Microsoft

µ §

, Software Design Engineer
Systems Developer Relations

Version 1.01

The information and code provided in this document is subject to change without notice and does not represent a commitment on the part of Microsoft Corporation or the author.

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 Server Implementation Guide

Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

Table of Contents

1 Introduction

This *Object Linking and Embedding (OLE) Server Implementation Guide* is intended to help you, as an applications programmer, add OLE server capabilities to a new or existing application. This guide provides OLE technical background information, suggestions to prepare an application for becoming an OLE server, 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 server, 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 easily debug a small part of code. The nature of OLE makes debugging difficult as it is, and trying to test all the OLE-specific code at once can take considerable time to determine where the bug really is, let alone how to fix it. In any case, you are always free to take your own approach.

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 implement OLE. "Integrating OLE" into an application is simply not necessary, because OLE only affects a few specific parts.

With this guide you should be able to add basic OLE support to a suitable server application within a week, give or take some days depending on the complexity of your application. The sample server demonstrates the steps described in this guide and also contains many pieces that you can immediately transplant to your application.

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.
Bitmaps	All OLE servers are required to produce a graphic representation of their data, and a bitmap is one format that a server provides for this purpose (see Metafile below). Does not necessitate knowledge of DIBs (Device Independent Bitmaps).
Callback functions	MakeProcInstance required for initializing function tables.

Clipboard I/O OLE Servers that can run stand-alone must provide several non-OLE and OLE-specific data formats on the clipboard.

DDE Since OLE 1.x works off the DDE protocol, a knowledge of DDE may help you understand how the OLE protocol works.

Dynamic memory allocation The application and possibly OLE-specific structures.

Dynamic Menu Changes Part of an OLE menu items for embedded objects. The function used is **ModifyMenu**.

File I/O Any server that will support linking (as opposed to embedding) must save information in a file.

Metafiles A required graphic representation of the server's data.

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*	M&T Publishing 1991
Wilton, Richard	*Windows Developer's Workshop*	Microsoft Press 1991
Yao, Paul and Norton, Peter	*Windows 3.0 Power Programming Techniques*	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:

```
    if (OLE_WAIT_FOR_RELEASE==os)
{
pOLE->oleDoc.fRelease=FALSE;
FOLEReleaseWait(&pOLE->oleDoc.fRelease, pOLE->oleSvr.lh);
}
```

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

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

4 Sample Server: SCHMOO

Accompanying this implementation guide is a sample OLE server called Schmoo, a Germanic twist on Lil' Abner's pets that also reminds us of college physics classes dealing with magnetic flux and electric fields. It's also a name that other servers will not use, so it does not interfere with your existing system.

Schmoo is a single-document application, capable of running multiple instances, that allows you to edit, save, and load a Schmoo Figure: a bunch of dots connected with lines. While the Schmoo Figure is somewhat uninteresting[1], Schmoo demonstrates the OLE server protocol and serves as a model for discussions in this document. In fact, the in-line code in this guide was taken directly from Schmoo.

5 Source Code Structure

Schmoo has a fair number of source files in which I have attempted to isolate the OLE-specific code leaving only a few parts of mostly non-OLE files touched by OLE. This isolation demonstrates how OLE can simply lay on top of an existing application requiring a few new files and minor additions to a few key sections of the existing application code.

There are two types of source files, those with OLE as a prefix and those without; the OLE files contain, as you might guess, OLE specific code. Some files, like OLEINIT.C and OLECLIP.C, correspond to non-OLE files like INIT.C and CLIP.C. The source files and contents are listed below. Also note that within these modules are various functions that are reusable in your application:

File	Contents
Non-OLE	
commdlg.c	Support for File/Open, File/Save As common dialogs.
fileio.c	File I./O functions.
misc.c	Functions without any other home.
polyline.c	Polyline window procedure and functions to generate bitmaps or metafiles of the image in the window. ALL data editing and creation is handled here.
polyline.h	Prototypes and definitions for polyline.c
schmoo.c	Main window procedure.
schmoo.h	Prototypes and definitions for non-OLE functions.

1Maybe a little interesting. The samples SHIP.MOO and TETRA.MOO show a few figures that you can make with Schmoo.

Files Dealing with OLE

clip.c	Pre-OLE clipboard support, calls OLECLIP.C functions for OLE clipboard formats.
exit.c	Pre-OLE cleanup function, calls OLEEXIT.C for OLE cleanup.
file.c	File menu commands: New, Open, Save, Save As, and Exit. Functions in this module are the most affected by OLE.
init.c	Pre-OLE initialization function, calls OLEINST.C and OLEINIT.C to perform OLE-specific installation and initialization responsibilities.

OLE-Specific Counterparts

oleclip.c	Functions to place OLE data on the clipboard.
oleexit.c	OLE cleanup, including calls to OLEVTBL.C to de-initialize the method tables.
olefile.c	Helper functions for File menu commands.
oleinst.c	Functions to install the server in the registration database.
oleinit.c	OLE initialization, including calls to OLEVTBL.C to initialize method tables.
olemisc.c	OLE functions without a home, such as modifying menus, notifying the client, and providing a message processing loop.
olevtbl.c	Functions to initialize and free method tables the for server, documents, and objects.

OLE-Specific Code

oledoc.c	OLESERVERDOCVTBL methods.
oleobj.c	OLEOBJECTVTBL methods.
olesvr.c	OLESERVERVTBL methods.
oleglobl.c	Declarations of OLE-specific global data.
oleglobl.h	External declarations of OLE-specific global data and prototypes of OLE-specific functions.
oleinst.h	Prototypes for functions in OLEINST.C that deal with the registration database. These are not in OLEGLOBL.H so that the installation code may be easily removed to a separate setup program.

6 Isolation of Global Data and Strings

Since the OLE protocol can quickly have you using global variables, I have isolated those dealing with OLE and those not dealing with OLE into two separate structures. The only true global variables are pointers to these two structures: pGlob and pOLE, and a pointer to a string array rgpsz (see below). pGlob points to application globals unrelated to OLE except the fOLE flag that indicates that the application is running as a linked or embedded server instead of stand-alone. pOLE points to OLE-specific globals.

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->fOLE or pOLE->fEmbed. This clearly

marks the use of a global as opposed to a local variable.

rgpsz is an array allocated in HLoadAppStrings (INIT.C) where the application's string table is read into local memory so that all strings can be referenced by a pointer using an index into rgpsz, such as rgpsz[IDS_CLASSSCHMOO]. Use of rgpsz will crop up in various places around this document.

7 OLE Technical Background

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

- OLESVR.DLL, OLE.H
- SHELL.DLL, SHELLAPI.H
- Library Redistribution and Installation
- OLE Communication Routes
- OLE Data Structures and Application-Specific Variations
- Mini-Servers vs. Full Servers
- Single-Document vs. Multiple-Document Servers
- Clipboard Formats and Conventions

8 OLESVR.DLL and OLESVR.LIB

The OLESVR library contains ten functions that enable a server to register itself with OLESVR and to inform OLESVR of changes to objects and documents. OLESVR.DLL contains the functions, and OLESVR.LIB is the import library to which you link your server application.

Function	When Used	Description
OleRegisterServer	Initialization	Registers the specified server with the library. Information registered included the class name, instance, and whether the server supports single or multiple instances.
OleRegisterServerDoc	Document creation	Registers a document with the server library.
OleRenameServerDoc	Document Save As	Renames the specified document.
OleRevertServerDoc	Document reload	Reverts a document to a previously saved state, without closing the document.
OleSavedServerDoc	Document save	Informs library that a document has been saved. Calling this function is equivalent to sending the OLE_SAVED notification.
OleRevokeServer	Server shutdown	Revokes access to the specified server, closing any documents and terminating communication with client applications. This function may return OLE_WAIT_FOR_RELEASE requiring the server to wait for OLESVR to call the Release callback.
OleRevokeServerDoc	Document close	Revokes access to the specified document. This function may return OLE_WAIT_FOR_RELEASE requiring the server to wait for OLESVR to call the Release callback.

OleRevokeObject	Object close/delete	Revokes access to the specified object.
OleQueryServerVersion	Anytime	Retrieves the version of the OLESVR.DLL library in use.
OleBlockServer	Anytime	Queues requests to the server until the server calls the **OleUnblockServer** function.
OleUnblockServer	Anytime	Processes a request from a queue created by calling the **OleBlockServer** function.

9 OLE.H

OLE.H is the standard include file for all OLE applications, clients and servers alike, and defines structures like OLESERVER, OLESERVERDOC, OLEOBJECT, OLESERVERVTBL, etc. Prior to a #include <ole.h> statement in your source files, #define SERVERONLY (or define it on the C compiler command line with -DSERVERONLY) to prevent OLE.H from including a number of additional fields in the OLEOBJECTVTBL structure that are used strictly for object handler DLLs[1], that is, a server application does not use them.

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

SHELL.DLL contains functions to manipulate the registration database.[2] SHELL.LIB is the import library to which you link your application. The include file SHELLAPI.H contains prototypes for these functions:

Function	Description
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 Server uses the RegCreateKey, RegSetValue, and RegCloseKey functions to install itself in the system registration database so that client applications are aware that the server is available. In the event of an error, the application can delete a key with RegDeleteKey. RegQueryKey can be used to detect if the server is already registered, and in most OLE **server** applications, RegOpenKey and RegEnumKey are not used.

11 Library Redistribution and Installation

You may ship various components that your server 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

1Object handlers are described in the Windows 3.1 Software Development Kit.
2SHELL.DLL also contains functions for the Windows 3.1 Drag/Drop interface; these will not be discussed in this document.

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 OLESVR.DLL, but if you ship that library you must also ship the matching OLECLI.DLL to insure compatibility between the two libraries. Since server applications must make use the registration database you must also ship SHELL.DLL. In addition, if you supply a REG.DAT file for your server, that we'll talk about later, include REGLOAD.EXE to facilitate merging your registration file with the user's. 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, 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" function in an OLECLIENTVTBL structure.
- 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 the CallBack pointer in the OLECLIENTVTBL structure provided by OLESVR. OLESVR intercepts these calls and may not pass the notification on to the client.
- OLECLI.DLL and OLESVR.DLL communicate through DDE messages.

Although the OLE 1.0 server library, OLESVR.DLL, uses DDE commands to communicate with the client library, a server 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 server application in this model makes calls to OLESVR.DLL functions and calls a CallBack functionto notify the client of changes. For example, when an OLE server is opened because of a linked object in a client, the server must notify the client, through CallBack, whenever a change is made to that document or object. When the client receives this notification, it requests

the updated data from the server by calling a function in OLECLI, sending a request to OLESVR that calls methods in the server to retrieve the data to return to OLECLI and eventually back to the client. The CallBack function to which the server sends notifications resides in OLESVR. OLESVR filters notifications before passing them to OLECLI who then passes them on to the client application.

A server's methods, specified through OLESERVERVTBL, OLESERVERDOCVTBL, and OLEOBJECTVTBL structures, are the only points where OLESVR requests actions in the server.

13 OLE Data Structures and Application-Specific Variations

There are eight data structures defined in OLE.H of interest to an OLE server 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.
OLEOBJECT	A single LPOLEOBJECTVTBL
OLEOBJECTVTBL	Far pointers to object methods (see below).
OLESERVER	A single LPOLESERVERVTBL
OLESERVERVTBL	Far pointers to server methods (see below)
OLESERVERDOC	A single LPOLESERVERDOCVTBL
OLESERVERDOCVTBL	Far pointers to document methods (see below)

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.

To fully utilize these structures, define application-specific modifications to each structure in your own server, adding any additional fields that you want to manipulate at the server, document, or object level. 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 Schmoo server defines its own modifications to these structures, like SCHMOOSERVER that contains the LPOLESERVERVTBL field but also contains other fields that essentially act as global variables. Documents and objects are treated in the same manner, where additional fields are added to the structures to act as globals within an object or document. These globals need only be visible to an object, a document, or the server, leaving few variables that need be truly global within the application.

14 Methods

A large part of the responsibilities of an OLE server is contained in the methods that handle requests from OLESVR:

OLESERVERVTBL	OLESERVERDOCVTBL	OLEOBJECTVTBL
Create	Close	DoVerb
CreateFromTemplate	GetObject	EnumFormats
Edit	Execute	GetData
Execute	Release	QueryProtocol
Exit	Save	Release
Open	SetColorScheme	SetBounds
Release	SetHostNames	SetColorScheme
		SetData
		SetTargetDevice
		Show

Each method is simply a callback function defined in and exported from the server application. The specific responsibilities of each of these methods will be detailed later in the **Step-by-Step OLE Server** section. Also, this document uses prefixes of **Server**, **Doc**, or **Obj** before various methods to identify which VTBL they are referring to, as in **ServerRelease.** This naming convention helps distinguish between methods with the same name, sich as the server, document, and object **Release** methods which become ServerRelease, DocRelease, and ObjRelease.

15 Relationship Between Server, Document, and Object

The OLE structures represent a loose hierarchy. The three *structures* OLESERVER, OLESERVERDOC, and OLEOBJECT are independent of one another, that is, an OLESERVER structure does not necessarily contain an OLESERVERDOC structure (although you may wish to include it there).

A *server* in the most abstract sense is just the top-level structure of a particular *object class*, and each server only deals with a single object class. A server in this sense can contain any number of documents, and each document contains any number of objects of that class. This relationship is contained in the server and document methods. The server methods ServerCreate, ServerCreateFromTemplate, and ServerOpen allocate and initialize documents. The document GetObject method allocates and initializes objects.

"Server," "Document," and "Object" are abstract names that roughly correspond to parts of an application. The Server is really the main application and the main window (like the Frame Windows in MDI) although a single application can support multiple object classes, meaning it's a multiple server.[1] Inside that server can be any number of documents; generally a document equates to a window containing that document. Finally, inside a document are the objects, which may exist

1For example, Microsoft Excel has a Worksheet object class and a Chart object class, both supported by the same application. The distinction is visible at the document level.

as child windows of the document window or may simply be separate editable pieces of the document, such as a range of cells in a larger spreadsheet.

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 the standard OLE structures. 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 Mini Servers vs. Full Servers

Mini Servers are *embedding-only* servers: they cannot run stand-alone, they cannot save and open files independently, and they have a simplified user interface. Examples of mini-servers are Microsoft Word-Art (font effects) and Microsoft Draw (metafile editor), both included with products such as Word for Windows 2.0 or Publisher 1.0. Mini Servers work best to generate small visual objects (as Word-Art and Draw) or where there is little point in providing the ability to save and open files independently.

The Native data generated by a mini-server should be small so the effect on a client document containing many of such embedded objects is not great. For example, a mini-server bitmap editor may consistently generate 50K objects so a client document containing 20 of such bitmaps rapidly exceeds 1 MB in size. On the other hand, a metafile from Microsoft Draw is relatively small,

perhaps 200 bytes for a simple image, since metafiles are descriptions of graphics operations instead of a pixel-by-pixel dump. If your server creates large objects, support linking with a full server to allow the user to separate the data into multiple linked files instead of a single huge file with many embedded objects.

Full Servers are applications that can run stand-alone, creating, opening, and saving files, can link *or* embed objects, and may be an OLE client themselves. Microsoft Excel is a good example of such an application. Since full servers usually have a menu, there are some minor user-interface changes that a full server must follow when it starts for editing an embedded object.

If you are converting an existing application to become an OLE server, your application most likely reads and writes its own files already. In this case it should become a full server, since supporting linking adds very little code to the application and requires no user-interface changes. Note that if you support linking, you must also support embedding–let the user decide which to use.

18 Single-Document vs. Multiple Document Servers

A server application can be either single-document (SDI) or multiple-document (MDI). When OLESVR initiates editing of an object, it starts another instance of an SDI server but directs an MDI server to simply create a new window in which to edit that object. MDI applications were created to optimize the workspace for a set of documents in an application. As OLE moves away from application-centric computing to document-centric, the need for MDI diminishes. Microsoft therefore encourages the use of a single-document interface for OLE servers.

There is one catch with the wording of Single-Document and Multi-Document servers when intermixed with Single-Instance and Multi-Instance servers. The words *document* and *instance* have nothing to do with each other. A *Single-Document* server is not necessarily *Multi-Instance*, and a *Multi-Document* server not necessarily *Single-Instance*. As we'll see later, this affects your choice of the last parameter to **OleRegisterServer** that can be OLE_SERVER_MULTI or OLE_SERVER_SINGLE, where OLE_SERVER_MULTI stands for *Multiple-Instance* and NOT MDI. OLE_SERVER_SINGLE stands for *Single-Instance*. **Be very careful not to confuse the two!**

19 Clipboard Formats and Conventions

OLE servers provide data to OLESVR (and indirectly to the client) in standard clipboard formats:

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

Native, **OwnerLink**, and **ObjectLink** are formats defined in the OLE protocol; all OLE

applications register these formats with RegisterClipboardFormat, which returns the exact 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. See the next section below for the structure of these formats.

The CF_METAFILEPICT format is required to provide the client application with some visual representation (presentation) of the embedded or linked data–this image is all that the client will show for the object. The server application must provide at least a metafile in lieu of the server using an object handler DLL. Object handlers, which are not discussed in this document, act in place of the server to generate images dynamically from Native data.

Since the clipboard must be able to access the data and pass it to other applications, always allocate the memory using GlobalAlloc(GMEM_DDESHARE | GMEM_MOVEABLE...). GMEM_DDESHARE insures that other applications will be able to make use of the memory.

Server applications are responsible for placing all the data formats above on the clipboard in order of decreasing fidelity, that is, place the data containing the most information first. Given the contents of all the formats, OLE applications follow a standard ordering:1. Application-specific data.

2. **Native**
3. **OwnerLink**
4. **CF_METAFILEPICT**
5. **CF_BITMAP**
6. **ObjectLink**
7. Any other data.

For all intents and purposes, ordering is only important from Native to CF_BITMAP in that Native and OwnerLink precede ObjectLink and that CF_METAFILEPICT precedes CF_BITMAP.

20 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:

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

String 1 Object class name
String 2 Document name
String 3 Object's individual 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's individual name, assigned by the server application

In OwnerLink and ObjectLink the *object class name* is the registered class of objects that the 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 current implementation of OLE (1.04) does not use the OwnerLink document name. However, the *document name* in ObjectLink must contain a path name of the linked file, allowing OLESVR to pass that filename back to the server when a client edits a linked object. The *object name* in ObjectLink specifies the object in the document to select when that document is loaded.

21 Preparing an Application to Become an OLE Server

Any application that produces some data can, with enough time and effort, become an OLE server. However, before starting to add OLE support code, you can make small changes in your existing application to simplify your experience with OLE. The suggested changes below will allow you to stay more focused on the OLE specifics instead of becoming distracted with other non-OLE code changes. While these suggestions are not a mandate, they can make OLE easier to implement:

- Version Number Data Structures and Files
- Isolate Data
- Create Metafiles and Bitmaps
- Clean Up Clipboard I/O
- Isolate Utility Functions

22 Version Number Data Structures and Files

Include version numbers in any private data structures as well as in files. With OLE, a client application may embed data from one version of a server and later request a newer version to edit that same data. Without version numbers, the server would malfunction when manipulating an old data structure. A version number in the structures allows a newer server to convert the old data to a new structure and allows an older server to notify the user that an updated version of the server application is available.

Also consider placing a unique identifier in the data structure that you could publish so other applications could provide conversion utilities to convert that data to their own formats.

23 Isolate Data

Consider reorganizing your existing global and static data with the objective of isolating non-OLE data from OLE-specific data that you might add later. OLE is a protocol that sits on top of any server application that allows other applications access to the server's data; in other words, OLE is not something that is "integrated" a great deal with the application, and OLE, in the future, 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.

The subsections below discuss more specifics about data isolation. One other small suggestion is to add a global variable such as **fOLE** to indicate if the server was started stand-alone or for linking/embedding. In dealing with File menu commands like File Save, you will need to distinguish the two cases to perform slightly different operations.

24 Use of Window-Specific Data

OLE is based on a loose object-oriented model where a server contains documents, and documents contain objects. Isolating object, document, and server data is simpler if you structure your application to have a server window that contains document windows, and document windows that contains object windows, where each of those windows maintains a private data structure visible only to that window. Not only does it simplify data management, but also can reduce global variables to track the number of objects or open files and possibly variable-length arrays of data structures.

For each server, document, and object window, register the window class with sizeof(HANDLE) in the **cbWndExtra** field of WNDCLASS and define a single data structure containing the data specific to that class. Allocate memory for this structure when the window receives a WM_CREATE message and free that memory on WM_DESTROY. Store the handle to this memory in the window's extra bytes. Whenever you enter the window procedure later, you can simply retrieve this handle and

have access to the entire data structure.[1] Note, however, that the Schmoo sample server does not use this method.

25 Application-Specific (global) Data

Most applications, either for performance reasons or just to save development time, maintain some global (application-visible) data. Some of this data is only manipulated through WinMain or the main window procedure, and some is necessary for all levels of functions. Separate the data that is manipulated only on the highest level into one data structure and that manipulated at all levels into another structure (or just leave it alone).

The data that is only manipulated at the high levels can be folded into the application's OLESERVER structure and shared only with the main window procedure and the OLESERVERVTBL methods. If any other module needs that data, simply pass a pointer to this structure. Not only does this provide some structure to otherwise global variables, but referencing the variables off a pointer visibly identifies that data as belonging to the global data block. For example, the Schmoo server has a single global pGlob that points to a data structure containing non-OLE global variables. Anywhere a global appears in the source code it always has pGlob-> as a prefix, quickly identifying that variable as global.

26 Document- and Object-Specific Data

Information that is global between documents or document windows can also be isolated into a data structure or attached to each document window. Isolating the data now will make it easier to fold this data into an OLESERVERDOC structure that holds OLE specific data for each document. By isolating this data now and always referencing it from a pointer, changing where the structure resides is a trivial matter of changing the name of the pointer from which you reference the variable.

In the same manner as you isolate document-specific data, also isolate object-specific data that can be folded into your application-defined OLEOBJECT structure.

27 Create Metafiles and Bitmaps

OLE will later require the server to provide a graphic representation of its native data in a metafile and possibly a bitmap. Before adding any OLE code, add the capability to generate these images in the form of a metafile handle (HMF) or a bitmap handle (HBITMAP). As we'll discuss below, isolate the functions to create these images so they may be called at any time. You will not only need these handles when copying or cutting data to the clipboard but the OLESVR will call the GetData method in your OLEOBJECTVTBL to request a handle to the

1Which is far more convenient and efficient than using separate window extra bytes for each field in this structure.

same image.[1]

In creating a metafile, use the MM_ANSITROPIC mapping mode so the metafile will scale properly in the clipboard viewer and in other applications (such as OLE clients). If your metafile contains text and you are targeting Windows 3.1, try to use TrueType fonts, simply because they will provide the best presentation no matter how that metafile is scaled. In addition, TrueType fonts are device-independent, relieving your application from dealing with device-specific fonts depending on the printer in use (see the **ObjSetTargetDevice** section at the end of this document).

28 Clipboard I/O

If you are converting your application to operate as a *full* OLE server, now is a great time to add clipboard I/O capabilities.[2] If you already have some clipboard interaction, verify that you manipulate your application's private data, CF_METAFILEPICT, and CF_BITMAP. Later you will need to add the ability to copy and cut the OLE-specific formats of **Native** and **OwnerLink**. You will probably use your application's private data as the **Native** format.

As you want a single function to generate a metafile image of your server's private data, include a function to create and return a global memory handle to a METAFILEPICT structure to send to the clipboard as the CF_METAFILEPICT format. In the METAFILEPICT structure, specify the extents of the metafile in HIMETRIC units which OLE uses exclusively to specify dimensions. Do not confuse the use of HIMETRIC units with the MM_ANISOTROPIC mapping mode required in the metafile itself. The extents and metafile are two independent things.

29 Isolate Utility Functions

As the previous sections began to mention, one of the best ways to simplify OLE implementation is to reorganize functions and code fragments in your application to make them callable from OLE methods. Again, if you spend time reorganizing before dealing heavily with OLE you can stay more focused on the OLE protocol instead of becoming bogged down in moving code and updating include files.

This section suggests that you isolate code in fourteen areas:

1. Application Installation (first instance only)
2. Application Initialization (all instances)
3. Exit and Cleanup Procedure
4. Background Processing Schedulers
5. Drawing to an Arbitrary DC
6. Clipboard Format Builders
7. MM_HIMETRIC to MM_TEXT (or other) Conversions

1OLE does not ask for an HMF directly, but does request a handle to a METAFILEPICT structure.
2Mini-servers do not generally interact with the clipboard.

8. Setting Private Data Dynamically
9. Manipulating the Dirty Flag
10. File I/O
11. Changing Window Title on File Open/Save As
12. Window Sizing, by User or Program
13. (optional) DDE Execute String Parsing and Dispatching

30 Application Installation (first instance only)

Some applications contain code that executes only if the instance of the application is the only one running. For example, the first instance may establish a system-wide network or database connection that all subsequent instances of that application also use.

During installation, an OLE server may want to verify that it exists in the system registration database and possibly register itself if not. This only needs to occur for the first instance run on any particular machine. By isolating your existing installation code, you create a definite place to quickly insert the code to deal with the registration database.

31 Application Initialization (all instances)

Application startup is greatly affected by OLE as the server must handle several responsibilities at that time such as registering clipboard formats and parsing the command line. Isolating existing initialization code will create obvious places to insert this OLE-specific code. If all initialization code resides in one function call that you make from WinMain, then that function can simply return FALSE indicating there was an error so that WinMain can terminate the application.

One of the complexities of initialization is parsing the command-line to detect an *Embedding* flag. To ease this extra step, consider making a function to just parse the command line into separate parameters. In the Schmoo sample server, a function called HListParse in INIT.C allocates an array of pointers, parses the command line, and stores a pointer to each parameter in the command-line that is separated by whitespace. The array pointers reference each parameter individually so you can simply walk through that array to find switches, flags, and filenames. When the OLE server checks for *Embedding*, it only has to compare the first item in the array, since *Embedding* is either first on the command line or not there at all.

When *Embedding* does occur, the nCmdShow parameter passed to WinMain will have to change, so include some method in your initialization function to return the modified nCmdShow. This can be accomplished by passing a pointer to nCmdShow to the initialization function that can modify it or leave it alone. WinMain then does not have to change, since nCmdShow was changed behind the scenes.

32 Exit and Cleanup Procedure

Applications generally have some code to clean up items such as allocations and GDI objects that were created in the application initialization code. So in the same fashion that you separate initialization code, isolate the exit procedure to create a place for OLE-specific cleanup.

33 Background Processing Schedulers

Some applications that perform background processing modify the message loop in WinMain to detect when there are no messages to process (by using PeekMessage) and then performing a step of some background process before checking for messages again. OLE requires a server to enter a separate message loop when certain operations require the server to wait until an object, document, or the server is *released*. The message loop is necessary to process DDE messages between OLESVR and OLECLI while synchronizing a sequence of calls in the server.

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.

34 Drawing to an Arbitrary DC

At any time OLESVR may ask an OLE server to produce an image on the screen (for it's own display), a metafile or bitmap (for providing an image to a client application), or even to a printer. Simplify this requirement by modifying whatever drawing code your application uses for its objects to work with an arbitrary DC, whether it be a screen DC, a metafile DC, or a memory DC containing a bitmap.

For example, the Schmoo server creates a window called Polyline that edits the object. The Polyline window's paint routine normally draws the image to the screen on a WM_PAINT message. By enabling it to draw to any DC, the same function can be called from the WM_PAINT case, a redraw case in WM_LBUTTONDOWN, or from code to generate a bitmap or metafile.

35 Clipboard Format Builders

OLESVR will ask an OLE server (through the Object's GetData method) to provide a handle to data in any single clipboard format, including Native, OwnerLink, CF_METAFILEPICT, and CF_BITMAP. To simplify your GetData method later, create functions like HGetMetafilePict and HGetBitmap (as in Schmoo's CLIP.C) that return handles for existing formats (prior to OLE) that are readily usable for clipboard I/O or in handling the GetData method.

36 MM_HIMETRIC to MM_TEXT (or other) Conversions

OLE expresses any rectangles or other dimensional quantities in HIMETRIC units, including any rectangles provided in the DocSetDocDimensions method and also in any metafile that the server copies to the clipboard. If your application does not deal in MM_HIMETRIC already, create two functions that convert MM_HIMETRIC to and from the mapping mode you normally use, such as MM_TEXT. The application can then continue to deal in it's usual mapping mode, only converting units when exchanging dimensions of an object with OLESVR.

The Schmoo server contains two functions in MISC.C to convert rectangles between MM_HIMETRIC and MM_TEXT units. RectConvertToHiMetric converts from MM_TEXT to MM_HIMETRIC (using DPtoLP) and RectConvertToDevice handles the opposite (using LPtoDP). These functions are readily usable in your application, but note that debug Windows will RIP if you pass a metafile DC to the LPtoDP or DPtoLP functions, so don't pass them to these functions in Schmoo.

37 Setting Private Data Dynamically

Applications that cut or copy private data to the clipboard are also capable of pasting that data back. However, the code to perform this paste may be buried inside a WM_COMMAND processing switch. OLESVR uses the object's SetData method to affect the same operation as paste, so isolate whatever code your application uses to take a private data structure and add it to the current document. When you receive the Paste, just get the data from the clipboard and pass it to this function. When the SetData method receives a handle you again simply pass it to this function, making SetData trivial.

38 Manipulating the Dirty Flag

Invariably you will have several different operations that cause some change to occur in the server document that the user is editing. Your application may currently set some global flag in all of these cases. Isolate code to change that flag into a separate function that you then call from each case.

When you add OLE linking support, you must notify the client application that the data has changed. By creating a function to call every time you change the dirty flag, you create a single place to add the client notification call and prevent yourself from having to find and modify (that is, *possibly miss*) all the cases where you change the dirty flag.

39 File I/O

OLE affects most File menu commands and may require loading or saving a file outside the context

of those commands. Therefore, isolate functions to read and write your data structures to a file. The server method CreateFromTemplate, for example, requires that you open and read the contents of a file to use as the initial contents of a new document in the server. You may have functions that create a new document from a file already, but the CreateFromTemplate method also requires that you *only* use the file for *initial* data and do not keep that file open.

Also create a single function where you prompt the user to save changes in a document before creating a new one, opening a new file, or exiting the application. Normally this small piece of code displays a message box, and depending on whether the user chooses Yes, No, or Cancel the function either saves the file, continues with the operation (new/open/exit), or cancels the operation. When an OLE server is started to edit an embedded object, the text in this message box will change, so centralizing this code now simplifies the necessary changes for OLE.

40 Changing Window Title on File Open/Save As

Most applications that load files display the loaded filename in the title bar of the application along with the usual application name. When a user opens a file or renames the file with Save As, the name of the file in the title bar generally changes.

The UI for OLE servers goes a little beyond filenames, requiring a different string in the title bar if the server is editing an embedded object. Isolate the code for manipulating the title bar by creating a function that takes the main window handle, a pointer to a string containing the document name (or filename), and a pointer to a string containing an object name. In stand-alone (or OLE linking) operation, just pass a NULL as the object name and have this function create a string like:

<Application Name> - <Filename>

If the function receives a non-NULL object name, use that to indicate that the server is editing an embedded object and change the title bar to the appropriate string:

<Application Name> - <Object Name> in *<Filename>*

As we'll see later, OLESVR provides the object name and filename in the embedding case through the document SetHostNames method.

41 Window Sizing, by User or Program

Some applications (Schmoo, for example) resize the document window and scale their image(s) when their main window changes size. Resizing may occur either through user action (where a window will receive a WM_SIZE message) or through a possible call to SetWindowPos. If your application only handles the user action case, OLE will change that.

Window Sizing and the Dirty Flag

If your application saves the dimensions of a document or object in a file, then window sizing should make the document or object "dirty." The same dirty flag that you may currently use for prompting the user to save changes to a file will also be used with OLE to prompt the user to update the object in a client document. Resizing the object in the server should be an operation that is reflected in the client document, so users who resize an object must be prompted to update the object if they attempt to close the server when the object is dirty.

The document SetDocDimensions method simply instructs a document to resize to a given rectangle. Schmoo resizes several windows to fit this new document size. If you have existing code in the WM_SIZE case to resize other windows based on a document size, isolate that code into a separate function that you can call from SetDocDimensions.

42 (Optional) DDE Execute String Parsing and Dispatching

If your application currently supports DDE Execute commands and your OLE server will support the **StdExecute** protocol, you will receive DDE Execute strings not only through DDE messages but also through the server and document Execute methods when those commands come through OLESVR. If you have existing code to parse an Execute string, isolate it to be callable from your WM_DDE_EXECUTE case or from either Execute method. You may also have code to actually execute the commands after parsing, which you can also isolate to be callable from these separate locations.

43 Two Hints for Debugging an OLE Server

Developers who have already struggled with implementing OLE servers (myself included) have found debugging to be one of the greatest challenges. Below are some hints aimed at making your experience easier.

44 Starting the Server in the Debugger

One of the greatest difficulties in debugging is breaking execution when OLESVR launches the server application. There are two ways to deal with this:

1. If your debugger supports debugging multiple instances of the same application, start one instance of the server as stand-alone in the debugger. Set a breakpoint on WinMain and let that instance run. When OLESVR launches another instance, you will break on WinMain and be able to debug this new instance from there. The disadvantage is that you must watch which instance you are running at the time.

2. Under Codeview for Windows you can debug two applications by loading one as an application and specifying the other as a DLL. To debug an OLE server, use a suitable client (preferably a sample client for which you have sources) as the main application and specify your server as a DLL.

   When Codeview starts, it loads the symbols for both applications, allowing you to set breakpoints anywhere in the server application, even if it has not yet started. When OLESVR launches your application you can break on entry to WinMain which is most convenient for debugging installation code and initialization functions. In addition, when your server terminates, you are still running the debugger on the client, so if you launch the server again you debug it again, from the start. You also have no need to watch which instance of the server is running, unless you wish to debug multiple instances.

45 Set Breakpoints on Entry to All Methods

A great technique to learn what is happening with your methods is to set a breakpoint at the beginning of each method. Anytime an OLE operation is carried out, you can debug each method as they are used. You will also see the sequence of calls to your methods giving you a good feel about what operations (in the client) cause which methods to be called and in what order they are called.

With that, let's get to the code...

46 Step-By-Step OLE Server

This section will take you through the necessary code additions and changes to make an existing application an OLE server. The discussion assumes that the application can operate stand-alone, independently loading, modifying, and saving documents in files. It also assumes that your application is *not* based on the Multiple-Document Interface (MDI) model, so special notes about MDI are given when necessary.

The incremental approach in this 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. Some points reference parts of the Schmoo server to demonstrate how you might implement the functionality just described.

This section is organized into the following steps:

Define OLE Data Structures	Modifying your include files to contain the necessary OLE structures.
Install the Server in the Reg. Database	Registering the server with the system registration database as a server for a particular object.
Place Data on the Clipboard	Clearing existing data, retrieving handles to all necessary formats (Native, OwnerLink, ObjectLink, metafile, and bitmap), and placing those formats on the clipboard in the right order.
Implement Skeleton (stub) Methods	Creating the necessary source files and function stubs for OLE Server Methods.
Initialize the Application and VTBLs	Registering OLE clipboard formats, allocating an OLESERVER structure, initializing VTBLs, registering the server with OLESVR, parsing the command line, and creating an initial document.
UI: Change Window Titles and Menus	Handling simple user interface requirements.
Implement Basic Methods	Making the server capable of embedding. *This is the longest section to work through.*
Handle Simple Shutdown	Revoking the server and waiting for release.

File Menu: New, Open, Save[As], Update Allocating, initializing, and registering new documents, notifying the client of changes, and handling a switch from embedding mode to stand-alone. Notifying the client and renaming documents.

Closing Objects, Documents, and the Server Updating the object if necessary, informing OLESVR that the document is closing, revoking the server and documents, and freeing the VTBLs and other structures.

Optional Methods and Functions Implementing optional methods and blocking OLESVR requests.

47 Define OLE Data Structures

OLESVR passes information concerning the server, documents, and objects to the server through pointers to OLESERVER, OLESERVERDOC, and OLEOBJECT structures. However, the definitions of these structures in OLE.H include only a single pointer to a method callback table:

```
typedef struct _OLESERVER
        {
        LPOLESERVERVTBL         lpvtbl;
        } OLESERVER;

typedef struct _OLESERVERDOC
        {
        LPOLESERVERDOCVTBL      lpvtbl;
        } OLESERVERDOC;

typedef struct _OLEOBJECT
        {
        LPOLEOBJECTVTBL         lpvtbl;
        } OLEOBJECT;
```

For you to make effective use of these structures, define your own structures, keeping the **lpvtbl** field first and adding any additional data. Feel free to name these structures anything you care as Schmoo does below:

```
typedef struct
  {
  LPOLEOBJECTVTBL    pvtbl;      //Standard
  BOOL               fRelease;   //Flag to watch if we need to wait
  LPOLECLIENT        pClient;    //Necessary for notifications
  HANDLE             hMem;       //Memory handle to this structure
  } SCHMOOOBJECT;

typedef struct
  {
  LPOLESERVERDOCVTBL  pvtbl;     //Standard
  LHSERVERDOC         lh;        //Required by OleRegisterServerDoc
```

```
   BOOL          fRelease;    //Flag to watch if we need to wait
   ATOM          aObject;     //Name of the object
   ATOM          aClient;     //Name of the connected client
   HANDLE        hMem;        //Memory handle to this structure
   LPSCHMOOOBJECT pObj;       //Last object allocated.
   } SCHMOODOC;

typedef struct
   {
   LPOLESERVERVTBL  pvtbl;    //Standard
   LHSERVER      lh;          //Required by OleRegisterServer
   BOOL          fRelease;    //Flag to watch if we need to wait
   BOOL          fEmbed;      //TRUE if we're embedding only
   BOOL          fLink;       //TRUE if we're linking only
   WORD          wCmdShow;    //OLE-modified window show command
   HWND          hWnd;        //Main application window
   HANDLE        hMem;        //Memory handle to this structure
   LPSCHMOODOC   pDoc;        //Last document allocated
   } SCHMOOSERVER;
```

Each of these structures contains a flag **fRelease** that the server watches in cases where a Revoke function returned OLE_WAIT_FOR_RELEASE. If we called OleRevokeServer, OleRevokeServerDoc, or OleRevokeObject then we must watch the fRelease flag in SCHMOOSERVER, SCHMOODOC, and SCHMOOOBJECT respectively. See the section **Closing Objects, Documents, and the Server**.

Also note that each structure uses ATOMs to store strings, such as the name of a document or client. Though using ATOMs is not required, they provide a convenient storage method for strings where you need to make no prior assumptions about string length. To use the string itself you do have to allocate string space and call GetAtomName, but that has little impact where an OLE server actually uses the strings.

Finally, the **lh** fields in the SCHMOOSERVER and SCHMOODOC structures are handles generated by OleRegisterServer and OleRegisterServerDoc that are required in other OLE calls. Storing a handle in these structures associates that handle with the structure, so whenever you have a structure pointer, you have the handle for that item.

Schmoo also defines a few additional types not included in OLE.H to eliminate the use of ugly type casting like (LPOLEOBJECT FAR *). The first three are used in various methods. The LPVOIDPROC is the type of the object QueryProtocol function and is typedef'd here to make other code cleaner:

```
typedef LPOLESERVER    FAR *LPLPOLESERVER;
typedef LPOLESERVERDOC FAR *LPLPOLESERVERDOC;
typedef LPOLEOBJECT    FAR *LPLPOLEOBJECT;

typedef LPVOID (FAR PASCAL *LPVOIDPROC) (LPOLEOBJECT, LPSTR);
```

48 Install the Server in the Registration Database

An OLE server must register itself with the system registration database specifying the objects it can edit. The database contains key/value pairs where keys and values are simple strings.

Windows uses the registration database for more than just OLE, such as storing associations between file extensions and executables; OLE is not the exclusive use of the database.

The discussion below describes how a server registers itself as the server for a single object, but a single application can be the server for multiple objects. For each object the application must repeat the registration process. Registration is accomplished through calling functions in SHELL.DLL (see the **OLE Technical Background** section above).

An alternate method to register the server is to ship your application with a APPNAME.DAT file and the REGLOAD.EXE utility. To generate the APPNAME.DAT file, start the REGEDIT.EXE application shipped with Windows 3.1 with **-v** on the command line to enable its editing mode. When you install your application, use REGLOAD.EXE to merge this APPNAME.DAT with the user's REG.DAT automatically, or instruct the user to merge the file themselves with RegEdit.

49 OLE Keys and Values

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. Both of these subkeys *must* have values:

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 ***in mixed case***	Edit
verb*n* (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

Verbs are the types of actions a user can perform on an object, such as "Play," "Edit," and "Open." For most graphical applications, like Schmoo, "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 document, 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. Note that when you register verbs, store them in mixed case.Be sure to include the **full pathname** to both server executable and object handler in the values for the *server* and *handler* subkeys. If you do not include the path, then your application must either reside in the PATH or OLE will force the user to provide that path *every time they edit an object*–a major headache. In addition, the verbs you register must be sequential, starting at 0 and increasing by one for each verb.

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 another application attempts to send commands through the OleExecute function in the OLECLI.DLL library. A server need not support **StdExecute**.

SetDataFormats and RequestDataFormats

A server may wish to support the OleSetData and OleRequestData calls available to client applications from OLECLI.DLL. These functions allow the client to set or retrieve the data for an object if that client understands the object's native format. The value strings for the SetDataFormats and RequestDataFormats keys are a comma-separated value (CSV) list of clipboard formats such as "Native, CF_METAFILEPICT" or "CF_DIB, CF_BITMAP, CF_TEXT."Note that while SetDataFormats and RequestDataFormats are optional in the StdFileEditing protocol, the object SetData and GetData methods, which are unrelated, are absolutely necessary.

50 Example: FOLEServerInstall and FKeyCreateThe Schmoo application uses its FOLEServerInstall function (OLEINST.C) to create the necessary keys in the registration database. FOLEServerInstall takes a single pointer to a structure that contains the strings necessary to build the keys and a few other key pieces of information. OLEINST.H defines this structure:

```
typedef struct
  {
  LPSTR     pszServerName;     //Full server name.
  LPSTR     pszServerClass;    //Short server class name.
  LPSTR     pszServerPath;     //Full path to server module.
  LPSTR     pszHandlerPath;    //Optional Full path to object handler DLL.
  LPSTR     pszExt;            //File extension for the server.
  LPSTR     *ppszVerbs;        //Pointer to array of LPSTRs to verbs.
  WORD      cVerbs;            //Number of verbs in array.
  LPSTR     pszSetFormats;     //Optional CSV list of accepted formats.
  LPSTR     pszRequestFormats; //Optional CSV list of requestable formats.
  BOOL      fExecute;          //Is OleExecute supported?
```

```
} REGINSTALL;.
```

FRegDBInstall (OLEINST.C) contains an example of filling this structure and calling FOLEServerInstall.

On entry, FOLEServerInstall checks if the server is already registered in the database, using RegQueryValue:

```
#include <shellapi.h>

...

BOOL FAR PASCAL FOLEServerInstall(LPREGINSTALL lpRI)
        {
        LONG                    lRet;

        ...

        //Check if this server is already around.
        lRet=RegQueryValue(HKEY_CLASSES_ROOT, lpRI->pszServerClass, szKey, &dw);

        if (ERROR_SUCCESS==lRet)
                return TRUE;

...
```

In RegQueryValue, HKEY_CLASSES_ROOT specifies the root key, lpRI->pszServerClass the *classname* used in all the subkeys, szKey is a dummy buffer to receive the value of this key, and dw is a DWORD that contains the length of the szKey buffer. If RegQueryValue returns ERROR_SUCCESS, then a value already exists for the *classname* key and there's no need to create any additional keys.[1]

If the server is not already registered, FOLEServerInstall proceeds to create keys for all desired values. For each key it calls FKeyCreate with a key name, a subkey name, and a value for the key. FKeyCreate handles the sequence of RegCreateKey, RegSetValue, and RegCloseKey:

```
BOOL PASCAL FKeyCreate(LPSTR pszKey, LPSTR pszSubKey, LPSTR pszValue)
        {
        char        szKey[128];
        HKEY                    hKey;
        WORD                    cch;
        LONG                    lRet;

        cch=lstrlen(pszValue)+1;

        lstrcpy(szKey, pszKey);
        lstrcat(szKey, pszSubKey);

        lRet=RegCreateKey(HKEY_CLASSES_ROOT, szKey, &hKey);

        if (lRet!=ERROR_SUCCESS)
                return FALSE;

        lRet=RegSetValue(HKEY_CLASSES_ROOT, szKey, REG_SZ, pszValue, cch);

        if (lRet!=ERROR_SUCCESS)
                {
```

1This assumes that if the *classname* key exists that all other necessary keys also exists, which is a safe assumption because end-users do not have the capability to modify the database. Only the SHELL.DLL API and the Windows 3.1 RegEdit application can change the database.

```
        //Delete key if we could not set a value.
        RegDeleteKey(hKey, pszSubKey);
        return FALSE;
        }

    lRet=RegCloseKey(hKey);

    if (lRet!=ERROR_SUCCESS)
        return FALSE;

    return TRUE;
    }
```

FKeyCreate appends the subkey pszSubkey to the basic key in pszKey. pszSubKey is passed separately in order to use RegDeleteKey if necessary. FKeyCreate uses RegDeleteKey only if RegCreateKey succeeded but RegSetValue failed.

Both FOLEServerInstall and FKeyCreate are reusable pieces for your application. As mentioned above, Schmoo uses FRegDBInstall (OLEINST.C) to fill a REGINSTALL structure. FRegDBInstall sets fExecute to FALSE indicating that the server does not support StdExecute. Also note that ppszVerbs must point to an array of LPSTR values, where each value in that array points to an actual string containing a verb. The Schmoo server, like most other servers with graphical data, supports only one verb, "Edit." However, FOLEServerInstall is capable of handling multiple verbs.

If you choose to use these functions, OLEINST.H contains the necessary structure definitions and function prototypes. OLEINST.H is the only include file independent of OLEGLOBL.H since installation code may reside in a separate installation program.

51 Verifying the Registration with RegEdit.

After running your server with the registration code, start the Windows Registration Database Editor, REGEDIT.EXE, to verify that your server was properly registered. In order to see all information and edit the database, start RegEdit with **-v** on the command line. In the figure below the Registration Editor shows that Schmoo successfully added the key **Schmoo1.0** with the value "Schmoo 1.0 Figure" and that subkeys of \protocol\StdFileEditing\verb\0 and \ protocol\StdFileEditing\server also exist. Schmoo also successfully added the key HKEY_CLASSES_ROOT\.MOO. In the event your code produced the wrong results, just delete the entries and try again.

52 Place Data on the Clipboard

Setting clipboard data is a simple matter of creating the data structures and passing handles to the clipboard, but it is necessary to allow the user to run the server stand-alone and copy data to the

clipboard for linking or embedding into another application. If you have already created functions to retrieve the handles for the individual clipboard formats, adding OLE clipboard support will be very quick using the following steps:

1. Call **OpenClipboard.**

2. Call **EmptyClipboard** to clear out existing data.

3. For each data format, retrieve a handle to the data and call **SetClipboardData**, observing the standard order of formats:
   a. Any private data supported by the application.
   b. "Native"
   c. "OwnerLink"
   d. CF_METAFILEPICT
   e. CF_BITMAP
   f. "ObjectLink," if the server has a known filename *and* the data is being *copied*, not cut. Linking to non-existent data is esoteric, at best.
   g. Any other standard clipboard format supported by the application.

4. Call **CloseClipboard**.

As far as pasting, there is no need for a server to know how to paste "Native," "OwnerLink," or "ObjectLink."

53 Example: FEditCopy, FOLECopyNative, FOLECopyLink and HLinkConstruct

Schmoo sends the basic non-OLE formats to the clipboard using the function FEditCopy in CLIP.C (FEditCut simply calls FEditCopy and deletes the current data). Prior to OLE implementation, it simply set data for its private format (a format registered as "Schmoo1.0"), CF_METAFILEPICT, and CF_BITMAP.

To support OLE, FOLECopy calls FOLECopyNative and FOLECopyLink (used for both OwnerLink and ObjectLink) in the correct order. Note that even when two formats have the same data (as the private and "Native" data have), send a different handle for each to the clipboard so that the clipboard can manage each format separately.

The function HLinkConstruct (OLECLIP.C) reusable in your application, verbatim. It simply takes three strings, allocates memory for the three plus an extra null-terminator, copies those three strings into that memory, and returns a memory handle that can be immediately sent to the clipboard.

µ §

54 Verifying Correct Data Placement

After adding basic OLE clipboard support, run your server and copy or cut data to verify that the proper clipboard formats are appearing on the clipboard in the correct order, depending on whether or not you have saved a file. The Display menu of the Window's Clipboard Viewer (clipbrd.exe) lists the data currently held in the clipboard:

No file saved:	**File saved**
<Owner Display>	<Owner Display>
<Private Data>	<Private Data>
Native	Native
OwnerLink	OwnerLink
Picture	Picture
Bitmap	Bitmap
	ObjectLink

At this time you should also verify that the metafile you place on the clipboard is scalable by viewing the CF_METAFILEPICT format (Picture) and resizing the clipboard viewer application. The image should scale smoothly to any rectangle without distortion, unlike a bitmap munged with StretchBlt.

To verify that the OwnerLink and ObjectLink data are correct, you will have to step through your code in a debugger and see exactly what string exists in that memory before the call to **SetClipboardData**.

You can now also open an OLE client application and paste the object into a document. The client application will simply retrieve the Native and OwnerLink data, if available, and a presentation format like a picture or a bitmap, displaying that image in its document. However, before you can double-click the object in the client document to edit it, there is more work to complete...

55 Implement Skeleton (stub) Methods

In the next section, **Initialize the Application and VTBLs**, you will add code to call MakeProcInstance for each of the server, document, and object methods. To do that, you have to have some methods, even stub methods.

As a starting point, use the Schmoo files OLESVR.C, OLEDOC.C, and OLEOBJ.C. These files contain implementations for each method that you can use as a starting point, commenting out any code you don't yet want yet. Each of the functions includes a comment describing what that method must do when called, as discussed later in this document under **Implementing Basic Methods**.

If the server is going to support multiple object classes, then each class needs its own set of methods on the server, document, and object level, so make copies of these source files as necessary. Of course, some of these methods may contain the same code, and there is no reason that they may not be the same function. When you initialize the VTBL for that object class you can simply use the

same function pointer across classes.

In all cases, these stubs return an appropriate OLE_ERROR_* value depending on the method. For information on the parameters and return values of all methods, consult the Windows 3.1 SDK Reference.[1]

56 Export the Methods

An extremely easy mistake that we have probably all made is to forget to export a function you pass to MakeProcInstance. During initialization, an OLE server passes pointers to all methods to MakeProcInstance and so you must list them in the EXPORTS section of your .DEF file in order for OLESVR to call them. Note that exported methods in the .DEF file need not be listed in any particular order, nor do they require particular ordinal number or any sort of standard name.

Compile here to weed out any syntax errors that have crept in and to insure that the source files are compiled and linked flawlessly. Also do a short run of the application to insure that the code addition did not cause any strange segmentation errors. Other than that, there is nothing to test.

57 Initialize the Application and VTBLs

This section describes what an OLE server must do during application (instance) initialization above what its normal operations prior to OLE. Instance initialization takes place before the application has created its main window and entered the message processing loop. If an error occurs during this phase, terminate the application, notifying the user if necessary.

Initialization steps specific for OLE:

1. Register clipboard formats for Native, OwnerLink, and ObjectLink types.
2. Initialize VTBLs containing thunks for the server, document, and object methods.
3. Allocate and initialize your application-specific OLESERVER structure.
4. Register the server application with OleRegisterServer.
5. Parse the application command line to determine if it's starting for a linked or embedded object.
6. Allocate and initialize your application's initial OLESERVERDOC structure.
7. Register the initial document with OleRegisterServerDoc.

This section also includes a summary of error conditions that may happen during OLE initialization and what action to take on those errors.

1The Windows 3.1 Software Development Kit contains the full OLE function and structure reference. Search for OLESERVERVTBL, OLESERVERDOCVTBL, and OLEOBJECTVTBL for server, document, and object methods, respectively, in the on-line help file.

58 Register Clipboard Formats

Regardless of whether or not the server actually does clipboard I/O, you need clipboard formats for "Native," "OwnerLink," and "ObjectLink." Store these registered formats in globally visible variables since the application uses them for clipboard I/O as well as inside object methods. The object methods GetData, SetData, and EnumFormats use these formats outside the clipboard context to exchange data with OLESVR. Register the three standard formats with **RegisterClipboardFormat**:[1]

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

if (0==pOLE->cfNative || 0==pOLE->cfOwnerLink || 0==pOLE->cfObjectLink)
   return FALSE;
```

If either call to RegisterClipboardFormat for Native and OwnerLink fails, then initialization of this server fails. If registering ObjectLink fails, then this server cannot do linking. It is your decision if this is to be a fatal error or will simply disable any ability to link, in which case the server acts as embedding only. The Schmoo server simply terminates if any of the three calls fail.

59 Initialize VTBLs and VTBL Pointers

Before the server application calls any function in OLESVR, it must initialize the method callback tables, called VTBLs, with the **MakeProcInstance** call, setting each field in the OLESERVERVTBL, OLESERVERDOCVTBL, and OLEOBJECTVTBL structures. Note that this requires you to allocate these structures, since the OLESERVER, OLESERVERDOC, and OLEOBJECT structures (or your variants) simply contain a *pointer* to the VTBL. If you have already implemented stubbed methods as described in the previous section, then you should have already exported them in your .DEF file.

For a single-document server application, such as Schmoo, initialize the server, document, and object VTBLs during application initialization. While it is acceptable to initialize the document and object VTBLs dynamically, it can become cumbersome and inefficient. At the bare minimum, however, you must initialize the OLESERVERVTBL, the pointer to which is stored in the OLESERVER structure (or your specific modification):

```
/*
 * pvt is of type LPOLESERVERVTBL, hInst is the application instance handle,
 * and the Server* functions are the names of the server methods.
 */
pvt->Create          =MakeProcInstance(ServerCreate,          hInst);
pvt->CreateFromTemplate=MakeProcInstance(ServerCreateFromTemplate, hInst);
pvt->Edit            =MakeProcInstance(ServerEdit,            hInst);
pvt->Execute         =MakeProcInstance(ServerExecute,         hInst);
pvt->Exit            =MakeProcInstance(ServerExit,            hInst);
pvt->Open            =MakeProcInstance(ServerOpen,            hInst);
pvt->Release         =MakeProcInstance(ServerRelease,         hInst);
```

1As mentioned earlier in this document, pOLE is a pointer to global OLE variables in the Schmoo sample server.

OLESERVERDOCVTBL and OLEOBJECTVTBL structures are initialized in the same manner:

```
pvt->Close          =MakeProcInstance(DocClose,          hInst);

pvt->GetObject      =MakeProcInstance(DocGetObject,      hInst);
pvt->Execute        =MakeProcInstance(DocExecute,        hInst);
pvt->Release        =MakeProcInstance(DocRelease,        hInst);
pvt->Save           =MakeProcInstance(DocSave,           hInst);
pvt->SetColorScheme  =MakeProcInstance(DocSetColorScheme,   hInst);
pvt->SetDocDimensions=MakeProcInstance(DocSetDocDimensions, hInst);
pvt->SetHostNames    =MakeProcInstance(DocSetHostNames,     hInst);
```

The one ugly part in initializing OLEOBJECTVTBL is the ObjQueryProtocol method that returns something other than an integer type (LPVOID), and where the OLEOBJECTVTBL field of QueryProtocol is not compatible with the standard FARPROC returned from MakeProcInstance. (The code below (from FOLEVtblInitObject) uses some locals to be a little more readable):

```
FARPROC    lpfn;
LPVOIDPROC  lpvp;

/*
 * Local variables are used here just to make this one assignment
 * more readable since it requires some typecasting to compile clean
 * at warning level 3.
 */
lpfn=(FARPROC)ObjQueryProtocol;
lpvp=(LPVOIDPROC)MakeProcInstance(lpfn, hInst);

pvt->QueryProtocol = lpvp;
pvt->DoVerb         =MakeProcInstance(ObjDoVerb,         hInst);
pvt->EnumFormats    =MakeProcInstance(ObjEnumFormats,    hInst);
pvt->GetData        =MakeProcInstance(ObjGetData,        hInst);
pvt->Release        =MakeProcInstance(ObjRelease,        hInst);
pvt->SetBounds      =MakeProcInstance(ObjSetBounds,      hInst);
pvt->SetColorScheme =MakeProcInstance(ObjSetColorScheme,   hInst);
pvt->SetData        =MakeProcInstance(ObjSetData,        hInst);
pvt->SetTargetDevice=MakeProcInstance(ObjSetTargetDevice,  hInst);
pvt->Show           =MakeProcInstance(ObjShow,           hInst);
```

If any MakeProcInstance call for any VTBL fails, then fail the initialization and terminate the application. An OLE server cannot function without the ability for OLESVR 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.

60 Allocate OLESERVER and Register the Server

Once you have initialized the OLESERVERVTBL structure, allocate your OLESERVER structure and initialize as necessary. Be sure to set the LPOLESERVERVTBL to point to the VTBL you just initialized. Then register the server application with OLESVR by calling **OleRegisterServer**:

```
os=OleRegisterServer(pszClass, (LPOLESERVER)pSvr, &pSvr->lh, hInstance, OLE_SERVER_MULTI);
```

pszClass	Points to the name of the object class that this server supports.
pSvr	Points to the relevant OLESERVER structure with an initialized

	VTBL.
&pSvr->lh	Points to the LHSERVER associated with the server. In this example, the handle is stored in the application-specific OLESERVER structure.
hInstance	Instance handle for the application.
OLE_SERVER_MULTI	Indicates that this server supports multiple instances as opposed to a single instance.

Warning

Do not confuse OLE_SERVER_MULTI with MDI. MDI applications specify OLE_SERVER_SINGLE, SDI applications specify OLE_SERVER_MULTI. If you specify OLE_SERVER_SINGLE then OLESVR, instead of launching a new instance of the application, always directs the existing instance to open a new document.

OLE Class Name vs. DDE Executable Name

If an OLE server application is also a DDE server, the class name passed to **OleRegisterServer** must not be identical to the application's module (executable) name. This restriction is due to the fact that OLESVR communicates with the server through DDE and attempts to initiate a DDE conversation with the server using the name given in OleRegisterServer. If the application uses that name for its own DDE, then OLE and non-OLE conversations will overlap and cause everything to malfunction.

Any OLE server application must *unconditionally* call OleRegisterServer. If OleRegisterServer returns OLE_OK, then the LHSERVER pointed to in the third parameter contains a crucial handle that you must pass to future OLESVR calls, such as OleRegisterServerDoc (see below under **Register the Document(s)**). Be sure to store this handle where it will be globally accessible and at the same time associated with the OLESERVER structure. Storing it directly in the application's OLESERVER meets both needs.

If OleRegisterServer returns anything other than OLE_OK, terminate the application. Treat this condition as if you could not register the main window class or as any other error that prevents the application from running.

The class name passed in **OleRegisterServer** must match the class name set in the registration database for the object class. The OLE libraries use this name to find your application if it's already running and the use edits another object of that class.

Finally, note that you must eventually free the OLESERVER structure, which is best accomplished during application shutdown after exiting the main message loop.

61 Parse the Command-Line and Determine the Initial Window State

In addition to any command-line parsing you already handle, check for an "Embedding" flag, which may appear (case-insensitive) as *-Embedding* or */Embedding*, possibly with a filename. What appears on the command line depends on whether the server is started as stand-alone, for embedding, or for linking, resulting in three possibilities:

- *Embedding* does not appear.
- *-Embedding* or */Embedding* appears with nothing else.
- *-Embedding* or */Embedding* appears with a single filename.

Note that when *Embedding* does appear there can only be an additional filename. No other command-line arguments or parameters will ever appear. Also note that the word "Embedding" is not localized in international versions of Windows.

In the first instance with a normal command-line, OLESVR did not start the application. In this case continue your initialization as usual, but also register the initial document (see below).

When *Embedding* appears, either alone or with a filename, do not show any window in the application–return a flag or value to WinMain indicating that the initial parameter to ShowWindow is SW_HIDE instead of WinMain's nCmdShow parameter. When OLESVR requires the server to become visible, it will call the ObjShow or ObjDoVerb methods at which time the server can show itself.

When *Embedding* appears alone, the server has been started in an embedding case where there's no filename known. When *Embedding* appears with a filename, the server has been started in a linking case and the filename references the linked file. When a server starts as linking, it must also register a document.

62 Allocate and Register the Initial Document(s)

Allocate your OLESERVERDOC structure and initialize as necessary, making sure to fill the LPOLESERVERDOCVTBL field with a pointer to the VTBL you created. Then, for all cases other than where *-Embedding* was the only thing on the command line, call **OleRegisterServerDoc** to inform OLESVR that this instance of the server has an open document. Again, DO NOT call OleRegisterServerDoc when the server starts to edit an embedded object.

If the command line contains a filename, with or without *-Embedding*, then you have a document name to pass to OleRegisterServerDoc; otherwise use a string like "(Untitled)" to register a new, unnamed document:

```
//Register a new Untitled document.
os=OleRegisterServerDoc(pSvr->lh, "(Untitled)", (LPOLESERVERDOC)pDoc, &pDoc->lh);
```

or

```
//Register the file given on the command line.
os=OleRegisterServerDoc(pSvr->lh, pszFile, (LPOLESERVERDOC)pDoc, &pDoc->lh)
```

where

pSvr->lh	Contains the LHSERVER generated in OleRegisterServer.
pszFile	Points a null-terminated string with the document name.
pDoc	Points to an OLESERVERDOC structure with an initialized LPOLESERVERDOCVTBL.
&pDoc->lh	Points to the LHSERVERDOC associated with the document. In this case, the handle is stored in the application-specific OLESERVERDOC structure.

If OleRegisterServerDoc returns OLE_OK in any case, then continue your initialization. Otherwise, immediately call OleRevokeServer and fail the initialization because you cannot do OLE without a document. OleRevokeServer will always return OLE_OK here because there can not yet be any conversations.

63 Handling Errors: Summary

Condition	Action
RegisterClipboardFormat fails on "Native"	Terminate application
RegisterClipboardFormat fails on "OwnerLink"	Terminat
RegisterClipboardFormat fails on "ObjectLink"	Disable li application
MakeProcInstance fails on any method.	Free any thunks previously created and terminate application. If the server does not support the StdExecute protocol, then you do not need to terminate if MakeProcInstance fails on ServerExecute or DocExecute.
Server allocation, OleRegisterServer fails.	Free the VTBL thunks and terminate application.
Doc allocation, OleRegisterServerDoc fails.	Call OleRevokeServer, free VTBL thunks, and terminate application.

64 Example: FApplicationInit, FOLEInstanceInit, and OLEVTBL.C

FApplicationInit (INIT.C) contains Schmoo's initialization code which calls FOLEInstanceInit (OLEINIT.C) to perform the OLE-specific initialization. Only parts of these functions are reusable, but they each call other functions that you may copy verbatim. The first is HListParse (INIT.C) which, as mentioned above, parses a command line into an array of pointers that can be walked one-by-one to check arguments. HListParse also uses PszWhiteSpaceScan in INIT.C.

The other reusable parts are all the functions in OLEVTBL.C, specifically the FOLEVtblInitServer, FOLEVtblInitDocument, and FOLEVtblInitObject functions. If you started with Schmoo's methods as a template and kept the method names the same, these functions in OLEVTBL.C will work as is. OLEVTBL.C also contains functions to handle VTBL cleanup on termination that we'll see later.

65 Verify Command-Line Parsing and Initialization

While the server is by no means functional at this point, verify that you correctly parse the command line and that you successfully register the server. For testing purposes, change the case where you use SW_HIDE in the first call to ShowWindow to use SW_SHOWNORMAL. Otherwise, you might start your application hidden and not be able to close it. You can then start your application stand-alone but with the *-Embedding* flag and other command-line arguments to verify that you are parsing it correctly and handling each case as necessary. You can also verify that OleRegisterServer and OleRegisterServerDoc work, as well as initializing your VTBLs.

Note that at this point, since we have not dealt with closing the server, you risk leaving OLESVR in an unstable state since you have registered the server with OLESVR but have done nothing to inform it that you are terminating. I only suggest these changes here to start giving you a feel of what is going on down in the initialization code.

66 UI: Change Window Titles and Menus

User interface standards for a server are quite simple and only affect the window's title bar and one or two of its menus and a message box. Complete information about UI standards for OLE is contained in the Microsoft *User Interface Style Guide*, Chapter 9.[1] For mini-servers, that never run stand-alone, the only UI change affects the window title as described below. Since mini-servers do not have menus, there are no menu changes to worry about.

All UI changes affect a server started for embedding; when a server is opened for linking it appears exactly like it was started stand-alone. The linked server only cares about OLE when it saves and changes documents so it can properly inform the client in which the document resides.

Note that after making these additions there is little testing you can perform, except to see if the routines to make the title bar and menu changes work as expected. You either call these functions

1The OLE Chapter of the Style Guide will be shipped with the final Windows 3.1 Software Development Kit.

from your file handling code or from the methods, such as DocSetHostNames.

67 Window Title Change

Most full servers that are able to open files display the name of the open file somewhere in their title bar. For example, the Windows 3.1 Paintbrush application displays the filename of the currently loaded bitmap if it has such a filename, or else it uses the filename "(Untitled)". The standard for filenames in the title bar of any application is:

<Application Name> - <Filename>

as in **Paintbrush - CHESS.BMP**. When the server is started for embedding, *and only for embedding*, the title bar should appear as:

<Application Name> - *<Object Type>* in *<Client Document>*

as Paintbrush shows: **Paintbrush - Paintbrush Picture in SERVER.DOC**
The names of objects and clients comes from the document SetHostNames method. When we implement that method in the next section we'll have a real string to use in these titles, but for now, just use a string like "Client Document."

68 Menu Changes for Embedding

When the focus in an MDI server changes from a window that had activated an embedded object to a window that is editing *a document that does not contain an embedded object*, the Update command should revert to Save and Save Copy As to Save As.
Like the changes to the title bar, changes to the menu are simple: they affect only the **File** menu in three locations using the **ModifyMenu** API call. First, change any menu item labeled **File/Save** to **File/Update <*Client Document*>** when the server starts for embedding where *<Client Document>* is the name given in the DocSetHostNames method covered in the next section. For now, just use a string like "Client Document" where the real name will appear later.

Second, change the menu item labeled **File/Save As...** to **File/Save Copy As...**. Finally, change the **File/Exit** menu item to **File/Exit & Return to <*Client Document*>**, where *<Client document>* is the same as for **File/Update** above.

69 Example: SCHMOO.C

SCHMOO.C contains a function called **MenuEmbeddingSet** that takes the window handle of the main server window, a string of the client name, and a flag indicating whether to set the menu to the OLE style or the stand-alone style:

```
void FAR PASCAL MenuEmbeddingSet(HWND hWnd, LPSTR pszClient, BOOL fOLE)
  {
  HMENU     hMenu;
  char      szTemp[130];
  LPSTR     pszT;
```

```
hMenu=GetMenu(hWnd);

//Change the File/Save menu to File/Update <client> or vice-versa
if (fOLE)
   wsprintf(szTemp, rgpsz[IDS_UPDATE], pszClient);
else
   lstrcpy(szTemp, rgpsz[IDS_SAVE]);

ModifyMenu(hMenu, IDM_FILESAVE, MF_STRING, IDM_FILESAVE, szTemp);

//Change the File/Save As menu to File/Save Copy As or vice-versa.
pszT=(fOLE) ? rgpsz[IDS_SAVECOPYAS] : rgpsz[IDS_SAVEAS];
ModifyMenu(hMenu, IDM_FILESAVEAS, MF_STRING, IDM_FILESAVE, pszT);

//Change "Exit" to "Exit & return to xx" or vice-versa.
if (fOLE)
   wsprintf(szTemp, rgpsz[IDS_EXITANDRETURN], pszClient);
else
   lstrcpy(szTemp, rgpsz[IDS_EXIT]);

ModifyMenu(hMenu, IDM_FILEEXIT, MF_STRING, IDM_FILEEXIT, szTemp);
return;
}
```

You can call a function such as this at any time to toggle the OLE menu on and off as necessary. Since this function just changes strings on the menu items, the application has to distinguish between File/Save and File/Update when it receives the WM_COMMAND message for this item. If you completely replace the item, changing the ID value, then each case will come as separate WM_COMMAND messages. The decision is simply what is most convenient for your application.

70 Implement Basic Methods

The real fun with an OLE server begins when you implement the basic methods for the server, document, and objects; after implementation, you can start seeing OLE working. Double-clicking an embedded object in a client will start an instance of your server, set data in an object, instruct the server to show itself, and so on. However, do not test these methods until you add minimal shutdown code, which is quickly covered in the next section. The remainder of this section will cover each basic method in the OLESERVERVTBL, OLESERVERDOCVTBL, and OLEOBJECTVTBL structures that are required for minimal OLE operation.

A Word of Caution:

Do not post any DDE messages or call any OLE functions from within a method. Methods are called as part of an asynchronous process that is currently affecting an object, document, or the server. In addition, do not display a message box or a dialog box since they process messages internally and would cause DDE messages to be processed before the server has finished executing the method it's in.

71 Thinking about the Methods

Implementing the basic 18 methods is not extremely complicated, but will take time. Before jumping into coding each method one by one, spend some time thinking about what should happen during each one of these methods and identify methods that will possibly share code. To make this easier, Schmoo's OLESVR.C, OLEDOC.C, and OLEOBJ.C files briefly describe the responsibilities of each method, which is repeated below. I mainly suggest you read through this section or the sample code before doing any code yourself, because it becomes increasingly difficult to see global issues between functions once you begin to immerse yourself in code.

Each of the following sections, given by Server, Document, and Object, lists the basic methods for that type and then deals with each method one-by-one, describing when OLESVR calls that method, the information passed to the method, the responsibilities of the method, and what actions to take. You will begin to see some overlap between methods. Note also that the names of the methods are given as ServerRelease and DocRelease instead of just Release, to make it easier to associate which method belongs to which OLE item. These names also match those in the sample code.

Much of this section is duplicated in the sample code, and the parameters to each method are documented in the Windows 3.1 SDK Programmer's Reference. However, since names of certain parameters are necessary in discussing the method's responsibilities, all the method parameters are given here. Also, the return value for the particular method is generally given in the responsibilities.

Method Parameter Types

Although the methods for server, documents, and objects are documented to accept pointers like LPOLESERVER, you may code your methods to accept your private LPOLESERVER data type. For example, Schmoo's ServerCreate in OLESVR.C accepts the type LPSCHMOOSERVER instead of LPOLESERVER as the first parameter. This relieves your code from ever having to declare a local variable for your private data structure and to assign the LPOLESERVER value to it on entry.

72 Basic Server Methods

Basic server methods encompass all methods in the OLESERVERVTBL except for ServerExecute.

ServerCreate	ServerExit
ServerCreateFromTemplate	ServerOpen
ServerEdit	ServerRelease

73 ServerCreate

Use: OLESVR calls ServerCreate when a client application has created a new object with the client's Insert Object command (client calls **OleCreate**).

Parameters: pServer (LPOLESERVER) Points to the OLESERVER structure passed in OleRegisterServer.
lhDoc (LHSERVERDOC) Identifies the handle to associate

(store) with the document.
pszClass (LPSTR) Provides the name of the document class.
pszDoc (LPSTR) Provides the name of the document as used in the client application. However, use the strings from DocSetHostNames for user interface changes.
ppServerDoc (LPLPOLESERVERDOC) Where to store a pointer to the new OLESERVERDOC structure.

Responsibilities:
1. Create a document of the specified class.
2. Allocate and initialize an OLESERVERDOC structure.
3. Store lhDoc in the OLESERVERDOC structure.
4. Store a pointer to the new OLESERVERDOC structure in ppServerDoc.
5. Return OLE_OK if successful, OLE_ERROR_NEW otherwise.

Steps 1 and 2, creating, allocating, and initializing, may be lumped together. Single-document servers can have a single OLESERVERDOC structure as a global variable that is initialized during startup, so creating and initializing a document is already done. Any server may allocate memory here instead, as the Schmoo sample demonstrates. Also note that an MDI server generally creates a new window for the new document.

74 ServerCreateFromTemplate

Use: ServerCreateFromTemplate is a fancy name for initializing a new document with the contents of a file, although from here on out the file is not referenced anywhere. OLESVR calls ServerCreateFromTemplate when a client application has created a new object specifying a template with the OleCreateFromTemplate function.

Parameters: Same as ServerCreate, with an additional LPSTR, pszTemplate, pointing to the filename to use as the template.

Responsibilities:
1. Create a document of the specified class.
2. Read the contents of the specified file and initialize the document.
3. Allocate and initialize an OLESERVERDOC structure.
4. Store lhDoc in the OLESERVERDOC structure.
5. Store a pointer to the new OLESERVERDOC structure in ppServerDoc.
6. Return OLE_OK if successful, OLE_ERROR_NEW otherwise.

Except for returning OLE_ERROR_TEMPLATE instead of OLE_ERROR_NEW, this method is identical to ServerCreate.

75 ServerEdit

Use: ServerEdit is exactly ServerCreate except that when the ServerEdit method is called the application can expect an ObjSetData call. OLESVR calls ServerEdit when a client application

has activated an *embedded* object for editing and the server must create a new document. The DocGetObject method will create the object and ObjSetData will initialize the object's data. At this point the server is not visible.

Parameters:
- pServer (LPOLESERVER) Identifies the server.
- lhDoc (LHSERVERDOC) Identifies the handle to store with the document.
- pszClass (LPSTR) Describes the class of document to create.
- pszDoc (LPSTR) Provides the name of the document as used in the client application. However, use the strings from DocSetHostNames for user interface changes.
- ppServerDoc (LPLPOLESERVERDOC) Where to store the pointer to the new document.

Responsibilities:
1. Create a document of the specified class.
2. Allocate and initialize an OLESERVERDOC structure.
3. Store lhDoc in the OLESERVERDOC structure.
4. Store a pointer to the new OLESERVERDOC structure in ppServerDoc.
5. Return OLE_OK if successful, OLE_ERROR_EDIT otherwise.

76 ServerExit

Use: OLESVR calls ServerExit when a fatal error requires the server to immediately terminate. The server is always hidden whenever ServerExit is called so there will not be dirty files that need to be saved.

Parameters:
- pServer (LPOLESERVER) Identifies the server.

Responsibilities:
1. Hide the window to prevent any user interaction.
2. Call OleRevokeServer. Ignore an OLE_WAIT_FOR_RELEASE return value.
3. Perform whatever action is necessary to cause the application to terminate, such as DestroyWindow.
4. Return OLE_OK if successful, OLE_ERROR_GENERIC otherwise.

Example:

```
ShowWindow(pGlob->hWnd, SW_HIDE);

pServer->fRelease=FALSE;
os=OleRevokeServer(pServer->lh);

DestroyWindow(pGlob->hWnd);
```

77 ServerOpen

Use: OLESVR calls ServerOpen when the user activates a linked object in an OLE client and the client calls OleActivate. The server simply creates a new document with that file loaded, much

like ServerCreateFromTemplate, but leaves that file open, unlike ServerCreateFromTemplate that uses the file to initialize the data but discards the filename. The server is still hidden at the point when ServerOpen is called.

Parameters:
- pServer (LPOLESERVER) Identifies the server.
- lhDoc (LHSERVERDOC) Identifies a handle to store with the document.
- pszDoc (LPSTR) Provides the filename of the document to open.
- ppServerDoc (LPLPOLESERVERDOC) Where to store a pointer to the new document.

Responsibilities:
1. Create a document of the specified class.
2. Read the contents of the specified file and initialize the document.
3. Save the filename of the loaded file with this document, if necessary.
4. Allocate and initialize an OLESERVERDOC structure.
5. Store lhDoc in the OLESERVERDOC structure.
6. Store a pointer to the new OLESERVERDOC structure in ppServerDoc.
7. Return OLE_OK if successful, OLE_ERROR_OPEN otherwise.

78 ServerRelease

Use: OLESVR calls ServerRelease after the server has called OleRevokeServer and when the DDE conversation with the client has been successfully closed. This informs the server that there are no conversations to it or any document or object and that it is free to terminate at this point. ServerRelease also causes the server to terminate if it was started from the client in order to update links (with the **OleUpdate** call). Do not free the structure holding server information if you still refer to that structure after the release.

Parameters:
- pServer (LPOLESERVER) Identifies the server.

Responsibilities:
1. Set a flag to indicate that Release has been called.
2. If the application is hidden *and it* has not called OleRevokeServer itself, then we must use ServerRelease to instruct the application to terminate, by posting a WM_CLOSE or otherwise effective message and immediately returning OLE_OK.
3. Otherwise, free any resources allocated for this server, including documents if necessary, and possibly the structure in which you hold server information.
4. Return OLE_OK if successful, OLE_ERROR_GENERIC otherwise.

ServerRelease is a tricky method to handle because it may be called twice in the life of a server application. If you run the server and close it, you will call OleRevokeServer which will eventually call ServerRelease. However, your server may start hidden and stay hidden to perform an invisible update in a client application. OLESVR will call ServerRelease before any other methods, in which case you must tell yourself to close because no user interaction could

close the application.
Schmoo uses the server handle to determine whether or not to close itself on ServerRelease. When the method is called, Schmoo checks if the window is hidden *and* that a non-NULL value exists in pOLE->pSvr->lh. If so, it posts a WM_CLOSE to kill the application and returns OLE_OK:

```
if (!IsWindowVisible(hWnd) && 0!=pOLE->pSvr->lh)
   {
   PostMessage(hWnd, WM_CLOSE, 0, 0L);
   return OLE_OK;
   }
```

Posting a WM_CLOSE message is the favorable way to terminate the application, since it by default calls DestroyWindow and allows centralization of the application's cleanup code. DestroyWindow sends a WM_DESTROY to the application where most applications call PostQuitMessage. Do not call PostQuitMessage directly from this method because it will possibly bypass your application's necessary cleanup procedures.
When Schmoo calls OleRevokeServer itself, it first clears the pOLE->pSvr->lh value to tell ServerRelease that we actually did call OleRevokeServer. Now when ServerRelease is called, it frees any document it's holding. If we did not make this distinction, the invisible update case above might have freed the document before that document was released.

79 Document Methods

Basic document methods are those in the OLESERVERDOCVTBL except for DocExecute, DocSetColorScheme, and DocSetDocDimensions. Those methods are not necessary for basic OLE server operation, but the remaining five are:

DocClose
DocGetObject
DocRelease
DocSave
DocSetHostNames

The return value of all these methods is, like the server methods, the type OLESTATUS. For method stubs, simply return OLE_OK for all methods.

80 DocClose

Use: OLESVR calls DocClose when the document must be unconditionally closed when the client containing a link (embedding or linking) to that document shuts down. This method is *always* called before the DocRelease method.

Parameters: pDoc (LPOLESERVERDOC) Identifies the document to close.

Responsibilities:
1. Call **OleRevokeServerDoc**; resources are freed when OLESVR calls DocRelease.
2. Return the return value of OleRevokeServerDoc, which will generally be

OLE_OK.

When DocClose is called, take no action to notify the user. The client application handles that responsibility.

```
os=OleRevokeServerDoc(pDoc->lh);    //lh was stored in ServerCreate, ServerOpen, etc.
return os;
```

81 DocGetObject

Use: OLESVR calls DocGetObject whenever a client application creates an object through functions like **OleCreate**. If the pszObj parameter below is NULL, then DocGetObject is called for an *embedded* object after ServerCreate, ServerEdit, or ServerCreateFromTemplate. If pszObj is non-NULL then ServerOpen was used to open a linked object.

Parameters:
- pDoc (LPOLESERVERDOC) Identifies the document affected.
- pszObj (LPSTR) Specifies the name of the object to create; if NULL, then OLESVR requests the entire document.
- ppObj (LPLPOLEOBJECT) Where to store a pointer to the new object.
- pClient (LPOLECLIENT) Identifies the client that will connect to this object.

Responsibilities:
1. Allocate and initialize an OLEOBJECT structure.
2. Store pClient in the object's OLEOBJECT structure for use in sending notifications to the client.
3. Store a pointer to the new OLEOBJECT structure in ppObj.
4. Return OLE_OK if successful, OLE_ERROR_NAME if pszObj is not recognized, or OLE_ERROR_MEMORY if the object could not be allocated.

The pClient pointer is not the same as a pointer that a client application passed to OLECLI. This client structure resides in OLESVR and acts on behalf of the client application. You *must* save this pointer so you can send notifications such as OLE_CHANGED to the pClient->lpvtbl->CallBack function.

82 DocRelease

Use: OLESVR calls DocRelease when all DDE conversations to the object have been closed, after the server has called OleRevokeServerDoc or OleRevokeServer. There will be no more calls to the document methods for this document after DocRelease. The document can free any resources or objects at this time but do not free the structure in which you hold additional document information if you still refer to this structure after the release.

Parameters:
- pDoc (LPOLESERVERDOC) Identifies the document released.

Responsibilities:
1. Free any resources allocated for this document and possibly the structure in which you hold document information.
2. Set a flag to indicate that Release has been called.
3. Return OLE_OK if successful, OLE_ERROR_GENERIC otherwise.

Example: The Schmoo server stores ATOMs for the document and client names in its SCHMOODOC structure and frees them on DocRelease:

```
if (NULL!=pDoc->aName) //Created in DocSetHostNames
        {
        DeleteAtom(pDoc->aName);
        pDoc->aName=NULL;
        }

if (NULL!=pDoc->aClient) //Created in DocSetHostNames
        {
        DeleteAtom(pDoc->aClient);
        pDoc->aClient=NULL;
        }
```

83 DocSave

Use: When a client application is closing and the user is saving the client's document that contains a linked object, OLESVR directs the server to save the linked file by calling the DocSave method. This insures that the data currently displayed in the client is saved when the client document is saved. OLESVR only uses this method when a server is editing a *linked* object, so it assumes that the server already knows a filename.

Parameters:
pDoc (LPOLESERVERDOC) Identifies the document to save.

Responsibilities:
1. Save the document to the known filename. How you save documents is application-specific.
2. Return OLE_OK if the save is successful, OLE_ERROR_GENERIC otherwise.

84 DocSetHostNames

Use: OLESVR calls DocSetHostNames to provide the server with the name of the client's document and the name of the object in the *client* application. These names are used to make the necessary window title bar and menu changes as described above in **UI: Change the Window Title and Menu**. OLESVR only calls this method for *embedded* objects.

Parameters:
pDoc (LPOLESERVERDOC) Identifies the document affected.
pszClient (LPSTR) Provides the name of the client application document.
pszObj (LPSTR) Provides the name of the object in the *client* application.

Responsibilities:
1. Change the title bar to reflect the embedded state with the appropriate names.
2. Change the File menu to reflect the embedded state and the name of the client application.
3. Store the object and client names in the OLESERVERDOC structure. These will be needed later for message boxes where the name of the client application must be displayed.
4. Return OLE_OK if all is successful, OLE_ERROR_GENERIC otherwise.

Once you have implemented this method, you can modify your previous work for the title bar and menus to reflect the names in pszClient and pszObj passed to this method. In addition, if you store names as ATOMs be sure to free any existing ATOMs before adding a new one, otherwise you will have a small memory leak.

Example:

```
if (NULL!=pDoc->aObject)
        DeleteAtom(pDoc->aObject);

pDoc->aName=AddAtom(pszDoc);

if (NULL!=pDoc->aClient)
        DeleteAtom(pDoc->aClient);

pDoc->aClient=AddAtom(pszClient);
```

85 Object Methods

Of the Object methods, ObjSetBounds, ObjSetColorScheme, and ObjSetTargetDevice are not necessary for basic operation of an OLE server, leaving only seven basic methods, each of which is very straightforward to implement:

ObjDoVerb	ObjRelease
ObjEnumFormats	ObjSetData
ObjGetData	ObjShow
ObjQueryProtocol	

86 ObjDoVerb

Use: OLESVR calls ObjDoVerb when the client application has called **OleActivate** on an *embedded* object. This method receives a verb identifier and must perform the necessary actions to execute that verb.

Parameters:
- pObj (LPOLEOBJECT) Identifies the object affected.
- iVerb (WORD) Index to the verb to execute.
- fShow (BOOL) Indicates if the server should show the object (TRUE) or remain in its current state (FALSE)
- fFocus (BOOL) Indicates if the server should take the focus (TRUE) or leave the focus unaffected (FALSE)

Responsibilities: 1. Execute the verb.

a. For a 'Play' verb, a server does not generally show its window or affect the focus.

b. For an 'Edit' verb, show the server's window and the object if fShow is TRUE, and take the focus if fFocus is TRUE. An ideal way to accomplish this is to call the ObjShow method *through the OLEOBJECTVTBL* since that method will handle showing the object and taking the focus itself.

c. An 'Open' verb is not clearly defined; depending on the application it may mean the same as 'Play' or 'Edit.' The Schmoo server, if it had an 'Open' verb, would treat it like 'Edit.'

2. Return OLE_OK if the verb was successfully executed, OLE_ERROR_DOVERB otherwise.

OLESVR will call the ObjSetData method before ObjDoVerb so the object has data to edit or play when so instructed through this method.

Example:

```
switch (iVerb)
   {
   case OBJVERB_EDIT:
      /*
       * Schmoo's edit is the same as just showing the object.  Call ObjShow
       * through the LPOLEOBJECTVTBL.
       */
      if (fShow)
         return (pObj->pvtbl->Show)((LPOLEOBJECT)pObj, fShow);
      //Return OLE_OK
      break;

   case OBJVERB_PLAY:
      //Unsupported (but perhaps known) verb.
      return OLE_ERROR_DOVERB;

   default:
      //Unknown verb.
      return OLE_ERROR_DOVERB;
   }

return OLE_OK;
```

87 ObjEnumFormats

Use: On various requests from OLECLI, OLESVR will ask the server for the types of data it can render for an object. OLESVR will make multiple calls to ObjEnumFormats until the method returns the format that OLESVR is looking for.

Parameters: pObj (LPOLEOBJECT) Specifies the object affected.
cf (WORD) Indicates the last format returned by this method. If cf is zero, then this is the first call to this method in a

series. The order in which formats are returned must be the same as the order that data is placed on the clipboard.

Responsibilities:
1. Depending on cf, return the 'next' clipboard format in which the server can render the object's data.
2. If there are no more supported formats after the format in cf, return NULL.

Example: A simple series of **if** statements determines the order of formats. The logic, not the order of appearance of the **if** statements, determines the order, which must match the ordering of placing the same data on the clipboard. Note also that you cannot use a **switch** statement since the format identifiers from RegisterClipboardFormat (like cfNative) are not constants:

```
if (0==cf)
        return pOLE->cfNative;

if (pOLE->cfNative==cf)
        return pOLE->cfOwnerLink;

if (pOLE->cfOwnerLink==cf)
        return CF_METAFILEPICT;

if (CF_METAFILEPICT==cf)
        return CF_BITMAP;

if (CF_BITMAP==cf)
        return pOLE->cfObjectLink;

//This IF is here just to be explicit.
if (pOLE->cfObjectLink==cf)
        return NULL;

return NULL;
```

88 ObjGetData

Use: Through ObjGetData, OLESVR requests the server to render an object in a specific format, such as Native or CF_METAFILEPICT. These requests occur any time the client needs to display an object or when the data must be written to a client file.

Parameters:
- pObj (LPOLEOBJECT) Specifies the object in question.
- cf (WORD) Identifies the data format requested.
- phData (LPHANDLE) Where to store the handle to the allocated data.

Responsibilities:
1. Allocate the requested data through GlobalAlloc (with GMEM_MOVEABLE *and* GMEM_DDESHARE). The exception is data for CF_BITMAP that uses a call like CreateBitmap.
2. Lock and fill the memory with the appropriate data.
3. Unlock the memory and store the handle in *phData.
4. Return OLE_OK if successful, OLE_ERROR_MEMORY otherwise.

Example: Some time ago in this document I recommended that you implement a routine or set of routines to create the necessary data structure for each format, specifically because OLESVR requests ObjGetData to return any data format at any time. Having a function to call for each format reduces ObjGetData to a trivial sequence of calls:

```
if (pOLE->cfNative==cf)
        hMem=HGetPolyline(pGlob->hWndPolyline);    //Polyline is Schmoo's native data.

if (CF_METAFILEPICT==cf)
        hMem=HGetMetafilePict(pGlob->hWndPolyline);

...

if (NULL==hMem)
   return OLE_ERROR_MEMORY;

*phData=hMem;
return OLE_OK;
```

89 ObjQueryProtocol

Use: OLESVR calls ObjQueryProtocol simply to determine which protocols the server supports: **StdFileEditing** and possibly **StdExecute**.

Parameters: pObj (LPOLEOBJECT) Specifies the object affected.
pszProtocol (LPSTR) Provides the name of the protocol.

Responsibilities:
1. If the protocol in pszProtocol is supported, return a pointer to an OLEOBJECT structure that contains an appropriate VTBL for that protocol, such as the pObj passed to this method.
2. If the protocol is not supported, return NULL.

Example:

```
//lstrcmp returns 0 if the two strings are identical.
if (0==lstrcmp(pszProtocol, "StdFileEditing"))
        return (LPVOID)pObj;

return NULL;
```

90 ObjRelease

Use: OLESVR informs an object that it is no longer connected to any client, after the client calls **OleDelete** or the server calls **OleRevokeServer**, **OleRevokeServerDoc**, or **OleRevokeObject**. Since this is the last object method that OLESVR calls for a given object, the application can free any resources for that object, but do not free the structure in which you hold additional object information if you still refer to this structure after the release.

Parameters: pObj (LPOLEOBJECT) Specifies the object being released.

Responsibilities:
1. Free any resources allocated for this object and possibly the structure in which you hold object information.

2. Set a flag to indicate that Release has been called.
3. NULL any saved LPOLECLIENT stores in the OLEOBJECT structure.
4. Return OLE_OK if successful, OLE_ERROR_GENERIC otherwise.

91 ObjSetData

Use: When OLESVR launches a server to edit an *embedded* object, it calls ObjSetData to provide the server with the data that is embedded in the client. ObjSetData is about the most important method in OLE as OLESVR calls ObjSetData before it calls other methods like ObjDoVerb and ObjShow. ObjSetData is the single method through which a server receives embedded data.

Parameters:
- pObj (LPSCHMOOOBJECT) Identifies the object affected.
- cf (WORD) The format of the data contained in hData.
- hData (HANDLE) A handle to global memory containing the data, allocated with GMEM_DDESHARE and GMEM_MOVEABLE.

Responsibilities:
1. If the data format is not supported, return OLE_ERROR_FORMAT.
2. Attempt to GlobalLock the memory to get a pointer to the data. If GlobalLock returns NULL, return OLE_ERROR_MEMORY.
3. Copy the data to the object identified by pObj.
4. Unlock and **GlobalFree** the data handle. The ObjSetData method is responsible for the memory.
5. Return OLE_OK.

Example: Schmoo accepts only "Native" through ObjSetData, copying that data to the object structure and passing the data to the Polyline window that was written to accept data through a window-specific message, PLM_POLYLINESET:

```
LPPOLYLINE      lppl;

//Check if we were given Native data.  We don't support anything else.
if (pOLE->cfNative!=cf)
                return OLE_ERROR_FORMAT;
lppl=(LPPOLYLINE)GlobalLock(hData);

//CHECK the return from GlobalLock since we don't know where this handle has been.
if (NULL==lppl)
                return OLE_ERROR_MEMORY;

//Set the data through the editing window.
SendMessage(pGlob->hWndPolyline, PLM_POLYLINESET, TRUE, (LONG)lppl);

//Server is responsible for freeing the data.
GlobalUnlock(hData);
GlobalFree(hData);
return OLE_OK;
```

92 ObjShow

Use: OLESVR calls ObjShow when an object must become visible, by making the server window visible and possibly scrolling the object into view. If the object is selectable (such as a range of cells in a spreadsheet), select the object as well.

Parameters:
- pObj (LPOLEOBJECT) Identifies the object to show.
- fTakeFocus (BOOL) Indicates if the server should SetFocus to itself or leave the focus unaffected.

Responsibilities:
1. Show the application window(s) if not already visible.
2. Scroll the object identified by pObj into view, if necessary.
3. Select the object if possible.
4. If fTakeFocus is TRUE, call SetFocus with the main window handle.
5. Return OLE_OK if successful, OLE_ERROR_GENERIC otherwise.

Example: (pGlob->hWnd contains the main window handle)

```
//Since we only have one object, we don't care what's in pObj.
ShowWindow(pGlob->hWnd, SW_NORMAL);

if (fTakeFocus)
   SetFocus(pGlob->hWnd);

return OLE_OK;
```

Now is a good time to compile all the new code added for basic methods so you can verify syntax and compilation. However, before you can successfully test the methods, you must implement a simple shutdown procedure where you call **OleRevokeServer**. After this simple addition, described in the next section, you can test your server handling embedded objects.

93 Handle Simple Shutdown

Closing the application requires that you call **OleRevokeServer** and perform application shutdown, such as freeing the thunks in the VTBLs created during initialization. OleRevokeServer automatically revokes any documents which revokes any objects. The other revoke functions, **OleRevokeServerDoc** and **OleRevokeObject** will be discussed in a later section. The best place to call OleRevokeServer is from the main application window's WM_CLOSE message case, just before calling DestroyWindow.

Any of the revoking functions may return OLE_WAIT_FOR_RELEASE in which case you would normally have to enter a message processing loop (see below). However, when calling OleRevokeServer you will not make any more OLE function calls and must assume that all OLE conversations are finished.

When we discuss handling File Menu commands later, you will at some points call

OleRevokeServerDoc. If that returns OLE_WAIT_FOR_RELEASE, you want to enter a message loop that does not exit until the DocRelease method has been called. This message loop allows DDE messages to be processed (which is critical to having DocRelease called) which not allowing the server to execute past a particular point. This is how OLE takes an asynchronous protocol like DDE and let's the application synchronize OLE calls.

The FOLEReleaseWait function below (in Schmoo's OLEMISC.C) demonstrates how to process messages while waiting for a flag, pointed to by pf, to change to TRUE. Since FOLEReleaseWait takes a pointer to any BOOL, you can use the function to wait for ObjRelease, DocRelease, or ServerRelease by passing a pointer to the fRelease flag contained in any of the application-specific OLEOBJECT, OLESERVERDOC, or OLESERVER structures.

FOLEReleaseWait uses PeekMessage to provide a space for background processing. If your application does no background processing, then replace PeekMessage with GetMessage and remove the **else** clause containing the WaitMessage. Do not concern yourself with OleUnblockServer for now.

```
BOOL FAR PASCAL FOLEReleaseWait(BOOL FAR *pf, LONG lhSvr)
   {
   MSG msg;
   BOOL fMsg=FALSE;

   *pf=FALSE;

   while (FALSE==*pf)
      {
      OleUnblockServer(lhSvr, &fMsg);

      //Process normal messages once we've cleared up the server queue.
      if (!fMsg)
         {
         /*
          * We use PeekMessage here to make a point about power
          * management and ROM Windows--GetMessage, when there's no more
          * messages, will correctly let the system go into a low-power
          * idle state.  PeekMessage by itself will not.  If you do
          * background processing with PeekMessage and have nothing to do,
          * call WaitMessage to let Windows detect the idle state.
          */

         if (PeekMessage(&msg, NULL, NULL, NULL, PM_REMOVE))
            {
            TranslateMessage (&msg);
            DispatchMessage (&msg);
            }
          else
            {
            WaitMessage();
            }
         }
      }

   return *pf;
   }
```

Note the use of **OleUnlblockServer** here because the call must appear in all message loops if you use OLE's Block/Unblock feature. See the section **Optional Methods and OLE Functions** later in this document.

Once you have added the single call to OleRevokeServer in your WM_CLOSE case, you are ready to compile and test your server with embedded objects.

After this compilation, you are ready to begin testing all the basic functionality of your OLE server. First, open the server stand-alone, create data, copy it to the clipboard, close the server, and paste it into a suitable client document. Then activate the object in the client for editing, launching your server. In a debugger with breakpoints on each method, you can begin to see the sequence of method calls and can verify that the correct actions are happening. When all goes well, your server will appear with the data from the embedded object.

The same goes for linking: start your application stand-alone, open a file, copy data to the clipboard, close the server, and paste the data into an appropriate client. When you activate the object in the client, you will see your server start and can observe what happens when OLESVR calls your methods.

At this time you can also verify the server's user interface, insuring that you change the title bar and File menu when the server starts for embedding. What we cannot do is properly update the object in the client, which is where the File Menu commands come in.

94 File Menu Commands: New, Open, Save [Copy] As, and Save/Update

When a full server carries out the File New, File Open, File Save [Copy] As, and File Save/Update commands, it must inform OLESVR about what is happening to its documents. This section describes the steps necessary to handle each case.

In an MDI server, the New or Open commands do not affect the window containing an embedded object as they simply create a new window. This eliminates the need to prompt the user to update the object.
In a single-document server, the File New and File Open commands break the link between the client application and the server document. The File Save Copy As command saves a silent copy of an embedded object without breaking the connection to the client application. If the object is linked, File Save As just informs the server that the document was renamed.

Before addressing the menu commands, two topics need brief treatment: maintaining a 'dirty' flag for the document and notifying the client. Also note that under the **File New and File Open** section is a special mention about an optional **File Import** command.

95 When to Consider the Document as Dirty

Before diving into each case, 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. An application normally tracks a dirty flag that is set or cleared on various conditions:

Flag	Condition
TRUE	OLESVR calls the ServerCreate method.
TRUE	OLESVR calls the ServerCreateFromTemplate method.
TRUE	A user action in the server changes the document.
TRUE	(Optional) The user resizes the server window, resizing the server's data.
FALSE	OLESVR calls the ServerEdit method.
FALSE	OLESVR calls the ObjGetData method requesting the "Native" data format.

The Schmoo server contains a single function, FDirtySet, to set or clear the dirty flag. The File menu commands are generally those that need to watch this flag in order to prompt the user to save changes.

The Microsoft *User Interface Style Guide* defines a standard message box for informing the user that an *embedded* object is dirty when the user *closes* a document (or the server). For a linked object, use what your application already has. In the message box below, CLIENT.DOC should be replaced by the client document name provided through the DocSetHostNames method.

96 Notifying the Client

File Save and File Update require that the server notifies the client (through OLESVR) of those actions but calling the OLECLIENTVTBL function CallBack; the pClient structure passed to DocGetObject contains a pointer to the client VTBL containing CallBack

Calls like **OleSavedServerDoc** generally handle these notifications automatically. The cases where the server must directly send a notification code to CallBack are listed below:

Notification Code	Condition
OLE_CHANGED	Any operation that changes the object, making it dirty. Not necessary for embedded objects.
OLE_SAVED	As a replacement for calling **OleSavedServerDoc**.
OLE_CLOSED	When the user *closes* a document with embedded objects that require updating (that is, dirty objects), before calling OleRevokeServerDoc.

The OLE_CHANGED notification works expressly for linked objects to allow the client application to dynamically the object as the user changes it in the server.

The Schmoo sample contains a function, OLEClientNotify (OLEMISC.C), that takes an LPOLECLIENT and a notification code and sends that notification to the CallBack referenced through the OLECLIENT pointer. It is only used in the File Exit (OLE_CLOSED) menu case and

from the FDirtySet function (OLE_CHANGED):

```
void FAR PASCAL OLEClientNotify(LPSCHMOOOBJECT pObj, WORD wMsg)
   {
   LPOLECLIENTVTBL pvt;

   if (NULL==pObj || NULL==pObj->pClient)
      return;

   pvt=pObj->pClient->lpvtbl;

   if (NULL==pvt)
      return;

   (pvt->CallBack)(pClient, wMsg, (LPOLEOBJECT)pObj);
   return;
   }
```

97 File New and File Open

1. (SDI Server only) Prompt the user to update the objects before proceeding. Updating takes the form of calling **OleSavedServerDoc**.
2. (SDI Server only) Close the old document and call **OleRevokeServerDoc**. If OleRevokeServerDoc returns OLE_WAIT_FOR_RELEASE, process messages until the DocRelease method is called..
3. Allocate, initialize, and register a new document with **OleRegisterServerDoc**.
4. For File Open, load the file as necessary.
5. (SDI Server Only) Reconfigure the user interface for a stand-alone server

File New and File Open follow the same sequence of steps, since File Open is really File New with the extra step of loading the contents of a file. Other than that, the two commands are generally equivalent.

Note that an SDI server that was performing embedding when the user chooses File New must reconfigure itself to appear as a stand-alone server by changing the title bar and menu back to the stand-alone interface: the new filename appears in the title bar and the File menu contains the standard "Save" and "Exit" items in place of "Update" and "Exit & Return to..." From this point on, the server operates as stand-alone. Be sure to reset any global flags in your application that track whether or not the server was started through OLE and whether it's linking or embedding.

98 File Import

Since the File Open command breaks the connection between the object in the server and the client application, it prevents users from importing data from existing files into an embedded or linked object. If you want to include this capability, your server application should support a File Import command. This command acts almost identically to File Open, but instead of renaming the server's document and breaking the connection to the client document, the command simply loads data from a file into the current object. That file is then closed and only referred to again when File Import is reused.

A File Import command may either fully replace the data in the object (as the Schmoo server does) or may simply add to it. If you wish to have both abilities, then you should support a File Import command (to fully replace the object's contents) and an Edit Paste From... command (to paste the

contents of a file into the object). As of the writing of this document, a standard interface for importing data from existing files has not been defined.

99 File Save [Copy] As

Save [Copy] As for an embedded object means to save a copy of the object into a file. For a linked object it means to save an untitled or existing file under a new name. Since File Save As affects a single document and does not create a new one, the steps are identical for SDI and MDI servers.

1. Retrieve the new filename.
2. Create and write the file.
3. If the object is embedded, you are done with this command. For linked objects, continue steps 4-6.
4. Call **OleRenameServerDoc** to inform OLESVR that the document has a new name.
5. Call **OleSavedServerDoc** to inform OLESVR that the document has been saved.
6. Change the window title bar to reflect the new filename for a linked file. If the object was embedded reconfigure the user-interface for a stand-alone server.

100 File Save/Update

1. If the object is linked, write the new data to a file.
2. Call **OleSavedServerDoc**.
3. Set your 'dirty' flag to FALSE.

If the server is running stand-alone and saves a file, it should simply send the OLE_SAVED notification to OLESVR as described above.

Handling the File Update command is quite simple, but once you call **OleSavedServerDoc** you will see that many of the methods, like ObjGetData, are called from OLESVR. The majority of the work in updating an object is handled through your various methods.

101 Verify File Commands

With all the File menu commands implemented, you are ready to test each of those commands as well as linked objects. To test a linked object, start the server stand-alone, edit and save a file, then copy an object to the clipboard and Paste Link it into a suitable client application. Close the server and double-click in the client to activate that linked object. Your server should start with *-Embedding <filename>* on the command line. As you make changes, verify that the changes are being noted to the client (sending the OLE_CHANGED notification). Whenever a change occurs and you send the notification, you will notice calls to your methods to retrieve the updated data.

When editing an embedded object, verify that the proper message boxes appear when necessary and that you are successfully registering new documents when closing an existing one. In addition, verify that updating the object causes the client to display the new data and that File Save Copy As makes a copy of the object without affecting the object being edited.

Overall, test your server with each menu command under stand-alone, linking, and embedding cases. If you have a multiple-instance, SDI server, you may also want to verify that when you edit an embedded object then create a new one (which should revert the server back to stand-alone configuration), double-clicking on an object in a client starts a new instance of your application to edit that other object. If a new instance does not start, check that you are *always* registering a document (regardless of whether or not it's untitled) and check your use of OLE_SERVER_MULTI and OLE_SERVER_SINGLE in your OleRegisterServer call.

102 Closing Objects, Documents, and the Server

At points in the life of the server, it may close an object (by deleting it), a document containing objects (by closing the file, with or without saving changes), or the server itself. Remember to call FreeProcInstance on the methods in the VTBLs when you terminate the application.

Closing any one of the three OLE items means to *revoke* it, or to terminate any conversations that any client may have with that item. *Release*, on the other hand, means to inform the item that no client document is connected to it, so it can perform any final cleanup. There are three revoke APIs in OLESVR to handle these cases:

Item Closing	**OLESVR Call**
Object	OleRevokeObject
Document	OleRevokeServerDoc
Server	OleRevokeServer

Note that **OleRevokeServerDoc** has the effect of revoking all objects in that document, and **OleRevokeServer** has the effect of revoking all documents *and* objects in that server. Any of the Revoke calls may return OLE_WAIT_FOR_RELEASE, and when receiving this return value, enter a message loop like that in Schmoo's FOLEReleaseWait until that particular item's Release method is called. Also note the special circumstance where the ServerRelease method may be called twice, as described in section 5.7.2.6 above.

When the user closes the server with an embedded object and that object requires updating, display the message box shown in the previous section. If the user chooses to save changes, then send the OLE_CLOSED notification to OLESVR through the OLECLIENT pointer given in DocGetObject and call **OleRevokeServerDoc**. If the user does not want to save changes, simply call **OleRevokeServerDoc**. An example of this can be seen in Schmoo's FCleanVerify function.

After successfully compiling and linking your server, verify that revoking objects, documents, and

the server occur at the proper time and that the server waits until the appropriate Release method is called. In addition, verify that you properly close the server when OLESVR calls ServerRelease and the application is hidden. The case where this normally occurs is when a client application updates a link, so in order to test this, open a suitable client application and create a linked object from your server. Save the client file and reload it, which will prompt you to update links. Updating the link to your server will start a hidden instance of it and eventually call ServerRelease. You can use the Windows SDK HeapWalk utility to verify that your servers modules are indeed freed from memory once the link update is complete.

103 Optional Methods and OLE Functions

This section discusses how to handle the remainder of the Server, Document, and Object methods that are optional for your server. Minimal stubs must exist for these methods, but they do not necessarily have to contain code. It also discusses the remaining OLESVR calls, such as OleBlockServer and OleRevertServerDoc.

104 Document and Object SetColorScheme

DocSetColorScheme and ObjSetColorScheme provide the document or object with a set of colors suggested by the client for foreground, background, fill, and lines. **This is NOT, repeat NOT, however, a palette from which you should call CreatePalette and RealizePalette.** The LPLOGPALETTE parameter to this function points to a LOGPALETTE structure, but the colors in the structure are not those to send to the video hardware. Instead, they are colors suggested from the client that the *server can display for color choices when editing the embedded or linked object.*

In the LOGPALETTE structure, the palNumEntries field contains the number of total colors in this color scheme. The palPalEntry array contains those colors. The first color is the suggested foreground color and the second is the suggested background color. The first half of the remaining colors are suggested fill colors, with the second half containing suggested line colors. If there are an odd number of entries, then give the extra color to the fill colors, that is, there is one less line color than fill colors. If there are 9 entries in the entire color scheme, use the first two for foreground and background, the next **4** for fill colors, and the final 3 for line colors.

If you are familiar at all with Microsoft PowerPoint for Windows, you know that is has a "color scheme" that consists of a foreground color, background color, fill colors, and line colors, that it displays on menus. These are the colors that all slides in the PowerPoint presentation are using. When you edit an object embedded in PowerPoint, it may send a color scheme to the server, which can then provide those colors as choices for editing.

If the server will make use of these colors, apply (or save) those colors to whatever color selection menus or dialogs the application displays and return OLE_OK. If the server does not manipulate colors with a palette, return OLE_ERROR_PALETTE.

Note that Microsoft PowerPoint is one of the few client applications that will use your SetColorScheme method.

105 DocSetDocDimensions

DocSetDocDimensions informs the document that the user changed the object's size in the client application. The server is not required to resize itself or the object accordingly, but doing so keeps the object in both client and server consistent.

DocSetDocDimensions receives an LPOLESERVERDOC parameter and a pointer to a RECT structure containing the size of the object in the client, given in MM_HIMETRIC units. *Be sure to convert these coordinates to the units manipulated by your server!* From this method, convert the units into those you need and carry out any object/document resizing, as the sample server does in OLEDOC.C:

```
OLESTATUS FAR PASCAL DocSetDocDimensions(LPSCHMOODOC pDoc, LPRECT pRect)

  {
  /*
   * OLESVR will call this method when the client has resized the
   * object.  In this case we try to make the parent window the correct
   * size to just contain the object.
   */

  //First, convert the rectangle to units we can deal with MM_TEXT.
  RectDeviceConvert(pGlob->hWnd, pRect);

  /*
   * Tell the Polyline document to use this rectangle, also notifying
   * the parent which will then resize itself.
   */
  SendMessage(pGlob->hWndPolyline, PLM_RECTSET, TRUE, (LONG)pRect);
  return OLE_OK;
  }
```

RectConvertToDevice is a private function that converts the MM_HIMETRIC units into MM_TEXT. The document resizing is done by sending a private message to the document window (hWndPolyline) with the new rectangle.

Note also that while resizing the document through user manipulation could be considered an action that *changes* the document (makes it dirty), resizing through the DocSetDocDimensions method should not, since the client object is already the given size, that is, the server and client are synchronized.

106 ObjSetBounds

In OLE version 1.x, ObjSetBounds is not used. If it were, handle it like you would handle DocSetDocDimensions, sizing only an object within a document and not the entire document itself.

107 ObjSetTargetDevice

The ObjSetTargetDevice method is called to inform the object that any rendering through the ObjGetData call should be for the screen or a printer. The hData parameter is either NULL or contains an OLETARGETDEVICE structure as defined in OLE.H.

If hData is non-NULL, then use the data in this structure to extract specific device information. If

your application can create a better metafile or bitmap for a specific device, it should use this method to detect that change and optimize the presentation. If you create the same metafile or bitmap for screen or printer, then there is no need to implement this method. This method is most useful for applications that deal with device fonts.

Note that once this method is called, any subsequent presentations created through the ObjGetData method should be rendered for the given device until ObjSetTargetDevice is called with a different device in hData (which might be NULL).

108 Server Execute

A server that supports the **StdExecute** protocol (by being both registered in the registration database as supporting StdExecute and returning an LPOLEOBJECT in ObjQueryProtocol for StdExecute) can expect to receive DDE-style execute strings through the ServerExecute method. This method is a central location to parse and execute these strings.

Execute strings are entirely application-specific. For example, the Windows Program Manager, although not an OLE server, supports a DDE Execute string "[CreateGroup(OLE Test Applications)]" which causes it to create a new group window. These execute strings generally instruct the receiving application to perform a set of command, not usually to generate and return data other than a TRUE or FALSE.

StdExecute is not an extremely important part of OLE at this point, but expect its importance to grow in the future where the Windows system will have a common macro language. The StdExecute protocol is central to providing an external macro language access to your application's operations. This has a great effect on application design where it must be structured to support operations not initiated by user interaction.

109 Blocking Requests (optional)

If you have a situation where your server is carrying out a long operation and does not want OLE in the picture, use the **OleBlockServer** function to instruct OLESVR to queue any OLE messages destined for the server. When the server is ready to process these requests, call **OleUnblockServer** prior to GetMessage in your main message loop *and* in any message loop used to wait for a Release. OleUnblockServer causes all queued requests to be processed before GetMessage can retrieve the next message. Alternately, return **OLE_BUSY** from any method that is called when you want to block requests. Returning OLE_BUSY allows the server to choose which requests to block.

Implementing a blocking mechanism requires that you maintain a global flag indicating you are ready to unblock. When you wish to block, call **OleBlockServer**. When you wish to unblock, set this flag to TRUE. In your message loops, call **OleUnblockServer** if this flag is true, and repeat until there are no more blocked requests, at which time you set the global flag back to FALSE.

A typical case where a server would use **OleBlockServer** is when a modal dialog appears. A modal dialog has its own message-processing loop inside Windows, so any DDE messages going between the OLE libraries will cause calls to your server's methods. This may cause unwanted actions while a dialog box is displayed, so block the server at that point, not allowing OLESVR to make any requests to your application until you unblock.

110 OleRevertServerDoc

OleRevertServerDoc is a special OLESVR function for applications that allow the user to open a file and make changes, but then reload the original file, discarding any changes. If you provide this sort of functionality, call the **OleRevertServerDoc** function when you reload a file without closing it.

Appendix A: Definitions

Term	*Definition*
Client	(or Client Application) An application that creates and edits compound documents containing objects from one or more server applications. Clients only store objects; servers actually edit them.
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.
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.

Term	*Definition*
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.
OLECLI.DLL	The OLE Client Library, that contains OLE API used by client applications.
OLESVR.DLL	The OLE Server Library, 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.

Term	*Definition*
Registration Database	The system database that holds names of applications that support the OLE protocol, the full pathnames to those application, the objects they can edit, what verbs those objects support, and whether or not 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 communication between a client application and a server, document, or object. When one of these items is revoked, the item will eventually become released. A client may also revoke communication between it and any objects it contains.
Server	(or Server Application) An application that creates and edits objects for storage in a client application's compound document.
SHELL.DLL	A dynamic link library that contains functions to manipulate the registration database.
Source	Synonym for Server.
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.