

Open Document Management API

Version 1.0a

Revision history:

Version 1.0 - with small BHC-325 at the end of the document
Brad Clements. This is the original version.

Version 1.0 - with small DM1\285 at the end of the document
Mike Gardiner. This contains additional information for registering
ODMA under Windows 95 and NT. It also adds a new table for
document ID constants.

Version 1.0a

Rod Schiffman. Additional information on the ODMA spec
background, usage of calls, suggestions on the use of document ID
character sets and the addition of `ODM_E_INUSE` as a valid return
code in `ODMActivate`.

The purpose of this update to the ODMA specification is to clarify a number of issues that have been raised by companies that have implemented ODMA in their products. It does not change any of the calls or functionality. It is a preliminary step to creating the next version of the specification. There is additional information about the usage of the calls, as well as more information about the goals of ODMA. The document, perhaps even more than in the past, retains an informal writing style.

The original impetus for ODMA was the recognition that there was no standard method for a client application to integrate with a DMS. Each DMS vendor wrote separate integration code for each of the major client applications they supported. Applications that did not have integrations written for them by the DMS vendors would have to write and support a separate integration for each DMS that was supported. This required a complete matrix of integrations, each with its own set of bugs, limitations and reliability issues. It seemed obvious that a high level standard for connecting applications and document management systems was a natural fit. A small group of application and DMS vendors started working together to create an API that would allow applications and document management systems to inter-operate through a single high level API. This implied the creation of a standard, however, the creation of a standard has many pitfalls. Probably the biggest problem is that a lot of work can be put into the creation of the standard, and nobody uses it.

The industry is filled with examples of standards that were obsolete by the time they made it through the standardization process. By time some standards make it through the standardization process, they are so large and unwieldy they are almost impossible to implement and maintain. Company politics and hidden agendas of the participants can play as big a role in the adoption of a standard as trying to solve the problem in the first place. The industry is also full of proprietary API's that claim to be standards, but are not. The initial group of vendors that met and formed the ODMA consortium wanted to avoid as many of these problems as possible.

The working rules of the ODMA consortium are fairly simple.

1. If the standard does not solve a problem it will not be used.
2. If the creation of the standard takes a long period of time, it does not solve the problem.
3. If the standard is difficult to implement, it does not solve the problem.
4. The standard must be vendor independent.
5. The standard must not try to solve all vendors problems, or it will be big, complex and take a long time to implement. This violates rules 1, 2 and 3 above.
6. It is the customers that lose if there is not a straightforward way to integrate applications that create documents, and applications that manage documents.
7. Easy integration between applications and document management systems will grow the industry and increase sales for the entire marketplace.

It is difficult to express the importance the initial members of the consortium placed on wanting to create a useful API that is vendor and platform independent while still simple to implement. They recognized that they could solve 80 percent of the problem easily and were willing to live with having to solve another 10 percent over time and probably never being able to solve the final 10 percent.

The Open Document Management API (ODMA) is a standardized, high-level interface between desktop applications and document management systems (DMSs). Its purposes are:

1. To make DMS services available to users of desktop application in a seamless manner so that these services appear to the user like extensions of the applications.

2. To reduce the application vendors' burden of having to deal with multiple DMS vendors. By writing to ODMA, an application vendor has potentially integrated his application with all supporting DMSs.
3. To reduce the DMS vendors' burden of integrating with multiple applications. By supporting ODMA a DMS vendor has potentially integrated with all applications that have written to ODMA.
4. To reduce effort and complexity needed to install and maintain DMSs.

ODMA specifies a set of interfaces that applications can use to initiate actions within a DMS. The API is intended to be relatively easy for application vendors to incorporate into updates of existing applications. It should not require major restructuring of an application to integrate it with ODMA. Note that this version of ODMA does not specify how DMSs may initiate actions within the applications.

The ODMA API is platform-independent. The associated data type definitions and binding information are platform-specific. Currently, most of the work has been done in Windows. It makes this document look Windows specific, but over time, the platform specific entries for other platforms will be added as they are defined.

Document IDs

Many of the ODMA functions accept or return a Document ID parameter. A document ID is a persistent, portable identifier for a document. It can be stored and used in a later session, and it can be passed across platforms via email or other processes.

Document IDs are case insensitive, null-terminated strings of printable characters. Although a document ID is case insensitive, an application should never change the case of a document ID. The format of a document ID is

`::ODMA\DM_ID\DM_SPECIFIC_INFO`

The DM_ID portion of a document ID will identify which DMS provided the ID. This information is primarily for the use of ODMA itself; applications using ODMA should not need to know which DMS provided a particular ID. The ODMA group members will coordinate these IDs to ensure their

uniqueness. The maximum length of the DM_ID portion of the document ID is specified by the constant **ODM_DMSID_MAX**.

The DM_SPECIFIC_INFO portion of the ID will vary depending on which DMS built the ID. The total length of the document ID including the terminating Null character cannot exceed **ODM_DOCID_MAX** bytes.

ODMA-aware applications should be able to handle a document ID anywhere they handle an externally-generated document filename. For example, if the application allows a document filename to be passed as a command line argument then it should allow a document ID to be passed in the same way. If the application allows document filenames to be used in DDE commands then it should also support the use of document IDs in the same commands.

Although the technical definition of a document ID is a *case insensitive, null-terminated strings of printable characters*, there are some general rules that are more likely to make a DMS and ODMA application work better together. ODMA was designed so that it would be easy to add to an application without major modifications in code or structure. If a DMS passes a document ID that breaks fundamental rules of normal file and path names it will probably run into problems if it is passed in on a command line. Special characters like ^, [,], |, *, -, >, < and ? are processed by the Unix shell even before they are seen by the application. It is possible to pass these characters by using special escape sequences, but that places a burden on the DMS vendor to process the document ID before giving it to the ODMA application. Some operating systems require the application to handle the reverse process of interpreting and removing the escape characters. The application may be able to support the escape removal on the command line, but not if the document ID with escape characters is returned in a procedure call. In most cases, it is easier to generate a document ID that contains a fairly simple set of characters. The following table suggests characters it may be wise to avoid for different platforms.

Platform	Characters to avoid
Windows 3.x	" ' < > * ? and the space character
Windows 95	" ' < > * ?
Other platforms to be defined	

Constants

CONSTANTS	Windows 3.x	Win32	Mac	Unix	Other
ODM_DOCID_MAX Maximum length of a document ID including the terminating Null character	80	255	255	255	255
ODM_FILENAME_MAX Maximum length of a path/filename returned by ODMA including the terminating Null character.	128	255	255	1024	255
ODM_API_VERSION The version of the ODMA API to which this header file corresponds. See the description of ODMRegisterApp .	100	100	100	100	100
ODM_DMSID_MAX Maximum length of a DMS ID including the terminating Null character.	9	9	9	9	9

Error Handling

Nearly all of the ODMA functions use the return value to indicate to the calling application whether the function succeeded, failed because the user canceled the operation, or failed for other reasons. The DMS is responsible for displaying informational error messages to the user where appropriate except when the **ODM_SILENT** flag is specified. The DMS must take care to return the appropriate error indication because applications may act differently depending on whether an ODMA call was canceled by the user or failed for other reasons. The calling application generally should not display error messages when an error value is returned from ODMA unless the **ODM_SILENT** flag was specified.

Connections and the ODMA Connection Manager

The ODMA connection manager is a small software module that sits between applications using ODMA and document management systems implementing ODMA. It manages the connections between these

components and routes ODMA calls to the appropriate provider. A freely-redistributable copy of the ODMA connection manager will be provided to any vendor wishing to implement or make use of the ODMA API. This is a place where it would be possible to provide mapping code that would allow ODMA to have truly platform independent document IDs, however, it currently only manages connections and does not touch document IDs.

Document Format Names

When new documents are registered with a DMS via ODMA and when an existing document's format is changed by an application, the application passes a document format name to ODMA. Document format names are Null-terminated strings defining the format of a document's content. Application vendors are encouraged to make public the format names they choose for their proprietary formats so that other ODMA users and providers can standardize on the same names. The **odma.h** header file contains names for common, non-proprietary formats such as text, RTF, and TIFF. This is a significant limitation in the ODMA 1.0 specification. There is a proposal in an appendix at the end of this document to create a standardization for document format names under Windows.

Character Sets

All strings passed to or returned from ODMA functions should be in the native character set of the system on which ODMA is being used. So for example, 8859-1 would be used on English Windows, Shift-JIS would be used on Japanese Windows, Unicode would be used on NT, etc. The term "Null-terminated" as used in this specification means terminated by the character set's natural Null character. For most character sets this means a single byte with the value 0x0; for Unicode it means a sequence of 2 bytes with the value 0x0. If an application obtains an ODMA document ID on one platform and later uses it on another platform, the application is responsible for translating the ID to the native character set of the second platform before using it there.

Application Interfaces

An ODMA-aware application can choose to communicate with the ODMA Connection Manager either through a traditional, function-oriented API or through Component Object Model (COM) interfaces. Prototypes and constants used for the function-oriented API are included in the **odma.h**

header file. Prototypes, constants, and interface definitions for the COM interface are included in the odmacom.h header file.

After calling **ODMRegisterApp** applications can obtain one or more COM interfaces to ODMA through the **ODMQueryInterface** function. The **IODMDocMan** interface provides an alternate entry point to most of the ODMA functions documented below. This interface and its interface ID are defined in the odmacom.h header file.

Note that IODMDocMan::QueryInterface will only query the default DMS for the calling application. The application must call **ODMQueryInterface** in order to query other DMSs.

ODMA API

ODMRegisterApp

ODMSTATUS ODMRegisterApp(ODMHANDLE FAR *pOdmHandle, WORD version, LPSTR lpszAppId, DWORD dwEnvData, LPVOID pReserved)

ODMRegisterApp registers an application with the appropriate Document Management System (DMS) and returns a handle that can be used in calls to other ODMA functions. An application must call this function before calling any of the other ODMA functions. A task may call **ODMRegisterApp** more than once; each call will return a different handle, each of which must be deregistered via **ODMUnRegisterApp**.

Parameters:

pOdmHandle - out - If successful, a handle is returned here that can be used in calls to other ODMA functions. If the registration fails then 0 is returned here.

version - in - Specifies the version of the API required by the application. 100 should be passed to indicate version 1.0, 110 should be passed to indicate version 1.1, etc. The macro **ODM_API_VERSION** can be used to get the correct version number at compile time. All versions of the ODMA API will be downward

compatible, so this should be interpreted as the minimum version number that the calling application expects the DMS to support. If the DMS does not support the specified version or a higher version then it should return an error.

lpzAppld - in - A unique identifier for the application. The maximum length for this string is 16 characters including the terminating Null, and it cannot begin with a digit. It is recommended that a Windows application use the File Manager ProgId for its primary document class, but this is not required.

dwEnvData - in - Environment data. On Windows platforms this is a Window handle for a parent window in the calling application. The DMS may use this window handle as the parent window for any dialogs or other windows it displays in response to ODMA calls. This handle must remain valid for the duration of the ODMA session (i.e. until `ODMUnRegisterApp` is called).

pReserved - in - Reserved for future use. Must be set to Null.

Return value: 0 is returned if successful. **ODM_E_NODMS** is returned if no Document Management System has been registered for the calling application. **ODM_E_CANTINIT** is returned if a DMS is registered for the calling application, but it fails to initialize itself. **ODM_E_VERSION** is returned if the DMS does not support the requested version of the API.

ODMUnRegisterApp

`void ODMUnRegisterApp(ODMHANDLE odmHandle)`

An application that previously registered itself with a DMS via **ODMRegisterApp** should call this function when it is finished using the DMS. This would typically be done when the application is shutting down. After this call returns, the DMS handle is no longer valid and cannot be used for subsequent calls to ODMA functions.

Parameters:

odmHandle - in - An ODMA handle obtained by a previous call to **ODMRegisterApp**.

ODMSelectDoc

ODMSTATUS ODMSelectDoc(ODMHANDLE odmHandle, LPSTR
lpzDocId, LPDWORD pdwFlags)

This function causes the DMS to return a document ID representing a document that has been selected for some action. Typically the DMS will display searching and other dialogs that allow the user to interactively select a document from among those managed by the DMS. An application would typically call this function whenever the user needs to select a document to be opened or imported.

Parameters:

odmHandle - in - An ODMA handle obtained by a previous call to **ODMRegisterApp**.

lpzDocId - out - A pointer to a buffer where the DMS will return the ID of the document selected by the user. This buffer needs to be at least **ODM_DOCID_MAX** bytes in length. If successful then a null-terminated document ID will be returned here. Otherwise the contents of the buffer will be undefined.

pdwFlags - in/out - On input, 0 or a combination of 1 or more of the following values:

ODM_SILENT - The DMS should not require user interaction while satisfying the call. If the call cannot be satisfied without user interaction then an error should be returned.

Upon return, one of the following flags will be set unless an error occurred:

ODM_MODIFYMODE - The user indicated that the selected document should be opened in a modifiable mode.

ODM_VIEWMODE - The user indicated that the selected document should be opened in a view-only mode.

Return value: 0 if successful. **ODM_E_CANCEL** if the user does not make a selection. **ODM_E_APPSELECT** if the user indicated that he wants to make a selection using the application's regular file selection

facilities rather than using the DMS. In this case the application should just display its regular selection dialog as though ODMA were not present. **ODM_E_OTHERAPP** if the user selected a document from a different application; generally the caller should treat this the same as **ODM_E_CANCEL**. **ODM_E_USERINT** if the **ODM_SILENT** flag was specified and the DMS could not make a selection without user interaction. **ODM_E_HANDLE** if *odmHandle* was invalid.

In the case that **ODM_E_APPSELECT** is returned and a document is opened through the application's regular file selection facilities, the following behaviors are recommended:

1. File | Save and File | Close should be handled as though the DMS were not present.
2. File | Save As should be handled through ODMA (specifically through the *ODMSaveAs* function). This will allow documents to be imported into the DMS.

ODMOpenDoc

ODMSTATUS ODMOpenDoc(ODMHANDLE odmHandle, DWORD flags, LPSTR lpszDocId, LPSTR lpszDocLocation)

This function causes the DMS to make a document available to the application. It performs any necessary pre-processing (mapping network drives, checking security, etc.) and then returns to the application a temporary filename that can be used to access the document during the current session. Note that this function does not open the document file; it merely makes the file temporarily available to the calling application. The application can then open, read, write, and close the file as needed. When the application is finished using the file, it should call **ODMCloseDoc**.

If **ODM_MODIFYMODE** is requested, the DMS may refuse the request if the user has view-only rights or if the document is currently checked-out to another user. It is recommended that the application retry the request specifying **ODM_VIEWMODE** in this case so that the user can at least view the document.

Parameters:

odmHandle - in - An ODMA handle obtained by a previous call to **ODMRegisterApp**.

flags - in - One or more of the following flags:

ODM_MODIFYMODE - The DMS should make the document available in a modifiable mode. This mode is assumed if **ODM_VIEWMODE** is not explicitly requested.

ODM_VIEWMODE - The DMS should make the document available in a view-only mode. Any changes made to the document will **not** be transferred back to the document repository. Note that it is invalid to specify both **ODM_VIEWMODE** and **ODM_MODIFYMODE** in the same call.

ODM_SILENT - The DMS should not require user interaction while satisfying the call. If the call cannot be satisfied without user interaction then an error should be returned.

lpszDocId - in - A document ID. This is typically obtained by a call to **ODMSelectDoc** or **ODMNewDoc**.

lpszDocLocation - out - A pointer to a buffer of at least **ODM_FILENAME_MAX** bytes in length. The DMS will store in this buffer a Null-terminated string indicating where the caller can access the document during the current session. Typically this will be the full path/file name of the specified document, but some document formats may dictate another type of location such as a directory name. If an error occurs then the contents of the buffer will be undefined.

Return value: 0 if successful. **ODM_E_ACCESS** if the user does not have the access rights requested (for example, modify mode was requested but the user only has view rights to the document). **ODM_E_INUSE** if the user is currently unable to access the document because it is checked out by another user; this differs from **ODM_E_ACCESS** in that it is expected that the user might be able to access the document in the specified mode at some point in the future. **ODM_E_DOCID** if the document ID is invalid or refers to a document that no longer exists. **ODM_E_USERINT** if the **ODM_SILENT** flag was specified and the DMS could not make the specified document available without user interaction. **ODM_E_FAIL** if the DMS is unable to make the

document accessible for any other reason. **ODM_E_HANDLE** if *odmHandle* was invalid.

ODMSaveDoc

ODMSTATUS ODMSaveDoc(ODMHandle odmHandle, LPSTR lpszDocId, LPSTR lpszNewDocId)

This function tells the DMS that the document should be saved to the document repository. An application would typically call this function after saving changes to the temporary file returned by **ODMOpenDoc**. A new document ID is returned to the application which should be used for all subsequent operations on the document. This ID replaces the previous document ID in the current session. The new ID may or may not be the same as the original ID. It will usually be the same unless the DMS saved the document as a new version or as a new document. If the new ID is different from the previous ID then the previous ID cannot be used subsequently without doing an **ODMOpenDoc** on it.

Parameters:

odmHandle - in - An ODMA handle obtained by a previous call to **ODMRegisterApp**.

lpszDocId - in - A document ID. This is typically obtained by a call to **ODMSelectDoc** or **ODMNewDoc**. This document must have been previously opened in modify mode by a call to **ODMOpenDoc**.

lpszNewDocId - out - A pointer to a buffer where the DMS will return the new ID of the document. This buffer needs to be at least **ODM_DOCID_MAX** bytes in length. If successful then a null-terminated document ID will be returned here. Otherwise the contents of the buffer will be undefined.

Return value: 0 if successful. **ODM_E_OPENMODE** if the document was not previously opened in modifiable mode. **ODM_E_FAIL** if the DMS is unable to save the document to the document repository. **ODM_E_HANDLE** if *odmHandle* was invalid.

ODMCloseDoc

ODMSTATUS ODMCloseDoc(ODMHANDLE odmHandle, LPSTR lpszDocId, DWORD activeTime, DWORD pagesPrinted, LPVOID sessionData, WORD dataLen)

An application that has opened a document by calling **ODMOpenDoc** must call **ODMCloseDoc** when it is finished using the document. The application should not call this function until after it has closed the document, because the DMS may move the document or make it inaccessible as a result of this call. Note that this function will not cause the document to be saved into the DMS's persistent repository unless **ODMSaveDoc** has been called previously.

Parameters:

odmHandle - in - An ODMA handle obtained by a previous call to **ODMRegisterApp**.

lpszDocId - in - A Null-terminated document ID. This is typically obtained by a call to **ODMSelectDoc** or **ODMNewDoc**. This document must have been previously opened by a call to **ODMOpenDoc**.

activeTime - in - If the application tracks time spent editing the document then it should pass the number of seconds here. Otherwise it should pass 0xFFFFFFFF.

pagesPrinted - in - If the application tracks the number of pages printed from this document during the current editing session, it should pass this number here. Otherwise it should pass 0xFFFFFFFF.

sessionData - in - The application may pass other information regarding the current editing session in this parameter. For example, an application might pass the number of keystrokes that were entered. The calling application is free to determine the format of this data, so DMSs that rely on this information will have to coordinate with each application supported. Null should be passed if the application has no meaningful information to pass through this parameter.

dataLen - in - The length of the data passed in the *sessionData* parameter. Ignored if *sessionData* is Null.

Return value: 0 if successful. **ODM_E_NOOPEN** if the document was not open. **ODM_E_HANDLE** if *odmHandle* was invalid.

ODMNewDoc

ODMSTATUS ODMNewDoc(ODMHANDLE odmHandle, LPSTR lpszDocId, DWORD dwFlags, LPSTR lpszFormat, LPSTR lpszDocLocation)

This function causes the DMS to create a new document profile and return the document ID for the new document to the calling application.

Parameters:

odmHandle - in - An ODMA handle obtained by a previous call to **ODMRegisterApp**.

lpszDocId - out - A pointer to a buffer where the DMS will return the ID of the document selected by the user. This buffer needs to be at least **ODM_DOCID_MAX** bytes in length. If successful then a Null-terminated document ID will be returned here. Otherwise the contents of the buffer will be undefined.

dwFlags - in - 0 or a combination of 1 or more of the following values:

ODM_SILENT - The DMS should not require user interaction while satisfying the call. If the call cannot be satisfied without user interaction then an error should be returned.

lpszFormat - in - A Null-terminated string naming the format of the new document's content. Note that this may be changed later via an **ODMSaveAs** call.

lpszDocLocation - in - Normally DMSs select the location for a new document. But if the document already exists and is large or resides on read-only storage then the calling application can use this parameter to tell the DMS where the document is currently stored. This is a hint to the DMS that the document should be left in this location. Note that some DMSs may ignore this hint and move the

document anyway. The calling application should **not** directly access the document in this location following the call to **ODMNewDoc**; it should use **ODMOpenDoc** to obtain a location for subsequent access to the document. In most cases the application should pass Null in this parameter to allow the DMS to determine the document's storage location.

Return value: 0 if successful. **ODM_E_CANCEL** if the user cancels the creation of the new document. **ODM_E_FAIL** if the DMS failed to create the new document. **ODM_E_USERINT** if the **ODM_SILENT** flag was specified and the DMS could not make the specified document available without user interaction. **ODM_E_APPSELECT** if the user indicated that he wants to select the document's filename using the application's regular file selection facilities rather than using the DMS. In this case the application should just display its regular filename selection dialog as though ODMA were not present. **ODM_E_HANDLE** if *odmHandle* was invalid.

ODMSaveAs

ODMSTATUS ODMSaveAs(ODMHANDLE odmHandle, LPSTR
IpszDocId, LPSTR IpszNewDocId, LPSTR IpszFormat,
ODMSAVEASCALLBACK pcbCallBack, LPVOID pInstanceData)

This function causes the DMS to return a new document ID for a document that is based on an existing document. An application would typically call this function in response to the user selecting a SaveAs menu option.

ODMSaveAs causes the DMS to display options to the user for selecting the destination for the new document. This might be an entirely new document or a new version of the current document.

Often the application will want to present additional options to the user at this point such as different file formats or encrypting the document. This is accomplished via the *pcbCallBack* parameter. ODMA implementors should provide a method for users to access this function if desired. For example, the DMS may show a dialog that includes an Options button. If the user clicks this button, the DMS would call the application's callback function which would give the application a chance to display other save options.

Note that following a successful call to **ODMSaveAs** the calling application may have two different document IDs to work with. This is different than

the situation with **ODMSaveDoc** where the new ID replaces the old one in the current session. The state of the document specified by the old ID remains the same after the call. The document specified by the new ID will be in the closed state following the call. A typical sequence of operations an application might follow in response to the user selecting File | Save As would be:

Application passes the currently open document's ID to **ODMSaveAs**. If Null is returned for the new ID then the sequence is complete. If a new ID for the document is returned then continue with the steps below.

Application calls **ODMOpenDoc** on the new ID. This returns a new filename for the document.

Application saves the document to the new filename and then calls **ODMSaveDoc** on the new ID to indicate to the DMS that the new document should be saved in the document repository.

Application calls **ODMCloseDoc** on the old ID.

The application can now forget about the old ID and use the new ID for all subsequent operations on the file. When the current session is completed it will call **ODMCloseDoc** on the new ID.

Parameters:

odmHandle - in - An ODMA handle obtained by a previous call to **ODMRegisterApp**.

lpzDocId - in - A Null-terminated document ID. This is typically obtained by a call to **ODMSelectDoc** or **ODMNewDoc**. This document may or may not be open at the time that **ODMSaveAs** is called. Its open status will remain the same after this call.

lpzNewDocId - out - A pointer to a buffer where the DMS will return the ID of the new document. This buffer needs to be at least **ODM_DOCID_MAX** bytes in length. If successful then a Null-terminated document ID will be returned here, unless the document is saved with the current ID. In this case the first byte of this buffer will be set to Null and 0 will be returned. Otherwise the contents of the buffer will be undefined.

lpzFormat - in - A Null-terminated string naming the format in which the application expects to save the document. This may be passed as a parameter to the *pcbCallBack* function which may return a different format.

pcbCallBack - in - A pointer to a callback function that can be used by the application to make other saving options available to the user. Any UI presented by this callback function should be task modal. The function should return a pointer to a format string which may or may not be the same as the original format string. This parameter may be null if the application does not wish to present any options. Under Microsoft Windows the procedure-instance address of the callback function (obtained from *MakeProcInstance*) should be used. The callback function's interface is as follows:

LPSTR SaveAsCallBack(DWORD
dwEnvData, LPSTR lpszFormat, LPVOID
pInstanceData)

dwEnvData - in - Environment data. On Windows platforms this is a Window handle for a parent window to associate with the callback function's display. If the DMS displayed a dialog in response to the **ODMSaveAs** call then it should pass the window handle for that dialog. Otherwise it should pass the window handle obtained from the **ODMRegisterApp** call. The callback function may use this window handle as the parent window for any dialogs or other windows it displays.

lpszFormat - The currently selected document format.

pInstanceData - The instance data passed from the calling application.

pInstanceData - in - A pointer to caller context information that will be passed to the *pcbCallBack* function. This data will not be accessed by ODMA.

Return value: 0 if successful. **ODM_E_CANCEL** if the user cancels the creation of the new document. **ODM_E_DOCID** if the document ID is invalid or refers to a document that no longer exists. **ODM_E_FAIL** if the DMS is unable to create the new document. **ODM_E_HANDLE** if *odmHandle* was invalid. **ODM_E_APPSELECT** if the user selected to save the document as a non-profiled document.

ODMActivate

ODMSTATUS ODMActivate(ODMHANDLE odmHandle, WORD action, LPSTR lpszDocId)

This function causes the DMS to perform actions that do not require cooperation from the calling application. Control is returned to the calling application after the specified action has been completed. Note that a DMS is not required to support all of these actions.

Parameters:

odmHandle - in - An ODMA handle obtained by a previous call to **ODMRegisterApp**.

action - in - One of the following action codes:

ODM_NONE - No specific action is requested. The DMS should simply make itself visible and let the user select the action to be performed.

ODM_DELETE - The DMS should delete the specified document. Note that most DMSs will not allow a deletion to occur if the document is currently in use. Normally the application would call **ODMSelectDoc** to get a document ID for the document to be deleted.

ODM_SHOWATTRIBUTES - The DMS should display the specified document's profile or attributes.

ODM_EDITATTRIBUTES - The DMS should display the specified document's profile or attributes, and the user should be put in edit mode. Note that some DMSs will not allow a document's attributes to be edited while the document is in use.

ODM_VIEWDOC - The DMS should display the specified document in a viewer window.

ODM_OPENDOC - The DMS should open the specified document in its native application. This function is intended for use by applications other than the document's native application (e-mail, workflow, annotation, etc.). Applications should use **ODMOpenDoc** to access their own documents.

lpszDocId - in - A document ID specifying the document on which to perform the requested action. This parameter may be Null if the action is **ODM_NONE**.

Return value: 0 if successful. **ODM_E_DOCID** if the document ID is invalid or refers to a document that no longer exists. **ODM_E_INUSE** if the document is currently in use or checked out by another user on actions where this would preclude the operation from completing correctly. **ODM_E_CANCEL** if the action was canceled by the user. **ODM_E_ITEM** if *action* is invalid or not supported by the DMS. **ODM_E_FAIL** if the action could not be completed by the DMS. **ODM_E_HANDLE** if *odmHandle* was invalid.

ODMGetDocInfo

ODMSTATUS ODMGetDocInfo(ODMHANDLE odmHandle, LPSTR lpszDocId, WORD item, LPSTR lpszData, WORD dataLen)

An application can use this function to obtain information about a document from the DMS.

Parameters:

odmHandle - in - An ODMA handle obtained by a previous call to **ODMRegisterApp**.

lpszDocId - in - A Null-terminated document ID. This is typically obtained by a call to **ODMSelectDoc** or **ODMNewDoc**. The specified document may or may not be open when **ODMGetDocInfo** is called.

item - in - One of the following:

ODM_AUTHOR - Author of the document.

ODM_NAME - Name of the document. This is a descriptive name for the document, not the filename.

ODM_TYPE - Type of the document. This is typically an indication of the format or content of the document, i.e. correspondence, memo, contract, etc.

ODM_TITLETEXT - Suggested text to display in the document window's title bar. This may include one or more fields from the document's profile and possibly other information as well.

ODM_CONTENTFORMAT - The format string indicating the format of the document's content.

ODM_DMS_DEFINED - The *lpszData* parameter contains a DMS-specific indication of the data to be returned. Note that an application must know which DMS it is talking to and must understand the data indications supported by that DMS in order to use this item name.

lpszData - in/out - On input, ignored if item is anything other than **ODM_DMS_DEFINED**. If item is **ODM_DMS_DEFINED** then *lpszData* contains an indication of the data to be returned. On output, the requested data is returned in the buffer pointed to by *lpszData*.

dataLen - in - length of the output buffer pointed to by *lpszData*. If the data to be returned is longer than this, it will be truncated. In either case the returned data will always be terminated with a Null character.

Return value: 0 if successful. **ODM_E_DOCID** if the document ID is invalid or refers to a document that no longer exists. **ODM_E_ITEM** if *item* is invalid or not supported by the document's DMS. **ODM_E_HANDLE** if *odmHandle* was invalid.

ODMSetDocInfo

ODMSTATUS ODMSetDocInfo(ODMHANDLE odmHandle, LPSTR lpszDocId, WORD item, LPSTR lpszData)

An application can use this function to pass information about the document to the DMS. The DMS may or may not accept the information; most DMSs validate attributes like document types and authors against predefined tables.

Parameters:

odmHandle - in - An ODMA handle obtained by a previous call to **ODMRegisterApp**.

lpszDocId - in - A Null-terminated document ID. This is typically obtained by a call to **ODMSelectDoc** or **ODMNewDoc**. The specified document may or may not be open when **ODMSetDocInfo** is called.

item - in - One of the following:

- ODM_AUTHOR** - Author of the document.
- ODM_NAME** - Name of the document. This is a descriptive name for the document, not the filename.
- ODM_TYPE** - Type of the document. This is typically an indication of the format or content of the document, i.e. correspondence, memo, contract, etc.
- ODM_CONTENTFORMAT** - The document's format string. Note that this merely informs the DMS of a change in the document's format; it does not cause a conversion process to take place within the DMS.
- ODM_DMS_DEFINED** - The *lpszData* parameter contains a DMS-specific indication of the data being passed as well as the data itself. Note that an application must know which DMS it is talking to and must understand the data indications supported by that DMS in order to use this item name.

lpszData - in - the data being passed to the DMS. Must be null-terminated.

Return value: 0 if successful. **ODM_E_DOCID** if the document ID is invalid or refers to a document that no longer exists. **ODM_E_ITEM** if *item* is invalid. **ODM_E_HANDLE** if *odmHandle* was invalid. **ODM_E_FAIL** if the specified data was invalid or the DMS was unable to accept it for other reasons.

ODMGetDMSInfo

ODMSTATUS ODMGetDMSInfo(ODMHANDLE odmHandle, LPSTR lpszDmsId, LPWORD pwVerNo, LPDWORD pdwExtensions)

This function returns information to the application about the currently active DMS.

Parameters:

odmHandle - in - An ODMA handle obtained by a previous call to **ODMRegisterApp**.

lpzDmsId - out - This should point to an area at least **ODM_DMID_MAX** bytes long. A Null-terminated ID identifying the DMS is returned here. This is the same ID embedded in document IDs returned by this DMS.

pwVerNo - out - The version of the ODMA API supported by this DMS is returned here.

pdwExtensions - out - Indications of extensions to the base ODMA API that are supported by this DMS are returned here. No extensions have been defined yet, so currently 0 will always be returned here.

Return value: 0 if successful. **ODM_E_HANDLE** if *odmHandle* was invalid.

ODMQueryInterface

HRESULT ODMQueryInterface(ODMHANDLE odmHandle, LPSTR lpzDocId, REFIID riid, LPVOID FAR *ppvObj)

An application can use this function to get a COM interface from an ODMA provider. All ODMA providers support the **IODMDocMan** interface, and individual DMSs may support other interfaces as well. Note that this function is prototyped in *odmacom.h* instead of *odma.h*, so that non-COM-aware applications do not have to `#include` the header files that define interface IDs.

Parameters:

odmHandle - in - An ODMA handle obtained by a previous call to **ODMRegisterApp**.

lpzDocId - in - An ODMA document ID or Null. If Null then the application's default DMS is queried for the interface. Otherwise the DMS that created this document ID is queried.

riid - in - The interface to be obtained from the DMS.

ppvObj - out - If the requested interface is supported by the DMS then it is returned here. Otherwise *ppvObj* is set to Null.

Return value: S_OK if successful. E_INVALIDARG if *odmHandle* or *lpszDocId* is invalid. E_NOINTERFACE if the requested interface is not supported by the DMS.

ODMGetLeadMoniker

ODMSTATUS ODMGetLeadMoniker(ODMHANDLE odmHandle, LPSTR lpszDocId, LPMONIKER FAR *ppMoniker)

Applications that are OLE 2 servers typically form composite monikers for their OLE links by combining a file moniker representing the document with one or more item monikers representing a particular section of the document. This approach often does not work in environments where document management systems are in use because the filename that the application sees is usually just temporary. This function lets the application obtain a leading moniker from the DMS that can be used in place of the file moniker.

This function will only be available on platforms supporting OLE 2. This function may not be supported by some DMSs; those DMSs will return **ODM_E_FAIL**. In this case the application should go ahead and use the file moniker as though ODMA were not present. Note that this function is prototyped in *odmacom.h* instead of *odma.h*, so that non-OLE-aware applications do not have to *#include* the OLE header files.

Parameters:

odmHandle - in - An ODMA handle obtained by a previous call to **ODMRegisterApp**.

lpszDocId - in - An ODMA document ID.

ppMoniker - out - A leading moniker for the specified document ID will be returned here if successful. Otherwise Null will be returned here.

Return value: 0 if successful. **ODM_E_FAIL** if the DMS that created the specified document ID does not support OLE moniker building. **ODM_E_DOCID** if the document ID is invalid or refers to a document that no longer exists. **ODM_E_HANDLE** if *odmHandle* is invalid.

DMS Interface

A DMS interfaces with ODMA through a function called **ODMGetODMInterface** and through COM interfaces. When an application calls the ODMA connection manager's **ODMRegisterApp** function, the connection manager calls the appropriate DMS's **ODMGetODMInterface** function to establish a connection with the DMS. This function is defined as follows:

ODMGetODMInterface

```
HRESULT ODMGetODMInterface( REFIID riid, LPVOID FAR *ppvObj,  
LPUNKNOWN pUnkOuter, LPVOID pReserved, LPSTR lpszAppId,  
DWORD dwEnvData )
```

The ODMA connection manager calls this function to get an interface from a DMS. This function is **not** available to ODMA-aware applications.

Parameters:

riid - in - ID of the interface being requested. This will typically be **IID_IODMDocMan**, but the connection manager may also pass through calls for other interfaces from the client. ODMA providers must support the **IID_IODMDocMan** interface.

ppvObj - out - The requested interface should be returned here. If the DMS cannot return the requested interface, *ppvObj* should be set to Null.

pUnkOuter - in - The controlling IUnknown interface for the aggregate object. ODMA providers **must** support aggregation.

pReserved - in - Reserved for future use. Will always be set to Null.

lpszAppId - in - The ID of the calling application. See **ODMRegisterApp**.

dwEnvData - in - Environment specific data. On Windows platforms this will be a window handle from the calling application. See **ODMRegisterApp**.

Return value: S_OK if successful. E_NOINTERFACE if the requested interface is not supported.

Binding to the API under Windows 3.x, Windows 95 and Windows NT

The ODMA connection manager, ODMA.DLL (Windows 3.x) or ODMA32.DLL (Windows 95 and NT), and a corresponding link library, ODMA.LIB (Windows 3.x) or ODMA32.LIB (Windows 95 and NT), will be made freely available to any application vendor wishing to use them. Applications can choose either of the methods described below to bind to ODMA:

1. Link to ODMA.LIB/ODMA32.LIB. Applications choosing this approach will need to install a copy of ODMA.DLL/ODMA32.DLL into the Windows system directory at the time the application is installed unless a newer copy of ODMA.DLL/ODMA32.DLL already resides there. At startup time the application can call **ODMRegisterApp** to determine whether or not an ODMA provider is present for the calling application.
2. Dynamically load ODMA.DLL/ODMA32.DLL at startup time with a call to LoadLibrary. With this approach the application has to call GetProcAddress to get a pointer to each ODMA function it will call, but the application does not have to do anything related to ODMA at installation time. If ODMA.DLL/ODMA32.DLL doesn't exist or if **ODMRegisterApp** returns **ODM_E_NODMS** then the application knows that no ODMA provider is present for the calling application.

There is currently no thunk layer that exists to allow a 32 bit application and a 16 bit DMS, or vice versa, to interact with each other.

Installing a DMS under Windows 3.x, Windows 95 and Windows NT

The ODMA connection manager, ODMA.DLL for Windows 3.x and ODMA32.DLL for Windows NT and Windows 95, will be provided to all DMS vendors that want to support the ODMA specification. These DLLs will serve as a common entry point for applications that wish to call ODMA functions. Its implementation of **ODMRegisterApp** will figure out which DMS is registered for default use by the calling application. It will then load the appropriate DMS's DLL and pass through the function call. All other

ODMA function calls that do not include a document ID will be passed to this default DMS.

When an ODMA call is made that includes a document ID, the ODMA connection manager will determine which DMS created the ID. If that DMS has not been initialized yet by the calling application, the connection manager will call the provider's **ODMGetODMInterface** function and will then pass through the current ODMA function call.

As part of its installation, a DMS should check whether a copy of ODMA.DLL or ODMA32.DLL already exists in the Windows system directory. If it does not or if the DMS has a newer version then it should copy its version of ODMA.DLL or ODMA32.DLL to the Windows system directory. Then it should register with the ODMA connection manager by adding one or more keys to the registration database.

Each DMS must add a subkey to the root level ODMA key. This subkey should be named the same as the DMS's DMS ID, and the subkey's value should specify the location of the DLL containing the DMS's ODMA entry points. For example, suppose a DMS with a DMS ID of DDD implemented its ODMA entry points in H:\ABC\XYZ.DLL. It would add the following key and value to the registration database:

ODMA\DDD = H:\ABC\XYZ.DLL for Windows 3.x
ODMA32\DDD = H:\ABC\XYZ.DLL for Win32 (i.e. WindowsNT and
Windows95)

If the DMS wanted to serve as the default ODMA provider to be used with all ODMA-aware applications that were not registered with a specific DMS, it would add the DEFAULT subkey to its DMS ID key as shown below:

ODMA\DDD\DEFAULT for Windows 3.x
ODMA32\DDD\DEFAULT for Win32 (i.e. WindowsNT and
Windows95)

If the DMS wanted to specifically register itself as the ODMA provider for one or more applications, it would also add a subkey to the relevant applications' root level keys. Note that this is only necessary if a different DMS is registered as the default ODMA provider. Each time an application calls **ODMRegisterApp**, the connection manager looks in the registration database for a root-level key that matches the specified application ID. If it finds such a key then it looks under it for a subkey called ODMA. The

value of this key is the DMS ID of the specifically registered DMS. If the application ID key is not found or if the subkey ODMA or ODMA32 does not exist under it, the connection manager looks at the subkeys of the root-level key ODMA or ODMA32 for a default ODMA implementation.

For example, suppose a document management system with a DMS ID of DDD wanted to support only one particular application that used QQQ as its application ID. It would add the following key and value to the registration database in addition to the ODMA subkey described above:

QQQ\ODMA = DDD	for Windows 3.x
QQQ\ODMA32 = DDD	for Win32 (i.e. WindowsNT and Windows95)

Appendix A

Preferred Call Usage for Initial File Creation

There has been some misunderstandings among both DMS and application vendors on the correct use of the ODMA calls to initially create a file. This appendix is basically from a proposal made in October 1994 by Rod Schiffman of Novell. There has been no dissension on the issue since the original proposal was made. Until the standard is modified to provide other methods for the initial creation of a new document in a DMS, this provides a sequence for the calls that will work with all major DMS vendors.

It is probably useful for me to describe the terminology and concepts I will use in this discussion. A DMS is fundamentally more complex than my description, but it will suffice for my purposes. A DMS is a black box that manages documents. My Local System is the particular machine and Operating System, or Environment, that I use. It has its own file system, applications and interfaces. When I need to do something to a existing document in the DMS, I select it using tools the DMS provides and retrieve it into my local system. I use an application to perform operations on the document. I may want to create a new document with my application. The new document will be stored on my local system until I save it into the DMS. The DMS may want to tell me where I should save it on the Local System until I eventually save it into the DMS. I may also want to update the copy in the DMS while I am still working on it. I call this writing the document through to the DMS. When I am finally done working on the document, I can save it to the DMS. It is then closed. At some point in the process of creating a new document, the DMS will require me to give it a profile. When I have saved the document to the DMS, it is closed and the document no longer has a presence on my local machine. ODMA has a vaguely defined concept of formats that are attached to a document, and while the timing of when and how formats are defined plays a central role in this discussion, formats, conversions and their association to documents does not play a role in this discussion.

ODMSaveDoc is the only ODMA function that actually saves a document into a DMS. It uses a Document ID (DocID) supplied by the DMS. The initial creation of the first DocID for the document can only be supplied by ODMNewDoc. ODMSaveAs and ODMSaveDoc can return a new DocID, but they must already have an initial DocID to start with. ODMSelectDoc only returns a DocID that already exists. This means that ODMNewDoc must be called to generate the initial DocID. After the application has a DocID, it can call ODMOpenDoc to get a local file name to save the

document to and ODMSaveDoc to write the document through to the DMS. This would imply that it is not necessary to call ODMSaveAs in the process of saving the document the first time. This has caused a number of people to assume that the usage of ODMNewDoc was synonymous with the initial "Save" function in most applications. However, the spec says that the format specified in ODMNewDoc "may be changed later via an ODMSaveAs call." This implies that it is OK to call ODMSaveAs after calling ODMNewDoc. ODMSaveAs is the only call that allows an application to interact with the DMS and supply application specific information such as formats and passwords. If an app only calls ODMNewDoc and does not call ODMSaveAs, it loses the ability to provide application specific information. ODMSaveAs requires that a user interface is supplied to the user. If ODMNewDoc also supplies a user interface, then the user interface comes up twice for each document creation. Once in ODMNewDoc and again in ODMSaveAs. This is probably not something that the user wants. Also, how do we reconcile the ability to call ODMNewDoc without a user interface. If ODMNewDoc is called with no user interface and a DocID is returned, then when is the profile filled in? ODMOpenDoc and ODMSaveDoc and ODMCloseDoc have no obvious provisions for providing a user interface.

ODMNewDoc was originally intended to be synonymous with the "New" menu choice, not the "Save" menu choice, in most applications. For example, although SoftSolutions and PC Docs do not care about a document until it is time to save it to the DMS the first time, DEC TeamLinks does have the concept of letting the DMS know a document is going through initial creation. There is nothing in the spec that indicates whether ODMNewDoc should be called at document creation or just before saving the document the first time. This is not an accident according to Brad Clements, the original author of the ODMA spec. It can be done at whatever time makes sense for the DMS and application. ODMNewDoc does not have provisions for the callback routine because it notifies the DMS about the creation of a document not the saving of a document. At document creation time, the user is probably not worried about the what the final format will be or if there will be a password. The user may not, and this is crucial, even save the document to the DMS. It is possible that if ODMNewDoc is called without a user interface, that the DocID returned to the application may not eventually be used, if the user is allowed to have the file saved to the local file system in a later call to ODMSaveAs.

It turns out that many applications and Document Management Systems do not have or need a concept of providing application and DMS interaction until the document is ready to be saved in the DMS for the first time.

Certainly a DMS must deal with applications that will not provide any interaction with the DMS until the document is already created in the edit buffer and is ready to be saved. At this point I would like to point out a scenario that follows the spec, and still works with major applications that may not have followed the spec according to the interpretation given here.

At the time of initial document creation, or when it is time to save the document the application should call ODMNewDoc with the ODM_SILENT flag. This will return a temporary DocID that the application can use. Technically, the application could call ODMOpenDoc to have a local temporary file to use until the document is finally saved into the DMS, but I know that every DMS vendor I have talked to would frown on this idea. They prefer the idea that all saves are written through to the DMS and that there is a full profile for anything that exists in the DMS. It is OK for the DMS vendor require this by returning an ODM_E_USERINT error to indicate that ODMOpenDoc cannot be called until the app makes a call to ODMNewDoc or ODMSaveAs that brings up an user interface. This implies that the application would be better to avoid the situation and not call ODMOpenDoc until after an user interface is provided to the user. After calling ODMNewDoc in silent mode, take the DocID that is returned and call ODMSaveAs to have the profile created, and allow access to the callback routines. If the ODMNewDoc returns an ODM_E_USERINT error because it does not support the silent mode, then call ODMNewDoc with a user interface and skip calling ODMSaveAs. This performs the same functionality, but does not provide access to the callback routines. If the application still wants the user to see the callback dialog, then it can call the callback dialog itself after calling the ODMNewDoc call. This is not optimal, but does work. In either case the user interface is only presented to the user one time.

There are a couple caveats for DMS vendors. The DocID returned by ODMNewDoc must be a temporary ID. It is possible that the user will throw away the file before it is saved, or that it will be saved to the native OS through the options button in the SaveAs dialog. This could be easily handled by returning a well known dummy value like ::ODMA\

Basically, this turns the ODMNewDoc into a fancy NOP routine. There may be situations where this may not be the best action, but since there is no compelling reason for the existence of the ODMNewDoc call in the 1.0 spec, it seems to solve more problems than it creates. It would probably be a good idea to add an extension to the 1.0 spec in ODMA 1.1 where the ODMSaveAs call can allow a NULL to be passed in as the IpszDocID parameter to indicate that the document is being saved for the first time. This would free the application from having to use the ODMNewDoc call. Of course, that is what the addition of a dummy DocID value returned from ODMNewDoc accomplishes. In the ODMA 1.1 time frame a new ODMNewDoc call can be created if someone comes up with a compelling reason for its existence. Simply using the ODMSaveAs call when an application is creating a new file in the DMS, whether it is the first save or an actual SaveAs, matches the way most applications work today.