a server immediately updates the object in the client. In addition, if the server for this object is open when the client loads the object from a file, the client knows to update that object immediately.
Manual	The object can only be updated through direct user command, either after loaded from a file or from the Links dialog.
Unavailable	The file to which this object is linked cannot be found when the object is loaded.

In the Links dialog a link may be canceled, converting the linked object to static. When this occurs, the word "Static" appears in the dialog where the other three types above normally appear.

Update Links After Loading a Document

There are two parts to updating links after loading a document:

1. Update any automatic links for which a server application containing that object is open:
   a. For each linked object in the document, check if its update option is **oleupdate_always**. If not, skip this object and count it for possible update in part 2

below.

b. Call **OleQueryOpen** to determine if the object is currently open in a server. If not, skip this object and count it for possible update in part 2 below.

c. Call **OleUpdate** to update the linked object and wait for release if necessary. If there is an error, count this object for possible update in part 2. Otherwise, mark this object as updated so we can skip it in part 2.

2. If the document contains any manual links or any automatic links that were not updated in part 1:

a. Display a message box asking if the user wants to update links contained in this document:

b. If the user chooses No, then exit.

c. If the user chooses Yes, then enumerate **all** linked objects:

1. If the object was marked as updated, clear the mark and skip the object.
2. Call **OleUpdate** for the object and wait for release if necessary. You must wait for release now to determine if OleUpdate works.
3. If an error occurs from OleUpdate, mark the object's link as unavailable. This will affect the display in the Links dialog if the user chooses to use it.

d. If there were any unavailable links, display the dialog box below allowing the user to invoke the Links... dialog. If the user presses Links... in this dialog, close this dialog and invoke the Links dialog described in the next section.

84 Create a Links Dialog

The Links dialog allows users to update, cancel, and change links:

When creating this dialog, use the LBS_EXTENDEDSEL and LBS_USETABSTOPS styles for the listbox. You may also want to include LBS_SORT, but sorting is not required.

The Links dialog the most complex user interface requirement of an OLE client, from displaying the strings in the listbox to handling the commands. This section describes eight different non-trivial steps to implement this dialog:

Implement Utility Functions	Implement three functions to make the Links dialog easier: one to build a listbox string, one to replace the listbox string, and one to enumerate all selected or non-selected items in the list.
Enable Buttons According to List Selections	Create a function to enable and disable five button controls in the dialog by analyzing the listbox selections.
Initialize List Tabstops and Items	Set the appropriate tab stops in the list, enumerate all linked objects, and build a string for each object to add to the listbox.
Prepare for Undo on Cancel	Use **OleClone** to make copies of all linked objects before changing them in the Links dialog. When Cancel is pressed, use these clones to revert any modified object to its previous state.
Change Update Options	Change update options when the Automatic and Manual radiobuttons are pressed.
Update Links	Update any selected links and locate any links that belong to the same source file. If any are found, let the user choose whether or not to update those links as well.
Cancel Links	Change any selected linked objects to static objects.
Change Links	Retrieve a new link filename from the user and change any selected links to that new file. If any other unselected links to the old file exist, ask if the user wants to change those as well. This step is much like updating links.

85 Implement Utility Functions

This section describes three functions that isolate frequently used code in the Links dialog.

1. **Create a listbox string given an object**. Each string in the listbox has the format:

   <***descriptive name***>\t<***linked document***>\t<***object identifier***>\t<***update option***>

where <*descriptive name*> is the readable English version of the classname, <*linked document*> and <*object identifier*> are the second and third strings in the object's ObjectLink data, and <*update option*> is "Automatic," "Manual," "Unavailable," or "Static."[1] Note that **\t** represents a tab in the string.

To build the string, first retrieve the object's ObjectLink data either from cached information or by calling **OleGetData**. Next, retrieve the descriptive name from the registration database for the class. Building the string is then just a matter of concatenating each string followed by a tab character into a single string. However, you'll notice in the dialog figure that each string is limited to the width of its column. This is critically important to insure that all columns line up correctly.

Therefore, limit each string to the number of characters that will fit into a tab width, one quarter of the listbox width, specified in some number of pixels (or dialog units). This requires repeated calls to GetTextExtent–using the hDC of the *listbox* to account for the font–until you find the number of characters for which the horizontal extent of the string is less than the tab width. Before adding the string to the listbox string, truncate it at this number of characters.

For an example, see Patron's **CchLinkStringCreate** in OLELINK2.C. It uses a function **CchLimitText** that inserts a null-terminator into a string to truncate it to fit into a tab width.

2. **Replace a listbox string**. Whenever you change a link or a link option, you need to update the listbox to show the change. For each link you modify you will recreate the listbox string (using the function above) and replace the existing string in the list, preserving the item's data and selection state. The message sequence below to accomplish this, assuming you have the item's index:

a. Send LB_GETITEMDATA and store the result in a temporary variable.

b. Send LB_GETSEL and store the result in another temporary variable.
c. Send LB_DELETESTRING to remove the item from the list.
d. Send LB_INSERTSTRING to insert the updated item in the list at the same point.
e. Send LB_SETITEMDATA using the result from step 1.
f. If the result from step **b** is non-zero, send LB_SETSEL with TRUE in wParam and the index in the low-word of lParam.

See the Patron's **ListStringChange** in OLELINK2.C for an example.

1Since the **OleGetLinkUpdateOption** will only return oleupdate_always or oleupdate_oncall (for automatic and manual links), you must define other codes to use in your object structure for unavailable and static links, such as the numbers -1 and -2.

3. **Enumerate selected or non-selected listbox items**. The operations you carry out in the Links dialog affect either all selected links in the listbox or all unselected links. An enumeration function finds selected, unselected, or all items in the listbox depending on a WORD that describes what type to look for. This function passes each enumerated list item to a callback function that carries out a specific action on each single object. Some operations in the callback may return OLE_WAIT_FOR_RELEASE for which the callback increments a release counter. Therefore the enumeration function also waits for all objects to be released after the enumeration is complete, if necessary:

a. Set the release counter to zero.

b. Send the LB_GETCOUNT message to the listbox to retrieve the number of items in the list.

c. Loop through the items (a **for** loop from 0 to the number of items works well):

1. Get the selection state for this item.
2. If the selection state does not match the desired selection state, skip this item and continue the loop.
3. Retrieve your application's OBJECT structure for this item.[1]
4. If the item is static, skip it and continue the loop.
5. Call the enumeration function, passing the listbox handle, the item index and at least a pointer to the object.[2]
6. If the enumeration function returns FALSE, end the enumeration, otherwise continue the loop.

d. If the release counter is non-zero, wait for release until your CallBack decrements it to zero. Always check the counter in case an enumeration function did not already wait for an object.

This enumeration function will be essential to simplify implementation of the other Links dialog functions. An example is found in Patron's **FLinksEnumerate** in OLELINK1.C. The application-defined data passed to the enumeration function is very useful as it can contain a variable that only has meaning to a particular type of enumeration as we shall soon see.

86 Enable Buttons According to List Selections

Five buttons in the Links dialog must either be enabled or disabled depending on the combination of the listbox selections:

Button	**Enable When...**
Automatic	Any items besides canceled links are selected.
Manual	Any items besides canceled links are selected.
Update Now	Any automatic or manual links and *no* unavailable links are selected and

1As described in the next section on initializing the list, the listbox item data is a great place to store a pointer to the OBJECT structure with the LB_SETITEMDATA message.

2Patron's implementation allows extra data to be passed into the enumeration callback which enables passing of information like a link filename or a new update option.

Cancel Link Any automatic or manual links are selected. Do not count unavailable or static links.
Change Link All the selections are linked to the same file.

In addition, if all the selected links are automatic, then check the "Automatic" button using **CheckDlgButton**; if all the selected links are manual, then check the "Manual" button. If the selections contain different update options, then uncheck both buttons.

To determine what items are selected, loop through *all* the listbox items, and for any selected item, increment a counter for its particular update option–automatic, manual, static, and unavailable. In addition, when you find the first selected link, save its link filename. For any subsequent selected link, compare the first item's filename to the current item's filename. If they do not match, then set a flag indicating the link files are different between the selections. Use this flag to enable or disable the Change Link button.

Patron's **EnableLinkButtons** function in OLELINK2.C retrieves the object for each item in the list and compares filenames stored in the aLink ATOM. When it encounters the first selection, that object's atom is kept in a variable to compare to all other selected links. Comparing ATOMs is much quicker and convenient that extracting and comparing strings. If any mismatch was found in comparing link files, then the Change Link buttin is disabled.

87 Initialize List Tabstops and Items

In the WM_INITDIALOG message case for your dialog, initialize tabstops in the listbox, fill the listbox with link items, and select the first item with the LB_SETSEL message. Finally, call your function to enable and disable buttons that you implemented in the previous section.

Using the LB_SETTABSTOPS message, set tabstops in the listbox at every quarter of the listbox width. Note that to use tabstops the listbox must be created with the LBS_USETABSTOPS style. In addition, the LB_SETTABSTOPS message requires you to provide tab positions in *dialog units*, not device units. Therefore, get the pixel width of the *entire* listbox (from **GetClientRect**), multiply the width of the entire listbox by four, and divide by the low-word of the value from **GetDialogBaseUnits**:

```
GetClientRect(hList, &rc);

//Convert pixel width to dialog width for LB_SETTABSTOPS
dwBase=GetDialogBaseUnits();
cx=((rc.right-rc.left) * 4)/LOWORD(dwBase);
```

This value (in cx above) is the width of the listbox in dialog units. To send the LB_SETTABSTOPS message, fill an array of three WORDs with one-fourth the width (the first tab), one half the width (the second tab), and three-fourths the width (the third tab stop). Pass a pointer to this array as the lParam of the message.

To fill the listbox with strings, enumerate all the *objects* (not links) in your application. For each **linked** object, create a string for it (using the utility function you created), add the string to the list with the LB_ADDSTRING message, and send the LB_SETITEMDATA message to save your OBJECT structure pointer for this object with the listbox item. By storing this pointer you will save yourself from ever having to find the object associated with a listbox string. Whenever you reference a listbox item, simply send the LB_GETITEMDATA message to retrieve the object pointer.

To select the first string in the list, call **SendMessage(hList, LB_SETSEL, 1, 0)**. Then finish your initialization by correctly enabling or disabling the other dialog box buttons using the function created in the previous section.

For an example, see the WM_INITDIALOG case in Patron's **LinksProc** (OLELINK1.C) and **FEnumLinksInit** (OLELINK1.C).

88 Prepare for Undo on Cancel

The Links dialog has both OK and Cancel buttons; OK means accept changes but Cancel means discard them. To undo changes when Cancel is pressed, you must make clones of each linked object using **OleClone**:

1. During WM_INITDIALOG, enumerate all linked objects. For each object:
    a. Call **OleClone** for each object. *This requires a new unique object name.*
    b. Store the LPOLEOBJECT value from step **a** in a list. This list must persist until the Links dialog is closed.
    c. Call **OleQueryOpen** to determine if the object was open or not. Store this condition in a flag associated with the cloned object.

2. When any object in the Links dialog is modified, mark that object as changed.

3. If the user presses Cancel in the Links dialog:
    a. Prompt the user to confirm discarding changes. There is no standard message for this prompt, but if the user chooses No, then stop this operation now.
    b. Enumerate all linked objects in the listbox.
    c. If the object was not modified, skip it. Otherwise continue.
    d. Retrieve the original object's name and call **OleDelete** for that object, waiting for release as necessary.
    e. Call **OleRename** to rename the cloned object from the name used in step **1a** above to the name from step **c** in this procedure.
    f. Install the clone OLEOBJECT and reinitialize any cached information.
    g. If the original object was open, call **OleReconnect** to attempt to reestablish the connection.
    h. Resize the object to its original size and repaint.

4. If the user presses OK in the Links dialog:
    a. Enumerate all linked objects in the listbox.
    b. Call **OleDelete** on the clone OLEOBJECT pointer associated with the object, waiting for release as necessary.

Patron makes clones of each object when initializing the Links dialog listbox (see FEnumLinksInit in OLELINK1.C). It stores the clone object and the open state of the object in the OBJECT structure for the link. In addition, a 'dirty' flag is kept in each object and set to TRUE whenever the Links dialog changes it. This information is then easily found when either OK or Cancel is pressed. Both cases use FLinksEnumerate, with **FEnumLinksUndo** performing step 3 above and **FEnumLinksAccept** performing step 4.

89 Change Update Options

When the user selects either Automatic or Manual, use your function to enumerate selected links and change each link's update options to **oleupdate_always** (for automatic) or **oleupdate_oncall** (for manual):

1. Set the release counter to zero, if waiting for all objects together.
2. Enumerate selected listbox items.
3. If the new update option already matches the object's update option, skip this item.
4. If the new option differs, then store that new option in your OBJECT structure and call **OleSetLinkUpdateOptions** for this object.
5. If OleSetLinkUpdateOptions returns OLE_WAIT_FOR_RELEASE, increment the release counter or wait for release immediately, as desired.
6. If there are no errors, change the string in the listbox for this item to reflect the new option.

Note that if your listbox string builder calls any OLE functions, then you must immediately wait for release after OleSetLinkUpdateOptions. Since Patron caches all the object's link information, it uses no OLE functions in the listbox string builder. See **FEnumOptionChange** (OLELINK1.C) for an example of this procedure.

90 Update Links

Updating links by themselves is quite straightforward–just call **OleUpdate** for each selected link. However you must detect when the update fails and change that link to unavailable if so. In addition, once you have updated selected links, search the *unselected* items to find any references to the same files in the selected items. For each file that is referenced by both selected and unselected items, display the message box below and update unselected items as necessary (SOURCE.XLS is generic for the source (server) document and CLIENT.DOC is generic for the client application's document):

Updating links follows the following process:

1. Enumerate selected links.

2. Call **OleUpdate** for the object associated with each enumerated item and wait for release if necessary. Check for errors using **OleQueryReleaseError**.

3. If there is an error, mark the link as unavailable and change its string in the listbox. If there is no error and the link was previously unavailable, change it to "Automatic" and update its string.

4. Check for a flag set in step **5d** below. If this is set, continue the enumeration with the next link. The first enumerated link will not have this flag set.

5. Otherwise, from within this first enumeration:
   a. Enumerate *unselected* links and find any item that is linked to the same file as the link updated in **2** above.
   b. If no matches are found, then continue the first enumeration in **1.**
   c. If the link files match, display the message box above.
   d. Enumerate all selected links that match the link file in **2**. For any that match, set a flag in the object that we used in **4** above. This flag prevents us from asking the user to update matching unselected links again when we encounter another selected link for this file.
   e. If the user responded **Yes** to the message box, repeat steps 1-3 for all *unselected* items.

6. Enumerate all selected links and reset the flag set in step **5d** above.

Patron implements this updating procedure through FLinksEnumerate and the callback **FEnumLinkUpdate** (OLELINK1.C) that performs a variety of different actions depending on the part of the process that we're in. In fact, FEnumLinkUpdate itself calls FLinksEnumerate using FEnumLinkUpdate again to perform the nested enumeration. A flag determines what action to perform in this callback.

91 Cancel Links

Canceling a link really means to create a new object that converts the original object into a 'static' copy of the original data that can no longer be edited through OLE means. The process is straightforward:

1. Enumerate selected links.
2. For each selected link, retrieve the object for that item and it's object name.
3. Call **OleObjectConvert** where the first parameter is the object and the second parameter is "Static."

4. If OleObjectConvert succeeds (and it does NOT return OLE_WAIT_FOR_RELEASE), call **OleDelete** for the original object and wait for release if necessary.
5. Replace the existing OLEOBJECT in your object structure with the new static object. If you cache such information, store the object type as it as OT_STATIC and the link update option as "static" (that is, whatever value you define for static).
6. Update the listbox string, changing the update option string (like "Automatic") to "Static."

92 Change Links

Changing links is a similar process to updating them. The process is slightly simpler since the Change Link button is disabled if the selected links reference different files. Therefore we do not need to nest link enumerations:

1. Enumerate selected links, stopping the enumeration on the first item found.

2. Extract the link filename for this object. Using this filename, invoke the common File Open dialog (using the title "Change Link"):
    a. Extract the file extension from the link filename.
    b. Find the class name associated with this extension in the registration database. You should already have a utility function to perform this lookup.
    c. Retrieve the descriptive name for the class name from the registration database. Again, you should have a function for this.
    d. Create a filter description string for the common dialog in the form "<***descriptive name***> (<***ext***>)" to appear in the file types list of the common dialog.
    e. Call **GetOpenFileName** using "Change Link" for the dialog title, passing the file extension (without the period) as the default extension, the string from step **d** as the filter, and the full path to the link file as the default path.

3. If the user cancels the Change Link dialog, then terminate this operation.

4. Enumerate all selected links:
    a. Create new ObjectLink data for the item, retaining the class name and object name but changing the document name to the new file.
    b. Call **OleSetData** using the new ObjectLink data.
    c. Call **OleUpdate** to update the link and wait if necessary. If an error occurs, skip this item.
    d. If no error occurred and the object was previously unavailable, mark it as "Automatic" and update the listbox string for this item.

5. Enumerate unselected links:
    a. If any unselected link matches the old link filename of the objects just changed, display a message box asking the user to change unselected links to the same file:

b. If the user responds No to this message, then the Change Link operation is finished. Otherwise repeat **4** for *unselected* links.

You made it! Now that Links are fully implemented, you can test all operations of your OLE client, including updating links on File Open and modifying various link attributes in the Links dialog. Note that to have a link marked unavailable, first create a linked object to an existing file, save and close the document holding that object, delete the source file, then attempt to reload the document and update all links. If **OleUpdate** fails at this stage, you should see the dialog informing you of unavailable links and allowing you to invoke the Links dialog which should have those links marked Unavailable.

93 Additional OLE Client Functions

Once you can compile and test updating links and the Links dialog, you now have a completely functional OLE client application! You may wish to expand the functionality of your client by taking advantage of the additional functions that OLECLI.DLL offers. This section briefly describes other OLE Client functions in OLECLI.DLL that have not yet been mentioned. These fall several categories: Object Creation, Object Handling, Server Information, and a few oddballs.

94 Object Creation

OleCopyFromLink Creates an embedded object from a linked object.

OleCreateFromTemplate Creates an embedded object using the contents of a file as the original data for the object. This is different from **OleCreateFromFile** in that the object's server is opened to allow initial editing.

OleCreateLinkFromFile Creates a linked object using a given filename. This function is similar to **OleCreateFromFile** that creates an embedded object.

95 Object Handling

OleEnumFormats Enumerates the data types supported by an object, allowing a client application to determine if it could retrieve a particular format (such as CF_DIB) from the object.

OleEqual Compares two objects returning OLE_OK if they are equal.

OleQueryOutOfDate Checks if an object is out of date and should be updated. In the current OLE libraries, this function always returns OLE_OK. When it does become implemented, it will provide a much easier method to determine if an object needs updating.

OleQueryProtocol Determines if the object supports a given protocol, either "StdFileEditing" or "StdExecute." This function can be used to determine if an object is static, as calling it with "StdFileEditing" will return NULL.

OleRequestData Similar to **OleGetData**, but requires that the object is already connected to the server. This simply provides a faster means of retrieving data when the server is already open.

96 Server-Related Functions

OleLockServer Instructs OLECLI to keep the server for an object in memory. The client application can use this to optimize operations for objects from the same server as OLECLI does not load and close the server for each object.

OleUnlockServer Instructs OLECLI that the server for an object can be unloaded. This function must be called for any server locked with **OleLockServer**.

OleExecute Sends a DDE Execute string to an object's server. Before sending any commands, you must insure that the server supports the StdExecute protocol by calling **OleQueryProtocol** for the object.

OleSetColorScheme Provides an object's server with colors used in the client application. This function takes a LOGPALETTE structure but is not related in any way to a *device* color palette. This function is only useful to applications to provide some set of color with which a user can draw text and graphics.

97 Miscellaneous

OleIsDcMeta Determines if a given hDC is a metafile DC.

OleQueryClientVersion Returns the version number of OLECLI.DLL.

OleQueryReleaseMethod Returns the type of operation for which the object was released. This is very useful when waiting for all objects to be released at one time. When one object is released it can call this function to know what it was just doing and act

accordingly, especially when combined with **OleQueryReleaseError**.

Appendix A: Definitions

Term	*Definition*
Class Name	A single word (or acronym) that identifies an object class. This is not expected to be used in a user interface.
Client	(or Client Application) An application that creates and edits documents that contain linked and/or embedded objects from one or more server applications. Clients only store objects; servers actually edit them.
CSV	Comma Separated Value string, where each item in the string is delimited with a comma.
Descriptive Name	A readable class name used in a user interface to describe an object.
Destination	Synonym for Client.
Document	A container for one or more objects, generally the same as a physical file.
Embed	To create and store an object completely within a client document. An embedded object contains a presentation format (bitmap or metafile), and OwnerLink data structure identifying the server, and the Native data provided by the server. Editing an embedded object starts the server and sends the Native data back to that server.
File	A physical file on a disk, usually containing a document.

Term	*Definition*
Key	Unit of storage in the registration database. There is one root key from which subkeys are attached. A key is physically a character string where each subkey is separated with a backslash (\).
Link	To create an object in a client document whose native data is stored in another file maintained by the server for that object. The client document contains only a presentation format and an ObjectLink data structure identifying the linked file.
Method	A callback function contained in the server application that the OLESVR library calls to perform specific actions such as creating documents or retrieving object data.
Native	An internal data structure manipulated by a server application that contains enough information to completely reconstruct an object. The server application is the only application that understands this data.
Object	A black box of information with a presentation that represents that data. A server application understands the internal data of an object it created, but a client application treats it like a number of bytes with a pretty picture on the box.
Object Link	Data structure that identifies the class, document filename, and the object name that is the source for a linked object.

Term	*Definition*
OLECLI.DLL	The OLE Client Library, OLECLI.DLL, that contains OLE API used by client applications.
OLESVR.DLL	The OLE Server Library, OLESVR.DLL, that contains OLE API used by server applications.
Owner Link	Data structure that identifies the class, document, and object names that describes the owner of an embedded object.
Registration Database	The system database that holds names of applications that support the OLE protocol, the full pathnames to those applications, the objects they can edit, what verbs those objects support, and whether an object handler exists for that class.
Release	A Released object, document, or server is one that no longer has any connections to any client documents. Servers, documents, and objects all have Release methods that inform the item that no client is connected to it.
Revoke	To close off communication from a client application from a server, document, or object. When one of these items is revoked, the item will eventually become released. A client may also revoke communication to a server, document, or object.
Server	(or Server Application) An application that creates and edits objects for storage in a client application's document.
SHELL.DLL	A dynamic link library that contains functions to manipulate the registration database.
Source	Synonym for Server.

Term	*Definition*
Subkey	A refinement of a key in the registration database. A key can have any number of subkeys and subkeys can have their own subkeys.
Thunk	A procedure-instance address created through a call to MakeProcInstance. Also called an instance thunk.

Appendix B: Guide to OLE Code in Patron

This appendix contains information about using Patron's OLE code in your own application. The sections are organized loosely along the structure of this entire implementation guide. This appendix is essentially documentation for an OLE client helper library made up from the files OLE*.C, OCLIENT.H, OCLIENT.RC, OCLIENT.DLG, REGISTER.C, and REGISTER.H. With little or no modification, this library should be immediately usable in your application.

The sections below will describe the interface for each component; specific implementation details are not provided. Much of the information is taken from header comments on the functions themselves. All functions are declared as FAR PASCAL.

Section	Contents and Files
Registration Database Helpers	Five functions in REGISTER.C to simplify information retrieval from the registration database. REGISTER.H contains function prototypes.
Resources	Dialog box templates and OLE-related strings in OCLIENT.DLG and OCLIENT.RC, with definitions in OCLIENT.H.
Utility Functions	Miscellaneous functions in OLELIB.C such as manipulating filenames, reading and writing >64K data to a file, and mapping mode conversion.
VTBL Constructors/Destructors	Four functions to alloc OLESTREAMVTBL structures in OLEVTBL.C.
The DOCUMENT Struct & Strings	Constructor, destructo DOCUMENT structure in OLEDOC.C. Includes loading OLE-specific strings into a globally visible array referenced through the PSZOLE macro.
STREAM and Default Methods	Constructor and destru as well as default Get and Put methods for the OLESTREAMVTBL in OLESTREA.C
OBJECT Manager	Constructor, initializer, enumeration, and destructor functions for the OBJECT structure in OLEOBJ.C
OBJECT Manipulations	Waiting for release, changing or retrieving object bounds, changing or retrieving object data, and drawing an object in OLEOBJ.C
Insert Object Dialog	A standard Insert Object dialog implementation to display the dialog, fill the listbox with class descriptive names, and create an OLEOBJECT within an OBJECT structure. Dialog is defined in OCLIENT.DLG and invoked through OLEINS.C.
Menu Manipulations	Two functions in OLEMENU.C to 1) enable or disable the Copy, Cut, Paste, and Paste Link menu

	items on a menu, and 2) to create an object verb menu at a given position.
Updating Links	Functions in OLELOAD.C to update open automatic links on loading a file, update all links in a document at the users request, and to track unavailable links and possibly invoke the Links dialog. This last function makes use of the Links dialog below.
Links Dialog	A standard Links dialog implementation to display the dialog, initialize the links listbox, enable and disable buttons depending on the selected links, and handling each of the dialog buttons. OLELINK1.C contains the function to display a dialog; OLELINK2.C contains utility functions to create and replace listbox strings.

Functions to manipulate the registration database, VTBL structures, the DOCUMENT structure, and the STREAM structure are independent of Patron's OBJECT manager implementation. However, to use the user interface functions (listed after the OBJECT Manager above), you do need to use this object manager. Of course, you can modify the code as necessary for your application, which should save you considerable time in just writing code to handle the user interface. No matter what you decide to do with this code, Patron's model will at least serve to illustrate how an object manager can simplify your implementation.

To create an OLE client using these functions, allocate a single DOCUMENT structure for each document with PDocumentAllocate. When closing the document call PDocumentFree. Before calling any OLE function to create an object, create an OBJECT in which to store it with PObjectAllocate, specifying the DOCUMENT structure for the document that contains that object. Once the OLE object is created, call PObjectInitialize. When that object is destroyed or the document is closed, call PObjectFree. With a document and object created, you can use any of the user interface helper functions.

B.1 Registration Database Helpers: REGISTER.C, REGISTER.H

WFillClassList

WORD WFillClassList(HWND *hList*)

Enumerates available OLE object classes from the registration database and fills a listbox with

those names. WFillClassList removes any prior contents of the listbox.

Parameter	Type	Description
hList	HWND	Listbox to fill.

Return Type	Description
WORD	Number of strings added to the listbox if successful, -1 otherwise.

WClassFromDescription

WORD WClassFromDescription(LPSTR *psz*, LPSTR *pszClass*, WORD *cb*)

Retrieves the OLE class name from the registration database for the given descriptive name.

Parameter	Type	Description
psz	LPSTR	Pointer to the descriptive name to find.
pszClass	LPSTR	Pointer to a buffer in which to store the class name.
cb	WORD	Maximum length of *pszClass*.

Return Type	Description
WORD	Number of characters copied to *pszClass* if successful, 0 otherwise.

WClassFromExtension

WORD WClassFromExtension(LPSTR *pszExt*, LPSTR *pszClass*, WORD *cb*)

Retrieves the OLE class name in the registration database for the given file extension.

Parameter	Type	Description
pszExt	LPSTR	Pointer to the extension to reference.
pszClass	LPSTR	Pointer to a buffer in which to store the class name.
cb	WORD	Maximum length of *pszClass*.

Return Type	Description
WORD	Number of characters copied to *pszClass* if successful, 0 otherwise.

CVerbEnum

WORD CVerbEnum(LPSTR *pszClass*, LPSTR *pszzVerbs*, WORD *cbMax*)

Builds a double-null terminated list of strings where each string is one of the supported verbs for

a for a particular class.

Parameter	Type	Description
pszClass	LPSTR	Pointer to the object classname.
pszzVerbs	LPSTR	Pointer to a buffer in which to store the verb list.
cbMax	WORD	Maximum length of *pszzVerbs*.

Return Type	Description
WORD	Number of verbs stored in *pszzVerbs* if successful, 0 otherwise.

WDescriptionFromClass

WORD WDescriptionFromClass(LPSTR *pszClass*, LPSTR *pszDescription*, WORD *cb*)

Looks up the descriptive name in the registration database for the given class name.

Parameter	Type	Description
pszClass	LPSTR	Pointer to the class name.
pszDescription	LPSTR	Pointer to a buffer in which to store the descriptive name.
cb	WORD	Maximum length of *pszDescription*.

Return Type	Description
WORD	Number of characters copied to *pszDescription* if successful, 0 otherwise.

B.2 Resources: OCLIENT.RC

OLE Strings in OCLIENT.RC

Identifier in OCLIENT.H	String
IDS_NATIVE	"Native"
IDS_OWNERLINK	"OwnerLink"
IDS_OBJECTLINK	"ObjectLink"
IDS_STDFILEEDITING	"StdFileEditing"
IDS_AUTOMATIC	"Automatic"
IDS_MANUAL	"Manual"
IDS_UNAVAILABLE	"Unavailable"
IDS_STATIC	"Static"
IDS_PACKAGE	"Package"

IDS_UPDATELINKS0	"The file contains links to other\ndocuments.\n\nUpdate links now?"
IDS_UPDATELINKS1	"The selected links to %s have been\nupdated. %s contains "
IDS_UPDATELINKS2	"additional links\nto %s.\n\nUpdate additional links?"
IDS_CHANGELINK	"Change Link"
IDS_CHANGELINKS1	"The selected links to %s have been\nchanged. %s contains "
IDS_CHANGELINKS2	"additional links\nto %s.\n\nChange additional links?"
IDS_INSERTTITLE	"Insert Object"
IDS_NOINSERT	"Could not create a new object or start object server."
IDS_VERBEDIT	"Edit"
IDS_OBJECT	"Object"
IDS_OBJECTBUSY	"The action cannot be completed because the object is busy."
IDS_OLEERROR	"OLE Error"
IDS_OLEERRORMSG	"Method: %d, Error: %d"
IDS_UPDATEMSG	"Updating Links"

Dialog Box Templates in OCLIENT.RC

Identifier in OCLIENT.H	**Description**
IDD_INSERTOBJECT	Insert Object dialog containing one listbox, an **OK** button, and a **Cancel** button.
IDD_LINKS	Links dialog containing a listbox for link strings, two radiobuttons **Automatic** and **Manual**, and push buttons for **OK**, **Cancel**, **Update Now**, **Cancel Link**, and **Change Link**.
IDD_BADLINKS	Message box informing the user that some links were unavailable and giving the user the option of invoking the links dialog. This message box must be a dialog to have the **Links...** button as well as an **OK**.

Control identifiers for these dialogs are also defined in OCLIENT.H.

B.3 Utility Functions: OLELIB.C

FFileDialog

BOOL FFileDialog(HWND *hWnd*, HANDLE *hInst*, LPSTR *pszExt*, LPSTR *pszFilter*, LPSTR *pszFile*, BOOL *fOpen*)

Invokes the COMMDLG.DLL GetOpenFileName dialog and retrieves a filename for saving or opening.

Parameter	**Type**	**Description**
hWnd	HWND	Window of the owning application.
hInst	HANDLE	Application instance.
pszExt	LPSTR	Pointer to the default extension.
pszFilter	LPSTR	Pointer to the filter description.
pszFile	LPSTR	Pointer to a buffer to receive the entered filename. Must be at least CCHPATHMAX (defined in OCLIENT.H) long.
pszCaption	LPSTR	Pointer to the title to use in the dialog box.
fOpen	BOOL	Flag indicating if we want file open or save.

Return Type	**Description**
BOOL	TRUE if the function retrieved a filename, FALSE if the user pressed CANCEL.

PszFileFromPath

LPSTR PszFileFromPath(LPSTR *pszPath*)

Returns a pointer within an existing pathname string to the last file of that string. Used to extract the filename from a path for use in window titles and message boxes. Note: This function does character comparisons to '\' and so may require more work to localize.

Parameter	**Type**	**Description**
pszPath	LPSTR	Pointer to the full pathname.

Return Type	**Description**
LPSTR	Pointer to a filename within *pszPath* if successful, NULL otherwise.

PszExtensionFromFile

LPSTR PszExtensionFromFile(LPSTR *pszFile*)

Returns a pointer within an existing filename string to the extension of that file. Used to extract

the extension from a file for use in File dialogs and so forth. The file is scanned backwards looking for a '.' or '\'. If no '.' is found before a '\' then this function returns a pointer to the null terminator. Note: This function does character comparisons to '.' and '\' and so may require more work to localize.

Parameter	Type	Description
pszFile	LPSTR	Pointer to a filename.

Return Type	Description
LPSTR	Pointer to an extension (starting with the .) within pszFile if successful, NULL otherwise.

PszWhiteSpaceScan

LPSTR PszWhiteSpaceScan(LPSTR *psz*, BOOL *fSkip*)

Skips characters in a string until a whitespace or non-whitespace character is seen. Whitespace is defined as \n, \r, \t, or ' '. NOTE: This function is not extremely well suited to localization. It assumes that an existing application seeking to become an OLE client probably already has such a string function available.

Parameter	Type	Description
psz	LPSTR	Pointer to string to manipulate.
fSkip	BOOL	TRUE if we want to skip whitespace. FALSE if we want to skip anything but whitespace.

Return Type	Description
LPSTR	Pointer to first character in the string that either non-whitespace (fSkip=TRUE) or whitespace (fSkip=FALSE), which may be the null terminator.

DwReadHuge

DWORD DwReadHuge(WORD *hFile*, LPVOID *pv*, DWORD *dwRead*)

Reads a data block that is potentially larger than 64K from a file. The data is read in 32K chunks.

Parameter	Type	Description
hFile	WORD	File handle from which to read.
pv	LPVOID	Pointer to the data buffer.
dwRead	DWORD	Number of bytes to read.

Return Type	Description
DWORD	Number of bytes read if successful, 0 otherwise.

DwWriteHuge

DWORD DwWriteHuge(WORD *hFile*, LPVOID *pv*, DWORD *dwWrite*)

Writes a data block that is potentially larger than 64K to a file. The data is written in 32K chunks.

Parameter	Type	Description
hFile	WORD	File handle to write to.
pv	LPVOID	Pointer to the data buffer.
dwRead	DWORD	Number of bytes to write.

Return Type	Description
DWORD	Number of bytes written if successful, 0 otherwise.

RectConvertMappings

void RectConvertMappings(LPRECT *pRect*, WORD *mmSrc*, WORD *mmDst*)

Converts the contents of a rectangle from one logical mapping into another. This function is useful since you have to convert logical->device then device->logical to do this transformation.

Parameter	Type	Description
pRect	LPRECT	Containing the source rectangle to convert.
mmSrc	WORD	Mapping mode of the source rectangle.
mmDst	WORD	Mapping mode of the destination rectangle.

Return Type	Description
None	

B.4 VTBL Constructors/Destructors: OLEVTBL.C

PVtblClientAllocate

LPOLECLIENTVTBL PVtblClientAllocate(LPBOOL *pfSuccess*, HANDLE *hInst*, FARPROC *pfn*)

Allocates and initializes an OLECLIENTVTBL structure.

Parameter	Type	Description
pfSuccess	LPBOOL	Pointer to a flag to store the outcome of the function. If this function returns non-NULL, but **pfSuccess*==FALSE, the caller must call the destructor function.
hInst	HANDLE	Application instance.
pfn	FARPROC	Pointer to the single client method to initialize. We call MakeProcInstance for this function.

Return Type	Description
LPOLECLIENTVTBL	Pointer to the allocated VTBL if successful, NULL if the allocation failed or a parameter is invalid.

PVtblClientFree

LPOLECLIENTVTBL PVtblClientFree(LPOLECLIENTVTBL *pvt*)

Frees all procedure instances in the LPOLECLIENTVTBL and frees the structure as well.

Parameter	Type	Description
pvt	LPOLECLIENTVTBL	Pointer to the structure to free.

Return Type	Description
LPOLECLIENTVTBL	NULL if the function succeeds, *pvt* otherwise

PVtblStreamAllocate

LPOLESTREAMVTBL PVtblStreamAllocate(LPBOOL *pfSuccess*, HANDLE *hInst*, FARPROC *pfnGet*, FARPROC *pfnPut*)

Allocates and initializes an OLESTREAMVTBL structure.

Parameter	Type	Description
pfSuccess	LPBOOL	Pointer to a flag to store the outcome of the function. If this function returns non-NULL, but **pfSuccess*==FALSE, the caller must call the destructor function.
hInst	HANDLE	Application instance.
pfnGet	FARPROC	Pointer to the stream's Get method.
pfnPut	FARPROC	Pointer to the stream's Put method.

Return Type	Description

LPOLESTREAMVTBL	Pointer to the allocated VTBL if successful, NULL if the allocation failed or a parameter is invalid.

PVtblStreamFree

LPOLESTREAMVTBL VtblStreamFree(LPOLESTREAMVTBL *pvt*)

Frees all procedure instances in the OLESTREAMVTBL and frees the structure.

Parameter	**Type**	**Description**
pvt	LPOLESTREAMVTBL	Pointer to the structure to free.

Return Type	**Description**
LPOLESTREAMVTBL	NULL if the function succeeds, *pvt* otherwise

B.5 The DOCUMENT Structure and PSZOLE: OLEDOC.C

PDocumentAllocate

LPDOCUMENT PDocumentAllocate(LPBOOL *pfSuccess*, HANDLE *hInst*, LPSTR pszCaption, FARPROC *pfnCallBack*, FARPROC *pfnGet*, FARPROC *pfnPut*)

Constructor method for the DOCUMENT data type used from application initialization. Allocates a DOCUMENT and sets the defaults in its fields:

- Initialize OLECLIENTVTBL and OLESTREAMVTBL
- Allocate and initialize an OLESTREAM structure (see OLESTREA.C)
- Register OLE clipboard formats
- Allocate scratch data and set pointers within it.
- Load OLE strings if this is the first DOCUMENT structure allocated. Strings are loaded once for all DOCUMENT allocations.

Parameter	**Type**	**Description**
pfSuccess	LPBOOL	Pointer to a flag to store the outcome of the function. If this function returns non-NULL, but **pfSuccess*==FALSE, the caller must call the destructor function.
hInst	HANDLE	Application instance.
pszCaption	LPSTR	Pointer to the application name.
pfnCallBack	FARPROC	Pointer to the single client method to initialize. We pass this function to PVtblClientAllocate.
pfnGet	FARPROC	Pointer to the document's Stream Get method, passed to PStreamAllocate.
pfnPut	FARPROC	Pointer to the document's Stream Put method, passed to

PStreamAllocate.

Return Type	Description
LPDOCUMENT	Pointer to the allocated DOCUMENT if successful, NULL if the allocation failed or a parameter is invalid.

PDocumentFree

LPDOCUMENT PDocumentFree(LPDOCUMENT *pDoc*)

Frees all data in the DOCUMENT and frees the structure. This includes freeing the OLE strings if this is the last DOCUMENT structure to be freed and the STREAM structure contained in this document.

Parameter	Type	Description
pDoc	LPDOCUMENT	Pointer to the structure to free.

Return Type	Description
LPDOCUMENT	NULL if the function succeeds, *pDoc* otherwise.

FDocumentFileSet

BOOL FDocumentFileSet(LPDOCUMENT *pDoc*, LPSTR *pszFile*)

3 Provides the document with an associated filename for use in OLE-related UI. An application should call this function whenever it loads a new file or when that filename changes.

Parameter	Type	Description
pDoc	LPDOCUMENT	Document in which to store the filename.
6 *pszFile*	LPSTR	Pointer to the filename of the document.

Return Type	Description
BOOL	TRUE if the function succeeds, FALSE otherwise.

PDocumentMsgProcSet

void PDocumentMsgProcSet(LPDOCUMENT *pDoc*, LPFNMSGPROC *pfn*)

Informs the DOCUMENT structure about a function in the main application that translates and dispatches messages. This prevents the DOCUMENT from having to carry an accelerator handle or window handle and allows the application to perform other actions we cannot predict (like IsDialogMessage). The *pfn* function should be in the form:

BOOL FAR PASCAL *MessageProc*(LPMSG pMsg)

The return value of *MessageProc* indicates whether or not the function processed the message.

Parameter	**Type**	**Description**
pDoc	LPDOCUMENT	Pointer to the structure concerned.
pfn	LPFNMSGPROC	Pointer to the message processing function.

Return Type	**Description**
None	

PDocumentBackgroundProcSet

void PDocumentBackgroundProcSet(LPDOCUMENT *pDoc*, LPFNMSGPROC *pfn*)

Informs the DOCUMENT structure about a function in the main application that performs background operations when there are no messages to process. This is necessary to provide a standard release waiting message loop such that the loop can call the background process function when it detects idle time. The *pfn* function should be in the form:

BOOL FAR PASCAL *BackgroundProc*(LPMSG pMsg)

The return value of *BackgroundProc* indicates whether or not the function performed any action. If the function calls WaitMessage, it should return TRUE, otherwise Patron will call WaitMessage again.

Parameter	**Type**	**Description**
pDoc	LPDOCUMENT	Pointer to the structure concerned.
pfn	LPFNMSGPROC	Pointer to the background processing function.

Return Type	Description
None	

PSZOLE (macro)

char NEAR * PSZOLE(WORD *i*)

The PSZOLE macro returns the a near string pointer stored in the **rgpszOLE** array at index *i*. This macro handles any subtraction on the index if the first OLE string has an identifier greater than zero.

Note that you *must* call PDocumentAllocate to load the strings from OCLIENT.RC and initialize **rgpszOLE** in order to use this macro.

B.6 STREAM and Default Methods: OLESTREA.C

PStreamAllocate

LPSTREAM PStreamAllocate(LPBOOL *pfSuccess*, HANDLE *hInst*, FARPROC *pfnGet*, FARPROC *pfnPut*)

Allocates and initializes a STREAM structure, using the function PVtblStreamAllocate to initialize its VTBL. If *pfnGet* is NULL then we use the StreamGet method from OLESTREA.C. Likewise, if *pfnPut* is NULL we use StreamPut from OLESTREA.C.

Parameter	Type	Description
pfSuccess	LPBOOL	Pointer to flag to store the outcome of the function. If this function returns non-NULL, but **pfSuccess*==FALSE, the caller must call the destructor function.
hInst	HANDLE	Application instance.
pfnGet	FARPROC	Pointer to the stream's Get method.
pfnPut	FARPROC	Pointer to the stream's Put method.

Return Type	Description
LPSTREAM	Pointer to the allocated STREAM if successful, NULL if the allocation failed or a parameter is invalid.

PStreamFree

LPSTREAM PStreamFree(LPSTREAM *pStream*)

Frees all data in the STREAM and frees the structure.

Parameter	Type	Description
pStream	LPSTREAM	Pointer to the structure to free.

Return Type	Description
LPSTREAM	NULL if the function succeeds, *pStream* otherwise.

StreamGet

DWORD StreamGet(LPSTREAM *pStream*, LPBYTE *pb*, DWORD *cb*)

Instructs the client to read a specified number of bytes (possibly over 64K) from wherever it stores objects.

Parameter	Type	Description
pStream	LPSTREAM	Pointer to the stream structure holding the file handle.
pb	LPBYTE	Pointer to which we read data. We have no idea what data we'll read since it's defined by OLECLI.
cb	DWORD	Number of bytes to read from the file into *pb*.

Return Type	Description
DWORD	Number of bytes actually read. If this value does not match *pb* then OLECLI assumes an error.

StreamPut

DWORD StreamPut(LPSTREAM *pStream*, LPBYTE *pb*, DWORD *cb*)

Instructs the client to write a specified number of bytes (possibly over 64K) to storage.

Parameter	Type	Description
pStream	LPSTREAM	Pointer to the stream structure holding the file handle.
pb	LPBYTE	Pointer to the data to write. We have no idea what data sits here.
cb	DWORD	Number of bytes to write from *pb*.

Return Type	Description
DWORD	Number of bytes actually written. If this value does not match *pb* then OLECLI assumes an error.

B.7 OBJECT Manager: OLEOBJ.C

PObjectAllocate

LPOBJECT PObjectAllocate(LPBOOL *pfSuccess*, LPDOCUMENT *pDoc*)

Allocates an OBJECT structure. This function only creates the structure and inserts it into the owner's list.

Parameter	**Type**	**Description**
pfSuccess	LPBOOL	Pointer to a flag to store the outcome of this function. If this function returns non-NULL, but **pfSuccess*==FALSE, the caller must call the destructor function.
pDoc	LPDOCUMENT	Pointer to the owner of this object, which contains the linked list into which we insert ourselves.

Return Type	**Description**
LPOBJECT	Pointer to the allocated OBJECT if successful, NULL if the allocation failed or a parameter is invalid.

PObjectInitialize

LPOBJECT PObjectInitialize(LPOBJECT *pObj*, LPDOCUMENT *pDoc*, LPSTR *pszDoc*)

Initializes an OBJECT structure and assumes that the pObj field in the structure already contains a valid OLEOBJECT pointer. Any existing initialized data is freed and overwritten.

Parameter	**Type**	**Description**
pObj	LPOBJECT	Pointer to the structure to initialize.
pDoc	LPDOCUMENT	Pointer to the owner of this object.
pszDoc	LPSTR	Pointer to the name of the client document name containing this object.

Return Type	**Description**
LPOBJECT	*pObj* if successful, NULL otherwise.

PObjectFree

LPOBJECT PObjectFree(LPDOCUMENT *pDoc*, LPOBJECT *pObj*)

Frees all data in the OBJECT and frees the structure.

Parameter	Type	Description
pDoc	LPDOCUMENT	Pointer to owner of the object.
pObj	LPOBJECT	Pointer to the structure to free.

Return Type	Description
LPOBJECT	NULL if the function succeeds, *pObj* otherwise

FObjectsEnumerate

BOOL FObjectsEnumerate(LPDOCUMENT *pDoc*, LPFNOBJECTENUM *pfn*, DWORD *dw*)

Enumerates all allocated OBJECT structures, passing them to a specified enumeration function given in pfn which should appear as:

BOOL *EnumFunc*(LPDOCUMENT *pDoc*, LPOBJECT *pObj*, DWORD *dwData*)

The return value of *EnumFunc* is TRUE to continue the enumeration, FALSE otherwise. This function provides a different enumeration method than OleEnumObjects since it contains the loop instead of embedding OleEnumObjects inside your own loop. The enumeration provided by this function is more consistent with other Windows Enum* functions.

Parameter	Type	Description
pDoc	LPDOCUMENT	Pointer to the owner of the objects.
pfn	LPFNOBJECTENUM	Pointer to the enumeration callback function.
dw	DWORD	Extra data to pass to the callback function.

Return Type	Description
BOOL	TRUE if ALL objects were enumerated, FALSE if the callback returned FALSE.

B.8 OBJECT Manipulations: OLEOBJ.C

FObjectPaint

BOOL FObjectPaint(HDC *hDC*, LPRECT *pRect*, LPOBJECT *pObj*)

Calls OleDraw for a specified object to draw the object ON THE SCREEN. If the object is

embedded and open, it also paints the object with an HS_BDIAGONAL hatch brush.

Parameter	Type	Description
hDC	HDC	Display context on which to paint.
pRect	LPRECT	Rectangle of the area to paint.
pObj	LPOBJECT	Pointer to the object to paint.

Return Type	Description
BOOL	TRUE if successful, FALSE otherwise.

FObjectRectSet

BOOL FObjectRectSet(LPDOCUMENT *pDoc*, LPOBJECT *pObj*, LPRECT *pRect*, WORD *mm*)

Provides the object with a screen-relative rectangle. The object itself assumes that the rectangle contains coordinates in units defined by the mapping mode in mm. The rectangle given here is sent to the OleSetBounds function for this object after which we call OleUpdate.

Parameter	Type	Description
pDoc	LPDOCUMENT	Pointer to the DOCUMENT containing OLE information. Necessary if we have to wait for release.
pObj	LPOBJECT	Pointer to the object concerned
pRect	LPRECT	Pointer to the rectangle of the object.
mm	WORD	Mapping mode of *pRect*.

Return Type	Description
BOOL	TRUE the set succeeds, FALSE if OleSetBounds fails or if an invalid pointer is passed.

FObjectRectGet

BOOL FObjectRectGet(LPOBJECT *pObj*, LPRECT *pRect*, WORD *mm*)

Retrieves the object's rectangle stored in the specified units.

Parameter	Type	Description
pObj	LPOBJECT	Pointer to the object concerned.
pRect	LPRECT	Rectangle in which to store the coordinates.
mm	WORD	Mapping mode in which to retrieve coordinates.

Return Type	Description

BOOL	TRUE if the function succeeds, FALSE if an invalid pointer is passed.

FObjectDataGet

BOOL FObjectDataGet(LPOBJECT *pObj*, WORD *cf*, LPSTR *psz*)

Calls OleGetData for a particular object to retrieve data in the specified format, either ObjectLink or OwnerLink. The contents are copied into a buffer pointed to by *psz* on which no length assumptions are made.

Parameter	**Type**	**Description**
pObj	LPOBJECT	Pointer to OBJECT concerned.
cf	WORD	Specifies the ObjectLink or OwnerLink format to retrieve.
psz	LPSTR	Pointer to buffer to store the string.

Return Type	**Description**
BOOL	TRUE if the function succeeds, FALSE otherwise.

FObjectDataSet

BOOL FObjectDataSet(LPDOCUMENT *pDoc*, LPOBJECT *pObj*, WORD *cf*, LPSTR *pszDoc*)

Calls OleSetData for a particular object to update data in the in the specified format, either ObjectLink or OwnerLink. The only changeable field in the data is the document (2nd string) which is provided in *pszDoc*. PszObjectDataSet passes the new data to OleSetData and stores it in *psz*; no assumptions are made about the length of *psz*.

Parameter	**Type**	**Description**
pDoc	LPDOCUMENT	containing OLE information.
pObj	LPOBJECT	containing the OLEOBJECT to receive the new data.
cf	WORD	specifying the ObjectLink or OwnerLink format.
pszDoc	LPSTR	to the new link file

Return Type	**Description**
BOOL	TRUE if the function succeeds, FALSE otherwise.

FOLEReleaseWait

BOOL FOLEReleaseWait(BOOL *fWaitForAll*, LPDOCUMENT *pDoc*, LPOBJECT *pObj*)

Enters a Peek/Translate/Dispatch message loop to process all messages to the application until one or all objects are released. This message processing is necessary because OLECLI.DLL and

OLESVR.DLL communicate with asynchronous DDE messages. The *fWaitForAll* flag indicates how to wait:

1. If TRUE==*fWaitForAll*, then wait for all objects, that is, until *pDoc*->cWait is zero.
2. If FALSE==*fWaitForAll*, then wait until *pObj*->fRelease is set to TRUE. This assumes that the caller previously set this flag to FALSE.

Parameter	**Type**	**Description**
fWaitForAll	BOOL	Flag indicating if we wait for one object by *pObj*->fRelease or for the *pDoc*->cWait counter to fall to zero.
pDoc	LPDOCUMENT	Pointer to a DOCUMENT containing the message processing function, the background processing function, and the cWait counter.
pObj	LPOBJECT	Pointer to the object to wait for if *fWaitForAll* is FALSE. Ignored if *fWaitForAll* is TRUE.

Return Type	**Description**
BOOL	TRUE if we yielded, FALSE otherwise. For what it's worth.

OsError

OLESTATUS OsError(OLESTATUS *os*, LPDOCUMENT *pDoc*, LPOBJECT *pObj*, BOOL *fWait*)

Provides a centralized error handler for OLE function calls, depending on the value in os:

OLE_OK	Return OLE_OK
OLE_BUSY	Display a message and return OLE_BUSY.
Any OLE error	Return that error.
OLE_WAIT_FOR_RELEASE	Call FOLEReleaseWait on the object if fWait is TRUE, then call OleQueryReleaseError for the return value. Otherwise increment *pDoc*->cWait.

Parameter	**Type**	**Description**
os	OLESTATUS	Error value to process.
pDoc	LPDOCUMENT	Pointer to a DOCUMENT containing OLE information.
pObj	LPOBJECT	Pointer to the OBJECT affected.
fWait	BOOL	Flag indicating if we are to wait for release on the object or increment *pDoc*->cWait

Return Type	**Description**

OLESTATUS	New error code depending on the original os.

B.9 Insert Object Dialog: OLEINS.C

PObjectInsertDialog

LPOBJECT PObjectInsertDialog(HWND *hWnd*, HANDLE *hInst*, LPDOCUMENT *pDoc*, LPSTR *pszObject*)

Displays the Insert Object dialog and creates an object of the selected name. The caller must provide the name of the object to create. NOTE: Uses *pDoc*->pszData1.

Parameter	**Type**	**Description**
hWnd	HWND	Window to use as the parent of the dialog box.
hInst	HANDLE	Application instance.
pDoc	LPDOCUMENT	Pointer to the owner of objects.
pszObject	LPSTR	Pointer to the name for a new object.

Return Type	**Description**
LPOBJECT	Pointer to a new OBJECT if the function succeeds, NULL otherwise or if the user pressed Cancel.

B.10 Menu Manipulations: OLEMENU.C

MenuOLEClipboardEnable

void MenuOLEClipboardEnable(HMENU *hMenu*, LPDOCUMENT *pDoc*, LPWORD *pWID*)

Enabled or disables Edit menu commands for Paste, Paste Link, Paste Special, and Links, depending on whether or not certain clipboard formats exists or if there are linked objects. This function should be called for the WM_INITPOPUPMENU message in the main window procedure.

Parameter	**Type**	**Description**
hMenu	HMENU	Handle to the Edit menu to modify.
pDoc	LPDOCUMENT	Pointer to the DOCUMENT owner of all objects.
pWID	LPWORD	Pointer to an array of four WORDs, where the caller stores menu ID values for Paste, Paste Link, Paste Special, and

Links. If any of these are zero that menu item is ignored.

Return Type	Description
None	

MenuOLEVerbAppend

void MenuOLEVerbAppend(HMENU *hMenu*, WORD *iVerbMenu*, WORD *wIDMin*, LPDOCUMENT *pDoc*, LPOBJECT *pObj*)

Appends the appropriate menu item(s) on the given menu depending on the given object. This function finds the verbs for the given object and if there's one, it creates a single menu item with *<verb> <object>*. If there's multiple verbs, it creates a cascading menu with:

<object> **>** *<verb 0>*
<verb 1>
...

Call this function when processing the WM_INITPOPUPMENU message. This function requires HVerbEnum function in register.c.

Parameter	Type	Description
hMenu	HMENU	Handle of the Edit menu to modify.
iVerbMenu	WORD	Position of the item to modify.
wIDMin	WORD	First menu ID value for a verb menu item.
pDoc	LPDOCUMENT	Pointer to DOCUMENT owning *pObj*. This must contain clipboard formats.
pObj	LPOLEOBJECT	Pointer to the object concerned.

Return Type	Description
None	

B.11 Updating Links: OLELOAD.C

FObjectAutoLinkUpdate

BOOL FObjectAutoLinkUpdate(LPDOCUMENT *pDoc*, LPOBJECT *pObj*)

Checks if the object link is automatic and if so, update it if the server is open, waiting as necessary.

Parameter	Type	Description
pDoc	LPDOCUMENT	Pointer to the object owner.
pObj	LPOBJECT	Pointer to the object in question.

Return Type	Description
BOOL	TRUE if the object was updated, FALSE if the object is a manual link or the server was not open.

FOLELinksUpdate

BOOL FOLELinksUpdate(HWND *hWnd*, HANDLE *hInst*, LPDOCUMENT *pDoc*)

Checks if the recently loaded file had objects requiring update. If so, then prompt the user to update links, and if they answer Yes, then call OleUpdate for all linked objects. If any of the links cannot be updated, then we prompt the user and invoke the Links dialog if they so choose.

Parameter	Type	Description
hWnd	HWND	Window handle to use as the parent of dialogs.
hInst	HANDLE	Application instance.
pDoc	LPDOCUMENT	Pointer to DOCUMENT holding list of object.

Return Type	Description
BOOL	TRUE if all objects could be updated or if the user used the Links dialog. FALSE if there are still non-updated links.

B.12 Links Dialog: OLELINK1.C and OLELINK2.C

FOLELinksEdit

BOOL FOLELinksEdit(HWND *hWnd*, HANDLE *hInst*, LPDOCUMENT *pDoc*)

Handles the Links dialog and the Update Now, Cancel Link, and Change Link commands.

Parameter	Type	Description
hWnd	HWND	Window to use as the parent of the dialog.
hInst	HANDLE	Application instance.
pDoc	LPDOCUMENT	

	Pointer to the owner of all objects.

Return Type	**Description**
BOOL	FALSE if we could not create the dialog or if the user pressed Cancel. TRUE otherwise.

FLinksEnumerate

BOOL FLinksEnumerate(HWND *hList*, LPDOCUMENT *pDoc*, LPFNLINKENUM *pfn*, WORD *wSelection*, DWORD *dwData*)

Enumerates links of a specific selection state in a listbox. Each item is passed to the *pfn* callback function which should appear as:

BOOL *EnumFunc*(HLIST *hList*, WORD *i*, LPDOCUMENT *pDoc*, LPOBJECT *pObj*, DWORD *dwData*)

Where *hList* is the listbox handle and *i* is the index of the current item. These are necessary if the callback needs to send any messages to the listbox to retrieve more item information.

The return value of *EnumFunc* is TRUE to continue the enumeration, FALSE otherwise. This function provides a different enumeration method than OleEnumObjects since it contains the loop instead of embedding OleEnumObjects inside your own loop. The enumeration provided by this function is more consistent with other Windows Enum* functions. If *EnumFinc* sees OLE_WAIT_FOR_RELEASE it should wait for that object immediately.

Parameter	**Type**	**Description**
hList	HWND	Handle to the listbox containing the items to enumerate.
pDoc	LPDOCUMENT	Pointer to DOCUMENT containing OLE information.
pfn	LPFNOBJECTENUM	Pointer to the callback function to which each item is passed.
wSelection	WORD	Specifies the type of items to enumerate: ENUMLINK_SELECTED, ENUMLINK_UNSELECTED, or ENUMLINK_ALL.
dwData	DWORD	Extra data to pass to the callback function.

Return Type	**Description**
BOOL	TRUE if ALL objects were enumerated, FALSE if the callback returned FALSE.

CchLinkStringCreate

WORD CchLinkStringCreate(LPSTR *psz*, LPDOCUMENT *pDoc*, LPOBJECT *pObj*)

Creates a Links... listbox string from an linked object. The ObjectLink data is stored in three ATOMs in this object which we created in PObjectAllocate (OLEOBJ.C). We also append the type of link (Automatic, Manual, or Unavailable) to the string. Each string is visually limited to a tab space in the listbox.

Parameter	**Type**	**Description**
psz	LPSTR	Pointer to the buffer to receive the string.
pDoc	LPDOCUMENT	Pointer to DOCUMENT containing OLE information.
pObj	LPOBJECT	Pointer to the object whose string we're building. We use the ATOMs from this object to create the string.

Return Type	**Description**
WORD	Number of characters in the string.

CchTextLimit

WORD CchTextLimit(LPSTR *psz*, HDC *hDC*, WORD *cx*)

Truncates a string at a point where it will fit into *cx* pixels using the display context in *hDC*.

Parameter	**Type**	**Description**
psz	LPSTR	Pointer to the string to limit.
hDC	HDC	Device context into which this string will be draw, assumed to have the correct font selected.
cx	WORD	Number of pixels to which we limit text.

Return Type	**Description**
WORD	Number of characters in *psz*.

LinkStringChange

void LinkStringChange(HWND *hList*, WORD *i*, LPSTR *psz*)

Changes a string in a listbox item to a new string, preserving the positioning, selection, and item data of the old string.

Parameter	**Type**	**Description**

hList	HWND	Window handle of the listbox.
i	WORD	Index of the item to change.
psz	LPSTR	Pointer to the new string.

Return Type	**Description**
None	