Legal Information

Messaging Application Programming Interface (MAPI) Programmer's Reference

About the MAPI SDK

The *Messaging Application Programming Interface (MAPI) Programmer's Reference* is the documentation that accompanies the MAPI Software Development Kit (SDK).

Intended Audience

The *MAPI Programmer's Reference* is written for C and C++ developers with a wide range of needs and experience with messaging. For those developers who want to use MAPI to augment their nonmessaging applications with messaging features, no specific prerequisite knowledge is required. However, for developers intending to use MAPI to create full-scale workgroup applications or drivers for specialized messaging system services, a background in messaging and a familiarity with the Component Object Model (COM) used with OLE is recommended.

How This Document is Organized

The *MAPI Programmer's Reference* is organized in the following parts:

- Guide
- Reference
- Appendixes
- Glossary

The Guide is organized in sections, beginning with a discussion of key concepts and an architecture overview. There is an introduction to MAPI programming, that includes information for developers of client applications and service providers. There are tutorials for application developers writing with the Simple MAPI and CMC set of API functions. For developers working with the object-based MAPI interface, there are sections that describe, at a conceptual level, the use and implementation of objects, interfaces, and properties. There are also guides to implementing and using some of the common MAPI objects and to writing a transport provider.

The Reference is divided into several sections. Each section is devoted to a different type of item within MAPI and contains entries with brief descriptions of the purpose of the item, and if appropriate, the correct syntax, parameters, and return values. The Reference includes the following sections:

- MAPI Interfaces
- MAPI Functions and Related Macros
- MAPI Properties
- MAPI Structures and Related Macros
- MAPI Data Types
- Common Messaging Calls (CMC)
- Simple MAPI

The Appendixes are:

- MAPI Versions of 32-Bit Windows Functions
- Address Types
- Transport-Neutral Encapsulation Format (TNEF)
- Mapping of X.400 P2 Attributes to MAPI Properties
- Mapping of Internet Mail Attributes to MAPI Properties
- Regular Expressions
- Functionality Groups

The Glossary provides definitions for commonly used MAPI terms and cross references to related topics in the Guide and Reference.

Document Conventions

This manual uses the following typographic conventions:

Convention	Description
`monospace`	Indicates source code, structure syntax, examples, user input, and program output. For example, `ptbl->SortTable(pSort, TBL_BATCH);`
Bold	Indicates an interface, method, structure, or other keyword in MAPI, the Microsoft® Windows® operating system, the OLE application programming interface, C, or C++. For example, **SpoolerYield** is a MAPI method. Within discussions of syntax, bold type indicates that the text must be entered exactly as shown.
Italic	Indicates placeholders, most often method or function parameters; these placeholders stand for information that must be supplied by the implementation or the user. For example, *lpMAPIError* is a MAPI method parameter.
UPPERCASE	Indicates MAPI flags, return values, and properties. For example, MAPI_UNICODE is a flag, S_OK is a return value, and PR_DISPLAY_NAME is a property. In addition, uppercase letters indicate filenames, segment names, registers, and terms used at the operating-system command level.
( )	Indicate one or more parameters that you pass to a function, in syntax.
[ ]	Indicate optional syntax items. Type only the syntax within the brackets, not the brackets themselves.
	Indicates that code continued from one line to the next should be typed all on one line. For example, `SharedExtensionsDir=  \\SERVER1\SHARE1\MAILEXTS`

Note The interface syntax in this book follows the variable-naming convention known as Hungarian notation, invented by the programmer Charles Simonyi. Variables are prefixed with lowercase letters that indicate their data type. For example, *lpszProfileName* is a long pointer to a zero-terminated string name *ProfileName*. For more information about Hungarian notation, see *Programming Windows 95* by Charles Petzold and *Code Complete* by Steve McConnell.

For More Information

For more information about OLE programming, see *Inside OLE*, *Second Edition*, by Kraig Brockschmidt (Redmond, WA: Microsoft Press, 1995) and the *OLE Programmer's Reference* in the Microsoft® Win32® Software Development Kit (SDK).

For more information about programming with Microsoft® Visual Basic® programming system, see your Visual Basic documentation.

Quick Start Procedures

Each topic has three sections: design tasks, implementation tasks, and references to sample source code. The goal is to call out design decisions and refer the reader to other topics for the necessary information, not to provide details right here.

Adding a Send Command to an Application

This topic explains the work necessary to add a **Send** command to a document-based application's **File** menu. This is a Microsoft Windows 95 logo requirement. The same logic applies to sending a message from any application.

Design Tasks

1. Decide whether to use Simple MAPI, CMC, MAPI, or the OLE Messaging Library. See Selecting a Client Interface.
2. Decide when to load and unload the selected interface, and when to create and destroy its session. You should share the messaging client's session, if it is active. See MAPI Sessions.

Implementation Tasks

1. Determine whether the **Send** command should be enabled. Installation of messaging support in the operating system is optional, and your **Send** command should be disabled if the selected interface is not available.
2. Load the selected interface DLL and the entry points you will need. See one of the following topics, depending upon which interface you have selected:
   - For Simple MAPI, see Initializing a Simple MAPI Client.
   - For CMC, see Starting a CMC Session.
   - For MAPI, see Initializing the MAPI Libraries and Initializing the OLE Libraries.
   - For the OLE Messaging Library, no explicit load step is necessary. Simply instantiate one of the top-level objects, MAPI.Session or MAPI.Message.
3. Send a document using the selected interface. See one of the following topics, depending upon which interface you have selected:
   - For Simple MAPI, see Sending Messages with Simple MAPI.
   - For CMC, see Sending Messages with CMC.
   - For the OLE Messaging Library, see Creating and Sending a Message.
   - For MAPI, several steps are necessary:
     1. Open the default message store. See Opening a Message Store.
     2. Open the Outbox. See About Opening Folders.
     3. Add recipients to the message's recipient table. See About Message Recipients.
     4. Add the document to the message as an attachment. See About Message Attachments.
     5. Optionally, add text to the message. See PR_BODY.
     6. Send the message. See About Sending Messages.
     7. Release all the objects created in the preceding steps.

About Sample Source Code

CMC client. Demonstrates the use of all CMC calls in a very simple messaging client. It can send and read messages, list the contents of the Inbox, and view the address book. The executable is named CMCCLI.EXE; the sample source code is in the CMC.CLI directory. It is written in C and supports all platforms.

Routing client. Constructs and sends a message with an attachment using MAPI. You can ignore the parts that deal with named routing properties. The executable is named ROUTE.EXE; the sample source code is in the ROUTE.CLI directory. It is written in C and supports all platforms.

Send RTF utility. Sends a message with a Rich Text Format content based on command line parameters. The executable is named SNDRTF.EXE; the sample source code is in SENDRTF.CLI. It is written in Microsoft Visual C++ and supports Microsoft Win32 platforms only.

Simple MAPI client. Demonstrates the use of all Simple MAPI calls in a very simple messaging client. It can send and read messages, list the contents of the Inbox, and view the address book. The executable is named SMPCLI.EXE; the sample source code is in the SIMPLE.CLI directory. It is written in C and supports all platforms.

Timecard application. Demonstrates creating and sending messages with custom fields using the OLE Messaging Library. The executables are named TMCLI.EXE and TMSRV.EXE; the sample source code is in the TIMECARD.CLI directory. It is written in Microsoft Visual Basic 4.0 and supports all platforms.

Adding Document Routing to an Application

This topic explains the work necessary to add a **Route** command – or another form of document routing support – to a **File** menu in a document-based application. Document routing can be used for many purposes, but all involve defining a list of people who are to read and act on information in the document in a specific order.

The following discussion assumes you want to base your routing application on an existing document-based application. If you prefer to base it on a messaging client, see also Creating a New Interpersonal (IPM) Message Class.

Note Work is in progress on a companion standard to MAPI for document routing and other workflow activities. Send e-mail to **mapi@microsoft.com** to receive further information or to become involved in the open design process for this technology.

Design Tasks

1. Decide whether to use Simple MAPI, CMC, MAPI, or the OLE Messaging Library. See Selecting a Client Interface.

   If your application requires routing across messaging domain boundaries, you must use MAPI or the OLE Messaging Library and store the routing information in named properties on the message. See About Sending Across Messaging Domains. If you use the OLE Messaging Library, the routing information must be stored in the message's Fields collection, using five special property sets. See Customizing a Folder or Message and Fields Collection Object.

   If you choose to use Simple MAPI or CMC, the routing information can be stored in a file attached to your message. For Simple MAPI, see Handling Attachments with Simple MAPI. For CMC, see Handling Attachments with CMC.
2. Design the message properties and the commands – called "verbs" in MAPI – specific to your routing functionality.

Implementation Tasks

1. Choose the message class name. It should begin with IPM. See About Message Classes.
2. Use the MAPI address book dialog box to let the user add names to the routing list. See the **IAddrBook::Address** method, About **ADRPARM** Structures and About **ADRLIST** Structures.
3. Store the routing list in the message, an attached file, or the document itself, as appropriate.
4. Support your routing verbs with message – or document – properties that refer to entries in the routing list.

About Sample Source Code

Routing client. Creates a message with an attached document and a simple linear route using MAPI. The only routing verb is "Route to Next." The executable is named ROUTE.EXE; the sample source code is in the ROUTE.CLI directory. It is written in C and supports all platforms.

Timecard application. Does not implement routing, but it demonstrates creating and sending messages with custom fields using the OLE Messaging Library. The executables are named TMCLI.EXE and TMSRV.EXE; the sample source code is in the TIMECARD.CLI directory. It is written in Visual Basic 4.0 and supports all platforms.

Creating a New Interpersonal (IPM) Message Class

This topic explains several ways to create a new message class used for person-to-person communication. By using MAPI properties to structure message content, you avoid writing code to parse message text or a binary attachment.

Design Tasks

1. Decide whether yours is an interpersonal messaging (IPM) application or a species of interprocess communication (IPC) application. There is an enormous variety of applications; following is a basic list of types and examples. For more information about IPC applications, see Creating a New Interprocess (IPC) Message Class.
   - Person-to-person. A person initiates the exchange of messages, and another person responds. This includes traditional e-mail as well as more structured exchanges such as document routing or expense approval.
   - Person-to-machine. A person initiates the exchange of messages, and a machine responds. Examples include submitting a database query by e-mail and subscribing to a mailing list.
   - Machine-to-person. A machine initiates the exchange of messages, and a person responds. Examples include news feeds and other types of document distribution, and opinion surveys.
   - Machine-to-machine. A machine initiates the exchange of messages, and another machine responds. Examples include link heartbeat monitoring, directory and database replication.
   - A more complex pattern with both people and machines in the route.
   - A more complex pattern where, instead of being transmitted, the message may be posted directly into a public folder or bulletin board forum supported by a message store, for consumption by other readers, an administrator, or a software agent.

   Logically, person-to-person applications, machine-to-person applications, and applications that post to public forums should be designated IPM, while person-to-machine and machine-to-machine applications are IPC message classes. The only real difference is that IPM messages in a message store are visible to messaging clients, while IPC messages usually are not. Anything that requires a person to respond must use an IPM message class. Applications involving a more complex pattern including people and machines often involve a mix of IPM and IPC messages.
2. Choose the message class name, beginning it with IPM. or IPC. according to the preceding guidelines.
3. Choose the application framework for sending and reading your messages. Alternatives include:
   - A form. A form is a MAPI object that is integrated into messaging clients; it can create, display, print, and perform custom commands – called "verbs" in MAPI – on a particular type of message. Forms take maximum advantage of integration with the messaging client. See MAPI Form Architecture.
   - A standalone application. This gives you the greatest control over the user interface to your messages but also requires the most work in creating and distributing your application and in making it easy for your users to find.
   - A client extension. This is a compromise. An extension is easier for users to discover than a standalone application and in some respects easier to write than a form, but the integration with the client is less complete than with a form. See Advantages of Extending Microsoft Exchange.
4. Decide whether to use Simple MAPI, CMC, MAPI, or the OLE Messaging Library. See Selecting a Client Interface. In addition to the considerations listed there, you must think about whether that interface fits well into the application framework you have chosen. For instance, both forms and extensions must use or furnish several COM interfaces which are not easily supported by Visual Basic programs.
5. Define properties specific to your message class. Decide whether to use named properties or properties from the 0x6800-0x7fff range. See About Property Identifiers and About Named Properties. Properties that contain e-mail addresses should be stored in a special way to provide for

translation when changing messaging domains. See About Sending Across Messaging Domains.

Consider the need for interoperability with messaging clients that can be only message text. It may be advisable to duplicate some or all of your message's properties in the message text if you expect the message to be read by people using such messaging clients.

6. Define the commands – called "verbs" in MAPI – specific to your message class, and choose which standard verbs to implement. See About Form Verbs.
7. Choose a distribution mechanism. For a standalone application, this generally involves making an executable and any necessary auxiliary files available to your users. Additional steps are necessary if you are implementing a form, including which library to register it in. See About Form Libraries. Additional steps are also necessary if you are implementing a client extension. See Registering Extensions.

Implementation Tasks

All IPM message classes should honor common IPM client options for disposition of sent messages and other features.

To implement a form

1. Choose a class identifier (CLSID) for the form. You can use the tool UUIDGEN.EXE for this purpose. See Using UUIDGEN.EXE.
2. Choose a message class name for the form. See About Message Classes.
3. Create an .EXE file that acts as a COM server for the form, registered with the form's class identifier. It must implement the OLE interface **IClassFactory**, and the **IMAPIForm** and **IPersistMessage** interfaces; it may optionally implement the **IMAPIFormAdviseSink** and **IMAPIFormFactory** interfaces. See Implementing the **IClassFactory** Interface for Form Servers and Implementing the **IMAPIForm** Interface for Form Servers. The class factory's **CreateInstance** method must return an **IMAPIForm** interface; that form object's **QueryInterface** method must be capable of returning an **IPersistMessage** interface.
4. Create a .CFG file listing the form's class identifier, message class, properties, verbs, extensions, and so forth. See Configuration File Format.
5. For each user of your form, make the .EXE file and any auxiliary files available, and register the .CFG file in a form library using PDKIN.EXE (MAPISDK), INSTFORM.EXE (EDK), or your own code. See About Form Libraries. If using the local form library, you must also register your .EXE file as a COM server for the form's class identifier, using REGEDIT.EXE.

For more information, see Developing MAPI Form Servers.

To add an extension to the messaging client

1. Decide which client contexts your extension should be available in, and whether your extension requires a specific message service to be available. See How Extensions Work.
2. Create a DLL with an entry point that returns an extension object – an object that implements the **IExchExt** interface and any other required interfaces. Implement the **IExchExt** interface and any other required interfaces.
3. For each user of your extension, make the DLL and any auxiliary files available, and register your extension in the system registry or WIN.INI file. See Registering Extensions.

For more information, see Command Extensions.

To create a standalone application using MAPI

1. Verify that MAPI is installed before proceeding if your installation process does not guarantee that MAPI is installed. See About MAPI Installations.
2. Load and initialize the MAPI DLL. See Initializing the MAPI Libraries and Initializing the OLE Libraries.
3. Log on to MAPI using the messaging client's session if available. See Logging On a MAPI Client.

4. Open the user's default message store, Inbox and Outbox folders. See Opening a Message Store and About Opening Folders.
5. Create and send a message using the following steps:
   a. Create a message in the Outbox folder, using the **IMAPIFolder::CreateMessage** method.
   b. Add recipients to the message's recipient table. See About Message Recipients.
   c. Add the document to the message as an attachment. See About Message Attachments.
   d. Optionally, add text to the message. See the PR_BODY property.
   e. Send the message. See About Sending Messages.
6. Scan the Inbox for new messages using the following steps:
   a. Optionally, force any incoming messages to be downloaded. See About Controlling Message Transmission.
   b. Open the Inbox folder's contents table.
   c. Restrict the contents table to messages of interest to your application.
   d. Optionally, register for new mail notifications on the message store object to be advised of further incoming messages.
7. For each user of your application, make the .EXE file and any auxiliary files available.

About Sample Source Code

Checkers form. Implements the MAPI form interfaces and a more elaborate message. Each message represents a move in a game of checkers between two players. The executable is named WCHECK.EXE; the sample source code is in the CHECKERS.FRM directory. It is written in C++ and supports all platforms.

Command extension. Adds a simple command to the client's menu. The executable is named CMDEXT.DLL; the sample source code is in the COMMAND.EXT directory. It is written in C++ and supports Win32 platforms only.

Routing client. A standalone application that implements a message class (the routing message) using MAPI. It can also send and read forms; you might find this helpful in testing your form. The executable is named ROUTE.EXE; the sample source code is in the ROUTE.CLI directory. It is written in C and supports all platforms.

Simple form. Implements the MAPI form interfaces and a very simple message containing only text and recipients. It is a good basis for reuse in a more complicated form. The executable is named SMPFRM.EXE; the sample source code is in the SIMPLE.FRM directory. It is written in C++ and supports Win32 platforms only.

Timecard application. Creates and sends messages – one IPM type and one IPC type – with custom fields using Visual Basic and the OLE Messaging Library. It's also an interesting example of how to distribute a standalone application through e-mail. The executables are named TMCLI.EXE and TMSRV.EXE; the sample source code is in the TIMECARD.CLI directory. It is written in Visual Basic 4.0 and supports all platforms.

Creating a New Interprocess (IPC) Message Class

This topic explains how to send and read structured messages between programs, or between programs and users. By using MAPI properties to structure message content, you avoid writing code to parse message text or a binary attachment.

Design Tasks

1. Decide whether yours is an interpersonal messaging (IPM) application or a type of interprocess communication (IPC) application. There is an enormous variety of applications; following is a basic list of types and examples. For more information about IPM applications, see Creating a New Interpersonal (IPM) Message Class.
   - Person-to-person. A person initiates the exchange of messages, and another person responds. Examples include traditional e-mail as well as more structured exchanges such as document routing or expense approval.
   - Person-to-machine. A person initiates the exchange of messages, and a machine responds. Examples include submitting a database query by e-mail and subscribing to a mailing list.
   - Machine-to-person. A machine initiates the exchange of messages, and a person responds. Examples include news feeds and other types of document distribution, and opinion surveys.
   - Machine-to-machine. A machine initiates the exchange of messages and another machine responds. Examples include link heartbeat monitoring, directory and database replication.
   - A more complex pattern with both people and machines in the route.
   - A more complex pattern where, instead of being transmitted, the message may be posted directly into a public folder or bulletin board forum supported by a message store, for consumption by other readers, an administrator, or a software agent.

   Logically, person-to-person applications, machine-to-person applications, and applications that post to public forums should be designated IPM, while person-to-machine and machine-to-machine applications are IPC message classes. The only real difference is that IPM messages in a message store are visible to messaging clients, while IPC messages usually are not. Anything that requires a person to respond must use an IPM message class. Applications involving a more complex pattern including people and machines often involve a mix of IPM and IPC messages.
2. Decide whether to use Simple MAPI, CMC, MAPI, or the OLE Messaging Library. See Selecting a Client Interface.
3. Decide whether your application can use the same profile as the messaging client or whether you must create one. See Creating and Configuring a Profile.
4. Choose the message class name, beginning it with IPM. or IPC. according to the guidelines established previously.
5. Define properties specific to the message class. Decide whether to use named properties or properties from the 0x6800-0x7fff range. See About Property Identifiers and About Named Properties. Properties that contain e-mail addresses should be stored in a special way to provide for translation when changing messaging domains. See About Sending Across Messaging Domains.

   One difference between IPM and IPC applications is that machines are much less tolerant of variations in message format than people are. Compatibility with messaging clients that can view only message text is often required for IPM applications, but IPC applications typically use MAPI properties to structure information in their messages and require MAPI everywhere. Nevertheless, for machine-to-person applications it may be necessary to duplicate some or all of the properties in the message text for compatibility with messaging clients that can view only message text.
6. Define the commands – called "verbs" in MAPI – specific to your message class, and choose which standard verbs to implement.
7. Decide how to handle sent messages and reports. It is usually not appropriate for IPC applications to save outbound messages in the IPM Sent Items folder, but the application may need to save outbound messages in a hidden folder for tracking purposes. See About Hidden Folders.

Implementation Tasks

1. Load the selected interface DLL and the entry points you will need. See one of the following topics, depending upon which interface you have selected.
   - For Simple MAPI, see Initializing a Simple MAPI Client.
   - For CMC, see Starting a CMC Session.
   - For MAPI, see Initializing the MAPI Libraries and Initializing the OLE Libraries.
   - For the OLE Messaging Library, no explicit load step is necessary; simply instantiate one of the top-level objects, MAPI.Session or MAPI.Message.
2. Log on to MAPI. IPM message classes should use the messaging client's session if available. IPC applications may need to use a specific profile they've created themselves.
   - For Simple MAPI, see Starting a Simple MAPI Session and related topics.
   - For CMC, see Starting a CMC Session.
   - For MAPI, see Logging On a MAPI Client.
   - For the OLE Messaging Library, see Session Handling.
3. For MAPI only, access the default message store. See About Opening Folders.
4. Create folders and receive folder mappings for your message class. For an IPC application the folders should be children of the message store's root folder, not of the IPM subtree. Include receive folder mappings for reports associated with your message class, if necessary. These steps can only be done using MAPI; if you have chosen a different interface, you can still write this code using MAPI and put it in your installation program.
   a. Open the root folder by calling the **IMsgStore::OpenEntry** method with a null entry identifier.
   b. Call the **IMAPIFolder::CreateFolder** method to create the folder.
   c. Call the **IMsgStore::SetReceiveFolder** method to establish a receive folder mapping for your message class. Several message classes can use the same receive folder; when processing incoming messages, you can restrict the folder's contents table on the PR_MESSAGE_CLASS property to isolate the messages you're interested in.
   d. Locate your folder. The most efficient way to find your folder once you've created it is a receive folder mapping. Even if you don't intend for the folder to receive messages, you can map it to a string that is not used as a message class. Named properties on the store object – the most obvious alternative – are not always supported by message store providers, and MAPI does not define a range of message store properties for application use.
5. Open the send and receive folders for your message class.
   - For Simple MAPI, this step is not necessary. Simply specify the message class in your call to the **MAPIFindNext** function.
   - For CMC, this step is not necessary. Simply specify the message class in your call to the **cmc_list** function.
   - For MAPI, see About Opening Folders, and note the following.

     To open the receive folder:

     1. Call the **IMsgStore::GetReceiveFolder** method.
     2. Pass the entry identifier returned to the **IMsgStore::OpenEntry** method.

     To open the Outbox (Send folder) for IPM only:

     1. Call the message store's **IMAPIProp::GetProps** method to retrieve the PR_IPM_OUTBOX_ENTRYID property.
     2. Pass the entry identifier to the **IMsgStore::OpenEntry** method.
   - For the OLE Messaging Library, see Accessing Folders and InfoStores Collection Object.

6. Send outgoing messages.

   Remember to handle the disposition of submitted messages. IPM message classes should honor common IPM client options for disposition of sent messages and the message store's PR_IPM_SENTMAIL_ENTRYID property.

   - For MAPI, see About Sent Message Processing and related topics.
   - For Simple MAPI, see Sending Messages with Simple MAPI.
   - For CMC, see Sending Messages with CMC.
   - For the OLE Messaging Library, see Creating and Sending a Message.
7. Handle incoming messages.
   - For Simple MAPI, call the **MAPIFindNext** function in a loop, specifying your message class.
   - For CMC, call the **cmc_list** and **cmc_read** functions in a loop, specifying your message class to **cmc_read**.
   - For MAPI, use the following steps:
     1. Open the contents table on the receive folder, using the **IMAPIContainer::GetContentsTable** method.
     2. Register for notification of new mail, using the **IMsgStore::Advise** method, the *fnevNewMail* event, and a null entry identifier. Do this before retrieving any messages to avoid a race condition that would result in missing messages.
     3. Create a property restriction to match PR_MESSAGE_CLASS with your message class and apply it to the contents table by calling **IMAPITable::Restrict**. Consider also limiting your contents table view to unread messages by creating a bitmask restriction with the MSGFLAG_UNREAD flag as the mask for the PR_MESSAGE_FLAGS property. See About Restrictions.
     4. Retrieve all the messages from the table, using the **HrQueryAllRows** function or the **IMAPITable::QueryRows** method, and process them.
     5. Rely on new mail notifications to advise you of further incoming messages. Your notification handler should check the **lpszMessageClass** member of the **NEWMAIL_NOTIFICATION** structure and ignore any messages that are not of your class.
   - For the OLE Messaging Library, access the Messages collection of your receive folder, and use the **GetFirst** and **GetNext** methods to retrieve the messages.
8. For each user of your application, make the .EXE file and any auxiliary files available.

About Sample Source Code

Timecard application. Creates and sends messages – one IPM type and one IPC type – with custom fields using Visual Basic and the OLE Messaging Library. It's also an interesting example of how to distribute a standalone application through e-mail. The executables are named TMCLI.EXE and TMSRV.EXE; the sample source code is in the TIMECARD.CLI directory. It is written in Visual Basic 4.0 and supports all platforms.

Creating and Configuring a Profile

This topic explains how to create a profile using MAPI, how to add messaging services to it and configure them, and how to establish certain default settings in the profile. See Message Services and Profiles.

Design Tasks

1. Understand what message services your profile requires, and whether you wish to give your user any choice of services.
2. Choose your tool. There are several tools available for creating profiles. Alternatives include:
   - Not creating one and using the messaging client's profile. This is usually the best choice for interpersonal (IPM) applications, especially if you want to use the same message store as the messaging client.
   - Using the New Profile wizard to create a profile and having your user select message services interactively. For this technique to succeed, the services you require must all support Wizard-based configuration, and your application must tolerate uncertainty in its configuration. See the **LAUNCHWIZARDENTRY** function.
   - Using the NEWPROF.EXE utility and a template contained in a .PRF file to create a profile from a setup program or batch environment, or by spawning the utility from your own program.
   - Writing C or C++ code to create a profile. This is often the best choice for a non-interactive application which requires a specific set of message services. See About Profile Administration.
3. Understand in what order to add services. Following are guidelines for ordering service providers. These guidelines will occasionally conflict; when they do, you will have to write C or C++ code to configure one or more of the provider orders. You can use the Delivery and Addressing pages of the Mail and Fax control panel applet to inspect a profile you have created and see if the providers have been ordered as you expected.
   - Default message store. The first message store added to a profile which is capable of being the default store becomes the default store. Ordinarily, the service you intend to contain the default store should be listed first. See **IMAPISession::SetDefaultStore**.
   - Personal address book (PAB), default directory, and address book search order. The PAB, if not set explicitly, is the first container that is writeable and can contain names (according to its PR_CONTAINER_FLAGS property). The default directory is the first container in the hierarchy that contains names and is not the PAB (unless the PAB is the only container that can contain names). The default search path is the PAB, followed by each directory that is of display type DT_GLOBAL, has names, and does not have the AB_NOT_DEFAULT flag set in PR_CONTAINER_FLAGS. If there are no directories of type DT_GLOBAL, the default search path is just the PAB and the default directory. See Resolving a Name, IAddrBook::SetPAB, and IAddrBook::SetSearchPath.
   - Transport order. If the transport order is not explicitly set, MAPI services transports in the same order they were added to the profile, except that transports which set the STATUS_XP_PREFER_LAST flag are serviced last. The transport order is unimportant unless your profile contains two or more transports which handle the same e-mail address type. See **IMsgServiceAdmin::MsgServiceTransportOrder**.

If your application must run unattended, perhaps as a background application or a Windows NT service, some special cautions apply. See Writing an Automated Client and Windows NT Service Client Applications.

If your application requires a specific profile other than the default profile, you must save its name in your own configuration database or use a specific naming convention. MAPI does not expose any profile attributes other than the name and default flag in the profile table, and the default profile flag is reserved for the messaging client and related IPM applications.

Implementation Tasks

To create a .PRF file for use with the NEWPROF.EXE utility

1. Assemble information from existing .PRF files. For each service you plan to use, copy the section that lists the name, type, and property identifier for each configuration property. The section must have the same name as the service. Services implemented by Microsoft are listed in Section 4 of the DEFAULT.PRF file that is installed with MAPI.
2. Place the profile name in the **ProfileName=** line of the [General] section.
3. Create the [Service List] section, in Section 2 of the file. In this section, list each service you require and the name of the section that contains its configuration properties. Refer to the earlier guidelines to establish the order of services.
4. For each service listed in the [Service List] section, create a section that lists the value for each configuration property; the section must have the same name as the service. This information appears in Section 3 of the file.
5. Invoke the utility. On 32-bit operating systems, it can be run from a command line or batch file. On 16-bit Windows, NEWPROF.EXE cannot be run from an MS-DOS shell; it can be run by a Windows-based setup script, by the **Run** command on the **File** menu in Program Manager or File Manager, or by the **WinExec** system call. It takes command line parameters as follows:
   - **-P** descriptor-file-path

     An absolute path to the .PRF file. The default value is DEFAULT.PRF in the Windows directory.
   - **-S**

     If this option is present, NEWPROF.EXE displays a logging window showing what is being executed and any errors it encounters. By default no window is displayed.
   - **-X**

     If this option is present, NEWPROF.EXE immediately creates the profile. By default, it waits until the user chooses the Execute button in its window. This option is required if NEWPROF.EXE is to run without displaying its window.
   - **-Z**

     If this option is present, NEWPROF.EXE displays error messages and other debugging information in its window. By default this information is not displayed.

Note References to Sections 1, 2, 3, and 4 in the preceding procedure refer to comments in the DEFAULT.PRF file distributed with the MAPI SDK.

To write custom C or C++ code to create a profile

1. Read the header file for each service. Understand what properties you need to configure and what values you will use.
2. Call the **MAPIAdminProfiles** function to obtain an **IProfAdmin** interface. Call the **CreateProfile** method to create your profile, and the **AdminServices** method to obtain an **IMsgServiceAdmin** interface.
3. Add message services to the profile. Refer to the previous guidelines for the order you should use. For each service, use the **IMsgServiceAdmin** interface to:
   a. Call the **CreateMsgService** method.
   b. Obtain the **MAPIUID** structure of the service you just created.
      1. Call the **GetMsgServiceTable** method to obtain an **IMAPITable** interface.
      2. Call the **HrQueryAllRows** function to retrieve all rows from the table.
      3. Get the PR_SERVICE_UID column from the last row. This is the **MAPIUID** structure of the last service added. You may wish to check with an assertion that other properties of

the service are as you expect.

   c. Call the **ConfigureMsgService** method, passing the **MAPIUID** structure of the service you just created and a property value array with its configuration properties.

4. If you must make configuration calls that require an **IMAPISession** interface, such as **IMAPISession::SetDefaultStore**, **IAddrBook::SetPAB**, or **IAddrBook::SetABSearchPath**, pass the MAPI_NO_MAIL flag to the **MAPILogonEx** function.
5. To make your profile temporary, call the **IProfAdmin::DeleteProfile** method immediately after logging on to the profile. It will be deleted after you log off, and will not be visible to other applications in the meantime.

About Sample Source Code

None available at this time.

Incorporating Formatted Text and Attachments

This topic explains what is necessary for a client application to do to send or receive a mesage that includes formatted text and attachments. Clients that want to work with formatted text must use the MAPI client interface. CMC, Simple MAPI, and the OLE messaging library do not support formatted text in messages.

Design Tasks

1. Decide on a mechanism for displaying formatted text. You can use the Rich Text Format (RTF) supported by Microsoft or your own customized format. See About Supporting Formatted Text.
2. Decide on the types of attachments that your client will handle. See About Message Attachments.
3. Decide if and how your client will activate and save its attachments. See Adding a Message Attachment.

Implementation Tasks

To send a message with formatted text

1. Call the **IMAPIProp::OpenProperty** method to open the PR_RTF_COMPRESSED property, setting both the MAPI_CREATE and MAPI_MODIFY flags. MAPI_CREATE insures that any new data replaces any old data and MAPI_MODIFY enables your client to make those replacements.
2. Call the **WrapCompressedRTFStream** function, passing STORE_UNCOMPRESSED_RTF if the message store sets the STORE_UNCOMPRESSED_RTF bit in its PR_STORE_SUPPORT_MASK property, to get an uncompressed version of the PR_RTF_COMPRESSED stream returned from **OpenProperty**.
3. Write the message text data to the uncompressed stream returned from **WrapCompressedRTFStream**.
4. If the message data is an attachment, write the character token "\objattph" followed by a space to the stream instead of the attachment.
5. Set the PR_RENDERING_POSITION property of the attachment to a value that will increase with each attachment. For example, the first attachment could be assigned 1 as its PR_RENDERING_POSITION, the second one 2, and so on.
6. Commit and release both the uncompressed and compressed streams.
7. If the message store does not support RTF as indicated by the absence of the STORE_RTF_OK setting in the PR_STORE_SUPPORT_MASK property, call **RTFSync** and pass the RTF_SYNC_RTF_CHANGED flag.

For more information, see Supporting Formatted Text in Outgoing Messages: Client Responsibilities.

To read a message with formatted text

1. Call **RTFSync** to synchronize the message text with the formatting if the message store is not RTF-aware and the PR_RTF_IN_SYNC property is missing or set to FALSE. The RTF_SYNC_BODY_CHANGED flag should be passed in the *ulFlags* parameter. Clients working with RTF-aware message stores need not make the **RTFSync** call because the message store takes care of it.
2. Call **IMAPIProp::SaveChanges** if the message has been updated.
3. Call **IMAPIProp::OpenProperty** to open the PR_RTF_COMPRESSED property.
4. Call the **WrapCompressedRTFStream** function, passing the STORE_UNCOMPRESSED_RTF flag if the message store sets the STORE_UNCOMPRESSED_RTF flag in its PR_STORE_SUPPORT_MASK property, to create an uncompressed version of the compressed RTF data.
5. Display the uncompressed RTF data.

If there are attachments in the message, perform the following tasks in addition to the preceding steps:

1. Before reading from the uncompressed RTF stream, sort the message's attachment table on the value of the PR_RENDERING_POSITION property. The attachments will now be in order by how they appear in the message.
2. As your client scans through the RTF stream, check for the token "\objattph". The character following the token is the place to put the next attachment from the sorted table. Handle attachments that have set their PR_RENDERING_POSITION property to -1 separately.

For more information, see Supporting Formatted Text in Incoming Messages: Client Responsibilities.

About Sample Source Code

Send RTF utility. Sends a message with formatted text based on command line parameters. The executable is named SNDRTF.EXE; the sample source code is in SENDRTF.CLI. It is written in C++ and supports 32-bit platforms only.

Finding a Message in a Message Store

This topic explains how to perform various tasks related to finding a message in a message store. The tasks are different, depending on the client interface you are using for your client and the message store. Some message stores support searching; others do not. Create a prototype to determine whether or not a particular search strategy will work.

If the message store supports searches, your client can use a search-results folder to locate messages. Clients written with the MAPI client interface can either create a new search-results folder or use a pre-existing one. Clients written with the OLE Messaging Library must use a pre-existing search-results folder.

If the message store does not support searches, your client will have to traverse the message store hierarchy, looking through each folder for the target message rather than using search criteria.

Specifically, this topic explains how to:

- Create a search-results folder using the MAPI client interface.
- Locate an existing search-results folder using the OLE Messaging Library.
- Perform a search in a searchable message store using the MAPI client interface.

Design Tasks

1. Determine whether or not the message store supports searches, meaning the ability to create search-results folders. Clients using message stores that support searches can use a search-results folder to find a message; clients using non-searchable message stores must traverse the folder hierarchy.
2. For clients using message stores that support searches, determine whether or not the message store supports the type of restrictions you will be imposing. For example, if you want to use a property restriction, make sure that the message store supports the specific property.

Implementation Tasks

1. Retrieve the PR_STORE_SUPPORT_MASK property from the message store by calling its **IMAPIProp::GetProps** method. If the STORE_SEARCH_OK flag is set, proceed. If this flag is not set, there are two choices:
   - Fail because the message store does not support searches.
   - Attempt the alternate strategy of enumerating the message store hierarchy.
2. Create a search-results folder if possible. See the following procedure and About Creating Folders.
3. If creating a search-results folder is not possible, locate an existing search-results folder. MAPI creates a default search-results folder called Search Root when it initializes a message store. Creating search-results folders is not allowed if you are using the OLE Messaging Library. See the following procedure.
4. Build a restriction that reflects your search criteria. See About Restrictions. If you are searching on a named property, call the **IMAPIProp::GetIDsFromNames** method to create a searchable property tag from the name. See About IMAPIProp::GetIDsFromNames.
5. Start the search by calling the search-results folder's **IMAPIContainer::SetSearchCriteria** method. Pass the BACKGROUND_SEARCH or FOREGROUND_SEARCH flag as appropriate, the restriction, and the entry identifier of the folder in which to search.
6. Retrieve the search results by opening the search-results folder's contents table and querying the rows. You can process this incrementally or wait for an *fnevSearchComplete* notification from the message store.

To create a search-results folder using the MAPI client interface

1. Locate and open the target message store. See Opening a Message Store or Opening the Default

Message Store.

2. Retrieve the message store's PR_FINDER_ENTRYID property, the entry identifier of the root search-results folder.
3. Open the root search-results folder by calling **IMsgStore::OpenEntry** with the entry identifier from the PR_FINDER_ENTRYID property.
4. Create a new folder in the root search-result folder by calling the root folder's **IMAPIFolder::CreateFolder** method. Set the *ulFolderType* parameter to FOLDER_SEARCH. See About Search-Results Folders.

To locate an existing search-results folder using the OLE Messaging Library

1. Locate and open the target message store using the Session.InfoStores collection. An InfoStore object provides access to the root folder for a message store.
2. Retrieve the Folder's FolderID property to identify its parent folder and pass this identifier in a call to Session.GetFolder to open the parent folder.
3. Work up through the folder hierarchy by repeating the previous step until the FolderID property is the same in two sequential iterations. When this occurs, you will be at the root folder of the message store.
4. Retrieve the Folders collection and browse for the folder whose name is Search Root. MAPI always creates a folder by that name when it initializes a message store.
5. Retrieve the root search-results folder's Folders collection and browse for your own search-results folder by name.

To locate an existing search-results folder using the MAPI client interface

1. Locate and open the target message store. See Opening a Message Store or Opening the Default Message Store.
2. Retrieve the message store's PR_FINDER_ENTRYID property, the entry identifier of the root search-results folder.
3. Open the root search-results folder by calling **IMsgStore::OpenEntry** with the entry identifier from the PR_FINDER_ENTRYID property.
4. Access the folder's hierarchy table by calling **IMAPIFolder::GetHierarchyTable**.
5. Look for a row in the table that has a PR_FOLDER_TYPE column set to FOLDER_SEARCH and a PR_DISPLAY_NAME set to Search Root. MAPI always creates a folder by that name when it initializes a message store.

About Sample Source Code

Spooler hook provider. Includes a function called **HrInitUnreadSearch** that shows how to create and initialize a search-results folder in the root folder of a message store's IPM subtree. The executable is named SMH.DLL. It supports 32-bit Windows platforms only.

Using Message Filtering to Manage Messages

The MAPI spooler makes calls to two different types of extensions by which code can be inserted in the message transmission process. These extensions, known as preprocessors and spooler hooks, can be used for a wide variety of purposes, including altering the recipient list or content of an outbound message; archiving outbound messages in local storage or on a central server; directing inbound messages to a particular folder based on arbitrary criteria; and responding automatically to inbound messages.

Design Tasks

1. Decide whether to use a preprocessor or a spooler hook.

   A preprocessor is called for outbound messages only. It may choose to be called for all messages, or for messages that have recipients of a particular type. Those recipients can be selected based on the address type (PR_ADDRTYPE), on the **MAPIUID** which qualifies the recipient's entry identifier, or both. See IMAPISupport::RegisterPreprocessor.

   A preprocessor can create new messages based on the input message, returning them through its *lpppMessage* parameter. In no case should the input message be placed in the *lpppMessage* parameter. If a preprocessor does not want the input message to be sent, it should delete all the recipients and set the PR_DELETE_AFTER_SUBMIT property; the input message should not be deleted using message store calls.

   A spooler hook may choose to be called for all inbound messages, all outbound messages, or both, by setting the HOOK_INBOUND flag, the HOOK_OUTBOUND flag, or both in its PR_RESOURCE_FLAGS property on the provider profile section, in MAPISVC.INF. See About the MAPISVC.INF File.

   To delete an inbound or outbound message, a hook should set the HOOK_DELETE flag in its *lpulFlags* parameter. It should not use message store calls to delete the message presented to it through its interface; it may use message store calls to create, modify, or delete other messages. A hook can prevent processing of a message by other hooks, including the default hook that processes receive folder settings, by setting the HOOK_CANCEL flag in *lpulFlags*.

   When working with a tightly coupled message store and transport, the store itself can transmit a message that is not destined for any spooler-based transports. Whether preprocessors and hooks are called in this situation depends upon the message store implementation. Microsoft Exchange Server, for example, calls preprocessors on outbound messages but does not call either inbound or outbound spooler hooks. See About Sending Messages.

   On an outbound message, hooks and preprocessors participate in a complicated sequence of events. The following steps involve hooks and preprocessors.

   a. The **PreprocessMessage** entry point of each preprocessor is called before any transport provider handles the message. It can update the recipient list, alter message content, or even create additional messages. The order in which preprocessors are called is the same as the order in which the transports that registered them are called to handle outbound messages. See Creating and Configuring a Profile.

   b. Transport providers are called to transmit the message. If any transport defers message processing, no hooks are called until the deferred processing is complete.

   c. The **RemovePreprocessInfo** entry point of each preprocessor is called after all transports have handled the message.

   d. The **ISpoolerHook::OutboundMsgHook** method is called for each hook that sets the HOOK_OUTBOUND flag in the PR_RESOURCE_FLAGS property. The order in which hooks are called is the same as the order in which they were installed into the profile; it does not follow the transport order, because a hook does not necessarily have an associated transport provider.

   For inbound messages, preprocessors are not called. Hooks are called as follows.

   a. The receiving transport provider completes its work.

b. The **ISpoolerHook::InboundMsgHook** method is called for each hook that sets the HOOK_INBOUND flag in the PR_RESOURCE_FLAGS property. The order in which hooks are called is the same as the order in which they were installed into the profile; it does not follow the transport order, because a hook does not have an associated transport provider.
c. For inbound messages, the receive folder assignment is honored only if a hook has not moved the message to another folder. Conceptually, the MAPI spooler implements an internal hook which is always called last and only if no previous hook returns the HOOK_CANCEL flag; it finds the receive folder based on the message's PR_MESSAGE_CLASS and places the message in that folder. See **IMsgStore::GetReceiveFolder**.

2. Define configuration parameters and understand how your hook or preprocessor is to be installed. A spooler hook is a distinct service provider type and is added to the user's profile as part of a message service, either alone or together with other, related service providers. Any configuration parameters are normally stored in the profile and edited using the property pages for the message service. A preprocessor is not a distinct service provider type; it must be registered by a transport provider. If necessary, a minimal transport provider can be created for this purpose alone.

Implementation Tasks

To implement a preprocessor

- Create a DLL containing the **PreprocessMessage** and **RemovePreprocessInfo** entry points. It is strongly recommended, though not required, that all the code and entry points listed here be implemented in the same DLL; this minimizes the delay in loading MAPI applications.
- Create a call to the **IMAPISupport::RegisterPreprocessor** method in a transport provider.
- Create a transport provider entry point, and the remainder of a minimal transport, if there is no existing transport. The minimal transport should initialize as recommended in the topic Required Functionality for Transport Providers. In response to the **IXPLogon::AddressTypes** call, the transport provider should return a single zero-length string in *lpppszAdrTypeArray* and NULL in *lpppUIDArray*.
- Create a MAPISVC.INF fragment for the transport provider and the message service that contains it. See Implementing a Message Service.
- Optionally, but strongly recommended, create a message service entry point for configuration. See About Message Service Entry Point Functions.
- Optionally, create an online Help file linked to your configuration property pages, wizard pages, or both, providing full details about all configuration options.
- Optionally, create a header file for custom programmatic configuration by MAPI clients.
- Optionally, create a service wizard entry point for interactive configuration by users. See Supporting the Profile Wizard.
- Optionally, create a .PRF file detailing configuration properties for the message service.

To implement a spooler hook

- Create a DLL containing the **HPProviderInit** entry point. It is strongly recommended, though not required, that all the code and entry points listed here be implemented in the same DLL; this minimizes the delay in loading MAPI applications. See About Provider DLL Entry Point Functions.
- Create a MAPISVC.INF fragment for the hook provider and the message service that contains it. See Implementing a Message Service.
- Optionally, but strongly recommended, create a message service entry point for configuration. See About Message Service Entry Point Functions.
- Optionally, create a header file for custom programmatic configuration by MAPI clients.
- Optionally, create a service wizard entry point for interactive configuration by users. See Supporting the Profile Wizard.
- Optionally, create an online Help file linked to your configuration property pages, wizard pages, or both, providing full details about all configuration options.

- Optionally, create a .PRF file detailing configuration properties for the message service.

About Sample Source Code

Peer-to-peer transport service. Uses files and directories to transmit and receive messages. It implements and registers a very simple preprocessor that appends a line of text to each outbound message. The executable is named SMPXP.DLL; the sample source code is in the PEER.XP directory. It is written in C and supports all platforms.

Sample spooler hook. Processes inbound and outbound messages. It can archive sent and deleted messages into folders by month and year, place incoming messages into folders based on variable criteria, and automatically respond to inbound messages with formatted text. The executable is named SMH.DLL; the sample source code is in the MANAGER.SH directory. It is written in C and supports Win32 platforms only.

Finding Information About a User in the Address Book

This topic explains how to find a user in the address book given a name, part of a name, or other information about the user; how to retrieve information from that entry; and how to search for several users at a time.

Design Tasks

1. Decide whether you're searching the entire address book, one or more containers belonging to a specific message service, or a particular type of container such as a personal or global address list.
2. Decide what properties are in your search criteria. The only property you can rely on being searching with acceptable performance is PR_DISPLAY_NAME. If you want to use another property such as an account name, employee identification number, or organization name, you must experiment with the address book providers you expect to use to verify that searching on those properties is supported and the performance is acceptable.
3. Understand whether or not the information you are given should identify each user uniquely. If it may be ambiguous – if it may match more than one entry in the address book – then decide whether your program should:
   - Prompt its user to choose the right entry.
   - Choose the right entry itself based on further information retrieved about each entry.
   - Fail.

Implementation Tasks

To search the entire address book by display name for one or more entries

- Use the **IAddrBook::ResolveName** method. You can optionally request a dialog box that prompts the user to choose the correct entry in case the information you were given matches more than one entry.

To open the user's personal address book (PAB)

1. Retrieve its entry identifier by calling the **IAddrBook::GetPAB** method.
2. Call the **IAddrBook::OpenEntry** method passing the PAB's entry identifier.

To open a specific container other than the PAB

1. Open the address book's root container by calling the **IAddrBook::OpenEntry** method with a NULL entry identifier. The root container is constructed by MAPI and contains the top-level containers of all the address book providers in the profile.
2. Get the list of top-level containers by calling the **IMAPIContainer::GetHierarchyTable** method.
3. If you want a specific container type such as the global address list, restrict the table on PR_DISPLAY_TYPE:
   a. Create a property restriction to match PR_DISPLAY_TYPE with the value for the particular type of container you want to open. For example, to open the Global Address Book, build a property restriction that matches PR_DISPLAY_TYPE with the DT_GLOBAL value.
   b. Create an **SPropTagArray** including PR_ENTRYID, PR_DISPLAY_TYPE, and any other columns of interest.
   c. Call the **HrQueryAllRows** function passing your property tag array and restriction. Ordinarily there will be a single global address list, but the call can return zero, one, or more rows depending on the address book providers in your profile. Be prepared to handle all these cases, not just one row.
   d. Call the **IAddrBook::OpenEntry** method with the PR_ENTRYID column from the row you want to open the container.
4. If you want a container belonging to a specific address book provider, restrict the table on PR_AB_PROVIDER_ID:

a. Create a property restriction to match PR_AB_PROVIDER_ID with the value that represents the target address book provider. You will find this property value in a header file created by the service provider. In the MAPI SDK, for instance, the value for the flat file address book sample is SAB_PROVIDER_ID in SMPAB.H.
b. Create an **SPropTagArray** structure including PR_ENTRYID, PR_AB_PROVIDER_ID, and any other columns of interest.
c. Call the **HrQueryAllRows** function passing your property tag array and restriction. The call will return zero rows if the provider you want is not in the profile. It can return one or more rows depending on the provider's containers are organized, but it will only return the top-level rows.
d. Call the **IAddrBook::OpenEntry** method with the PR_ENTRYID column from the row you want to open the container. If the container you want is not a top-level container, you will then have to navigate down through the latter's hierarchy table.

To search in a specific address book container

1. Open its contents table by calling its **GetContentsTable** method.
2. Use **IMAPITable::FindRow**, **IMAPITable::SortTable**, and **IMAPITable::Restrict** to search the contents table just as you can any MAPI table. See Table Positioning and About Restrictions. These methods are subject to limitations in the provider's implementation, and their speed is unpredictable; careful testing against any providers you expect to use is indispensible. In addition to the normal **IMAPITable** methods, there are two search techniques specific to address book containers:
   - Restriction on PR_ANR (ambiguous name resolution). Although formatted as an ordinary **SPropertyRestriction** structure, this restriction actually invokes a special display name matching algorithm based on separating the string you give it into words and prefix-matching those words against display names in the address book. See Implementing the PR_ANR Property Restriction.
   - **IABContainer::ResolveNames**. This method does the same work as a PR_ANR restriction for multiple names, and can be much faster. Providers are not required to implement it, and generally do not unless it carries a significant performance benefit. It is not necessary to open the contents table in order to user this method. See Implementing **IABContainer::ResolveNames**.
3. After applying any of the restriction methods listed above, call the **IMAPITable::QueryRows** method to retrieve any rows that match the restriction. You may get back zero, one, or more rows that match.
4. Retrieve additional information for a single address book entry by calling the **IAddrBook::OpenEntry** method passing its entry identifier, then call **GetProps** on the resulting **IMAPIProp** interface. To retrieve additional properties for multiple address book entries, you can of course call **OpenEntry** for each one, but it is usually much more efficient to call the **IAddrBook::PrepareRecips** method. **PrepareRecips** requires an **ADRLIST** structure.

About the Sample Code and SDK Tools

Note All executable names have "32" appended to them when built for 32-bit platforms. The routing client, for instance, is named ROUTE.EXE when built for 16-bit Windows, but ROUTE32.EXE when built for 32-bit Windows.

Address book viewer. Displays address book container, user, and distribution list properties in raw form as well as through the standard address book dialog box, and exercises nearly all methods on address book provider interfaces. It is useful both for exercising address book providers and for understanding what properties are present on objects in an address book. The executable is named ABVIEW.EXE; the sample source code is in the ABVIEW.CLI directory. It is written in C++ using the Microsoft Foundation Classes and supports all platforms.

Checkers form. Implements the MAPI form interfaces and a more elaborate message; each message represents a move in a game of checkers between two players. The executable is named WCHECK.EXE; the sample source code is in the CHECKERS.FRM directory. It is written in C++ and supports all platforms.

CMC client. Demonstrates the use of all CMC calls in a very simple messaging client. It can send and read messages, list the contents of the Inbox, and view the address book. The executable is named CMCCLI.EXE; the source code is in the CMC.CLI directory. It is written in C and supports all platforms.

Command extension. Adds a simple command to the messaging client's menu. The executable is named CMDEXT.DLL; the sample source code is in the COMMAND.EXT directory. It is written in C++ and supports Win32 platforms only.

Docfile message store. Uses directories as folders and OLE 2.0 docfiles as messages (with the MAPI message-on-storage utility). It supports all features required of a default message store. Neither its **IMAPIProp** nor its **IMAPITable** interfaces are native; they are provided by the message-on-storage facility and **ITableData** respectively. The executable is named SMPMS.DLL; the sample source code is in the DOCFILE.MS directory. It is written in C and supports all platforms.

Event extension. Adds a new-mail event handler to the messaging client. The executable is named EVEXT.DLL; the sample source code is in the EVENT.EXT directory. It is written in C++ and supports Win32 platforms only.

Flat file address book service. Supports a single read-only container with names read from a flat binary file containing display names and e-mail addresses. It supports one-off templates and all configuration options except the wizard. The executable is named SMPAB.DLL; the sample source code is in the FLATFILE.AB directory. It is written in C and supports all platforms.

Property viewer. Displays the properties directly accessible through an **IMAPIProp** interface implementation, and exercises all methods of the interface. The executable is named PROPVU.DLL; the sample source code is in the PROPVU.CLI directory. It is written in C++ using the Microsoft Foundation Classes and supports all platforms.

Table viewer. Displays the properties directly accessible through an **IMAPITable** interface implementation, and exercises all methods of the interface. The executable is named TBLVU.DLL; the sample source code is in the TABLEVU.CLI directory. It is written in C++ using the Microsoft Foundation Classes and supports all platforms.

Message store viewer. Displays message, folder, and message store properties in raw form, and exercises nearly all methods on message store provider interfaces. It is useful both for exercising message store providers and for understanding what properties are present objects in a store. The executable is named MDBVU.EXE; the sample source code is in the MDBVIEW.CLI directory. It is written in C++ using the Microsoft Foundation Classes and supports all platforms.

Peer-to-peer transport service. Uses files and directories to transmit and receive messages. It illustrates how to split message content between TNEF and text. It also supports all configuration options (property sheets, wizard, and programmatic configuration) and message options. It does not support the remote transport interfaces. The executable is named SMPXP.DLL; the sample source code is in the PEER.XP directory. It is written in C and supports all platforms.

Property sheet extension. Adds to the messaging client's standard message property sheets, for messages belonging to the IPM.Document class. The new property sheet displays the document's summary properties – the author, title, date, and so forth. The executable is named PRSHEXT.DLL; the sample source code is in the PROPSH.EXT directory. It is written in C++ and supports Win32 platforms only.

Remote transport service. Uses named pipes to communicate with a server process running on Windows NT. It uses TNEF exclusively to encode message content. It supports all configuration options and the remote transport interfaces. The transport executable is named XPWDSR.DLL, and the source code is in the REMOTE.XP directory. The sample server source code is in the REMOTE.SRV directory. Both are written in C++; the transport supports Win32 platforms only, and the server supports Windows NT only.

Routing client. Creates a message with an attached document and a simple linear route using MAPI. It can also send and read forms. The executable is named ROUTE.EXE; the sample source code is in the ROUTE.CLI directory. It is written in C and supports all platforms.

Sample profile tool. MKPROF.EXE, creates and configures a profile using some of the Microsoft message services. Its source code is in the PROFILE.CLI directory. It is written in C++ and supports all platforms.

Sample spooler hook. can archive sent and deleted messages into folders by month and year, place incoming messages into folders based on variable criteria, and automatically respond to incoming messages with a rich-text message. The executable is named SMH.DLL; the sample source code is in the MANAGER.SH directory. It is written in C and supports Win32 platforms only.

Send RTF utility. Sends a message with Rich Text Format content based on command line parameters. The executable is named SNDRTF.EXE; the sample source code is in SENDRTF.CLI. It is written in C++ and supports Win32 platforms only.

Simple form. Implements the MAPI form interfaces and a very simple message, containing only text and recipients. It is a good basis for reuse in a more complicated form. The executable is named SMPFRM.EXE; the sample source code is in the SIMPLE.FRM directory. It is written in C++ and supports Win32 platforms only.

Simple MAPI client. Demonstrates the use of all simple MAPI calls in a very simple messaging client. It can send and read messages, list the contents of the Inbox, and view the address book. The executable is named SMPCLI.EXE; the sample source code is in the SIMPLE.CLI directory. It is written in C and supports all platforms.

Simple MAPI stress mailer. Generates message traffic using Simple MAPI. It can send a message any number of times, at any interval, with message text and attachments of any size. The executable is named SEND.EXE; the sample source code is in the SEND.CLI directory. It is written in C++ using the Microsoft Foundation Classes and supports all platforms.

Status table viewer. Displays the MAPI status table in raw form, and exercises provider **IMAPIStatus** interfaces. The executable is named STATVU.DLL; the sample source code is in the STATUSVU.CLI directory. It is written in C++ using the Microsoft Foundation Classes and supports all platforms.

Timecard application. Demonstrates creating and sending messages with custom fields using the OLE Messaging Library. The executables are named TMCLI.EXE and TMSRV.EXE; the sample source code is in the TIMECARD.CLI directory. It is written in Visual Basic 4.0 and supports all platforms.

Concepts and Architecture

As business becomes more competitive, the success of an organization increasingly depends on how quickly, smoothly, and efficiently people within that organization work together. The key to a successful organization is how well that organization manages and distributes information. Networking is an important part of teamwork because it enables fast and efficient information exchange. But networking is only part of the solution; organizations must also keep track of the information and manage its distribution. Electronic messaging systems provide these capabilities.

Electronic messaging has become critically important to enterprise computing. In fact, many organizations are looking to their electronic messaging systems to take on the role of a central communications backbone, used not only for electronic-mail (e-mail) messages, but to integrate all types of information. Electronic messaging provides a way for users in organizations to retrieve information from a variety of sources, to exchange information automatically, and to store, filter, and organize the information locally or across a network.

Today, powerful enterprise-wide workgroup applications that manage group scheduling, forms routing, order processing, and project management are built on electronic messaging systems. Hundreds of different messaging systems are offered by different vendors, and a wide range of applications have been built to use them. But each of these messaging systems has a different programming interface, making an extensive development effort necessary to enable an application to interact with more than one system.

To solve this problem, Microsoft, along with more than 100 independent software vendors (ISVs), messaging system suppliers, corporate developers, and consultants from around the world, has created the Messaging Application Programming Interface (MAPI). MAPI is a messaging architecture that enables multiple applications to interact with multiple messaging systems seamlessly across a variety of hardware platforms.

MAPI is made up of a set of common application programming interfaces and a dynamic-link library (DLL) component. The interfaces are used to create and access diverse messaging applications and messaging systems, offering a uniform environment for development and use and providing true independence for both. The DLL contains the MAPI subsystem, which manages the interaction between front-end messaging applications and back-end messaging systems and provides a common user interface for frequent tasks. The MAPI subsystem acts as a central clearinghouse to unify the various messaging systems and shield clients from their differences.

About MAPI Features

MAPI has several key features that enable it to provide a consistent way for developers to work with and use different messaging systems in a seamless fashion. These features are:

- An open programming interface
- A dual-purpose interface, separated into two independent parts
- A comprehensive interface
- Integration with the operating system
- Support for industry standards

Because MAPI is an open programming interface, it provides services in a generic way, allowing its users to add any necessary customization now and in the future. The users of the MAPI interface include all levels and types of applications, also called client applications or clients, and service providers, or driver-like components that translate the features of a specific messaging system into features that any MAPI client application can access. Applications with such diverse messaging needs as a word processing application requiring only the ability to send documents and a workgroup application requiring the ability to share and store different types of data can use the MAPI programming interface to fulfill their needs. In fact, any application that has to either exchange or store information in a particular format can benefit from using the MAPI programming interface. Any service provider can use the interface to expose the unique features of its messaging system, selecting those features that provide the most benefit to application users.

MAPI provides separation between the programming interface used by the front-end messaging client applications and the programming interface used by the back-end service providers. Separating the client interface from the service provider interface enables a single application to use multiple messaging systems and a single service provider to be used by multiple applications. Every component works with a common, Windows-based user interface. This is a great benefit to users. Users can select from a variety of systems, depending on their needs at any one time, and be able to work consistently with each selected system, thereby providing true independence from specific messaging systems.

For example, one messaging client application can be used to receive messages from a fax, a bulletin board system, a host-based messaging system, and a LAN-based system. Messages from all of these systems can be placed in a single location upon arrival, or universal inbox. Having a single application handle all of these systems lessens the cost of development, user training, and system administration.

Separating the client interface from the provider interface removes any programming dependencies placed on the application by the messaging system and vice versa. Developers of client applications and service providers code to a standard set of MAPI features, rather than a diverse set of application-specific or messaging system-specific features. Developers focus only on their component, whether it be a client or service provider, and MAPI handles the rest, thereby reducing development time and costs.

The MAPI programming interface provides a comprehensive set of features. MAPI is aimed at the powerful, new market of workgroup applications – applications that communicate with such different messaging systems as fax, DEC All-In-1, voice mail, and public communications services such as AT&T Easylink Services, CompuServe, and MCI MAIL. The MAPI interface allows service providers to be made available for all of these systems.

Because workgroup applications demand more of their messaging systems, MAPI offers much more than basic messaging in the programming interface and supports more than local area network (LAN) -based messaging systems. Applications can, for example, format text for a single message with a variety of fonts and present to their users a customized view of messages that have been filtered, sorted, or preprocessed.

MAPI is built into the Microsoft Windows family of operating systems, currently a component of

Microsoft® Windows® 95 operating system and soon to be a component of Microsoft® Windows NT® operating system. Because MAPI is an integrated part of the operating system, developers of 16-bit and 32-bit Windows-based applications can have access to a consistent interface. This approach is similar to the approach used by the Windows printing system. Just as a word-processing program can print to many different printers through the Windows printing system as long as the necessary drivers are installed, so can any MAPI-compliant application communicate with any messaging system as long as the appropriate service providers are installed.

The programming interface and subsystem contained in the DLL are based on the object oriented concepts introduced with another integrated part of the Windows operating systems, OLE. MAPI-compliant objects conform to the OLE methodology known as the Component Object Model (COM). COM objects implement a set of methods that belong to one or more interfaces, or collections of related functions that define how objects behave and operate within the COM world. Users of COM objects access them only through pointers to these interfaces.

In addition to the DLL and programming interface, several other components are included with the operating system as part of MAPI. There are standard messaging client applications that demonstate different levels of messaging support. There are sample address book, message store, and transport providers written in both C and C++. You can use these components either for testing your own client or service provider or as a learning tool.

MAPI provides cross platform support through such industry standards as SMTP, X.400, and Common Messaging Calls (CMC). MAPI applications can be run on all 16-bit Windows 3.x platforms and both 16-bit and 32-bit platforms on Windows 95 and Windows NT. MAPI is the messaging component of an open architecture standard known as the Microsoft Windows Open Services Architecture (WOSA). WOSA standards exist or are being developed in the areas of database, directory, security, and messaging technology among others. WOSA programming interfaces allow developers to create front- and back-end components with the confidence that these products can be easily connected in a distributed computing environment.

About the MAPI Architecture

MAPI defines a modular architecture, as shown in the following illustration.

{ewc msdncd, EWGraphic, groupx827 0 /a "MAPI_43.WMF"}

There are basically three types of client applications: messaging-aware, messaging-enabled, and messaging-based. These applications are known as client applications because they are clients of the MAPI subsystem. Messaging-aware and messaging-enabled applications are typically applications not primarily focused on messaging; these are word processing or spreadsheet applications, for example, that incorporate messaging as an added feature. Messaging-based applications employ messaging as a central part of their processing and offer extensive messaging features, such as the exchange of information of various types in various formats and the ability to save and organize the information locally. Electronic mail, scheduling, and work flow applications are examples of messaging-based applications.

The MAPI subsystem is made up of the MAPI spooler, a common user interface, and the programming interfaces. The MAPI spooler is a separate process responsible for sending messages to and receiving messages from a messaging system. The common user interface is a set of dialog boxes that gives client applications a consistent look and users a consistent way to work.

MAPI has programming interfaces that are used by the MAPI subsystem, by client application writers, and by service provider writers. The main programming interface is an object-based interface known as the MAPI programming interface. Based on the OLE Component Object Model, the MAPI programming interface is used by the MAPI subsystem and by messaging-based client applications and service providers written in C or C++.

Client application writers have a choice of three other interfaces:

- Simple MAPI. An API function-based client interface for applications written in C, Microsoft Visual C++, or Microsoft Visual Basic.
- Common Messaging Calls (CMC). An API function-based client interface for applications written in C or C++.
- OLE Messaging Library. An object-based client interface for applications written in C, C++, Visual Basic, or Visual Basic for Applications.

The Simple MAPI, CMC, and OLE Messaging Library client interfaces are primarily for messaging-aware and messaging-enabled client applications. These interfaces are less complex; applications that require fewer messaging features can use Simple MAPI, CMC, or the OLE Messaging Library to implement these features quickly and easily.

Client application writers have the option of making MAPI calls either directly through the MAPI programming interface or indirectly through one of these three client-only interfaces. Messaging can be implemented with a single MAPI client interface or a combination of interfaces. A single application can make calls to methods or functions belonging to any of the interfaces.

The service provider programming interface is the part of the object-based MAPI programming interface that applies specifically to service providers. Service providers, positioned between the MAPI subsystem and the underlying messaging systems, translate requests from MAPI-compliant client applications into tasks a specific messaging system can understand. When tasks are complete, the service providers translate again, converting status and information that is messaging system-specific into a MAPI format. As with client applications, there are different types of service providers. Each type handles a different messaging system service. The address book provider, for example, works with directory information while the transport provider handles message transmission and reception.

About Client Applications

A MAPI client application is any application that uses one of the three MAPI client interfaces (Simple MAPI, CMC, and the OLE Messaging Library) or the MAPI programming interface. Client applications implement messaging tasks as either their primary or secondary focus. Messaging client applications, such as applications that send and receive electronic mail, implement messaging as their primary focus while for non-messaging client applications, such as inventory or configuration applications, it is a secondary feature.

Client applications can be organized into one of the following three categories:

- Messaging-aware applications
- Messaging-enabled applications
- Messaging-based workgroup applications

A messaging-aware application does not require the services of a messaging system, but includes messaging options as an additional feature. For example, a word processing application that includes a **Send** command in its **File** menu to allow documents to be sent is considered messaging-aware.

A messaging-enabled application requires the services of a messaging system and typically runs on a network or an on-line service. An example of a messaging-enabled application is Microsoft Mail.

A more advanced client application is the messaging-based workgroup application. The workgroup application requires full access to a wide range of messaging system services, including storage, addressing, and transport services. These applications are designed to operate over a network without users having to manage the applications' network interaction. Examples of such applications include work flow automation programs and bulletin board services.

Client applications can either include the user to create an interactive environment or operate without a user in an automated environment. While MAPI supplies a set of common dialog boxes with its standard user interface, client applications are not required to present a user interface. In fact, all processing can be handled within the application if desired. An example of an automated client application would be an inventory management application that is programmed to route items of a particular type to standard recipients on a regular basis.

About the MAPI Subsystem

The MAPI subsystem consists of the following parts:

- Client and service provider programming interfaces
- MAPI spooler
- Common user interface

The MAPI programming interface is based on a powerful, object-oriented interface that subscribes to OLE's Component Object Model, a model for object interaction. MAPI defines a set of objects that share structure and behavior, enabling developers to create and use objects in a consistent manner. There is a large feature set available, enabling client developers, for example, to provide their users with access to message or recipient properties and customized views of message and address book information. Although all types of client applications can use the object-oriented MAPI interface, typically only messaging-based applications and service providers need its power and complexity. A simpler, more restrictive API is usually sufficient for messaging-aware and messaging-enabled applications.

To support a wider audience of client application developers, there are three other API sets on top of MAPI: Common Messaging Calls (CMC), Simple MAPI, and the OLE Messaging Library. Simpler to use and understand, these API sets provide messaging functionality through either C standard function calls or Visual Basic. Client application developers can choose the API that is most suitable for their needs.

The following table describes each of the interfaces available to client applications.

Client interface	Description
Simple MAPI	Supports existing messaging-enabled and messaging-aware applications. For C, C++, or Visual Basic client applications.
CMC	Supports cross-platform applications written in C or C++.
OLE Messaging Library	Supports OLE automation controllers written in C, C++, Visual Basic, or Visual Basic for Applications.
MAPI	Supports full-featured client applications and service providers written in C or C++.

The following illustration shows how Simple MAPI, CMC, and the OLE Messaging Library are layered between MAPI and client applications. Messaging-based clients call directly into MAPI while messaging-aware and messaging-enabled clients call into Simple MAPI, the OLE Messaging Library or CMC. Calls to these higher level APIs are forwarded to MAPI.

{ewc msdncd, EWGraphic, groupx827 1 /a "MAPI_45.WMF"}

About the MAPI Spooler

The MAPI spooler is a separate process within the MAPI subsystem, responsible for sending messages to and receiving messages from a messaging system. The MAPI spooler plays a vital role in message receipt and delivery. When a messaging system is unavailable, the MAPI spooler stores the messages and automatically forwards them at a later time. This ability to hold onto or send data when necessary is known as store and forward, a critical feature in environments where remote connections are common and network traffic is high. The MAPI spooler runs as a background process, doing much of its work when a client application is idle, thus improving the responsiveness of the client application.

The MAPI spooler has additional responsibilities related to message distribution. These extra duties include:

- Keeping track of the recipient types that are handled by specific transport providers.
- Informing a client application when a new message has been delivered.
- Invoking message preprocessing and postprocessing.
- Generating reports that indicate that message delivery has occurred.
- Maintaining status on processed recipients.

The following illustration shows at a high level how a message flows from a client to the messaging system. The user of a client application sends a message to one or more recipients. The message store provider initiates the sending process, formatting the message with additional information needed for transmission. Unless the message store is tightly coupled with a transport provider that can handle all of the recipients to whom the message is addressed, the MAPI spooler receives the message, performs any required preprocessing, and delivers it to the appropriate transport provider. The transport provider gives the message to its messaging system which sends it to its intended recipient.

With incoming messages, the flow is reversed. The transport provider receives a message from its messaging system and notifies the MAPI spooler. The spooler performs any necessary postprocessing and informs the message store provider that a new message has arrived. This notification causes the client to refresh its message display, enabling the user to read the new message.

{ewc msdncd, EWGraphic, groupx827 2 /a "MAPI_46.WMF"}

About Service Providers

Between the MAPI subsystem and the messaging systems are the various service providers. Service providers are drivers that connect MAPI client applications to an underlying messaging system. There are several types of service providers, but not all types are common. Most messaging systems include three types of services: message store providers, address book or directory providers, and message transport providers. Other less typical service providers include messaging hook providers and profile providers. MAPI supports each type of service independently, allowing a vendor to offer one or more custom service providers. For example, a vendor might want to create an address book provider that uses a corporate telephone book directory of employees or create a message store provider that uses an existing database.

Service providers are typically written by software developers with specialized knowledge or experience with a messaging system. In many cases, this is the company or organization which promotes a specific messaging system. For instance, CompuServe supplies address, message store, and transport providers for the CompuServe Information Service. MAPI supplies two of its own service providers: a message store provider known as Personal Folders and an address book provider known as the Personal Address Book (PAB). These providers can be used in isolation, as a client's only message store or address book provider, or in combination with other service providers.

MAPI presents client applications with a unified view of address book and transport provider information. This integrated approach saves the client application from having to map data to the appropriate provider and the user from having to negotiate among addressing schemes used by multiple address book and transport providers. Message store provider information, however, is not unified and clients that use multiple message store providers are responsible for handling them individually.

The service providers work with MAPI to create and send messages in the following way. Messages are created using a form that is appropriate for the specific type, or class, of message. Many messages are made with the standard note form that comes with the MAPI subsystem, either by the user of a client application or programmatically without user interaction. The completed message is addressed to one or more recipients, a user or group of users designated to receive the message. A recipient might or might not have an entry in a directory owned by one of the installed address book providers. Recipients that are not associated with an installed address book provider are called custom recipients. A custom recipient can be temporary, lasting only until the message is submitted, or more permanent if it is saved in the Personal Address Book (PAB) or another modifiable address book.

When the client application sends the message, the message store provider checks that each recipient has a unique and valid address and that the message has all of the information necessary for transmission. If there is a question about a recipient, such as can occur when there are multiple recipients with the same name, an address book provider takes care of resolving the ambiguity. The message is then placed in the outbound queue.

If the message store is tightly coupled to a transport provider that can handle all of the recipients and there is no required preprocessing, the message goes directly to the transport provider without intervention from the MAPI spooler. Otherwise the MAPI spooler performs preprocessing if necessary and attempts to route the message to a transport provider based on the address type of the recipient. If the appropriate transport provider is available, the MAPI spooler transfers the message and the transport provider delivers it. If the transport provider is not available, the MAPI spooler either holds on to the message until the transport provider becomes available or sends it using another appropriate transport provider.

About Address Book Providers

Address book providers handle access to directory information. Directory information consists of data for two types of message recipients: individual messaging users and groups of messaging users who are commonly addressed together called distribution lists. Depending on the type of recipient and the address book provider, there is a wide range of information that can be made available. For example, all address book providers store a recipient's name, address, and address type.

Each address book provider organizes this data using one or more containers. The number and structure of the containers is dependent on the address book provider's implementation. For example, one address book provider might use a single container to hold all of the information, another might use one top-level container that holds subcontainers, a third might use several top-level containers, each holding subcontainers. An address book container hierarchy can be quite deep; there is no limit to the number of subcontainers that can be used.

The following illustration shows the typical MAPI address book organization.

{ewc msdncd, EWGraphic, groupx827 3 /a "MAPI_04.WMF"}

MAPI integrates all the information supplied by the installed address book providers into a single address book, presenting a unified view to the client application. The integrated list shows the top-level containers displayed by each of the installed address book providers. Most address book providers expose only a few containers (typically one to three) at the top level for inclusion in the top level of MAPI's integrated address book. For example, an address book provider might make available All Users and Local Users as two containers at the top level.

The users of client applications can view the contents of address book containers and in some cases modify it. Address book containers can be created with different access levels, depending on the address book provider. MAPI's Personal Address Book (PAB) is an example of a modifiable address book container that allows new entries to be added and existing entries to be modified or deleted. The PAB is a special container because it allows users to store copies of frequently used addresses and maintains custom recipient entries, or new entries that do not exist in the information maintained by one of the installed address book providers. Custom recipients may be created for the duration of the session only or be stored in the PAB for greater longevity.

Whereas the PAB is an address book provider supplied by MAPI, it is also a role that any modifiable address book container can play. A user can take another modifiable address book container and assign it as the PAB, replacing the MAPI Personal Address Book as the default PAB.

About Message Store Providers

Message store providers handle the storage and retrieval of messages and other information for the users of client applications. The message information is organized using a hierarchical system known as a message store. The message store is implemented in multiple levels, with containers called folders holding messages of different types. There is no limit to the number of levels in a message store; folders can contain many subfolders.

The hierarchical message store architecture is shown in the following illustration. There are two folders, one with a subfolder. Client application users can access a summary view of the messages contained within each folder or view them individually with a form. Whether the client displays a standard form supplied by MAPI or a custom form supplied by a form developer depends on the type, or class, of the message. The first folder contains note messages and uses the MAPI standard note form. The second folder contains inventory request messages and uses a custom inventory form. The information on both forms represents the properties of the message.

{ewc msdncd, EWGraphic, groupx827 4 /a "MAPI_03.WMF"}

Message store data can be used in a variety of ways. Besides the traditional electronic mail usage, folders can be used as a forum for public discussion, as a repository for reference documents, or as a container for bulletin board information, to name a few. A single message store can hold many types of information, some modifiable and some not. Multiple clients can install the same message store, making the sharing of data easy and fast.

Message store folders provide the means for sorting and filtering messages and for customizing their view in a user interface display. Links to filtered messages are held in special folders called search-results folders. The user of a client application enters filtering criteria, which MAPI refers to as a restriction, and the criteria is applied to the messages stored in one or more folders. For example, a user might want to view only those messages dealing with a particular subject with arrival dates that are more recent than last week. References to the messages that match the criteria are listed in the search-results folder and the real messages remain in their regular folders.

Messages are the units of data transferred from one user or application to another user or application. Every message contains some message text, formatted simply or more intricately, and message envelope information that is used for transmission. Some messages include one or more attachments, or additional data related to and transported with a message in the form of a file, another message, or an OLE object.

Depending on the message store provider, a user can save a new message currently under composition as well as messages that have been sent or received. Messages can be copied or moved from one folder to another with each copy becoming a separate message that can be copied, deleted, or modified individually. Another feature that some message store providers allow is the ability to change a message, once it has been received, and store it back in its folder. A user might take advantage of this feature for rotating a fax message that has arrived upside down. The correct view can be stored in the folder for later viewing.

About Transport Providers

Transport providers handle message transmission and reception; they control the interaction between the MAPI spooler and the underlying messaging system and implement security if necessary. They also take care of any preprocessing and postprocessing tasks that are necessary. There is typically one transport provider for every active messaging system.

Client applications communicate with the transport provider through a message store provider. The message store and transport provider communicate through the MAPI spooler. When an incoming message is detected, the transport provider informs the MAPI spooler and the message is delivered to the appropriate message store. To handle outgoing messages, the processing happens in reverse: the message store provider moves the message to the outbound queue, informs the MAPI spooler, and the MAPI spooler transfers it to the appropriate transport provider. When possible, calls to the transport provider are made when client applications are idle. Transport providers and the MAPI spooler operate in the background except at logon time and when prompted by the client application to flush the transmit and receive queues.

Transport providers register with MAPI to handle one or more particular types of recipient entries. When a message is ready to be sent, the MAPI spooler looks at each recipient and determines which transport provider should handle the transmission. Depending on the type of recipient, the MAPI spooler can even call upon more than one transport provider. If the MAPI spooler's first choice is unavailable, and another transport provider has also registered to handle the specific recipient type, the MAPI spooler sends the message to the alternate provider. If the unavailable transport provider is the only one that can handle the recipient, there is no choice except to wait until a connection with that provider can be reestablished.

Some messaging systems are secure systems; all potential users are required to enter a set of valid credentials before access is permitted. MAPI prevents unauthorized access to such secure messaging systems by having the transport provider validate credentials at logon time.

About Message Services

A message service defines a group of related service providers, typically service providers that work with the same messaging system. Whereas service providers perform the work of interfacing between messaging systems and the MAPI subsystem, message services perform the work of interfacing between service providers that work with a common messaging system and the user.

Message services exist to make the installation and configuration of service providers easier for users. Users never directly install or configure a service provider; this installation and configuration is completely handled by the message service, making knowledge of service provider configuration requirements unnecessary. For example, when a user wants to install the service providers for the CompuServe Information Service, that user installs a single message service. The installation component in the message service handles the installation of each of the service providers that belong to the service.

The following illustration shows the relationship between a messaging-based client application and two message services. The user invokes the installation code of each message service to add the service and its service providers to a profile. In one of the message services, there are three service providers and in the other there are two. At some later time after installation is complete, typically at logon time, the service providers in each message service are configured. The configuration code in each message service handles the configuration of the providers in the service.

{ewc msdncd, EWGraphic, groupx827 5 /a "MAPI_44.WMF"}

When a message service is installed, its installation program copies necessary files from the installation source to the user's local disk and updates a configuration file, MAPISVC.INF. The MAPISVC.INF file holds configuration settings for all of the message services and service providers that can be installed on the computer. It is organized in hierarchical sections, with links between each section at each level. The section at the top level contains information relevant for the MAPI subsystem, such as a list of all available message services, and for the online help installation. The next level has sections for each message service, with information such as the DLL filename of the message service and the name of its configuration entry point function. The third level has sections with configuration data for each service provider in the message services.

To handle configuration, a message service implements an entry point function that conforms to a prototype defined by MAPI and a tabbed dialog box known as a property sheet. MAPI calls the entry point function to service client requests that relate to profile management and the management of service providers within the message service. Property sheets are used for viewing and changing message service and service provider configuration properties.

About Profiles

A profile is a collection of information about the message services and service providers that a user of a client application wants to be available during a particular MAPI session. Every user has at least one profile; many users keep several. For example, a user might have one profile to work with a server-based message store service and another profile to work with a message store service on the local computer. A user might want to access one set of messaging systems using the appropriate transport services for part of the day and another set for the rest of the day. Profiles provide a flexible way to select combinations of messaging system services.

Profiles can be given names up to 64 alphanumeric characters in length. The names can include accent characters, the underscore, and embedded spaces and cannot include leading or trailing spaces. Profiles can have passwords, but are not required to have them because not all operating systems support them. Currently, Windows NT and Windows 95 do not support passwords, but Windows version 3.1 does.

Profiles are organized hierarchically and divided into sections, with one section for each message service and one section for each service provider within a service. The related sections are linked, making it easier to navigate through the information. Each section contains a series of entries that MAPI or a client application uses for configuration.

The entries included in a profile vary from message service to message service. Some of the common entries include:

- Name of each message service or service provider
- Name of the DLL files that contain service providers and message services
- Name of each message service's entry point function
- A list of the service providers that make up each message service

Profiles can be created at installation time, when MAPI or a message service is loaded onto a computer workstation, or at any time after. MAPI provides three programs for profile administration: the Control Panel applet, the Profile Wizard, and the NEWPROF utility.

The Control Panel applet is a full-featured configuration application, allowing a user to create new profiles and maintain existing profiles either by deleting them or adding, modifying, or deleting entries.

The Profile Wizard is an application that creates new profiles with a minimum amount of work. Default values for settings are used wherever possible, saving users time and effort. Users enter values only when absolutely necessary.

The NEWPROF utility is a tool for creating new profiles with a template file similar to MAPISVC.INF called DEFAULT.PRF. The entries that are placed in DEFAULT.PRF are written into the new profile as properties. NEWPROF can be invoked from the command line, a message service installation program, or from within a client application.

About MAPI Forms

A MAPI form is a viewer for a message. Every message has a message class that dictates the particular form that is used as its viewer. MAPI defines a few message classes and has implemented the forms for viewing messages of these classes. Client application developers can create new message classes and custom forms for viewing messages created with the new classes.

Every custom form implements a set of standard menu commands, such as open, create, delete, and reply and a set of commands that are specific to the particular form. Some of the form commands are integrated with the user interface of the client application when the form is active; other form commands totally replace the client commands.

The MAPI form architecture involves three main components:

- Form library provider
- Form server
- Form viewer

The form library provider maintains a library of information about all of the forms available on the computer, enabling the client to select the form suitable for the message to be displayed.

The form server is responsible for displaying the form and providing the information for the display. The form server manages the user's interaction with the form, interpreting the menu selections and processing the messages. A form server is similar in implementation to an OLE compound document application.

The form viewer is a component within a client application that contains the display and presents it to the user.

The following illustration shows the relationship between the various components that make up the MAPI form architecture.

{ewc msdncd, EWGraphic, groupx827 6 /a "MAPI_53.WMF"}

Introduction to MAPI Programming

Before beginning serious development work, you need to consider the following information about using the MAPI Software Development Kit (SDK), the logon process, and how profiles and message services are created and configured. Client application developers need to consider how to choose between the four client APIs (MAPI, Simple MAPI, CMC, and the OLE Messaging Library) and how to create a client application that takes advantage of more than one client API.

About MAPI Installations

Information about a computer's MAPI installation can be found in the system registry on 32-bit client applications and in the Windows initialization file, WIN.INI, for 16-bit client applications. The system registry and WIN.INI file contain the same information about the MAPI libraries that are installed on the computer and options about use. All values in the registry entries are character strings.

Message service installation programs are responsible for creating the installation information in the [Mail] section of the WIN.INI file and in the following system registry key:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows Messaging Subsystem
```

Message services can add entries to either the WIN.INI file, the system registry, or both resources depending on the installation.

The following table summarizes how clients retrieve version information and determine if Simple MAPI, CMC, MAPI, or the OLE Messaging Library are available on their machine.

To check	**Registry/WIN.INI entry**
Availability of Simple MAPI	Look for `MAPI=1`
Availability of MAPI	Look for `MAPIX=1`
Availability of CMC	Look for `CMC=1`
Availability of OLE Messaging Library	Look for `OLEMessaging=1`
Available version of MAPI	Look for MAPIXVER string of the form: `x.x.x`"

Note Client applications that are running Windows NT 3.51 or earlier must check the WIN.INI file rather than the registry for the `MAPIX=1` entry to determine the availability of Simple MAPI.

Determining Which Client Interfaces are Installed

The information that is added to the [Mail] section of the WIN.INI file or the system registry specifies the types of MAPI client application programming interfaces (APIs) that are installed on a computer. These types include Simple MAPI, MAPI, the OLE Messaging Library, and CMC. If CMC is installed, another entry exists to identify the name of the CMC DLL file that should be loaded for your system.

The entries that identify client API installations are as follows:

```
[Mail]
MAPI=0 or 1
MAPIX= 0 or 1
OLEMessaging=0 or 1
CMC=0 or 1
```

Each entry that represents an installed MAPI client API is set to 1. If a particular client API is not installed, its entry either is missing from the WIN.INI file or the registry or is set to 0. The MAPI entry identifies a Simple MAPI installation, the MAPIX entry identifies a MAPI installation, and the OLE Messaging Library and CMC entries indicate the client API of the same name.

Computers with CMC installations can also set the CMCDLLNAME or CMCDLLNAME32 entries to specify the filename of the CMC DLL. CMCDLLNAME identifies the 16-bit version; CMCDLLNAME32 identifies the 32-bit version. If these entries are omitted and a CMC installation exists, MAPI searches for a file named CMC.DLL regardless of whether the installation is for a 16-bit or 32-bit platform. The format of the CMCDLLNAME entries is as follows:

```
CMCDLLNAME=full path to file
CMCDLLNAME32=full path to file
```

Simple MAPI, MAPI, and the OLE Messaging Library installations always reside in a standard DLL file. For MAPI, the DLL is called MAPIX.DLL for 16-bit platforms and MAPIX32.DLL for 32-bit platforms. For Simple MAPI, the DLL is called MAPI.DLL for 16-bit platforms and MAPI32.DLL for 32-bit platforms. Both 16-bit and 32-bit Simple MAPI client applications can run in an environment that supports both platforms (for example, the Windows 95 and Windows NT operating systems) if there is a Simple MAPI installation available for at least one of the platforms.

If the OLE Messaging Library is installed, the type library resides in MDISP.TLB for 16-bit platforms and MDISP32.TLB for 32-bit platforms. Client applications do not need the name of the server executable file; information contained in the OLE registry automatically binds the appropriate file.

For CMC installations, MAPI places the 16-bit CMC DLL into MAPI.DLL and the 32-bit CMC DLL into MAPI32.DLL.

Microsoft supplies an import library for MAPI. Client applications can load the MAPI DLL statically using the import library or load it dynamically using the Windows API functions **LoadLibrary** and **GetProcAddress**.

There is no import library for either the Simple MAPI DLLs or the CMC DLLs. These DLLs must always be loaded dynamically by calling the Windows functions **LoadLibrary** and **GetProcAddress**. Applications that use these Windows functions to load a CMC, Simple MAPI, or MAPI DLL should always check either the appropriate CMC, MAPI, or MAPIX entry to determine whether the API is installed before attempting to load the DLL.

The specific version of MAPI is indicated by another string value, which is a four-part number as follows:

```
MAPIXVER=w.x.y.z
```

About Installation Defaults

MAPI supplies a default profile provider implementation and a set of common dialog boxes. Typically service providers will use what MAPI provides. However, it is possible to replace either the profile provider implementation or the set of common dialog boxes. A provider can specify versions of these components in the [MAPI] section of the WIN.INI file or the system registry. Service providers that use the default components need not include this section.

The following entries can be included in the [MAPI] section:

```
Profile DLL=full path to file
Dialogs=full path to file
ProfileDirectory16=full path to file
```

The Profile DLL entry specifies the name of the DLL for the MAPI profile provider. The default setting is MAPIX.DLL for 16-bit platforms and MAPIX32.DLL for 32-bit platforms. The filename included in the path should be the profile provider's base DLL name, that is the DLL name without the suffix.

The Dialogs entry specifies the name of the DLL that contains the implementation for the common dialog boxes. The default setting is WMSUI.DLL for 16-bit platforms and WMSUI32.DLL for 32-bit platforms.

The ProfileDirectory16 entry specifies the path to the directory where the profile files are kept. This entry is supported for 16-bit Windows environments only. The default location is a subdirectory named MAPI under the Windows directory. The ProfileDirectory16 entry can be set either by the user or automatically – for example, by a network logon script. Specifying an explicit path for profile files allows you as a service provider to store these files on a server computer rather than locally, allowing them to be accessed from anywhere in the network.

About Time Zone Installation Information

The [MAPI 1.0 Time Zone] section stores information that is used by the MAPI 16-bit implementation of the time-zone API functions provided by 32-bit Windows. The entries that appear in this section are as follows:

```
[MAPI 1.0 Time Zone]
Bias=minutes
StandardName=string
StandardStart=time structure
StandardBias=minutes
DaylightName=string
DaylightStart=time structure
DaylightBias=minutes
```

The **TIME_ZONE_INFORMATION** structure is defined as follows in the Microsoft Win32 SDK documentation.

```
typedef struct _TIME_ZONE_INFORMATION
{
    LONG       Bias;
    WCHAR      StandardName[ 32 ];
    SYSTEMTIME StandardDate;
    LONG       StandardBias;
    WCHAR      DaylightName[ 32 ];
    SYSTEMTIME DaylightDate;
    LONG       DaylightBias;
} TIME_ZONE_INFORMATION;
```

Using the MAPI SDK

The MAPI Software Development Kit (SDK) includes tools to help in the development of MAPI-compliant client applications, service providers, and message services. The MAPI SDK is distributed with the Microsoft Developer's Network (MSDN) Level 2. There are two versions of the MAPI SDK: one for 16-bit developers and one for 32-bit developers.

You can install and use the MAPI SDK on any computer that meets the minimum qualifications for Microsoft Windows version 3.1 (enhanced mode only), Microsoft Windows NT, or Microsoft Windows 95.

The MAPI SDK contains the following components:

- MAPI setup program
- MAPI header files
- MAPI libraries and DLLs
- MAPI executable files
- Sample client applications and service provider DLLs
- MAPI spooler
- Help files

For configuration, MAPI provides two applications: a Control Panel applet for full configuration support and the Profile Wizard for basic profile administration.

The 32-bit version of the MAPI SDK is installed with the Win32 SDK. To install the MAPI SDK, select the check boxes for MAPI in the Win32 installation dialog box.

To install the 16-bit version of the MAPI SDK, insert the CD-ROM disk into the appropriate drive and run the setup program, SETUP.EXE, in the root directory. SETUP.EXE will automatically perform the rest of the installation, decompressing and copying the SDK software from the CD to the local disk. The setup program will also add a MAPI program group to the Program Manager window.

The SETUP.EXE program prompts you to choose a complete or a custom installation. A complete installation copies all of the MAPI files to the appropriate directories. A custom installation enables you to select specific files to be copied. SETUP.EXE displays information about each individual component and prompts for a decision on whether or not to install each one.

Using the Samples

The MAPI SDK contains a variety of samples and tools in C, C++, and Visual Basic. There are samples for different types of client applications, service providers, and extensions to the Microsoft Exchange client. There are also several tools for accessing some of the basic MAPI objects.

The following table describes the MAPI SDK samples and tools in order by type:

SDK component	Type	Description
FLATFILE.AB	Address book provider	Address book provider written in C.
SIMPLE.CLI	Client application	Simple MAPI client application.
CMC.CLI	Client application	CMC client application.
ROUTE.CLI	Client application	MAPI client application for routing.
EVENTS.EXT	Client extension	Extends the Microsoft Exchange client with message send and read hooks.
COMMAND.EXT	Client extension	Extends the Microsoft Exchange client with a folder command.
PROPSH.EXT	Client extension	Extends the Microsoft Exchange client with a new property sheet for document messages.
SIMPLE.FRM	Form	Simple form.
CHECKERS.FRM	Form	Form for playing checkers by e-mail.
DOCFILE.MS	Message store provider	Message store provider written in C.
MANAGER.SH	Messaging hook provider	Messaging hook provider written in C.
REMOTE.SRV	Server	Server-based messaging host.
ABVIEW.EXE	Tool	Tool for displaying and changing address book objects.
MDBVU.EXE	Tool	Tool for displaying and changing message store objects.
PROPVU.DLL	Tool	Tool for displaying and changing properties.
SEND.EXE	Tool	Tool for generating message traffic.
STATUSVU.DLL	Tool	Tool for displaying the status table.
TBLVU.DLL	Tool	Tool for displaying tables.
PEER.XP	Transport provider	Transport provider written in C.
REMOTE.XP	Transport provider	Remote transport for server-based host written in C++.

For developers interested in writing a profile provider, SAMPLE.PR is available upon request. SAMPLE.PR is a sample profile provider written in C.

Using Functions

The MAPI SDK contains two sets of functions: a core set that is needed to implement the MAPI basic functionality and a set that can be used to implement auxiliary features. Client applications and service providers can find alternate ways, if necessary, to implement the features supported by this latter set of functions. All of the messaging features required by clients and service providers can be implemented using the set of basic functions. Because the auxiliary set of functions are regarded as nonessential and used predominately to implement utility features, it is possible that MAPI will not support them in future releases.

As a developer of a MAPI-compliant client application or service provider, be aware that the use of auxiliary functions might result in code having to be re-implemented for future versions. You can ensure the portability of your client or service provider code by using only the core MAPI functions.

The following is a list of the auxiliary functions:

ChangeIdleRoutine	**CloseIMsgSession**
DeinitMapiUtil	**EnableIdleRoutine**
FBadColumnSet	**FBadEntryList**
FBadProp	**FBadPropTag**
FBadRestriction	**FBadRglpNameID**
FBadRglpszW	**FBadRow**
FBadRowSet	**FBadSortOrderSet**
FBinFromHex	**FEqualNames**
FNIDLE	**FPropCompareProp**
FPropContainsProp	**FPropExists**
FtAddFt	**FtgRegisterIdleRoutine**
FtMulDw	**FtMulDwDw**
FtNegFt	**FtSubFt**
GetInstance	**HexFromBin**
HrAllocAdviseSink	**HrComposeEID**
HrComposeMsgID	**HrDecomposeEID**
HrDecomposeMsgID	**HrEntryIDFromSz**
HrGetOneProp	**HrSetOneProp**
HrSzFromEntryID	**HrValidateIPMSubtree**
LPropCompareProp	**MAPIDeInitIdle**
MAPIInitIdle	**MapStorageSCode**
NOTIFCALLBACK	**OpenIMsgSession**
PpropFindProp	**PreprocessMessage**
PropCopyMore	**RemovePreprocessInfo**
ScBinFromHexBounded	**ScCopyNotifications**
ScCopyProps	**ScCountNotifications**
ScCountProps	**ScLocalPathFromUNC**
ScRelocNotifications	**ScRelocProps**
ScUNCFromLocalPath	**UlAddRef**
UlPropSize	**UlRelease**

Using UUIDGEN.EXE

UUIDGEN.EXE is a Microsoft utility that generates unique, byte-order independent identifiers. UUIDGEN.EXE has several uses in the MAPI environment. The following table indicates some of the ways that MAPI components can use this utility.

MAPI component	**Use of UUIDGEN.EXE**
Service provider	Creating new hardcoded **MAPIUID**s.
Form	Creating new class identifiers for forms.
Client application or service provider	Identifying hardcoded global profile section.
Client application or service provider	Creating new property sets for named properties.

Selecting a Client Interface

Client application developers can choose to use one or more of the MAPI client interfaces: Simple MAPI, CMC, MAPI, and the OLE Messaging Library. Consider the following factors before making a decision:

- Programming language
- Time constraints
- Resource constraints
- Type of client application

The language in which your client application will be written or is written is an important issue. If you are modifying an existing application, you must use a client interface that supports your application's language. If you are writing a new client application, the choice of language depends on your experience with the supported languages and any requirements for interoperability that might exist with other components. All of the client interfaces work with C and C++, whereas Visual Basic developers are limited to using the OLE Messaging Library and Simple MAPI.

The amount of time you need to create or modify your client application is also a consideration. A simple API such as Simple MAPI or CMC is more appropriate if you have limited time. Resource constraints might also be an issue. To successfully develop a client application, you should have experience with messaging and the appropriate programming language.

Messaging-enabled and messaging-aware client applications have different requirements than messaging-based applications. Messaging-enabled and messaging-aware applications have fewer and simpler messaging features to implement. Messaging-based applications have more complex messaging requirements because they have more direct contact with and control over the underlying messaging system services like address books, message stores, and transports. These applications often implement a wide variety of messaging features, such as rules processing, automatic forwarding, and supporting Rich Text Format. Shared group applications like schedulers and calendars, work flow and message management applications, electronic mail clients, and rules-based inbox managers are examples of messaging-based applications. Complex messaging-based applications like these require the MAPI client interface.

The OLE Messaging Library is a good choice for applications that require a moderate amount of messaging support. The OLE Messaging Library is an object oriented API used primarily by Visual Basic and Visual C/C++ client application developers. It provides programmable objects that publish properties and methods which can then be managed by Visual Basic and Visual Basic for Applications programs or other OLE Automation controllers. The OLE Messaging Library is based on the capabilities provided by OLE Automation. In terms of messaging functionality, it offers more than CMC and Simple MAPI offer, but less than MAPI offers.

Simple MAPI or CMC is the best choice for messaging-enabled and messaging-aware applications. These interfaces provide a fast and easy way to build basic applications from scratch or add messaging functionality to existing applications.

About the Client Interfaces

The following table summarizes the differences between the characteristics of the four client interfaces.

Issues	CMC	Simple MAPI	MAPI	OLE Messaging Library
Messaging support	Low	Low	High	Medium
Prerequisite knowledge	None	None	COM and OLE	OLE automation and Visual Basic programmable objects
Forms support	None	None	Full support	Some support
Language support	C and C++	C, C++, Visual Basic	C and C++	C, C++, Visual Basic
Platform support	Cross-platform	Windows-based	Windows-based	Windows-based

Choosing Between Simple MAPI and CMC

The features provided by Simple MAPI and CMC are nearly identical, but CMC is slightly more compact, combining multiple features in a single function. Both interfaces provide a straightforward function call interface that enables users to create, send, receive, reply to, forward, and edit messages.

The following table lists all the functions in the CMC and Simple MAPI APIs together with the features they provide.

Purpose	**Simple MAPI function**	**CMC function**
Establish a session	**MAPILogon**	**cmc_logon**
Terminate a session	**MAPILogoff**	**cmc_logoff**
Free memory	**MAPIFreeBuffer**	**cmc_free**
Send a message with or without a user interface	**MAPISendMail**	**cmc_send**
Send a message with a user interface	**MAPISendDocuments**	**cmc_send_documents**
Find messages that match a set of search criteria	**MAPIFindNext**	**cmc_list**
Save a message	**MAPISaveMail**	**cmc_act_on**
Delete a message	**MAPIDeleteMail**	**cmc_act_on**
Assign recipients to a message	**MAPIAddress**	**cmc_look_up**
Show recipient details	**MAPIDetails**	**cmc_look_up**
Resolve ambiguous names in recipient list	**MAPIResolveName**	**cmc_look_up**
Retrieve configuration data	**-**	**cmc_query_configuration**

The main difference between the two APIs is that CMC is designed to be independent of the operating system. CMC was developed in conjunction with the X.400 Application Programming Interface Association (XAPIA) standards organization and electronic mail vendors and users. Because CMC runs on Windows, MS-DOS, and UNIX systems, it is a good choice for client applications that need to use multiple messaging systems on multiple platforms. Because Simple MAPI runs on Windows-based platforms only, it is a good choice for single platform Windows-based client applications that require a feature CMC lacks or that need to be compatible with an existing Simple MAPI application.

Using Multiple Client Interfaces

Most client applications will use one client API – either the OLE Messaging Library, Simple MAPI, CMC or MAPI. However, some clients will want to take advantage of what more than one client API has to offer. These clients will want to use MAPI for some tasks and a simpler API for others. The MAPI architecture allows client applications to do this fairly easily by providing several API functions for converting between the environments.

One common conversion task involves identifiers. Identifiers are used throughout MAPI to uniquely represent a component, such as a message or a service provider. MAPI uses a binary structure called an entry identifier; the OLE Messaging Library uses a hexadecimal entry identifier, and Simple MAPI and CMC use a string called a message identifier. Message identifiers can be simple or compound.

MAPI only provides the means to convert between MAPI entry identifiers and all other identifiers. It does not provide the means for converting between Simple MAPI and CMC, Simple MAPI and the OLE Messaging Library, or CMC and the OLE Messaging Library.

The following API functions are used to translate identifiers used by the different client API sets:

FBinFromHex
HexFromBin
HrSzFromEntryID
HrEntryIDFromSz
HrComposeEID
HrDecomposeEID
HrComposeMsgID
HrDecomposeMsgID

The **FBinFromHex** and **HexFromBin** functions are used to translate the binary entry identifiers used in the MAPI programming interface and the hexadecimal entry identifiers used in the OLE Messaging Library.

The **HrEntryIDFromSz** function creates a MAPI entry identifier from a Simple MAPI string identifier and the **HrSzFromEntryID** function performs the opposite operation, producing a Simple MAPI string identifier from a MAPI entry identifier.

The **HrComposeMsgID** function also creates a Simple MAPI identifer. However, **HrComposeMsgID** creates a compound entry identifier, or identifier that can be used by Simple MAPI clients to open a non-default message store. Compound entry identifiers for messages are built from the record key of the message store and the message's entry identifier. A message store's record key is a unique binary value that can be used for comparison. Calling **HrComposeMsgID** enables Simple MAPI clients to create the identifiers necessary for working with multiple message stores.

The **HrComposeEID** function is similar to **HrComposeMsgID** because it is also used for creating compound entry identifiers. However, whereas **HrComposeMsgID** is primarily for Simple MAPI clients, **HrComposeEID** is for CMC clients. Callers of **HrComposeEID** pass a session pointer, a message store record key, and the entry identifier of an object. **HrComposeEID** produces a MAPI entry identifier based on both the record key and entry identifier, if possible, or only the entry identifier if the record key is not available.

HrDecomposeEID separates a compound entry identifier into its parts: a message store record key and an entry identifier that represents a MAPI object. Use **HrDecomposeEID** with care; it is an expensive call. Both **HrComposeEID** and **HrDecomposeEID** can handle binary message identifiers that require string versions.

Because **HrComposeEID**, **HrComposeMsgID**, **HrDecomposeEID**, and **HrDecomposeMsgID** all require a MAPI session pointer as input, client applications that are not started with the MAPI logon function, **MAPILogonEx**, must translate the current session handle into a MAPI session before calling any of these functions. **HrEntryIDFromSz**, a function that does not require a session pointer, can also be used to create a compound entry identifier. However, subsequent uses of this entry identifier can fail. If possible, clients should use the **HrComposeMsgID** or **HrComposeEID** function instead.

Converting session types is possible with one of two API functions: **ScMAPIXFromCMC** or **ScMAPIXFromSMAPI**. To convert CMC sessions into MAPI sessions, use **ScMAPIXFromCMC**. To convert Simple MAPI sessions, use **ScMAPIXFromSMAPI**. Both functions take the current session handle as input and return a pointer to a MAPI session object.

When using multiple client APIs, exercise caution in interpreting return values. Client APIs do not share the same set of return values, nor do they return the same type of values. For example, Simple MAPI and CMC functions return unique sets of unsigned long values whereas MAPI functions and methods return values that are of type **HRESULT**. In all cases, zero means a successful result.

The unsigned long values and the **HRESULT** values are based on a numeric code which, in many cases, is the same for all caller types. However, there are a few cases where the values are different. The following table lists the differences between the values returned by Simple MAPI and MAPI.

Simple MAPI return value	**MAPI return value**
MAPI_E_NOT_SUPPORTED	MAPI_E_NO_SUPPORT MAPI_E_INTERFACE_NOT_SUPPORTED MAPI_E_INVALID_PARAMETER MAPI_E_VERSION
MAPI_E_DISK_FULL	MAPI_E_NOT_ENOUGH_DISK
MAPI_E_NETWORK_FAILURE	MAPI_E_NETWORK_ERROR
MAPI_E_USER_ABORT	MAPI_E_USER_CANCEL
MAPI_E_ACCESS_DENIED	MAPI_E_NO_ACCESS
MAPI_E_AMBIGUOUS_RECIPIENT	MAPI_E_AMBIGUOUS_RECIP

MAPI User Interface

MAPI provides a set of common dialog boxes that offer a consistent way for users to interact with messaging systems. Simple MAPI, CMC, and OLE Messaging Library client applications are presented with these dialog boxes automatically. MAPI client applications and service providers can use the MAPI dialog boxes or implement their own. Unless you have very customized needs, you should take advantage of the MAPI user interface. By doing so, you will save time and give users tools that are consistent across applications and messaging systems.

The MAPI user interface includes many different dialog boxes, some for session start-up, some for addressing, some for message composition. The message composition dialog boxes are used to create interpersonal messages; to create messages of other types, you can implement special dialog boxes, or forms, that can be provided by the originator of the message type. For information about implementing and displaying these special forms, see Developing MAPI Form Servers.

MAPI Sessions

Before your client application can call an underlying messaging system, it must establish a session, or connection, with the MAPI subsystem. Sessions are initiated when a user logs on, a process that consists of accessing a valid profile, validating messaging system and message service credentials, and insuring that all of the profile's message services are properly configured. The client interface you use determines the logon call. MAPI clients call the **MAPILogonEx** function, Simple MAPI clients call the **MAPILogon** function, CMC clients call the **cmc_logon** function, and OLE Messaging Library clients call the Session object's **Logon** method.

MAPI supports the validation of credentials, such as passwords, as an optional feature. The use and implementation of passwords depends on the messaging system being used and the operating system. Some operating systems support security that is based on user accounts; others do not. Passwords can either be saved in the profile or entered by the user with every logon.

The following table lists the MAPI operating systems and whether or not each type implements user account security.

MAPI platform	User account security
Windows 95	Yes
Windows NT	Yes
Windows 3.x	No
Windows for Workgroups 3.x	No

Message service configuration is one of the most important parts of the logon process. The profile is the initial source for configuration information. If information for a particular message service is missing, the logon process attempts to prompt the user to supply it, an attempt which is not always successful for two reasons. Prompting the user requires the display of a dialog box. It is possible for clients to disallow the display of a user interface by passing a flag into the logon call. A second possible reason for failure is the cancellation of the dialog box by the user before the needed information can be added.

When a logon process fails once, the user is informed of the failure and given the opportunity to retry or correct the error condition. Once again, a user interface will be displayed, if possible, and the user will be asked to enter whatever data is missing. If this second attempt proves unsuccessful, MAPI disables all service providers in the message service for the duration of the session. In effect, the whole message service is disabled, meaning that none of the service providers in the message service can work. This is done because if one provider fails logon, the other providers usually also fail. The logon process can fail due to an invalid path for a necessary resource, an incompatible version of MAPI, an unavailable messaging server, or data corruption.

Clients can specify one of two types of sessions to be established in the logon call: an individual session or a shared session. Individual sessions are private connections; there is a one-to-one relationship between a client application and the session it is using. As a consequence, client applications sharing a session also share a profile. Shared sessions are established once but can be "used" by other client applications who need to use them. The profile and credentials are specified only with the initial logon.

MAPI also supports unified logon, in which a single initial logon to the messaging system results in access to multiple workgroup applications. When working with unified logon, users do not have to enter a name and password for each application they want to use.

Clients can log on multiple times as the same user or as multiple users. MAPI does not prevent this. Some service providers, however, might not be as flexible, returning the error value MAPI_E_SESSION_LIMIT on subsequent logon attempts. Service providers with underlying hardware limitations can be required to enforce a session limit.

The function calls for establishing a session have a collection of flags and parameters that control how the session is created. The client specifies an optional profile name, password, and a window handle that acts as the parent window for any dialog boxes that are displayed. The flags include MAPI_NEW_SESSION to request that a new, individual session be established rather than establishing a connection to a shared session, and two user interface flags, MAPI_LOGON_UI and MAPI_PASSWORD_UI. The user interface flags are set to request either a logon dialog box or a password dialog box.

The following illustration shows how these various parameters and flags establish a MAPI session.

{ewc msdncd, EWGraphic, groupx828 0 /a "MAPI_47.WMF"}

Message Services and Profiles

Some users require the services of several messaging systems, each with one or more service providers. Because it is cumbersome to have to install and configure each of these service providers individually and because a messaging server usually requires a group of related providers to expose all of its functionality, MAPI includes the concept of a message service. Message services help users install and configure their service providers.

To create a message service, a developer writes a message-service entry point program to handle the configuration of each provider in the service and a setup program to:

- Install each provider in the service.
- Create registry and initialization file entries.
- Create entries in the MAPI configuration file, MAPISVC.INF.

The MAPISVC.INF file contains information relating to the configuration of all message services and service providers installed on the computer workstation. It is organized in hierarchical sections, with each level linked to the next. At the top are three sections: one listing message service help files, one listing the most important, or default, message services, and one listing all of the services on the workstation. The next level contains sections for each message service and the last level contains sections for each service provider in a service. MAPI requires developers of service providers and message services to add certain entries to MAPISVC.INF; other entries can be added at the developer's discretion. Most of the information in MAPISVC.INF ends up in one or more profiles, a collection of configuration information for a user's preferred set of message services. Because a workstation can have multiple users and a single user can have multiple sets of preferences, many profiles can exist on a workstation. Each profile describes a different set of message services. Having multiple profiles enables a user to work, for example, at home with one set of message services and at the office with a different set.

Profiles are created at message service installation or logon time by a client application that provides configuration support. MAPI provides two such client applications: the Control Panel applet and the Profile Wizard. The Control Panel applet is a full service configuration application that allows users to create, delete, edit, and copy profiles as well as make modifications to the entries within a profile. The Profile Wizard is a simple application designed to make adding a message service to a profile as easy as possible. The Profile Wizard consists of a series of dialog boxes, called property pages, that prompt the user through the process of installing and configuring a service. The user is prompted only for values for the most critical settings; all other settings inherit default values. Once the profile has been created, users are not allowed to make changes.

Whereas the Control Panel applet is always invoked through the Control Panel, there are a variety of scenarios that can cause the Profile Wizard to be called. Client applications can call the Profile Wizard to create a default profile at logon time when one has not yet been created. Rather than re-implementing code to add a profile, the Control Panel applet or another client application can rely on the functionality already in the Profile Wizard. A message service, in its entry point function, can call the Profile Wizard when the service needs to be added to the default profile. Message services that use the Profile Wizard must write an extra entry point function and a standard Windows dialog procedure. The Profile Wizard calls the entry point function to retrieve the service's configuration dialog box while the dialog procedure handles the messages that are generated when this dialog box is in use.

Profiles are organized in a similar way to the MAPISVC.INF file. There are linked hierarchical sections with service providers owning sections in the lowest level, message services owning sections in the middle level, and MAPI owning sections in the highest level. Each section is identified with a unique identifier known as a **MAPIUID**. The MAPI sections contain information internal to MAPI, such as the identifiers of all of the message service profile sections and links to each of the other sections. Each message service section stores links to its provider sections and each provider section stores a link to its service section.

The following illustration shows the contents of two typical profiles. Sam has two profiles on his computer, one for home use and one for office use. The home profile contains three message services. Message Service X is a single provider service for address book management. Message Services Y and Z have three providers − an address book provider, a message store provider, and a transport provider. Sam's Work Profile contains two different message services; each of which has an address book provider, a message store provider, and a transport provider.

{ewc msdncd, EWGraphic, groupx828 1 /a "MAPI_56.WMF"}

Some profiles, depending on the platform, are password-protected. Password protection is only supported on platforms that do not provide user account security. It can be desirable to password protect profiles if it is important to ensure single user access.

Note Although password-protected profiles can only be changed by valid users, anyone with access to the system can delete them. This is because profiles are not considered critical components; they can be easily recreated.

The following illustration shows the pieces that make up a message service. The message service code for installation and configuration of each of its three service providers resides in a single DLL. This code reads information from the profile at logon time, adding missing information if necessary with or without the help of the user. Requests from a client application to view or change configuration settings for any of the providers are handled by this common code.

{ewc msdncd, EWGraphic, groupx828 2 /a "MAPI_55.WMF"}

Operating Environment Issues

The issues of operating system and operating environment are important when choosing a development environment and an environment for MAPI SDK installation. MAPI developers must be aware that these choices affect the operating system that their component will run on and the range of components that they will be able to target.

MAPI currently runs on the 32-bit environments provided with Windows NT and Windows 95, the native 16-bit environment of Windows 3.1 or later, and a 16-bit compatibility mode environment that allows 16-bit client applications to run on 32-bit platforms. Compatibility mode does not work for service providers.

The choice of development environment affects the environment that your component will be able to operate in. For service providers, the environment used for development is the only environment that their component will be able to operate in. That is, service providers developed in 16-bit environments can operate in 16-bit environments and service providers developed in 32-bit environments can operate in 32-bit environments.

Client applications are slightly more flexible. Clients written in 32-bit environments are like 32-bit service providers; they can only operate in 32-bit environments. Clients written in 16-bit environments, however, can operate in any environment.

Environment Issues Affecting Client Applications

MAPI-compliant client applications can run in multiple environments, depending on the environment used for development. A client written in a 16-bit development environment can operate either in a native 16-bit Windows environment or in any 32-bit environment under 16-bit compatibility mode. Compatibility mode allows a 16-bit application to simulate a 32-bit application. A client written in a 32-bit development environment can run on any 32-bit operating system, such as Windows NT or Windows 95.

The environment into which a client installs the MAPI SDK affects the scope of usable service providers. If the MAPI SDK is installed in a generic 32-bit environment or in the 16-bit compatibility mode environment, a client can use two types of service providers:

- Service providers written for 32-bit operating environments
- Single-source service providers

Single-source service providers are built with a single set of source files. This set of source files can be used to build binaries that will run in any environment.

If the MAPI SDK is installed in a 16-bit environment, a client can use only a 16-bit service provider.

Windows NT Service Client Applications

To enable client applications that are written as Windows NT services to operate with MAPI-compliant service providers, MAPI imposes several limitations and requirements. Simple MAPI, CMC, and MAPI clients have the following limitations:

- They cannot allow a user interface.
- They can only send messages through a tightly coupled message store and transport provider. In addition, MAPI clients can only send and receive messages using the Microsoft Exchange server or another server-based transport provider. Because of identity and security issues between client applications and the MAPI spooler, most transport providers are not supported in a Windows NT service.

All client applications, whether or not they are implemented as Windows NT services, must call **MAPIInitialize** to initialize the MAPI libraries. A call to **OleInitialize** is also necessary to use the OLE libraries. Both **MAPIInitialize** and **OleInitialize** make calls to **CoInitialize** to initialize the Component Object Model (COM) libraries. Clients that are Windows NT services must set a special flag, MAPI_NT_SERVICE, on the **MAPIInitialize** call to inform MAPI of their special implementation.

Windows NT service clients written with the MAPI client interface have a few additional requirements:

- They must set the MAPI_NO_MAIL flag in the call to **MAPILogonEx**. Other types of Windows NT service clients need not set a flag for logon because it is automatically set by MAPI.
- They must add the service's account to the group of accounts called Administrators on the machine that the service is using. This is necessary for profile access. If the account does not belong to this Administrators group of accounts, the **MAPIInitialize** function will fail when called.

Note The requirement that the account belong to the Administrators group will be removed in a future release of Windows NT.

To handle messages in an initialization thread, a MAPI client that is implemented as a Windows NT service

1. Calls **MsgWaitForMultipleObjects** when the main thread blocks.
2. Calls the **GetMessage**, **TranslateMessage**, and **DispatchMessage** sequence of Windows functions to handle the message when **MsgWaitForMultipleObjects** returns the sum of the value of the *nCount* parameter and the value of WAIT_OBJECT_0, indicating that a message is in the queue.

To produce application log entries through asserts and error traces

1. Use the debug version of MAPI
2. Add the following lines to the debug initialization file, MAPIDGB.INI:

```
[General]

DebugTrace=1
EventLog=1
```

Environment Issues Affecting Service Providers

Service providers can only operate in the environment for which they are created. That is, a 16-bit service provider works only in 16-bit environments and a 32-bit service provider works only in 32-bit environments. To create a service provider that runs on any environment, there are two choices: create separate versions for each environment or use an approach known as single sourcing.

Single sourcing allows developers to use a special set of API functions provided by MAPI to reduce the amount of 16 bit-specific work. MAPI has ported a few dozen Win32 API functions to 16 bit. With these functions, 16-bit code can be written as if it were being written with the standard Win32 library of API functions. The code is compiled for the 16-bit environment using some special built-in support. All of the sample services installed with the MAPI SDK are single source services written using these APIs.

The environment used to install the MAPI SDK determines the set of client applications that a service provider can support. Service providers can be written to support clients that run on any of the following operating systems and environments:

- Windows NT (32-bit)
- Windows 95 (32-bit)
- Win16

When the MAPI SDK is installed under Windows NT, a service provider can be created that can be used by 16-bit or 32-bit Windows NT clients. When the MAPI SDK is installed under Windows 95, a service provider can be created that can target 16-bit clients, Windows 95 clients, and in some cases, Windows NT clients. When the MAPI SDK is installed in a 16-bit environment, a service provider can be created that can target 16-bit clients only. Because MAPI does not support the 32-bit subset, Win32s, these service providers will only run in the 16-bit environment on Windows 3.1 or later.

Service providers should keep their future audience in mind when selecting an installation environment. If 16-bit clients are the only type of clients that will be using a service provider, a 16-bit environment for installation is appropriate. Likewise, if these clients are 32-bit clients, a 32-bit environment will work.

Some service providers, however, need to target clients that operate in all three operating environments: 32-bit Windows NT, 32-bit Windows 95, and 16-bit Windows. To target all environments, a service provider must operate in a 32-bit environment and must install both the 32-bit version of the MAPI SDK and the 16-bit version.

Service provider developers that are planning to target either Windows NT or Windows 95 should write their provider to be used by clients using both operating environments. Targeting both 32-bit environments is easy to do because the additional code required for the second environment is minimal.

Programming with CMC

This section contains conceptual information for programming with the Common Messaging Calls (CMC) application programming interface (API). You should understand this material before writing client applications that use the CMC APIs.

About CMC

The Common Messaging Calls (CMC) client interface is a set of ten functions that enables you to add simple messaging capabilities to your client applications quickly. For example, your client can send a message with a single CMC function call and receive a message with two CMC function calls.

Because CMC is built on top of the core MAPI subsystem, it shares the advantage of messaging system independence. The CMC API is especially valuable because it is also independent of the operating system and the underlying hardware used by the messaging system, providing a common messaging interface that can be used in virtually any environment. The CMC API is therefore a good choice for a messaging API when your client must run on multiple platforms and provide simple messaging functionality on each of these platforms.

Because CMC isolates clients from the complexities of MAPI, service providers, networking, and any other mechanisms which implement messaging, the remainder of this document will use the term "messaging system" to mean all of those things taken together. As far as a CMC client is concerned, the implementation layer underneath CMC is the messaging system that the client is using.

Although the Microsoft CMC implementation is built on top of a MAPI implementation, it is important to note that other implementations of CMC from other vendors might not be. In this document, "CMC" and "CMC implementation" mean "Microsoft's implementation of the CMC API using the MAPI API."

The CMC API was developed in conjunction with the X.400 API Association (XAPIA) standards organization and a group of e-mail vendors and users. It is supported on Microsoft Windows, MS-DOS, OS/2, Macintosh, and UNIX platforms. Because CMC is supported on several different platforms, a client application written to the CMC standard can be ported to other platforms. In contrast, MAPI is a Windows-only standard.

The CMC API supports three principal tasks: sending messages, retrieving messages, and looking up addressing information.

Note that these tasks do not have to be performed in isolation from each other. Your client application can establish a session once with a call to the **cmc_logon** function and use the resulting CMC session for the duration of the user's interaction with the application. During that time, your client can make many CMC calls to perform messaging tasks such as send messages, receive them, and make directory queries.

The CMC API works as a layer between a messaging-enabled application and the messaging system. The messaging system can support multiple messaging protocols, each using different messaging formats and protocols, for example, X.400, RFC 822, and Simple Mail Transport Protocol. The design of the CMC interface specifies that its functions be independent of the messaging protocols. However, the API does enable developers to use extensions to invoke protocol-specific functions. For more information, see Using CMC Data Extensions.

A directory, a submission queue, and a receiving mailbox are the three messaging system components in the CMC API model.

{ewc msdncd, EWGraphic, groupx829 0 /a "MAPI_08.WMF"}

Using the directory, the messaging-enabled application can look up information about users of connected messaging systems and can resolve users' names to actual addresses. Some messaging systems can also provide an interface enabling users to create recipient lists for messages or find out details about specific recipients.

The CMC implementation assigns a submission queue for each messaging-enabled application. By doing so, CMC provides each application with synchronous submission of messages to the underlying messaging system; once a call to send a message has returned, the calling application is guaranteed that the submission process has completed (although no guarantee is made about whether the message was successfully delivered). When the call has returned, the CMC implementation has all further responsibility for submitting the message to the underlying messaging system.

On the receiving side, a mailbox receives all messages for a user. The messaging system maintains mailboxes on behalf of messaging users. These mailboxes are accessible to users of messaging-enabled applications who have the proper permissions. With the CMC API, your client application can retrieve summaries of the contents of a mailbox, along with identifiers for the particular messages summarized. Your client can use these identifiers to select and retrieve individual messages.

The CMC API uses a fixed set of API functions, data structures, and data types. The *API functions* are the functions that your client calls to carry out messaging tasks. The *data structures* are the groupings of information that your client must provide to the CMC APIs, and are sometimes returned by the CMC APIs. The *data types* are basic elements that comprise the data structures. They each have a specific range of values and specific memory storage characteristics (for example, Boolean and floating point) and can have specific operations performed on them. For reference information, see Functions, Structures and Data Types, and Data Extensions.

Starting a CMC Session

CMC function calls occur within the context of a messaging session. Your client application establishes a session with a call to the **cmc_logon** function. The **cmc_logon** function checks the user's credentials for the messaging system, sets session attributes, and returns a session handle for use in later CMC calls. Session attributes include the character set and version number for the CMC API. Currently, the CMC API provides no support for sharing sessions among clients. Your client ends a session with a call to the **cmc_logoff** function.

Your client can convert the CMC session handles to MAPI session handles with the **ScMAPIXFromCMC** function. For example, your client can convert the session handle returned from the **cmc_logon** function to an LPMAPISESSION value and use it to access the methods in the **IMAPISession** interface. This enables your client to use CMC for the bulk of its messaging tasks without sacrificing access to MAPI functions. CMC session handles cannot be converted to Simple MAPI session handles. Clients that need both CMC and Simple MAPI functionality must maintain separate sessions for each.

Your client can also use the **cmc_query_configuration** function to determine logon identity and messaging options before calling the **cmc_logon** function.

Addressing Messages with CMC

You client application addresses messages with CMC by calling the **cmc_look_up** function to find addresses that it can then use with the **cmc_send** function. By calling **cmc_look_up**, your client can search for names in the address book and resolve names into the addresses used by the underlying messaging system.

Both **cmc_look_up** and **cmc_send** make heavy use of the **CMC_recipient** structure. The **cmc_look_up** function uses it on input to receive the name of the recipient your client is trying to look up. On output, **cmc_look_up** passes back an array of **CMC_recipient** structures that contain the results of the search. The **cmc_send** function uses a **CMC_recipient** structure to receive the address or addresses of the recipients of the message being sent.

To look up names and address information in the directory

1. Establish a session either through the **cmc_logon** function or interactively by sending the CMC_LOGON_UI_ALLOWED flag value with the **cmc_look_up** function.
2. Translate a user's display name (what the user sees) into a messaging address (what the underlying messaging system uses) by calling **cmc_look_up**. With this function, your client can also request that the standard CMC address dialog box be displayed for the user to view recipient-specific details or create addressing lists.
3. Release memory allocated by CMC by calling the **cmc_free** function.
4. End the session by calling the **cmc_logoff** function.

Sending Messages with CMC

Your client application sends messages with CMC by building a CMC_message structure and passing it to the cmc_send function. Your client fills in the members of the **CMC_message** structure depending on the parameters used to invoke **cmc_send**. For example, if your client passes the CMC_LOGON_UI_ALLOWED flag to **cmc_send**, then your client does not need to provide a list of recipients, a subject line, or message text because CMC will query the user for that information.

When your client calls the **cmc_send** function, it must provide a session handle, a pointer to a **CMC_message** structure, any necessary flags, an optional user interface identifier if the flags include the CMC_LOGON_UI_ALLOWED flag, and optional extensions. Once **cmc_send** returns, your client has no further responsibility for sending the message.

To send a message

1. Establish a session with the messaging system either through the cmc_logon function or interactively by sending the CMC_LOGON_UI_ALLOWED flag value with the cmc_send function.
2. Submit a message to the submission queue. Usually, a client application does so through the **cmc_send** function. If your client calls **cmc_send**, it must first create a **CMC_message** structure to pass to the **cmc_send** function. Your client can also call the more limited cmc_send_documents function to send a message; this function is primarily used to send messages or files from macro languages.
3. End the session by calling the cmc_logoff function.

Receiving Messages with CMC

It is slightly more complex for your client application to receive messages with CMC than to send messages.

To receive a message

1. Establish a session by calling the **cmc_logon** function.
2. Retrieve a summary of mailbox information by calling the **cmc_list** function.
3. Retrieve an individual message by calling the **cmc_read** function.
4. Optionally, enable a user to act on the message in the mailbox (for example, delete it) by calling the **cmc_act_on** function.
5. Release memory allocated by CMC by calling the **cmc_free** function.
6. End the session by calling the **cmc_logoff** function.

There are a number of ways your client can invoke the **cmc_list** function in order to control the sorts of messages listed by the API. For example, your client can pass in the string "CMC: NDR" in the *message_type* parameter to obtain a list of nondelivery reports, or can include the CMC_LIST_UNREAD_ONLY flag in the *list_flags* parameter to obtain a list of unread messages.

Handling Attachments with CMC

Your CMC client application receives attachments in the form of an array of **CMC_attachment** structures on a message. Each element of this array specifies a file containing an attachment. The one exception to this rule is for embedded messages, which are received as a CMC message in the attachment extension.

CMC clients have no explicit method for distinguishing between OLE 1 and OLE 2 attachments. Your client should try opening the attachment as an **IStorage** object to determine whether the attachment is an OLE 2 object. If this fails, your client should try opening it as an OLE 1 **IStream** object. If both of these fail, the attachment should be treated as an ordinary data file.

Getting Information About the CMC Implementation

Your client can get information about the CMC implementation by calling the **cmc_query_configuration** function. The client must pass in an item code that specifies the type of configuration information for the **cmc_query_configuration** function to return. Your client must also pass in a pointer to a buffer where the **cmc_query_configuration** function can pass back the requested information. It is your client's responsibility to ensure that the buffer is large enough to hold data of the requested type.

Releasing Memory with CMC

In some circumstances, your client application will have to release memory that has been allocated by CMC. CMC provides a single function, **cmc_free**, for releasing memory that it has allocated. This memory is typically the result of a CMC function call. For example, the **cmc_look_up** function allocates an array of **CMC_recipient** structures to hold the results of an address book search. Your client application is responsible for calling the **cmc_free** function on that array after it is no longer needed. To free memory allocated by CMC, your client only needs to pass a pointer to the memory to the **cmc_free** function.

Your client should not try to free the memory itself. The memory is not guaranteed to be in a single contiguous block, so one call to an underlying memory management function − such as the C run time library function **malloc** − is not guaranteed to free all of the memory that CMC has allocated. A memory leak will result if your client does this. The **cmc_free** function will make all the necessary memory management calls to properly free the memory CMC has allocated.

Ending a CMC Session

When your client no longer needs any message services, it should call the **cmc_logoff** function to end the CMC session that it established with the **cmc_logon** function. Depending on the nature of your client and whether it is possible to establish a CMC session without the user's assistance, it can be practical to close the session as soon as a group of messaging calls are finished. For other clients it is practical to establish one CMC session when the client starts and retain it until the client exits.

Using CMC Data Extensions

The data structures and functions defined by CMC can be expanded through the use of data extensions to add members to data structures and parameters to function calls.

Data extensions have two roles in CMC messaging. First, they are a mechanism to provide features not common across all messaging systems. Second, they enable extension of CMC in the future while also minimizing backward-compatibility issues. Use caution when using data extensions in your client application to take advantage of features specific to a messaging system. Reliance on specific features limits the portability of your client across messaging systems; also, such features might not be preserved properly when a message passes through multiple gateways in a mixed messaging network. If you do use data extensions in your client, your code should test for the presence of the extensions and gracefully handle the absence of the extensions. Doing so will make your client portable to other CMC implementations.

Data extensions are grouped into extension sets. Extension sets have unique identifiers (defined constants) assigned to them that represent the set as a whole and that represent the individual extensions in the set. These identifiers are assigned by the X.400 API Association, which guarantees that there are no conflicts between identifiers in officially recognized extension sets. There are also provisions for allowing a CMC implementation to define its own data extensions without obtaining identifiers from the X.400 API Association. If this is done, clients that use those extensions might not be portable to other CMC implementations.

The CMC common extension set contains those function and data extensions that are common to most messaging systems but are not in the CMC base specification. The common extension set is identified by the CMC_XS_COM constant. Individual extensions in the common extension set are identified by constants with names that start with CMC_X_COM. For a full list of common extensions, see Data Extensions.

A generic data structure, **CMC_extension**, is the base from which these extensions are created. A **CMC_extension** structure consists of an **item_code** member identifying the extension, an **item_data** member containing the length of the extension data or the data itself, an **item_reference** member pointing to the location where the extension value is stored or that is NULL if there is no related item storage, and an **extension_flags** member containing a bitmask of extension flags. The **item_code** member identifies a particular extension and determines the meanings of values in the other members.

Extensions that are additional parameters to a function call can be either input or output parameters. That is, an extension can be passed either as an input parameter from your client application to CMC or as an output parameter from CMC to your client. If an extension is an input parameter, your client allocates memory for the extension structure and any other structures associated with the extension. If an extension is an output parameter, CMC allocates the memory for the extension result, if necessary, and your client must free the allocated memory with a call to the **cmc_free** function.

To use a data extension

1. Create a **CMC_extension** structure and fill in the members according to the extension you want to use.
2. Pass a pointer to that structure in a call to a CMC function which can use it. The CMC function will use the information in the extension structure to enable additional functionality beyond the functionality defined for that CMC function in the X.400 API Association's CMC specification. The exact nature of the additional functionality is described in the documentation for the extension set you are using, which should be obtained from the extension set vendor.
3. Retrieve any return values from the members of the **CMC_extension** structure if the CMC function uses the structure to pass values back to your client.

CMC Programming Examples

This section includes example code that illustrates the following common messaging operations:

- Logging on and off and using **cmc_query_configuration** to get system information
- Sending the same message using the **cmc_send** and **cmc_send_documents** functions
- Listing, reading, and deleting the first unread message in the **Inbox**
- Looking up a specific recipient and getting recipient details from the address book
- Specifying use of the common extensions during a session logon

Error checking code has been deliberately left out of these examples for the sake of clarifying the use of the CMC functions themselves. Also, the way that an application recovers from errors is often dependant on the structure of the application, so it is unlikely that any error checking code in these examples would be of any use to other developers.

Logging On and Off: CMC Sample

The following code example demonstrates how your client application can log on and off and use **cmc_query_configuration** to get system information.

```
/* Local variables used */

CMC_return_code Status;
CMC_boolean     UI_available;   /* True if an interface is allowed */
CMC_session_id  Session;

/* Find out if user interface (UI) is available with this
   implementation before starting.*/

Status = cmc_query_configuration(
              0,                        /* No session handle          */
              CMC_CONFIG_UI_AVAIL,      /* See if UI is available.    */
              (void *)&UI_available,    /* Return value               */
              NULL);                    /* No extensions              */
    /* Error handling */

/* Log onto system using UI. */

Status = cmc_logon(
              NULL,                     /* Default service            */
              NULL,                     /* Prompt for user name       */
              NULL,                     /* Prompt for password        */
              NULL,                     /* Default character set      */
              0,                        /* Default UI ID              */
              CMC_VERSION,              /* Version 1 CMC calls        */
              CMC_LOGON_UI_ALLOWED |    /* Full logon UI              */
              CMC_ERROR_UI_ALLOWED,     /* Use UI to display errors. */
              &Session,                 /* Returned session ID        */
              NULL);                    /* No extensions              */
    /* Error handling */

/* Do various CMC calls. */

/* Log off from the implementation. */

Status = cmc_logoff(
              Session,              /* Session ID                     */
              0,                    /* No UI will be used.            */
              0,                    /* No flags                       */
              NULL);                /* No extensions                  */
    /* Error handling */
```

Sending a Message: CMC Sample

The following code example demonstrates how your client application can send a message, using first the **cmc_send** and then the **cmc_send_documents** function.

```
/* Local variables used */

CMC_attachment  Attach;
CMC_session_id  Session;
CMC_message     Message;
CMC_recipient   Recip[2];
CMC_return_code Status;
CMC_time        t_now;

/* Build recipient list with two recipients.  Add one "To" recipient. */

Recip[0].name       = "Bob Weaver";       /* Send to Bob Weaver.      */
Recip[0].name_type  = CMC_TYPE_INDIVIDUAL;/* Bob's a person.          */
Recip[0].address    = NULL;               /* Look_up Bob's address.   */
Recip[0].role       = CMC_ROLE_TO;        /* He's a "To" recipient.   */
Recip[0].recip_flags= 0;                  /* Not the last element     */
Recip[0].recip_extensions = NULL;         /* No recipient extensions  */

/* Add one "Cc" recipient. */

Recip[1].name       = "Mary Yu";           /* Send to Mary Yu.        */
Recip[1].name_type  = CMC_TYPE_INDIVIDUAL; /* Mary's a person.        */
Recip[1].address    = NULL;                /* Look_up Mary's address. */
Recip[1].role       = CMC_ROLE_CC;         /* She's a "Cc" recipient. */
Recip[1].recip_flags= CMC_RECIP_LAST_ELEMENT;/* Last recip't element  */
Recip[1].recip_extensions = NULL;          /* No recipient extensions */

/*  Attach a file. */

Attach.attach_title = "stock.wks";         /* Original filename       */
Attach.attach_type  = NULL;                /* No specific type        */
Attach.attach_filename  = "tmp22.tmp";     /* File to attach          */
Attach.attach_flags   = CMC_ATT_LAST_ELEMENT;  /* Last attachment     */
Attach.attach_extensions = NULL;           /* No attachment extension */

/*  Put it together in the message structure. */

Message.message_reference   = NULL;     /* Ignored on cmc_send calls. */
Message.message_type        = NULL;     /* Interpersonal message type */
Message.subject             = "Stock";    /* Message subject          */
Message.time_sent           = t_now;    /* Ignored on cmc_send calls. */
Message.text_note           = "Time to buy";   /* Message note        */
Message.recipients          = Recip;      /* Message recipients       */
Message.attachments         = &Attach;    /* Message attachments      */
Message.message_flags       = 0;          /* No flags                 */
Message.message_extensions  = NULL;       /* No message extensions    */

/*  Send the message. */

Status = cmc_send(
              Session,         /* Session ID - set with logon call   */
              &Message,        /* Message structure                  */
              0,               /* No flags                           */
              0,               /* No UI will be used.                */
              NULL);           /* No extensions                      */
    /* Error handling */


/* Now do the same thing with the send documents call and UI. */

Status = cmc_send_documents(
              "to:Bob Weaver,cc:Mary Yu",      /* Message recipients  */
              "Stock",                         /* Message subject     */
              "Time to buy",                   /* Message note        */
              CMC_LOGON_UI_ALLOWED |
              CMC_SEND_UI_REQUESTED |
              CMC_ERROR_UI_ALLOWED,/* Flags (allow various UIs)  */
              "stock.wks",         /* File to attach             */
              "tmp22.tmp",         /* Filename to carry on attachment */
              ",",                 /* Multivalue delimiter       */
              0);                  /* Default UI ID              */
    /* Error handling */
```

Receiving a Message: CMC Sample

The following code example demonstrates how your client application can list, retrieve, and delete the first unread message in the **Inbox**.

```
/* Local variables used */

CMC_message_summary     *pMsgSummary;
CMC_message             *pMessage;
CMC_uint32              iCount;
CMC_session_id          Session;
CMC_return_code         Status;

/* Read the first unread message and delete it. */

iCount  = 5; /* Maximum number of messages to get summary info about. */

Status = cmc_list(
              Session,               /* Session handle                */
              NULL,                  /* List ALL message types.       */
              CMC_LIST_UNREAD_ONLY,  /* Get only unread messages.     */
              NULL,                  /* Starting at the top           */
              &iCount,               /* Input/output message count    */
              0,                     /* No UI will be used.           */
              &pMsgSummary,          /* Return message summary list.  */
              NULL);                 /* No extensions                 */
    /* Error handling */

Status = cmc_read(
              Session,                            /* Session ID       */
              pMsgSummary->message_reference,     /* Message to read  */
              CMC_MSG_AND_ATT_HDRS_ONLY,    /* Don't get attach files.*/
              &pMessage,                          /* Returned message */
              0,                                  /* No UI            */
              NULL);                              /* No extensions    */
    /* Error handling */

Status = cmc_act_on(
              Session,                           /* Session ID        */
              pMsgSummary->message_reference,    /* Message to delete */
              CMC_ACT_ON_DELETE,                 /* Message to read   */
              0,                                 /* No flags          */
              0,                                 /* No UI             */
              NULL);                             /* No extensions     */
    /* Error handling */

/* Free the memory returned by the implementation. */

Status = cmc_free(pMsgSummary);
Status = cmc_free(pMessage);


/* Do the same thing without the list call, because the read call can
   get the first unread message. */
```

```
Status = cmc_read(
             Session,                  /* Session ID                  */
             NULL,                     /* Read the first message.     */
             CMC_READ_FIRST_UNREAD_MESSAGE | /* Get first unread msg. */
             CMC_MSG_AND_ATT_HDRS_ONLY,/* Don't get attach files.     */
             &pMessage,                /* Returned message            */
             0,                        /* No UI                       */
             NULL);                    /* No extensions               */
    /* Error handling */

Status = cmc_act_on(
              Session,                           /* Session ID        */
              pMessage->message_reference,       /* Message to delete */
              CMC_ACT_ON_DELETE,                 /* Message to read   */
              0,                                 /* No flags          */
              0,                                 /* No UI             */
              NULL);                             /* No extensions     */
    /* Error handling */

/* Free the memory returned by the implementation. */

Status = cmc_free(pMessage);
```

Getting Recipient Details: CMC Sample

The following code example demonstrates how your client application can look up a specific recipient and get details on that recipient from the address book.

```
/* Local variables used */

CMC_session_id  Session;
CMC_recipient   *pRecipient;
CMC_recipient   Recip;
CMC_return_code Status;
CMC_uint32      cCount;

/* Look up a name to pick correct recipient. */

Recip.name       = "Bob Weaver";            /* Send to Bob Weaver.     */
Recip.name_type  = CMC_TYPE_INDIVIDUAL;     /* Bob's a person.         */
Recip.address    = NULL;                    /* Look_up Bob's address.  */
Recip.role       = 0;                       /* Role not used           */
Recip.recip_flags      = 0;                 /* No flag values          */
Recip.recip_extensions = NULL;              /* No recipient extensions */

Status = cmc_look_up(
              Session,                      /* Session handle          */
              &Recip,                       /* Name to look up         */
              CMC_LOOKUP_RESOLVE_UI |       /* Resolve names using UI. */
              CMC_ERROR_UI_ALLOWED,         /* Display errors using UI.*/
              0,                            /* Default UI ID           */
              &cCount,                      /* Only want one back      */
              &pRecipient,                  /* Returned recipient ptr  */
              NULL);                        /* No extensions           */

/* Display details stored for this recipient. */

Status = cmc_look_up(
              Session,                     /* Session handle           */
              pRecipient,                  /* Name to get details on   */
              CMC_LOOKUP_DETAILS_UI |      /* Show details UI.         */
              CMC_ERROR_UI_ALLOWED,        /* Display errors using UI. */
              0,                           /* Default UI ID            */
              0,                           /* No limit on return count */
              NULL,                        /* No records returned      */
              NULL);                       /* No extensions            */

/* Free the memory returned by the implementation. */

cmc_free(pRecipient);
```

Using Common Extensions: CMC Sample

The following code example demonstrates how your client application can specify use of the common extensions during a session logon.

```
/* Local variables used */

CMC_return_code     Status;
CMC_session_id      Session;
CMC_extension       Extension;
CMC_extension       Extensions[10]; /* show how to handle multiple */
CMC_X_COM_support   Supported[2];
CMC_uint16          index;
CMC_boolean         UI_available;


/* Find out if the common extension set is supported, but
   COM_X_CONFIG_DATA support is not required. */

Supported[0].item_code =  CMC_XS_COM;
Supported[0].flags     =  0;

Supported[1].item_code =  CMC_X_COM_CONFIG_DATA;
Supported[1].flags     =  CMC_X_COM_SUP_EXCLUDE;

Extension.item_code       =   CMC_X_COM_SUPPORT_EXT;
Extension.item_data       =   2;
Extension.item_reference  =   Supported;
Extension.extension_flags =   CMC_EXT_LAST_ELEMENT;

Status = cmc_query_configuration(
              0,                           /* No session handle       */
              CMC_CONFIG_UI_AVAIL,         /* See if UI is available. */
              &UI_available,               /* Return value            */
              &Extension);                 /* Pass in extensions.     */
    /* Error handling */

if (Supported[0].flags & CMC_X_COM_NOT_SUPPORTED)
    return FALSE;    /* Needed common extensions are not available. */

/* Log onto system and get the data extensions for this session.    */

Supported[0].item_code =  CMC_XS_COM;
Supported[0].flags =      0;

Supported[1].item_code =  CMC_X_COM_CONFIG_DATA;
Supported[1].flags =      CMC_X_COM_SUP_EXCLUDE;

Extension.item_code =         CMC_X_COM_SUPPORT_EXT;
Extension.item_data =         2;
Extension.item_reference =    Supported;
Extension.extension_flags =   CMC_EXT_REQUIRED | CMC_EXT_LAST_ELEMENT;

Status = cmc_logon(
              NULL,                   /* Default service              */
              NULL,                  /* Prompt for user name.       */
              NULL,                  /* Prompt for password.        */
              NULL,                  /* Default character set       */
              0,                     /* Default UI ID               */
              CMC_VERSION,           /* Version 1 CMC calls         */
              CMC_LOGON_UI_ALLOWED | /* Full logon UI               */
              CMC_ERROR_UI_ALLOWED,  /* Use UI to display errors.   */
              &Session,              /* Returned session ID         */
              &Extension);           /* Logon extensions            */
    /* Error handling */
if (Supported[0].flags & CMC_X_COM_NOT_SUPPORTED)
    return FALSE;     /* Needed common extensions are not available, */
        /* the common data extensions will be used for this session. */

/* Example of how to free data returned from the CMC implementation in
   function output extensions.  */

for (index = 0; 1; index++) {
    if (Extensions[index].extension_flags & CMC_EXT_OUTPUT) {
        if (cmc_free(Extensions[index].item_reference) != CMC_SUCCESS){
            /* Handle unexpected error here. */
        }
    }
    if (Extensions[index].extension_flags & CMC_EXT_LAST_ELEMENT)
         break;
}

/* Do various CMC calls. */

/* Log off from the implementation. */

Status = cmc_logoff(
              Session,             /* Session ID                  */
              0,                   /* No UI will be used.         */
              0,                   /* No flags                    */
              NULL);               /* No extensions               */
    /* Error handling */
```

Programming with Simple MAPI

This section describes how and why you can use the Simple MAPI functions to add messaging functionality to your client application. The topics in this section cover the major areas of functionality that your Simple MAPI client needs to implement, such as:

- Initializing your client so that it can use the Simple MAPI functions.
- Creating messages.
- Managing attachments.

About Simple MAPI

Simple MAPI provides a set of functions that enables you to add a basic level of messaging functionality to Microsoft Windows-based applications.

Because Simple MAPI is only intended for the Microsoft Windows environment and offers limited functionality, it is recommended primarily for backward compatibility with older client applications. You are encouraged to use CMC or MAPI for development of new clients whenever possible.You should link Simple MAPI clients with either the MAPI.DLL or the MAPI32.DLL library, depending on whether you are writing for 16-bit or 32-bit Windows platforms.

Simple MAPI contains 12 high-level functions that enable a client application to send, address, receive, and reply to messages. Messages sent using Simple MAPI can even include file attachments and OLE objects. Windows-based clients that are not primarily used for messaging, such as spreadsheets, word processors, or applications that require only basic messaging can use Simple MAPI to provide messaging functionality quickly and easily.

Simple MAPI has several features that make it a good option for clients that do not have extensive messaging needs:

- Simple MAPI takes advantage of the MAPI subsystem to maintain independence from the underlying messaging system and network.
- Simple MAPI enables your client to define message types as well as the content of messages, and enables flexibility in the management of stored messages.
- Simple MAPI includes an optional common user interface (dialog boxes), so with little effort you can make your clients look consistent with each other and with other Windows-based applications. Through these dialog boxes, your users can address, compose, and send messages. Using the common dialog boxes is not mandatory, however, so if your client does not need a user interface, it can call the Simple MAPI functions without displaying any dialog boxes.
- Although designed to be called from C programs, your client can call the functions with little or no parameter modification from application-specific and standalone scripting packages such as Visual Basic, Actor, Smalltalk, and ObjectVision.

If you are writing a new client, you should implement it using MAPI or CMC. However, using MAPI involves learning about objects, interfaces, and the Component Object Model, and your client might only need the limited messaging functionality available with Simple MAPI. In this case, you should use CMC because it has the same functionality as Simple MAPI and it works with Microsoft Windows NT, Microsoft Windows for Workgroups 3.1, and Microsoft Windows 95. You should only use Simple MAPI for maintenance of older clients that were written using it.

Initializing a Simple MAPI Client

Before starting a Simple MAPI messaging session you need to initialize your client application. This involves checking the computer's WIN.INI file to determine if Simple MAPI is available, loading the correct dynamic link library (DLL) that contains the Simple MAPI functions, and setting a pointer to each function.

To initialize your client

1. Determine that Simple MAPI is available by checking the [MAIL] section in the computer's WIN.INI file for the MAPI entry. This entry will have a value of 1 if Simple MAPI is installed, or 0 if uninstalled.
2. Load the correct DLL for your operating system by calling the Windows **LoadLibrary** function as follows:

   ```
   hlibMAPI = LoadLibrary("MAPI.DLL");     // 16 bit clients
   hlibMAPI = LoadLibrary("MAPI32.DLL");   // 32 bit clients
   ```

   The two DLLs for Simple MAPI are MAPI.DLL for 16-bit clients and MAPI32.DLL for 32-bit clients.
3. Set the pointer variable to the actual address of the **MAPILogon** function as follows:

   ```
   lpfnMAPILogon = (LPFNMAPILOGON) GetProcAddress
       (hlibMAPI, "MAPILogon")))
   ```

Similarly defined constants exist for the other Simple MAPI functions, so your client simply needs to set the value of the function pointers using the Win32 function **GetProcAddress** as in the previous example.

Starting a Simple MAPI Session

Most MAPI calls are made in the context of a session, defined as an active connection between a client application and the MAPI subsystem. Each session uses a particular profile that specifies the set of available message services and the providers to manage those services. Before your client application can send or receive messages, it needs to log on and establish a session.

There are two types of sessions, temporary and persistent. A temporary session exists only for the lifetime of a single Simple MAPI call. A persistent session exists until the session is explicitly closed. Establishing a temporary session is referred to as implicit log*on*; establishing a persistent session is called explicit logon. Clients can use a persistent session for all calls that require the same set of providers and a temporary session for single calls that do not require the same context.

Note Some messaging systems can allow a limited number of sessions, so your client application must be able to handle a MAPI_E_TOO_MANY_SESSIONS return value. This value can be returned when a user is logged on to the system through an e-mail application and a mail-enabled application attempts to log on with a different identity.

Sharing Simple MAPI Sessions

Workgroup applications should all be able to work from the same profile without presenting logon or profile-selection dialog boxes when each application begins. A MAPI shared session is established once but can be used by other clients. Only one shared session can exist on a given computer at a time. The shared session can be used with any profile. If your client application must use a non-shareable session − that is, your client needs to be guaranteed that no other clients will use its session − then it must use the **MAPILogon** function in the MAPI API, which has a broader range of functionality than the Simple MAPI logon function.

By default, Simple MAPI attempts to use the shared session if it exists. If no shared session exists, Simple MAPI will create a new shared session when your client calls **MAPILogon**. If your client logs on with the MAPI_NEW_SESSION flag set, then Simple MAPI will create a new shared session for your client to use regardless of whether one already exists.

The session handles returned to clients using a shared session are not the same. Each client, regardless of whether it is using a unique session or a shared session, has its own unique handle. Session handles are not valid across tasks, even when those handles represent the same shared session.

Explicit Logon with Simple MAPI

Your client application logs on explicitly by calling the **MAPILogon** function. **MAPILogon** automatically opens the default address book for the Simple MAPI caller. When **MAPILogon** returns to your client, the session is ready to service all messaging requests. The default message store provider is loaded and opened by the first Simple MAPI call that needs the message store.

The session handle produced by **MAPILogon** represents the MAPI session that your client can use in further Simple MAPI calls. This session handle is passed to most Simple MAPI calls and can also be used with MAPI. The **ScMAPIXFromSMAPI** support method, prototyped in the MAPI.H file, converts a Simple MAPI session handle into a MAPI session object pointer. It is not possible to convert a Simple MAPI session handle into a CMC session handle. Clients needing both Simple MAPI and CMC functionality must maintain separate sessions for each.

Simple MAPI function calls produce return values that provide information about the success of a call. These return values are unsigned long values. Successful calls return the SUCCESS_SUCCESS value. Unsuccessful calls return error values starting with MAPI_E_, such as MAPI_E_INSUFFICIENT_MEMORY.

Implicit Logon with Simple MAPI

Your client application logs on implicitly by using a 0 session handle when calling Simple MAPI functions. Not all of the Simple MAPI functions allow the session handle to be 0; these calls cannot be made outside of an established session. For example, the **MAPIDeleteMail** function can only delete a message from a pre-existing session; a 0 session handle causes **MAPIDeleteMail** to return an error value.

When the session handle is 0, Simple MAPI function calls use the *flFlags* parameter to determine how to create the temporary session. If *flFlags* is set to MAPI_NEW_SESSION, the call tries to establish a new session, either programmatically if possible or with a logon dialog box. If MAPI_NEW_SESSION is not set, the call tries to use the shared session. If the call cannot establish a temporary session or use the shared session, it will return an error value.

When the call completes, the state of the messaging system is as it was before the call was made. That is, the implicit session opened for the call is closed by the time the call returns.

Specifying a Profile During Simple MAPI Logon

A call to the **MAPILogon** function produces different results depending on whether your client application specifies a profile name. Normally a profile name is required. If your client specifies NULL for the *lpszProfileName* parameter of the **MAPILogon** function, MAPI first checks for an existing shared session. If possible, the shared session is used. If there is no shared session or if it cannot be used, the call to **MAPILogon** will fail.

Your Simple MAPI client can use this behavior to test for the presence of an existing shared session before creating one of its own.

Specifying Passwords with Simple MAPI

Whether a password is required for your client application to log on depends on the operating system which is running on the computer. If the operating system integrates messaging system credential checking into the initial system logon, your client need not provide a password to the **MAPILogon** function. If not, a password is required from the user or from a cache location implemented by the client. Microsoft Windows NT and Microsoft Windows 95 do not require passwords for **MAPILogon** since the user's initial logon to the operating system suffices for both. Clients running with Microsoft Windows for Workgroups are required to provide a password for **MAPILogon**.

Requesting a Logon User Interface with Simple MAPI

Your client application can log on either by displaying a logon dialog box to provide user interaction or by providing necessary credentials (profile name and password if necessary) programmatically. If your client requires user interaction, you can use the common dialog box provided by Simple MAPI or create your own. However, it is recommended that you use the standard MAPI dialog boxes whenever possible to promote a consistent look and ease of use.

The presence or absence of the MAPI_LOGON_UI value in the *flFlags* parameter of the **MAPILogon** function controls the logon dialog box display.

To force the dialog box to be displayed

Set MAPI_LOGON_UI in the *flFlags* parameter and set the *lpszProfileName* and *lpszPassword* parameters to NULL as follows:

```
flFlags |= MAPI_LOGON_UI;
MAPILogon ((ULONG) hWnd, NULL, NULL, flFlags, 0, &lhSession);
```

If your client provides a value for *lpszProfileName* (and *lpszPassword* if necessary), **MAPILogon** attempts to establish a session without a user interface. Only if this attempt fails will **MAPILogon** display the logon dialog box for the user to enter new credentials.

Ending a Simple MAPI Session

To end a Simple MAPI session

1. Call the **MAPILogoff** function to close a session when it is no longer needed.
2. Call the **MAPIFreeBuffer** function to release any buffers that were allocated by Simple MAPI calls and returned to your client for its use. Calling **MAPILogoff** does not cause buffers allocated by Simple MAPI calls to be released. Your client has that responsibility.

Addressing Messages with Simple MAPI

Message headers form the "envelope" of a message. The message header contains all the information about the recipients of the message, as well as a subject for the message.

Addressing messages involves three major tasks:

- Generating recipient lists for different recipient classes. There should be separate lists of addresses for primary recipients, carbon copy (CC) recipients, and blind carbon copy (BCC) recipients.
- Checking names that the user enters against the address book to resolve them into actual addresses.
- Getting details about address book entries to help the user choose between ambiguous entries.

Creating a subject line is a simple matter of generating a text string containing the subject.

Generating Recipient Lists with Simple MAPI

Your client application calls the **MAPIAddress** function to generate a list of recipients for a message. If some of the recipients are known already, for example when forwarding a message, your client can pass in an initial array of **MapiRecipDesc** structures which the user can then modify.

MAPIAddress always displays a dialog box where the user can modify the list of recipients. If your client needs to generate a recipient list without presenting a dialog box, it should get recipients' names or addresses through some other means, by calling the **MAPIResolveName** function if necessary.

A successful call to the **MAPIAddress** function produces a buffer containing one or more **MapiRecipDesc** structures that your client must release with the **MAPIFreeBuffer** function when the buffer is no longer needed.

Checking Names with Simple MAPI

Checking names is the process of using the **MAPIResolveName** function to take a recipient's display name – usually their real name or something derived from their real name – or partial display name and retrieve the address book entry corresponding to it.

To check names

Call the **MAPIResolveName** function. Your client has the option of allowing the display of address dialog boxes while resolving the name by setting the MAPI_DIALOG flag in the *flFlags* parameter. If the MAPI_DIALOG flag is not set and the recipient's display name resolves to more than one address book entry, **MAPIResolveName** will return the MAPI_E_AMBIG_RECIP error value.

If the MAPI_DIALOG flag is set, Simple MAPI will display a dialog box for the user to resolve the ambiguity. If the MAPI_AB_NOMODIFY flag is also set, the properties of addresses displayed in this dialog box will not be modifiable, even if the addresses are stored in the user's personal address book.

Like the **MAPIAddress** function, a successful call to **MAPIResolveName** produces a buffer containing one or more **MapiRecipDesc** structures that must be released with a call to the **MAPIFreeBuffer** function.

Getting Details About a User with Simple MAPI

The **MAPIDetails** function presents a dialog box to the user that displays the properties of an address book entry. This dialog box cannot be suppressed. Like the **MAPIResolveName** function, **MAPIDetails** supports the MAPI_AB_NOMODIFY flag to prevent modification of the properties displayed in the dialog box. **MAPIDetails** is most often used with this flag set when the user needs more information about an address in order to resolve an ambiguity. This function is most often used without the MAPI_AB_NOMODIFY flag set when the user needs to create or edit entries in the personal address book.

Displaying and Editing Addresses with Simple MAPI

The **MapiRecipDesc** structure, which describes a recipient, contains both the recipient's address and the recipient's "friendly name." The address, contained in the **lpszAddress** member, contains the address that the messaging system uses to deliver the message. This can take a number of forms depending on what type of address it is. All addresses are of the form [*AddressType*][*E-mail Address*]. For example, an internet address could be SMTP:jdoe@microsoft.com, while a fax gateway address could be FAX:206-555-1212.

The friendly name, stored in the **lpszName** member, typically contains the recipient's actual name or some variation of it. Whenever possible, the messaging system places a value in this member when a message is retrieved from a message store. However, not all addresses can be assigned a friendly name. For example, a fax gateway address might not correspond to a specific person, so the address book might not contain a name associated with that address. Similarly, the nature of internet addresses is such that the messaging system might not be able to determine a friendly name for messages that come into the system through an internet gateway; the address book cannot contain entries for every valid internet address, and the format of internet addresses does not mandate that a friendly name be attached to them.

Your client application should display friendly names to the user when these names are available. When they are not, it is acceptable to display the actual address instead. When forwarding or replying to a message, your client should never allow the user to edit the friendly name or address fields of a **MapiRecipDesc** structure that the messaging system has returned with a message. Both friendly names and addresses should be treated as indivisible entities.

When displaying a message to the user, your client should make a distinction between different types of recipients that might be present in the message's **lpOriginator** and **lpRecips** members. The **lpOriginator** member contains the address and friendly name of the individual who sent the message. Your client should display this recipient (note that the originator of a message is still called a recipient because both originators and actual recipients are described by **MapiRecipDesc** structures) so that the user is aware of this recipient's role.

Recipients stored in the **lpRecips** member can have one of four different roles. The **ulRecipClass** member of the **MapiRecipDesc** structure stores this information.

Role	Description
MAPI_TO	The recipient is a primary recipient of the original message.
MAPI_CC	The recipient received a carbon copy of the original message.
MAPI_BCC	The recipient received a blind carbon copy of the original message.
MAPI_ORIG	The recipient sent the message.

In practice, MAPI_BCC and MAPI_ORIG are less common than the others. MAPI_BCC should only occur if the user reading the message is the one who received a blind carbon copy. The semantics of MAPI_BCC are such that no one besides the originator and a MAPI_BCC recipient should ever know that the recipient received the message. MAPI_ORIG should only occur if the user sent a message to himself or herself.

Sending Messages with Simple MAPI

To compose a message, your client application either creates a new message, forwards an existing message, or replies to an existing message. The new message can contain new material. Replying to existing messages involves creating a new message and copying the contents of the original message to it. All new messages can specify a return receipt as well.

To send a message

1. Create a **MapiMessage** structure to contain the message.
2. Create one or more **MapiRecipDesc** structures describing the recipients of the message and place them in the **lpRecips** member of the **MapiMessage** structure.
3. Create a text string containing the subject, if any, and place it in the **lpszSubject** member of the **MapiMessage** structure.
4. Create a text string containing the message text, if any, and place it in the **lpszNoteText** member of the **MapiMessage** structure.
5. Create an array of **MapiFileDesc** structures, if necessary, to contain any attachments and place it in the **lpFiles** member of the **MapiMessage** structure.
6. Submit the message by calling the **MAPISendMail** function.

Sending messages can involve more or less effort on the part of your client application, depending on the way you invoke various Simple MAPI functions to perform the above steps. For more information, see Controlled Sending of a Message: Simple MAPI Sample.

To create a new message

1. Allocate a **MapiMessage** structure.
2. Fill in the **MapiMessage** structure members with values appropriate for the message the user wants to send.
3. Submit the **MapiMessage** structure to the messaging system by calling the **MAPISendMail** function.

To forward an existing message

1. Retrieve the message by calling the **MAPIReadMail** function.
2. Modify the message as appropriate for forwarding:
   - Put one or more new recipients' addresses in the message's **lpRecips** member.
   - Modify the subject line to indicate that the message has been forwarded.
   - Allow the user to edit the message text.
3. Submit it to the messaging system by calling the **MAPISendMail** function.

To reply to a message

1. Retrieve the message by calling the **MAPIReadMail** function.
2. Modify the message as appropriate for replying:
   - Put the original sender's address into the **lpRecips** member.
   - Change the subject line to indicate that the message is a reply to an earlier message.
   - Allow the user to edit the message text.
3. Submit the message to the **MAPISendMail** function.

When your client submits a message to **MAPISendMail**, it has the option of requesting a return receipt. Return receipts are automatically generated messages that senders receive to inform them that messages they sent, for which a return receipt was requested, were delivered successfully to the recipient. To request a return receipt, your client should include the MAPI_RECEIPT_REQUESTED flag in the **flFlags** member of the **MapiMessage** structure:

```
// MapiMessage mymessage; declared and initialized elsewhere.
mymessage.flFlags |= MAPI_RECEIPT_REQUESTED;
```

Using the Subject with Simple MAPI

When your client application displays a message , the subject is simply a null-terminated string stored in the **lpszSubject** member of a **MapiMessage** structure.

When replying to or forwarding a message, your client should modify the subject line. The subject of a reply to a message should begin with the string "RE:", or an appropriate localized equivalent. Similarly, forwarded messages should have subject lines that begin with "FW:" or an appropriate localized equivalent. If the subject line already begins with "RE:" or "FW:", your client should not add it again.

Handling Attachments with Simple MAPI

Simple MAPI handles attachments by means of an array of MapiFileDesc structures pointed to by the **lpFiles** member of a MapiMessage structure. The **nFileCount** member of the **MapiMessage** structure indicates the length of the array, with a value of 0 indicating there are no attachments. All Simple MAPI attachments are returned by means of temporary files.

Simple MAPI recognizes three types of attachments: data files, editable OLE objects, and static OLE objects. Your client can use the values of the **flFlags** and **lpszFileName** members of a **MapiFileDesc** structure to determine the type of attachment it describes. OLE 2 attachments are identified by the extension ".STG" in the **lpszFileName** member. OLE 1 attachments are identified by the MAPI_OLE flag in the **flFlags** member. OLE 1 compliant clients which receive OLE 2 attachments can convert them by handling the OLE **IStorage** interface themselves or by using an OLE server to convert them to OLE 1. Embedded messages are data files that have the extension ".MSG" in the **lpszFileName** member.

Each attachment has a **nPosition** member that stores its position within the message. The position is an index into the message's **lpszNoteText** member (which can be treated as an array of characters). The attachment will appear at the character at its position in the message; that is, the attachment will be rendered instead of the character **lpszNoteText[nPosition]**. The special value **-1** (**0xFFFFFFFF**) indicates that the attachment is not rendered in this way; in this case, the client application receiving the message is responsible for providing the user with a way of accessing the attachment. Note that the position is a character position within the text note, not a byte offset into the text note. This is an important distinction when double byte character sets are in use.

Note The attachment should be rendered *instead* of the character stored in **lpszNoteText[nPosition]**. The actual value of **lpszNoteText[nPosition]** should be completely immaterial to the way the message is displayed to the user.

To add attachments to a message

1. Allocate an array of **MapiFileDesc** structures, one for each attachment.
2. Set the members of each array element to values appropriate for the data files or OLE objects that are being attached.
3. Set the message's **nFileCount** member to the number of attachments.
4. Set the message's **lpFiles** member to the address of the first element of the array of **MapiFileDesc** structures.

To handle attachments in a message your client has received

1. Scan the attachments in the array of **MapiFileDesc** structures and note their character positions.
2. When displaying the message text for the user, place graphic representations of data file attachments or OLE objects at the appropriate positions.
3. Provide a mechanism for the user to interact with the attachments. You might choose to implement a point-and-click interface, or allow users to select actions such as saving and opening from a menu.

Message Options with Simple MAPI

Simple MAPI supports a few options for messages and for sending messages. Some of these options are accessed through members of the **MapiMessage** structure, and some through parameters of the **MAPISendMail** function.

About MapiMessage

The **lpszMessageType** member of the MapiMessage structure indicates the type of the message. The most common type is the interpersonal message, or IPM. A NULL value for **lpszMessageType** or a pointer to an empty string indicates an interpersonal message. Client applications can use this member to define their own message types. Be aware that not all messaging systems support message types other than IPM. Those messaging systems will ignore **lpszMessageType**.

The **lpszMessageType** member is also used to indicate when a message is a nondelivery report (NDR). The messaging system formats the nondelivery for a message type as "REPORT." plus the message type plus ".NDR". For example, if your client uses a message type of "mytype", the nondelivery report for such a message will have "REPORT.mytype.NDR" as its **lpszMessageType** string. For clients that deal with multiple types of messages, your client can compare the last four characters of the **lpszMessageType** string against ".NDR" to determine whether the message is a nondelivery report. Naturally, your client should not choose a string that ends in ".NDR" as its internal message type.

Note While nondelivery reports generated by MAPI follow this convention, nondelivery reports generated by transport providers for external mail delivery systems might not.

The **flFlags** member can be used to request a receipt and to detect the read or unread and sent or unsent status of a message. When sending a message, your client can set the MAPI_RECEIPT_REQUESTED flag to request a receipt. When reading a message, your client can test for the MAPI_UNREAD flag to determine whether the message has not yet been read. Similiarly, your client can test for the MAPI_SENT flag to determine whether the message has been sent. When saving a previously unsaved message, your client should set the MAPI_UNREAD and MAPI_SENT flags as appropriate for the message.

About MAPISendMail

The **MAPISendMail** function supports options through its *flFlags* parameter. Your client application can set the following flags when sending a message.

Flag	Description
MAPI_LOGON_UI	Allows a logon interface if required. This flag should be set if your client is using an implicit session.
MAPI_NEW_SESSION	Prevents Simple MAPI from using an existing shared session if one is present.
MAPI_DIALOG	Displays a dialog box which allows the user to create a message. This flag should be set if your client does not have a **MapiMessage** structure already constructed, or if the user is composing a message interactively.

Simple MAPI Programming Examples

The code examples in this section are provided to illustrate in detail the concepts discussed earlier. Additional code examples can be found in the SMPCLI application included with the MAPI Software Development Kit (SDK). SMPCLI is not a typical Simple MAPI application because it exists solely to illustrate every Simple MAPI function. It is good for illustrating the use of all the Simple MAPI APIs, but it is not good for illustrating typical use of those APIs in real situations.

The following three examples show how a client application can send and receive messages with varying degrees of control. The first example sends a message very simply, leaving creation of the message's contents almost entirely up to Simple MAPI through user interaction. The second example takes more control over creation of the message and demonstrates how to use the **MAPIResolveName** function to check that addresses are valid before sending. The third example shows how to receive and display a message. Comments precede sections that benefit from explanation.

Sending a Message Simply: Simple MAPI Sample

This example shows the simplest way your client application can send a message. An essentially blank message is created and passed to the **MAPISendMail** function with parameters that cause Simple MAPI to use dialog boxes to create the content of the message. First, the client defines the variables it needs. Note that your client will not hard-code the attachment path name or filename in the **MapiFileDesc** structure.

```
// Example 1:
// Send a mail message containing a file and prompt for
// recipients, subject, and note text.

ULONG err;
MapiFileDesc attachment = {0,          // ulReserved, must be 0
                           0,          // no flags; this is a data file
                           (ULONG)-1, // position not specified
                           "c:\\tmp\\tmp.wk3",  // pathname
                           "budget17.wk3",      // original filename
                           NULL};               // MapiFileTagExt unused
// Create a blank message. Most members are set to NULL or 0 because
// MAPISendMail will let the user set them.
MapiMessage note = {0,            // reserved, must be 0
                    NULL,         // no subject
                    NULL,         // no note text
                    NULL,         // NULL = interpersonal message
                    NULL,         // no date; MAPISendMail ignores it
                    NULL,         // no conversation ID
                    0L,           // no flags, MAPISendMail ignores it
                    NULL,         // no originator, this is ignored too
                    0,            // zero recipients
                    NULL,         // NULL recipient array
                    1,            // one attachment
                    &attachment}; // the attachment structure
```

Next, the client calls the **MAPISendMail** function and stores the return status so it can detect whether the call succeeded. Your client should use a more sophisticated error reporting mechanism than the C library function **printf**.

```
err = MAPISendMail (0L,          // use implicit session.
                    0L,          // ulUIParam; 0 is always valid
                    &note,       // the message being sent
                    MAPI_DIALOG, // allow the user to edit the message
                    0L);         // reserved; must be 0
if (err != SUCCESS_SUCCESS )
    printf("Unable to send the message\n");
```

Controlled Sending of a Message: Simple MAPI Sample

This example shows how your client application can take more control over a message it sends by specifying more of the contents of the message, validating addresses before sending, and by denying a sending interface to the user. Again, the client starts by defining its variables.

```
// Example 2:
// Send a mail message containing a spreadsheet and a short note
// to Sally Jones and copy the Marketing group. Don't prompt the user.

ULONG err;
MapiRecipDesc recips[2],     // this message needs two recipients.
             *tempRecip[2];  // for use by MAPIResolveName

// create the same file attachment as in the previous example.
MapiFileDesc attachment = {0,          // ulReserved, must be 0
                           0,          // no flags; this is a data file
                           (ULONG)-1, // position not specified
                           "c:\\tmp\\tmp.wk3",  // pathname
                           "budget17.wk3",      // original filename
                           NULL};               // MapiFileTagExt unused
```

The client then uses the **MAPIResolveName** function to generate **MapiRecipDesc** structures for the recipients of the message. It can create them directly, as in the previous example, but then no error checking is possible. Since this client is creating and sending the message without any interaction from the user, it is important to make sure the addresses are valid before sending the message.

```
// get Sally Jones as the MAPI_TO recipient:
err = MAPIResolveName(0L,            // implicit session
                      0L,            // no UI handle
                      "Sally Jones", // friendly name
                      0L,            // no flags, no UI allowed
                      0L,            // reserved; must be 0
                      &tempRecip[0]);// where to put the result
if(err == SUCCESS_SUCCESS)
    { // memberwise copy the appropriate fields in the returned
      // recipient descriptor.
        recips[0].ulReserved   = tempRecip[0]->ulReserved;
        recips[0].ulRecipClass = MAPI_TO;
        recips[0].lpszName     = tempRecip[0]->lpszName;
        recips[0].lpszAddress  = tempRecip[0]->lpszAddress;
        recips[0].ulEIDSize    = tempRecip[0]->ulEIDSize;
        recips[0].lpEntryID    = tempRecip[0]->lpEntryID;
    }
else
    printf("Error: Sally Jones didn't resolve to a single address\r\n");

// get the Marketing alias as the MAPI_CC recipient:
err = MAPIResolveName(0L,            // implicit session
                      0L,            // no UI handle
                      "Marketing",   // friendly name
                      0L,            // no flags, no UI allowed
                      0L,            // reserved; must be 0
                      &tempRecip[1]);// where to put the result
if(err == SUCCESS_SUCCESS)
```

```
    { // memberwise copy the appropriate fields in the returned
      // recipient descriptor.
        recips[1].ulReserved   = tempRecip[1]->ulReserved;
        recips[1].ulRecipClass = MAPI_CC;
        recips[1].lpszName     = tempRecip[1]->lpszName;
        recips[1].lpszAddress  = tempRecip[1]->lpszAddress;
        recips[1].ulEIDSize    = tempRecip[1]->ulEIDSize;
        recips[1].lpEntryID    = tempRecip[1]->lpEntryID;
    }
else
    printf("Error: Marketing didn't resolve to a single address\r\n");
```

Now the client creates the message. Again, your client should not hard-code the actual values of the **MapiMessage** structure's members.

```
MapiMessage note = {0, "Budget Proposal",
                   "Here is my budget proposal.\r\n",
                   NULL, NULL, NULL, 0, NULL,
                   2, recips, 1, &attachment};
```

Again, the client sends the message and records the return value. This time no user interface is displayed. After the **MAPISendMail** call, the **MapiRecipDesc** structures allocated by **MAPIResolveName** must be released.

```
err = MAPISendMail (0L,    // use implicit session.
                    0L,    // ulUIParam; 0 is always valid
                    &note, // the message being sent
                    0L,    // do not allow the user to edit the message
                    0L);   // reserved; must be 0
if (err != SUCCESS_SUCCESS )
    printf("Unable to send the message\n");
MAPIFreeBuffer(tempRecips[0]);  // release the recipient descriptors
MAPIFreeBuffer(tempRecips[1]);
```

Receiving a Message: Simple MAPI Sample

The following example shows how your client application can receive a message with Simple MAPI. For simplicity, this example assumes that a character interface is being used.

First, the client application defines needed variables and logs on to get a session handle. Unlike when sending a message, the Simple MAPI functions for reading messages can't log on implicitly and require an explicit session handle.

```
ULONG ReadNextUnreadMsg()
{
ULONG err;
LHANDLE lhSession;      // Need a session for MAPIFindNext.
CHAR rgchMsgID[513];    // Message IDs should be >= 512 CHARs + a null.
MapiMessage *lpMessage; // Used to get a message back from MAPIReadMail.
int i,                  // Ubiquitous loop counter.
    totalLength;        // Number of characters printed on a line.
err = MAPILogon(0L,               // ulUIParam; 0 always valid.
               "c:\\pst\\myprofil.pro",// Shouldn't hardcode this.
               NULL,              // No password needed.
               0L,                // Use shared session.
               0L,                // Reserved; must be 0.
               &lhSession);       // Session handle.
if (err != SUCCESS_SUCCESS)       // Make sure MAPILogon succeeded.
{
    printf("Error: could not log on\r\n");
    return(err);
}
```

Next, the client searches for the first unread message in the default folder in the store (probably the user's Inbox). Since there might not be any unread messages in the folder, the client first tests the return value from the **MAPIFindNext** function against MAPI_E_NO_MESSAGES before checking against SUCCESS_SUCCESS. If the call is successful, the client will have a valid message identifier to use to retrieve the first unread message.

```
// find the first unread message
err = MAPIFindNext(lhSession, // explicit session required
                   0L,        // always valid ulUIParam
                   NULL,      // NULL specifies interpersonal messages
                   NULL,      // seed message ID; NULL=get first message
                   MAPI_LONG_MSGID | // needed for 512 byte rgchMsgID.
                   MAPI_UNREAD_ONLY, // only get unread messages.
                   0L,        // reserved; must be 0
                   rgchMsgID);// buffer to get back a message ID.

if (err == MAPI_E_NO_MESSAGES) // make sure a message was found
{
    printf("No unread messages.\r\n");
    return(err);
}
if (err != SUCCESS_SUCCESS)    // make sure MAPIFindNext didn't fail
{
    printf("Error while searching for messages\r\n");
    return(err);
}
```

The client application can now be sure it is safe to retrieve the message. However, it is still a good idea to check the return value from the MAPIReadMail function. If the call fails, the memory pointed to by the client's **lpMessage** pointer will not be accessible by the client. The client should not try to display a message at that location. Note that this example sets the MAPI_SUPPRESS_ATTACH flag so the returned message will not have any attachments in it and Simple MAPI will not create any temporary files for them.

```
// retrieve the message
err = MAPIReadMail(lhSession,   // Explicit session required.
                   0L,          // Always valid ulUIParam.
                   rgchMsgID,   // The message found by MAPIFindNext.
                   MAPI_SUPPRESS_ATTACH, // TO DO: handle attachments.
                   0L,          // Reserved; must be 0.
                   &lpMessage); // Location of the returned message.
if(err != SUCCESS_SUCCESS)      // Make sure MAPIReadMail succeeded.
{
    printf("Error retrieving message %s\r\n",rgchMsgID);
    return(err);
}
```

Now, the client can display the message. As expected, it begins by displaying the addressing information attached to the message before displaying the subject line and message text. When displaying the addressing information, it is best if your client can display friendly names. However, since friendly names are not always available, your client must verify that each recipient structure's **lpszName** member points to a valid string, and that the string is not a null string.

```
// Display the sender's name or address; use the friendly name
// if it is present.
if((lpMessage->lpOriginator->lpszName != NULL) &&
   lpMessage->lpOriginator->lpszName[0] != '\0')
    printf("From: %s\r\n",lpMessage->lpOriginator->lpszName);
else
    printf("From: %s\r\n",lpMessage->lpOriginator->lpszAddress);
```

Displaying the recipients' addresses is complicated by the need to avoid breaking a recipient's name or address across two lines. This code can be further improved by differentiating between recipients based on their **ulRecipClass** members so that they can be properly displayed as To, CC, or BCC recipients. As this code shows, handling recipient data can be the most complex part of reading a message.

```
// Display the recipients' names or addresses.  To Do: enhance
// this code to separate the recipients into lists of MAPI_TO,
// MAPI_CC, and MAPI_BCC recipients for separate display.
if(lpMessage->nRecipCount == 0)
    printf("Warning: no recipients present for this message\r\n");
else
    for(i = 0; i < lpMessage->nRecipCount; i++) // For each recipient...
    {
        // This code uses lstrlen to calculate the length of strings and
        // to validate that the strings have some content, since the
        // length is needed anyway. This avoids the more verbose checks
        // as were done for lpMessage->lpOriginator->lpszName earlier.

        // lpszT references a name or address; simplifies later code.
        // length is the length of the name or address.
        LPSTR lpszT = lpMessage->lpRecips[i]->lpszName;
        int length = lstrlen(lpszT);

        if(i == 0) // First recipient; need to do some initialization.
        {
            printf("Recipients:");
            totalLength = 11;      // since strlen("Recipients:") = 11.
        }

        // Decide whether to use the friendly name or the address.
        if(length == 0)
            length = lstrlen(lpszT=lpMessage->lpRecips[i]->lpszAddress);

        // Verify that the line has room for this name or address. If
        // not, print a CR LF pair to go to the next line.
        if(totalLength + length + 1 > LINE_WIDTH)
        {
            printf("\r\n");
            totalLength = 0;
        }

        printf(" %s",lpszT);      // Finally, print the name or address.
        totalLength += length + 1;// Maintain the line length.

        // If there are more addresses, separate them with semicolons.
        if(i < (lpMessage->nRecipCount - 1))
        {
            printf(";");
            totalLength++;
        }
    }
```

Now, the client displays the subject line and message text if they are present. Note that the message text can be printed with a simple call to the C library function **printf**. Since the message was read with the MAPI_SUPPRESS_ATTACH flag set, there will be no attachments in it.

```
// Display the subject and message body. Not printing anything for the
// subject is fine, but something should always be printed for the
// message body since it is the last thing that this function displays.
if(lpMessage->lpszSubject    != NULL &&   // Standard validity check
   lpMessage->lpszSubject[0] != '\0')
    printf("Subject: %s\r\n",lpMessage->lpszSubject);
if(lpMessage->lpszNoteText    != NULL &&  // Standard validity check
   lpMessage->lpszNoteText[0] != '\0')
    printf("Message Text:\r\n%s",lpMessage->lpszNoteText;
else
    printf("No message text.\r\n");
```

Finally, the client releases the memory that the **MAPIReadMail** function allocated for the message, closes the session, and returns a successful return value.

```
MAPIFreeBuffer(lpMessage);
MAPILogoff(lhSession,    // The session.
           0L,           // 0 always valid for ulUIParam.
           0L,           // No logoff flags.
           0L);          // Reserved; must be 0.
return SUCCESS_SUCCESS;  // Inform the caller of our success.
```

```
} // End of ReadNextUnreadMsg.
```

Objects and Interfaces

A MAPI object is a C++ object class or C data structure inherited from one or more MAPI interfaces, or collections of related functions. These collections of related functions are known to C++ developers as pure virtual functions. For a pure virtual function, MAPI supplies only the function prototype, not an implementation. It is expected that a client application, a service provider, or MAPI will provide this implementation by creating an object class inherited from the interface and conforming to the function descriptions documented in the *MAPI Programmer's Reference*. A MAPI interface can only be instantiated through an inherited class.

There are many different MAPI objects, each object inheriting from an interface that is ultimately inherited from **IUnknown**. **IUnknown** is the OLE Component Object Model (COM) base interface. It provides MAPI objects with a standard mechanism for communication and control. The Component Object Model dictates how object implementors handle issues such as memory management, parameter management, and multithreading. By conforming to this model, an object implementor adheres to a contract as specified by the interfaces included in the object.

Many MAPI interfaces are inherited directly from **IUnknown** while others are inherited indirectly through one of two other base interfaces: **IMAPIProp** for property management and **IMAPIContainer** for folder and address book access. Base interfaces are never implemented as separate, standalone objects; they are always implemented as part of other objects, objects that implement derived interfaces.

MAPI defines many types of objects, each implemented by one or more MAPI components. Objects implemented by clients are used by MAPI, by service providers, and by custom form components. Objects implemented by service providers are typically used by MAPI and by clients. Objects implemented by form library providers and form servers are used by other form components and by clients.

Component Object Model and MAPI

The Win32 SDK includes a comprehensive discussion of the rules for implementing objects that conform to the Component Object Model (COM). There are rules in the section called "Programming Rules for COM - A Quick Reference" that address how to:

- Design interfaces and objects.
- Implement **IUnknown**.
- Manage memory.
- Handle reference counting.
- Implement apartment threaded objects.

Although all MAPI objects are considered COM-based because they implement interfaces that inherit from **IUnknown**, MAPI deviates in some situations from the standard COM rules. This deviation allows developers more flexibility in their implementations. For example, a MAPI interface, like any COM interface, describes a contract between implementor and caller. Once the interface is created and published, its definition cannot and does not change. MAPI does not deviate from this description, but it relaxes the description somewhat. Implementors can choose to not implement particular methods, returning one of the following error values to the caller:

MAPI_E_NO_SUPPORT
MAPI_E_TOO_COMPLEX
MAPI_E_BAD_CHARWIDTH
MAPI_E_TYPE_NO_SUPPORT

The other deviations from the standard COM rules are described in the following table.

COM programming rule	MAPI variation
All string parameters in interface methods should be Unicode.	MAPI interfaces are defined to permit either Unicode or ANSI string parameters. Any method that has a string parameter also has a *ulFlags* parameter; the width of the string parameters is indicated by the value of the MAPI_UNICODE flag in *ulFlags*. Most MAPI interfaces to not support Unicode and return MAPI_E_BAD_CHARWIDTH when the MAPI_UNICODE flag is set.
All interface methods should have a return type of HRESULT.	MAPI has one method that returns a non-HRESULT value: **IMAPIAdviseSink::OnNotify**.
Callers and implementors should allocate and free memory for interface parameters using the standard COM task allocators.	All MAPI methods use the linked allocators **MAPIAllocateBuffer**, **MAPIAllocateMore**, and **MAPIFreeBuffer** to manage memory for interface parameters. All MAPI implementations of interfaces defined by OLE, such as **IStream**, use the standard COM task allocators.

All out pointer parameters must explicitly be set to NULL when a method fails.	MAPI interfaces require that out pointer parameters either be set to NULL or remain unchanged when a method fails. All MAPI implementations of interfaces defined by OLE explicitly set out parameters to NULL on failure.
Implement aggregatable objects whenever possible.	MAPI interfaces are not aggregatable.

Objects and the MAPI Architecture

All of the objects defined by MAPI fall into one or more layers in the MAPI architecture. The client interface layer is made up of all the objects that a client application, form viewer, or form server can implement. The service provider interface layer contains the objects that a service provider of any type can implement. This layer includes objects implemented by address book, message store, transport, form library, and messaging hook providers. The layer that represents the MAPI subsystem is positioned between the client and service-provider interface layers. The MAPI layer contains all of the objects that MAPI implements for clients or service providers to use.

The following illustration shows where each of the MAPI objects fits into the MAPI architecture. The objects are represented with the names of their derived interfaces. For example, an advise sink object is shown as **IMAPIAdviseSink**, the interface deriving from **IUnknown** that every advise sink object implements. The interfaces that bridge layers are either used or implemented by multiple components. Although the MAPI layer appears to separate the client and provider layers, implying that all communication must flow through MAPI, this is not the case. Clients can and do communicate directly to service provider objects.

{ewc msdncd, EWGraphic, groupx830 0 /a "MAPI_38.WMF"}

Object Inheritance Hierarchy

All interfaces implemented by MAPI objects ultimately inherit from **IUnknown**, the OLE interface that enables objects to communicate. Most interfaces directly inherit from **IUnknown**, but some inherit from one of two other base interfaces: **IMAPIProp** and **IMAPIContainer**. The following illustration shows the complete inheritance hierarchy in MAPI.

{ewc msdncd, EWGraphic, groupx830 1 /a "MAPI_06.WMF"}

Object Containment Hierarchy

The containment relationship between objects specifies the dependencies that some objects have on other objects for access. For a client application, access to particular objects enables access to others. In some cases, the containment relationship between objects implemented by a service provider follows a logical hierarchy. In other cases, it is arbitrary.

A client must obtain access to a MAPI session object before using many other objects, for example, service providers and the MAPI address book.

Message store containment is based on the hierarchical relationship between objects in the message store: the message store object itself, folders, messages, and attachments. Logically, attachments are contained in messages, messages in folders, and folders in the message store. The containment relationship matches this logical hierarchy. To gain access to a message, for example, a client must first access the folder in which the message is contained. Profiles and status objects are examples of a more arbitrary containment relationship. Both of these objects are available through the session.

With some objects, containers provide the only access path. Attachments and recipients are examples of objects that are totally dependent on their containers. The only access route to an attachment or a recipient is through the message to which they belong. Other objects have alternate access paths. These objects are assigned binary identifiers, known as entry identifiers, by the service providers that create them. Entry identifiers can be used to access their objects directly, enabling clients to bypass the containment tree.

The following illustration shows the MAPI containment hierarchy. The session is at the top of the tree because it is through the session that a client gains access to all other objects. The next level includes the message store table, a table object that lists properties for all of the message store providers in the current session, and the address book to supply access to all of the address book providers. The message store table and address book are used to access the objects implemented by particular service providers, shown next, in containment order.

{ewc msdncd, EWGraphic, groupx830 2 /a "MAPI_41.WMF"}

Objects Implemented by MAPI

MAPI implements several objects for use by client applications and service providers. The session object allows clients to use session services, to access tables, and to communicate with service providers. The address book object provides clients with integrated access to all of the different address book providers.

Multiple table and status objects are supplied by MAPI for clients to use for viewing and monitoring session and service provider information. For example, MAPI provides a profile table with information about all of the profiles that are installed on the computer and a message service table with information about all of the message services in the current profile. MAPI provides three different status objects: one that represents the overall subsystem, one for the MAPI spooler, and one for the integrated address book.

MAPI implements four different objects for managing the configuration of message services, service providers, and profiles. Both clients and service providers use provider administration and profile section objects; these objects enable them to configure service providers and access profile properties. Clients only use message service and profile administration objects, the objects that support the administration of message services and profiles.

MAPI provides two objects for service providers: a support object and a Tnef object. All service providers use one or more support objects; there are four different support object implementations. MAPI supplies an implementation to support configuration as well as specific implementations to support address book, message store, and transport providers. The Tnef object is used by transport providers that support the Transport Neutral Encapsulation Format (TNEF).

Two utility objects, table data and property data, are typically used by service providers. Table data objects help in the implementation of table objects; property data objects help to set and view property access and help in the implementation of **IMAPIProp : IUnknown**, the base property interface.

The following table summarizes the purpose for each object implemented by MAPI.

MAPI object	**Description**
Address book	Provides access to the integrated view of recipient information belonging to all of the address book providers in the active profile.
Message service administration	Provides access to message service information for configuration.
Profile administration	Provides access to profile information for configuration.
Profile section	A part of a profile used to describe a particular message service or service provider.
Property data	Maintains access to properties and helps implement **IMAPIProp**.
Provider administration	Provides access to service provider information for configuration.
Session	Represents a connection to underlying messaging systems and provides clients with access to MAPI resources.
Status	Provides access to the state of the MAPI subsystem, the address book, or the MAPI spooler.
Support	Provides service providers with help in

	handling client requests.
Table	Provides access to a summary view of object data in row and column format, similar to a database table.
Table data	Maintains access to underlying table data and implements table objects.
Tnef	Supports the use of the Transport Neutral Encapsulation Format (TNEF).

The following illustration shows the relationship between the objects that MAPI implements, the interfaces from which they inherit, and the components that use them.

{ewc msdncd, EWGraphic, groupx830 3 /a "MAPI_68.WMF"}

About Client Objects

Standard messaging client applications implement only one object: an advise sink. Advise sinks inherit from the **IMAPIAdviseSink : IUnknown** interface and are used by MAPI and service providers for event notification. Some clients also implement progress objects to support the display of progress dialog boxes.

More complex clients that support custom forms implement another advise sink object and a few other objects, such as the message site object that inherits from the **IMAPIMessageSite : IUnknown** interface and the view context object that inherits from the **IMAPIViewContext : IUnknown** interface. The additional advise sink object inherits from the **IMAPIViewAdviseSink : IUnknown** interface.

The following table summarizes the MAPI objects implemented by standard messaging clients and by clients that support the viewing of custom forms.

Client object	**Description**
Advise sink	Provides a callback function for events that occur in the message store, address book, or the session.
Message site	Handles the manipulation of form objects.
Progress	Displays a dialog box to show the progress of an operation.
View advise sink	Provides callback functions for events that occur in a form.
View context	Supports commands for printing and saving forms and for navigating between forms.

The following illustration shows the relationship between these different client objects, the interfaces from which they inherit, and the MAPI components that use them.

{ewc msdncd, EWGraphic, groupx830 4 /a "MAPI_65.WMF"}

Clients use many more objects than they implement. All clients use a session object to gain access to a wide variety of service provider objects and objects implemented by MAPI. Clients interact with service providers either indirectly, through the session, the address book, or the status objects supplied by MAPI, or directly through a variety of objects implemented by particular service providers. To make direct contact with address book providers, clients use address book containers, messaging users, and distribution lists. To access a message store provider directly, clients use the message store object, folders, messages, and attachments. When service providers support a status object, clients can use the status object to monitor the service provider's state.

Clients that support service provider and message service configuration use three objects implemented by MAPI: the message service administration object, profile administration object, and provider administration object. Clients that display custom forms use several form objects implemented by either a form library provider or a form server.

About Service Provider Objects

Service providers implement many objects, some that are used primarily by MAPI and some that are used by client applications. A few objects are implemented by all types of service providers; the rest are specific to a single provider type. The following table describes all of the service provider objects.

Service provider object	**Description**
Address book container	Contains recipient information for one address book provider in the active profile; address book providers can have one or more address book containers.
Attachment	Contains additional data, such as a file or OLE object, to be associated with a message.
Control	Enables or disables a button and initiates processing when the button is pressed.
Distribution list	Describes a grouping of individual message recipients.
Folder	Contains messages and other message containers.
Logon	Handles service provider event notification and client requests.
Messaging user	Describes an individual recipient of a message.
Message	Contains information that can be sent to one or more recipients with a messaging system.
Message store	Acts as a database of messages, organized hierarchically.
Provider	Handles service provider startup and shutdown.
Spooler hook	Performs special processing on inbound and outbound messages.
Status	Provides access to the service provider's state.
Table	Provides access to a summary view of object data in row and column format, similar to a database table.

All service providers implement a provider object and a logon object. Provider objects are strictly for bookkeeping; they are used by MAPI to control the start up and shutdown processes. Logon objects service some client requests indirectly. For example, the message store provider's logon object handles notification registration and requests to open message store objects.

Provider and logon objects implement a different interface depending on the type of service provider supplying the implementation. A message store provider implements the **IMSProvider** and **IMSLogon** interfaces in its provider and logon objects, an address book provider implements the **IABProvider** and **IABLogon** interfaces, and a transport provider implements the **IXPProvider** and **IXPLogon** interfaces.

Message hook providers implement spooler hook objects, or objects that filter inbound and outbound messages.

Service providers typically use only a few objects. Most frequently, they use a support object provided by MAPI that helps implement client requests. The support object is customized for the type of provider using it. For all service providers, the support object includes methods for handling event notification, displaying configuration properties, opening objects, and error handling. The rest of the methods are specific to its use; there are customized versions for address book, message store, and transport providers and for configuration support. For example, the address book support object displays details and custom recipient dialog boxes. The message store support object supports copy and move operations for folders and messages. The transport provider support object includes methods for facilitating interaction with the MAPI spooler.

Some service providers use table data and property data objects, utility objects implemented by MAPI. Table data objects enable service providers to manage the underlying data of a table. Property data objects enable service providers to set object and property access.

Transport providers that support the Transmission-Neutral Encapsulation Format (TNEF) for transferring properties use a Tnef object implemented by MAPI that supports the **ITnef** interface. For more information, see Developing a TNEF-Enabled Transport Provider.

About Address Book Provider Objects

In addition to the standard provider and logon objects, address book providers implement address book containers, distribution lists, messaging users, tables, status objects, and controls. The following illustration shows these objects, their corresponding interfaces, and the MAPI components that use them.

{ewc msdncd, EWGraphic, groupx830 5 /a "MAPI_64.WMF"}

About Message Store Provider Objects

Message store providers implement provider and logon objects as do all service providers. They also implement a message store object, folders, messages, attachments, and tables. As an option, some message store providers implement status objects.

The following illustration shows each message store object with its corresponding interface and the MAPI component that uses it.

{ewc msdncd, EWGraphic, groupx830 6 /a "MAPI_63.WMF"}

About Transport Provider Objects

In addition to the standard provider and logon objects implemented by all service providers, transport providers are required to implement a status object. For the other service provider types, implementing a status object is optional. However, MAPI requires it for transport providers. Transport providers that support the downloading of message headers from a remote server also implement a folder and a table.

The following illustration shows each of the objects that transport providers can implement with their corresponding interfaces and indicates whether MAPI or a client is the object's user.

{ewc msdncd, EWGraphic, groupx830 7 /a "MAPI_66.WMF"}

About Custom Form Objects

Objects for custom forms are implemented by three different components:

- Form server
- Form library provider
- Form viewer

A form server is similar in functionality to an OLE compound document object application. It is an executable component that implements the form; it controls its display and the operations that a user can perform. MAPI launches a form server upon request when a user wishes to view a message with a message class that is displayed using a form supported by the form server. Form servers implement three objects: a form factory object that resembles the standard OLE class factory, a form advise sink for handling form-specific events, and the form itself.

A form library provider supplies clients with access to a form's property set, to its container, and to the object that links messages of a specific class with the server that can activate the form for that class. Form library providers implement three objects: a form information object, a form container, and a form manager for binding a message to the appropriate form server based on the message's class.

A form viewer is a component that is included in clients that support the display of custom forms from within their folder viewers. Form viewers are not independent MAPI components as are form library providers and form servers. They invoke form servers and provide context for them. Form viewers implement three objects: a message site, a view context, and an advise sink specific for handling view-specific events.

The following table describes all of the custom form objects.

Form object	**Description**
Form	Controls the display and operation of a custom form for viewing messages of a specific class.
Form advise sink	Handles notifications from the form viewer.
Form factory	Creates an instance of a form and allows its server to remain in memory.
Form container	Contains form information.
Form information	Contains messages and other message containers.
Form manager	Provides access to an integrated view of custom form information relating to all of the installed forms. Also matches message classes with corresponding form class identifiers.
Message Site	Handles the manipulation of form objects from inside the client and provides access to a form manager object.
View Context	Supports form commands for activating next and previous messages and for saving or printing.
View Advise Sink	Handles notifications from the form server.

The following illustration shows the relationship between custom form components, the objects and interfaces that they implement, and the components that are users of the objects. Notice that, unlike most other MAPI objects, the form object implements two interfaces that are not related by direct inheritance. When an object exposes multiple independent interfaces, a user of the object that has a pointer to one of the interfaces can retrieve a pointer to any of the other interfaces. This ability to navigate between an object's interface implementations is a feature of the **QueryInterface** method, one of the methods in the base interface **IUnknown**.

{ewc msdncd, EWGraphic, groupx830 8 /a "MAPI_67.WMF"}

Using MAPI Objects

Clients and service providers use MAPI objects by calling the methods in their interface implementations. This is the only way that MAPI objects can be used; methods that are implemented by an object outside of a MAPI interface are not publicly accessible. Because all of an object's interfaces are related through inheritance, an object's user can call methods in either the base interface or one of the inherited interfaces as if they belong to the same interface.

When an object's user wants to make a call to a method and that object implements several interfaces related through inheritance, the user need not know to which interface the method belongs. The user can call any of the methods on any of the interfaces with a single pointer to the object. For example, the following illustration shows how a client application uses a folder object. Folder objects implement the **IMAPIFolder : IMAPIContainer** interface wihich inherits from **IUnknown** indirectly through **IMAPIProp** and **IMAPIContainer**. A client can call one of the **IMAPIProp** methods, such as **GetProps**, and one of the **IMAPIFolder** methods, such as **CreateMessage**, in the same way with the same object pointer. A client is not aware of or affected by the fact that these calls belong to different interfaces.

{ewc msdncd, EWGraphic, groupx830 9 /a "MAPI_40.WMF"}

These calls translate into code differently depending on whether the client making the calls is written in C or C++. Before any call to a method can be made, a pointer to the interface implementation must be retrieved. Interface pointers can be obtained by:

- Calling a method on a different object.
- Calling an API function.
- Calling **IUnknown::QueryInterface** on the target object.

MAPI provides several methods and API functions that return pointers to interface implementations. For example, clients can call the **IMAPISession::GetMsgStoresTable** method to retrieve a pointer to a table object that provides access to message store provider information through the **IMAPITable : IUnknown** interface. Service providers can call the API function **CreateTable** to retrieve a pointer to a table data object. When there is no function or method available and clients or service providers already have a pointer to an object, they can call the object's **QueryInterface** method to retrieve a pointer to another of the object's interface implementations.

Implementing MAPI Objects

MAPI objects are implemented using C++ classes or C data structures, depending on the language and the API set a client or service provider is using. Service providers are written in C or C++ with the MAPI service provider interface; client applications can use any of the three supported languages and the four supported client interfaces. If possible, clients and service providers using the object-oriented programming interface should use C++.

C++ is the preferred choice because MAPI is an object oriented technology and C++ lends itself more readily to object oriented development. The resulting code is simpler and more straightforward making it easier to maintain. Also, the MAPI documentation is written primarily for C++ developers; all of the syntax descriptions for the MAPI interface methods in the *MAPI Programmer's Reference* are in C++.

When a MAPI object is implemented, a client or service provider creates code for all of the interface methods, code for any private methods that are specific to the implementation, and code to support private data members for maintaining state information. The code for the interface methods must follow the specifications published by MAPI that document expected behavior.

There are many macros in the MAPIDEFS.H header file and OLE header files that clients and service providers in either language can use to help them with their definitions of MAPI objects. For example, there is a macro to define the methods of each of the MAPI interfaces. The macro to define the methods of **IUnknown** appears in MAPIDEFS.H as follows:

```
#define MAPI_IUNKNOWN_METHODS(IPURE)          \
    MAPIMETHOD(QueryInterface)                \
        (THIS_ REFIID riid, LPVOID FAR * ppvObj) IPURE;    \
    MAPIMETHOD_(ULONG,AddRef)  (THIS) IPURE;              \
    MAPIMETHOD_(ULONG,Release) (THIS) IPURE;   \
```

Implementing Objects in C++

C++ clients and service providers define MAPI objects by creating classes that inherit from the interfaces they are implementing. Each of the interface methods is public as are the constructor and destructor for the class. If the class has additional methods, they can be public or private, depending on the implementation. All data members are private.

The following example code shows how to define a C++ status object. The CMyMAPIObject class inherits from the **IMAPIStatus : IMAPIProp** interface. Many of the macros used in this example are defined in the OLE header file COMPOBJ.H.The first members of the class are the methods of the base interface, followed by the methods of the inherited interfaces in order of inheritance. Following the interface definitions are any additional methods, the constructor and destructor, and the data members.

```
class  CMyMAPIObject : public IMAPIStatus
{
public:
// Methods of IUnknown
    STDMETHODIMP QueryInterface (REFIID riid, LPVOID * ppvObj);
    STDMETHODIMP_(ULONG) AddRef ();
    STDMETHODIMP_(ULONG) Release ();

    MAPI_IMAPIPROP_METHODS(IMPL);
    MAPI_IMAPISTATUS_METHODS(IMPL);

// Other methods specific to CMyMAPIObject
    BOOL WINAPI Method1 ();
    void WINAPI Method2 ();

// Constructors and destructors
public :
    CMyMAPIObject () {};
    ~CMyMAPIObject () {};

// Data members specific to CMyMAPIObject
private :
    ULONG               m_cRef;
    CAnotherObj *       m_pObj;
};
```

To use an instance of the CMyMAPIObject class, C++ clients or service providers make a call to one of its methods as follows.

```
lpMyObj->ValidateState(ulUIParam, ulFlags);
```

Implementing Objects in C

Client applications and service providers written in C define MAPI objects by creating a data structure and an array of ordered function pointers known as a virtual function table, or vtable. A pointer to the vtable must be the first member of the data structure. In the vtable itself there is one pointer for every method in each interface supported by the object. The order of the pointers must follow the order of the methods in the interface specification published in the MAPIDEFS.H header file. Each function pointer in the vtable is set to the address of the actual implementation of the method. In C++, the vtable is set up automatically by the compiler. In C, it is not.

The following illustration shows how this works. The box on the far left represents a client that needs to use a service provider object. Through the session, the client obtains a pointer to the object, *lpObject*. The vtable appears first in the object followed by private data and methods. The vtable pointer points to the actual vtable containing pointers to each of the implementations of the methods in the interface.

{ewc msdncd, EWGraphic, groupx830 10 /a "MAPI_42.WMF"}

The following sample code shows how a C service provider can define a simple status object. The first member is the vtable pointer; the rest of the object is made up of data members.

```
typedef struct _MYSTATUSOBJECT
{
     const STATUS_Vtbl FAR *lpVtbl;

     ULONG              cRef;
     ANOTHEROBJ        *pObj;
     LPMAPIPROP         lpProp;
     LPFREEBUFFER       lpFreeBuf;

} MYSTATUSOBJECT, *LPMYSTATUSOBJ;
```

Because this object is a status object, the vtable includes pointers to implementations of each of the methods in the **IMAPIStatus : IMAPIProp** interface as well as pointers to implementations of each of the methods in the base interfaces: **IUnknown** and **IMAPIProp**. The order of methods in the vtable match the specified order as defined in MAPIDEFS.H.

```
static const MYOBJECT_Vtbl vtblSTATUS =
{
     STATUS_QueryInterface,
     STATUS_AddRef,
     STATUS_Release,
     STATUS_GetLastError,
     STATUS_SaveChanges,
     STATUS_GetProps,
     STATUS_GetPropList,
     STATUS_OpenProperty,
     STATUS_SetProps,
     STATUS_DeleteProps,
     STATUS_CopyTo,
     STATUS_CopyProps,
     STATUS_GetNamesFromIDs,
     STATUS_GetIDsFromNames,
     STATUS_ValidateState,
     STATUS_SettingsDialog,
     STATUS_ChangePassword,
     STATUS_FlushQueues
};
```

Clients and service providers written in C use objects indirectly through the vtable and add an object pointer as the first parameter in every call. Every call to a MAPI interface method requires a pointer to the object being called as its first parameter. C++ defines a special pointer known as the *this* pointer for this purpose. The C++ compiler implicitly adds the *this* pointer as the first parameter to every method call. Since in C there is no such pointer; it must be explicitly added.

The following code fragment demonstates how a client can make a call to an instance of MYSTATUSOBJECT:

```
lpMyObj->lpVtbl->ValidateState(lpMyObj, ulUIParam, ulFlags);
```

Implementing the IUnknown Interface

The methods of the **IUnknown** interface, implemented in every MAPI object, support interobject communication and object management. **IUnknown** has three methods: **QueryInterface**, **AddRef**, and **Release**. **QueryInterface** enables one object to determine if another object supports a particular interface. With **QueryInterface**, two objects with no prior knowledge of each other's functionality can interact. If the object implementing **QueryInterface** does support the interface in question, it returns a pointer to the implementation of the interface. If the object does not support the requested interface, it returns the E_NOINTERFACE value.

When **QueryInterface** returns a requested interface pointer, it must also increase what is known as the new object's reference count. An object's reference count is a numeric value used to manage the object's lifespan. When the reference count is greater than one, the object's memory cannot be freed because it is actively being used. It is only when the reference count drops to zero that the object can be released safely.

The other two **IUnknown** methods, **AddRef** and **Release**, manage the reference count. **AddRef** increments the reference count while **Release** decrements it. All methods or API functions that return interface pointers, such as **QueryInterface**, must call **AddRef** to increment the reference count. All implementations of methods that receive interface pointers must call **Release** to decrement the count when the pointer is no longer needed. **Release** checks for an existing reference count, freeing the memory associated with the interface only if the count is 0.

Note Because **AddRef** and **Release** are not required to return accurate values, callers of these methods must not use the return values to determine if an object is still valid or has been destroyed.

Implementing IUnknown in C++

Implementing **QueryInterface**, **AddRef**, and **Release** in C++ is fairly simple. After some standard validation of the parameters passed in, an implementation of **QueryInterface** checks the identifier of the requested interface against the list of supported interfaces. If the requested identifier is among those supported, **AddRef** is called and the *this* pointer returned. If the requested identifier is not on the supported list, the output pointer is set to NULL and the E_NOINTERFACE value is returned.

The following example shows how **QueryInterface** can be implemented in C++ for a status object, an object that is a sub-class of the **IMAPIStatus : IMAPIProp** interface. **IMAPIStatus** inherits from **IUnknown** through **IMAPIProp**. Therefore, because these interfaces are related through inheritance, if a caller asks for any of them, the *this* pointer can be returned.

```
HRESULT CMyMAPIObject::QueryInterface (REFIID   riid,
                                       LPVOID * ppvObj)
{
    // Always set out parameter to NULL, validating it first
    if (IsBadWritePtr(ppvObj, sizeof(LPVOID)))
        return E_INVALIDARG;

    *ppvObj = NULL;

    if (riid == IID_IUnknown || riid == IID_IMAPIProp ||
        riid == IID_IMAPIStatus)
    {
        // Increment reference count and return pointer
        *ppvObj = (LPVOID)this;
        AddRef();
        return NOERROR;
    }

    return E_NOINTERFACE;
}
```

Note that this example is a 32-bit implementation; 16-bit implementations must include a call to **ResultFromScode** on their return statements to translate the 16-bit status code values to HRESULT values. For example, the final return statement would be re-coded for 16-bit platforms as:

```
return ResultFromScode(E_NOINTERFACE);
```

The following code samples show how to implement the **AddRef** and **Release** methods for the CMyMAPIObject object. Because implementing **AddRef** and **Release** is so simple, many service providers choose to implement them inline. The calls to the Win32 functions **InterlockedIncrement** and **InterlockedDecrement** are to ensure thread safety. The memory for the object is freed by the destructor which is called when the **Release** method deletes the object.

```
ULONG CMyMAPIObject::AddRef()
{
    InterlockedIncrement(m_cRef);
    return m_cRef;
}

ULONG CMyMAPIObject::Release()
{
    // Decrement the object's internal counter
    ULONG ulRefCount = InterlockedDecrement(m_cRef);
    if (0 == m_cRef)
    {
        delete this;
    }

    return ulRefCount;
}
```

Implementing IUnknown in C

Implementations of **QueryInterface** in C are very similar to C++ implementations. There are two basic steps to the implementation:

1. Validating parameters.
2. Checking the identifier of the requested interface against the list of interfaces supported by the object and returning either the E_NO_INTERFACE value or a valid interface pointer. If an interface pointer is returned, the implementation should also call the **AddRef** method to increment the reference count.

The main difference between an implementation of **QueryInterface** in C and C++ is the additional first parameter in the C version. Because the object pointer is added to the parameter list, a C implementation of **QueryInterface** must have more parameter validation than a C++ implementation. The logic for checking the interface identifier, incrementing the reference count, and returning an object pointer should be identical in both languages.

The following code sample shows how to implement **QueryInterface** in C for a status object:

```
STDMETHODIMP STATUS_QueryInterface(LPMYSTATUSOBJ lpMyObj, REFIID lpiid,
                                   LPVOID FAR * lppvObj)
{

     HRESULT hr = hrSuccess;

     // Validate object pointer
     if (IsBadReadPtr(lpMyObj, sizeof(MYSTATUSOBJECT))
           || lpMyObj->lpVtbl != &vtblSTATUS )
     {
           hr = ResultFromScode(E_INVALIDARG);
           return hr;

     }

     // Validate other parameters
     if (IsBadReadPtr(lpiid, (UINT) sizeof(IID))
           || IsBadWritePtr(lppvObj, sizeof(LPVOID)))
     {
           hr = ResultFromScode(E_INVALIDARG);
           return hr;
     }

     // Set output pointer to NULL
     *lppvObj = NULL;

     // Check interface identifier
     if (memcmp(lpiid, &IID_IUnknown, sizeof(IID)) &&
           memcmp(lpiid, &IID_IMAPIProp, sizeof(IID)) &&
           memcmp(lpiid, &IID_IMAPIStatus, sizeof(IID)))
     {
           hr = ResultFromScode(E_NOINTERFACE);
           return hr;
     }

     // Interface is supported. Increment reference count and return
     lpMyObj->lpVtbl->AddRef(lpMyObj);
```

```
    *lppvObj = lpMyObj;
    return hr;
}
```

Whereas the implementation of **AddRef** in C is similar to a C++ implementation, a C implementation of **Release** can get more elaborate than a C++ version. This is because much of the functionality involved with freeing an object can be incorporated into the C++ constructor and destructor and C has no such mechanism. All of this functionality must be included in the **Release** method. Also, because of the additional parameter and its explicit vtable, more validation is required.

The following **AddRef** method call illustrates a typical C implementation for a status object:

```
STDMETHODIMP_(ULONG) STATUS_AddRef(LPMYSTATUSOBJ lpMyObj)
{
    LONG cRef;

    // Check to see if it has a lpVtbl object member
    if (IsBadReadPtr(lpMyObj,
                offsetof(MYSTATUSOBJECT, lpVtbl)+sizeof(STATUS_Vtbl *)))
    {
          return 1;
    }

    // Check size of vtable
    if (IsBadReadPtr(lpMyObj->lpVtbl,
           offsetof(STATUS_Vtbl, AddRef)+sizeof(STATUS_AddRef *)))
    {
          return 1;
    }

    // Check method
    if (STATUS_AddRef != lpMyObj->lpVtbl->AddRef)
    {
          return 1;
    }

    InterlockedIncrement(lpMyObj->cRef);
    cRef = ++lpMyObj->cRef;
    InterlockedDecrement (lpMyObj->cRef);

    return cRef;
}
```

A typical implementation of **Release** for a C status object follows. If after decrementing the reference count, it becomes zero, a C status object implementation should perform the following tasks:

- Release any held pointers to objects.
- Set the vtable to NULL, facilitating debugging in the case where an object's user has called **Release** yet continued to try to use the object.
- Call **MAPIFreeBuffer** to free the object.

```
STDMETHODIMP_(ULONG) STATUS_Release(LPMYSTATUSOBJ lpMyObj)
{
    LONG cRef;
    // Check size of vtable
    if (IsBadReadPtr(lpMyObj, sizeof(MYSTATUSOBJ)))
    {
          return 1;
    }

    // Check if correct vtable
    if (lpMyObj->lpVtbl != &vtblSTATUS)
    {
          return 1;
    }

    InterlockedIncrement(lpMyObj->cRef);
    cRef = --lpMyObj->cRef;
    InterlockedIncrement (lpMyObj->cRef);

    if (cRef == 0)
    {
          lpMyObj->lpVtbl->Release(lpMyObj);
          DeleteCriticalSection(&lpMyObj->cs);

          // release IMAPIProp pointer
          lpMyObj->lpProp->Release(lpMyObj->lpProp);
          lpMyObj->lpVtbl = NULL;
          lpMyObj->lpFreeBuff(lpMyObj);
          return 0;
    }

    return cRef;

}
```

Implementing a Sample Object

Advise sink objects, or objects that support the **IMAPIAdviseSink : IUnknown** interface, are an example of simple MAPI objects that are implemented by client applications for processing notifications. **IMAPIAdviseSink** inherits directly from **IUnknown** and contains only one method, **OnNotify**. Therefore, to implement an advise sink object, a client creates code for the three methods in **IUnknown** and for **OnNotify**.

The MAPIDEFS.H file defines an **IMAPIAdviseSink** interface implementation using DECLARE_MAPI_INTERFACE as follows:

```
#define      INTERFACE  IMAPIAdviseSink

DECLARE_MAPI_INTERFACE_(IMAPIAdviseSink, IUnknown)
{
    BEGIN_INTERFACE
    MAPI_IUNKNOWN_METHODS(PURE)
    MAPI_IMAPIADVISESINK_METHODS(PURE)
};
```

Clients implementing advise sink objects can define their interfaces in their objects manually or with the MAPI_IUNKNOWN_METHODS and MAPI_IMAPIADVISESINK_METHODS macros. Object implementors should use the interface macros whenever possible to ensure consistency across objects and to save time and effort.

Implementing the **IUnknown::AddRef** and **IUnknown::Release** methods is relatively simple because typically only a few lines of code are needed. Therefore, clients and service providers implementing objects can make their **AddRef** and **Release** implementations inline. The following code shows how to define a C++ advise sink object with inline implementations of **AddRef** and **Release**.

```
class  CMAPIAdviseSink : public IMAPIAdviseSink
{
public:
    STDMETHODIMP QueryInterface
                    (REFIID                     riid,
                     LPVOID *                   ppvObj);
    inline STDMETHODIMP_(ULONG) AddRef
                    () { InterlockedIncrement(m_cRef);
                         ++m_cRef;
                         InterlockedDecrement(m_cRef);
                         return m_cRef; };
    inline STDMETHODIMP_(ULONG) Release
                    () { InterlockedIncrement(m_cRef);
                         ULONG ulCount = --m_cRef;
                         InterlockedDecrement(m_cRef);
                         if (!ulCount)  delete this;
                         return ulCount;};
    MAPI_IMAPIADVISESINK_METHODS(IMPL);

    BOOL WINAPI AddConnection (LPMDB pMDBObj, ULONG ulConnection);
    void WINAPI RemoveAllLinks (LPMDB pMDBObj);

// Constructors and destructors
public :
    inline CMAPIAdviseSink  (CStoreClient * pStore)
                            { m_cRef = 1;
```

```
                              m_ulConnection = 0;
                              m_pStore = pStore;
                              AddRef;};
    ~CMAPIAdviseSink () {Release};

private :
    ULONG               m_cRef;
    CStoreClient *      m_pStore;
    ULONG               m_ulConnection;
};
```

In C, the advise sink object is made up of:

- A pointer to a vtable that contains pointers to implementations of each of the methods in **IUnknown** and **IMAPIAdviseSink**.
- Data members.

The following code sample shows how to define an advise sink object in C and construct its vtable.

```
// Object definition
typedef struct _ADVISESINK
{
     const ADVISE_Vtbl FAR * lpVtbl;

     ULONG             cRef;
     STORECLIENT      *pStore;
     ULONG             ulConnection;

} ADVISESINK, *LPADVISESINK;

// vtable definition
static const ADVISE_Vtbl vtblADVISE =
{
     ADVISE_QueryInterface,
     ADVISE_AddRef,
     ADVISE_Release,
     ADVISE_OnNotify
};
```

After an object is declared in C, it must be initialized by setting the vtable pointer to the address of the constructed vtable as is shown following:

```
LPADVISESINK lpMyObj = NULL;

HRESULT hr = (*lpAllocateBuffer) (sizeof(ADVISESINK),
                 (LPVOID *)&lpMyObj);
lpMyObj->lpVtbl = &vtblADVISE;
```

Properties

A property is an attribute of a MAPI object of a particular type. Properties describe something about the object, such as the subject line of a message or the address type of a distribution list. MAPI defines many properties that client applications and service providers can use for their objects. Clients and service providers can extend this set of predefined properties by creating new, custom properties. Clients can define properties to describe new message classes and service providers can define properties to expose the unique features of their messaging system.

Properties can be stored persistently with their objects either with the data of the object or in the profile or exist only for the duration of the current session. There are two mechanisms for displaying properties to a user: a table and a configuration property sheet. Tables provide users with a read-only view of property data in row and column format. Each row represents several properties for an object; each column represents a property. Configuration property sheets allow users to view or change property data, depending on the service provider.

MAPI limits the use of some of its properties to objects of particular types, states, or classes. The scope associated with a property can vary from a single object type to global usage. Properties that are used with many types of MAPI objects include the entry identifier, a reference property used to open objects, the object type, a constant used to identify the kind of object, and a display name, a character string that can be used to describe an object to the user. Properties that are more narrow in scope apply only to one type of object. Examples of single object properties include the message class, a character string that describes the type of a message object, and the telephone number, method, a character string used only with individual address book entries.

Properties that are applicable only when an object is in a particular state or belongs to a particular class are typically message properties. When a message is first created, its set of properties is very small. As a message progresses through transmission to delivery, its property set enlarges. Some of these added properties appear only on the message as it is being delivered while others appear only on the message as it is being sent. Messages also have properties that are associated with the class to which they belong. Report messages, for example, have a set of unique properties that do not appear on note messages.

Clients and service providers implement and use properties through the methods of the **IMAPIProp** interface. The **IMAPIProp** interface enables client applications and service providers to manipulate the properties of a MAPI object. There are methods for copying properties, making changes and saving those changes, mapping between property names and their identifiers, and retrieving information about a prior error.

Most properties have three characteristics:

- A value that describes the actual data of the property, such as the characters in the message class string or the numbers in a telephone number.
- A type that describes the kind of data, such as string or numeric. There is a standard set of property types that are used by all properties, whether predefined by MAPI or created by clients or service providers.
- A numeric identifier that uniquely describes the property.

MAPI defines ranges of identifiers for differentiating between properties; the identifier for a message content property defined by MAPI falls in a different range than the identifier for a message content properties defined by a client for a custom message class. A special range is used to identify properties that are described only by name. These properties are eferred to as named properties.

The identifier and type are combined to form a property tag, a constant that can be used to easily refer to the property. MAPI defines property tag constants for its properties. For custom properties not in the named property range, property tags must be created by the responsible client or service provider by following the MAPI specification. The type must be based on one of MAPI's predefined types and the identifier must be within the appropriate range. Named properties are referenced using methods in the standard property interface, **IMAPIProp**.

About Property Values

A property value is the data for the property. Property values are represented with the **SPropValue** data structure. The **SPropValue** structure is made up of two fields that client applications and service providers can work with: a property tag and a value. The **SPropValue** data structure is defined as follows.

```
typedef struct _SPropValue
{
    ULONG            ulPropTag;
    ULONG            dwAlignPad;
    union _PV        Value;
} SPropValue, FAR *LPSPropValue;
```

The **ulPropTag** member is a 32-bit unsigned integer that contains the property's unique identifier in the high order 16 bits and the property's type in the low order 16 bits.

The **Value** member is a union of data structures. The specific data for a property's **Value** member depends on the type of property. A character string property, for example, can have data of type LPSTR or LPWSTR, depending on whether the data is ANSI or Unicode. A boolean property uses an unsigned short integer for its **Value**. The **Value** member for an binary property is an **SBinary** data structure.

The **SPropValue** structure includes an alignment member, **dwAlignPad**, that is designed for computers that require 8-byte alignment for 8-byte values. Clients and service providers cannot use this member; it is reserved for MAPI. The alignment member makes it possible for clients and service providers to allocate an array of property values on an 8-byte boundary and have the array align correctly. Developers who write code on such computers should use memory allocation routines that allocate the **SPropValue** arrays on 8-byte boundaries.

About Property Types

A property type represents the format of a property value, the underlying data type of a property. MAPI defines a set of property types in the MAPIDEFS.H header file. All properties, whether they are defined by MAPI, by client applications, or by service providers, use one of these types. All of the types are represented by underlying operating system datatypes and some relate to OLE properties.

Property types are represented by constants that follow a similar naming convention to the one used for property tags. Many property types have both a single value and multivalue version. Single valued properties contain one value of its type such as a single integer or character string. The constant used to represent a single value property has two parts: the prefix "PT_" and a string describing the actual type, such as LONG or STRING8. Multivalued properties contain more than one value of its type. The constant used to represent multivalued properties is created by combining the MV_FLAG flag with the corresponding single value constant representing the base type. There are three parts: the prefix "PT_" followed by "MV_" followed by a string that describes the type. For example, the type for a property containing an array of character strings is PT_MV_STRING8.

The following illustration shows a property value structure with all of the potential property types. Notice that not all of the single-valued types have multivalued equivalents.

{ewc msdncd, EWGraphic, groupx831 0 /a "MAPI_11.WMF"}

The following illustration shows the structure of an **SPropValue** structure to describe a multivalued property with the base type of PT_LONG. The value is no longer a single variable; it is made up of a pointer that points to an array of values and a count of the number of values in the array.

{ewc msdncd, EWGraphic, groupx831 1 /a "MAPI_12.WMF"}

Although support for multivalued properties is optional, MAPI recommends that clients and service providers support both types of properties because doing so enables greater interaction between MAPI-compliant components.

Property Type Summary

The following table describes the property types that are supported by MAPI.

Property type	Multivalued type	Underlying data type
PT_APPTIME	PT_MV_APPTIME	Double. Interpreted as date and time. Same as OLE type VT_DATE and compatible with the Visual Basic time representation.
PT_BINARY	PT_MV_BINARY	SBinary. Counted byte array.
PT_BOOLEAN	PT_MV_12	16-bit Boolean; 0 = False, non-zero = True. Same as OLE type VT_BOOL.
PT_CLSID	PT_MV_CLSID	CLSID structure. Class identifier. VT_CLSID.
PT_CURRENCY	PT_MV_CURRENCY	64-bit integer intepreted as decimal. Compatible with Visual Basic CURRENCY type. Same as OLE type VT_CY.
PT_DOUBLE	PT_MV_DOUBLE	Double; 64-bit floating point, PT_R4. Same as OLE type VT_R8.
PT_ERROR	(None)	SCODE; 32-bit unsigned integer. Same as OLE type VT_ERROR.
PT_FLOAT	PT_MV_FLOAT	32-bit floating point, PT_R4. Same as OLE type VT_R4.
PT_I2	PT_MV_I2	Signed 16-bit integer, PT_SHORT. Same as OLE type VT_I2.
PT_I4	PT_MV_I4	Signed or unsigned 32-bit integer. Same as OLE type VT_I4. Same as PT_LONG.
PT_I8	PT_MV_I8	Signed or unsigned 64-bit integer. Same as OLE type VT_I8. Uses the structure LARGE_INTEGER.
PT_LONG	PT_MV_LONG	Signed or unsigned 32-bit integer, PT_I4. Same as OLE type VT_I4.
PT_LONGLONG	PT_MV_LONGLONG	Signed or unsigned 64-bit integer, PT_I8. Same as OLE type VT_I8.
PT_NULL	(None)	Indicates no property value. Same as OLE type VT_NULL. Reserved for interface use.

PT_OBJECT	(None)	A pointer to an object that implements IUnknown. Similar to several OLE types, including VT_UNKNOWN.
PT_R4	PT_MV_R4	4-byte floating-point data. Same as OLE type VT_R4.
PT_R8	PT_MV_R8	8-byte floating-point data. Same as OLE type VT_DOUBLE.
PT_SHORT	PT_MV_SHORT	Signed 16-bit integer, PT_SHORT. Same as OLE type VT_I2.
PT_STRING8	PT_MV_STRING8	Null-terminated 8-bit character string data. Same as OLE type VT_LPSTR.
PT_SYSTIME	PT_MV_SYSTIME	64-bit integer date/time value in the form of a **FILETIME** structure. Same as OLE type VT_FILETIME.
PT_TSTRING	PT_MV_TSTRING	Set to PT_UNICODE when compiling with the UNICODE symbol; else PT_STRING8. Either the OLE type VT_LPSTR or VT_LPWSTR.
PT_UNICODE	PT_MV_UNICODE	Null-terminated wide string data. Same as OLE type VT_LPWSTR.
PT_UNSPECIFIED	(Not supported)	Indicates that the client application does not supply the property type. Reserved for interface use.

About the Boolean Property Type

Some of the property types have special meaning and use. Boolean properties can have two states, set or unset, and two values in its set state: TRUE or FALSE. Your client or service provider may need to differentiate in its handling of boolean properties when properties are not available, when they are available and set to TRUE, or when they are available and set to FALSE. Specifically, either the same course of action or separate courses of actions may be appropriate when a property is set to FALSE and when it does not exist.

For example, consider a transport provider's handling of the message property PR_SEND_RICH_INFO:

- When PR_SEND_RICH_INFO is set to TRUE, the transport provider transmits the full context of the message.
- When PR_SEND_RICH_INFO is set to FALSE, the transport provider discards unnecessary message content.
- When PR_SEND_RICH_INFO does not exist, the transport provider follows its default course of action, whatever that is.

About the PT_UNSPECIFIED Property Type

MAPI defines a special property type, PT_UNSPECIFIED, that your client or service provider can use to retrieve a property when the property type is not known.To retrieve a property without advance knowledge of its type, a client or service provider calls **IMAPIProp::GetProps** and passes a property tag made up of the property's identifier and the PT_UNSPECIFIED property type. **GetProps** returns an **SPropValue** structure for the property, replacing PT_UNSPECIFIED with the appropriate type. Service providers implementing **GetProps** are required to support PT_UNSPECIFIED.

About the Object Property Type

Some MAPI objects support properties that are themselves objects. For example, folders and address book containers support two table object properties:

PR_CONTAINER_HIERARCHY
PR_CONTAINER_CONTENTS

PR_CONTAINER_HIERARCHY provides a summary view of a folder or address book container's hierarchy and PR_CONTAINER_CONTENTS provides a summary view of objects within a folder or address book container. Your client or service provider can access table object properties in one of two ways:

- A call to the **IMAPIProp::OpenProperty** method, specifying the appropriate interface for access.
- A call to a method on the object supporting the property.

For example, to access PR_CONTAINER_HIERARCHY with the **OpenProperty** method, your client or service provider would specify IID_IMAPITable as the interface identifier because PR_CONTAINER_HIERARCHY is a table object. To access it using the second approach, your client or service provider would call the folder or address book container's **IMAPIContainer::GetHierarchyTable** method.

About the String Property Type

String properties are described with another special property type, PT_TSTRING. The PT_TSTRING property type compiles conditionally to one of the following two other string property types that describe null-terminated strings, depending on the operating system's character set:

- PT_STRING8 for 8-bit ANSI character strings
- PT_UNICODE for double-byte character strings

Either or a client or a service provider or both client and provider can opt to support Unicode character strings. It is not required. A client that supports only PT_STRING8 strings can operate with a provider that supports Unicode and vice versa. To allow this interoperability, clients and service providers pass a flag, the MAPI_UNICODE flag, to indicate that Unicode is supported in methods that involve the exchange of character strings.

For example, suppose your client supports Unicode and needs to retrieve the display name of a folder. All of its PT_TSTRING properties are compiled to type PT_UNICODE. When your client calls the folder's **IMAPIProp::GetProps** method to retrieve its PR_DISPLAY_NAME property, it passes the MAPI_UNICODE flag. Specifying MAPI_UNICODE informs the message store provider that your client expects a Unicode display name to be returned.

Clients and service providers need to be aware that specifying MAPI_UNICODE in a method call is only a request because the implementor of the method might not support Unicode. Supporting Unicode is optional. However, service providers are encouraged to support both environments because it allows them to achieve more widespread distribution than providers that support only one environment.

String properties can grow to be quite large as can binary properties, properties that use the property type PT_BINARY. To facilitate the management of large string and binary properties that are read using **IMAPIProp::GetProps** and set using **IMAPIProp::SetProps**, MAPI enables service providers to enforce size limits. These limits can vary, depending on the following considerations:

- Whether the properties are being read or written.
- The service provider implementing the **IMAPIProp** methods.
- Runtime considerations such as memory constraints.
- Character set translation issues.

Size limits can also be placed on string and binary properties when they are used in the column set of a table because it is sometimes impossible to make all of a large property's value visible. Many service providers truncate large string or binary properties that are used in tables to 255 bytes.

When a client calls **GetProps** or **SetProps** to work with a large string or binary property and exceeds the limits of the particular service provider, the provider returns the error value MAPI_E_NOT_ENOUGH_MEMORY. If it is **GetProps** that is failing for a specific property, the client can recover by calling **IMAPIProp::OpenProperty**, specifying IID_IStream as the interface identifier. With **OpenProperty**, the client can retrieve a large property using an interface, such as **IStream**, that is better suited for working with large properties.

About Property Identifiers

A property identifier is a number that is used to indicate what a property is used for and who is responsible for it. Property identifiers are divided by MAPI into ranges; where an identifier falls in the range indicates its use and ownership.

The range of property identifiers runs from 0001 through FFFF. Property identifiers 0000 and FFFF are reserved in all cases, meaning that these identifiers must remain unused. The range for MAPI's sole use runs from 0001 to 3FFF. The range 4000 to 7FFF belongs to message and recipient properties, and either clients or service providers can define properties in this range. Beyond 8000 is the range for what is known as named properties, or properties that have a name associated with their identifier. Clients use named properties to customize their property set.

Service providers can use the range 0x67F0 thru 0x67FF to define secure profile properties. Secure profile properties are used for information that requires additional protection, such as passwords. These properties can be hidden and encrypted. Whether or not secure properties are included in the default list of properties returned by the **IMAPIProp::GetPropList** method depends on the provider's implementation. Usually these properties are not included. The **IProfSect** interface is used for accessing secure profile properties.

Some properties are defined as being either transmittable or non-transmittable. Transmittable properties are transferred with a message; non-transmittable properties are not transferred with a message. Non-transmittable properties usually contain information that is of value only to the client and service providers operating with the current session. These properties would not necessarily be useful to another messaging system and another set of service providers. The concept of transmittable properties applies primarily to transport providers. To determine whether a property tag is transmittable or not, use the **FIsTransmittable** macro (see the MAPITAGS.H header file).

The following table summarizes the different ranges for property identifiers, describing the owner for the properties in each range.

Range	**Owner**	**Type of property**
0000	MAPI	Reserved for the special value PR_NULL
0001 - 0BFF	MAPI	Message envelope properties
0C00 - 0DFF	MAPI	Recipient properties
0E00 - 0FFF	MAPI	Non-transmittable message properties
1000 - 2FFF	MAPI	Message content properties
3000 - 3FFF	MAPI	Properties for objects other than messages and recipients
4000 - 57FF	Transport providers	Message envelope properties
5800 - 5FFF	Transport and address book providers	Recipient properties
6000 - 65FF	Clients	Non-transmittable message properties
6600 - 67FF	Any service provider	Various properties; visible or invisible to users at provider's option

6800 - 7BFF	Creators of custom message classes	Properties for custom message classes
7C00 - 7FFF	Creators of custom message classes	Non-transmittable properties for custom message classes
8000 - FFFE	Clients and occasionally service providers	Properties identified only by name, through the methods in **IMAPIProp** for mapping property names to identifiers
FFFF	MAPI	Reserved for the special error value PROP_ID_INVALID

The range between 3000 and 3FFF is reserved for properties that are not related to either messages or recipients. MAPI divides this range into sub-ranges by types of object; the following table shows this further breakdown.

Identifier range	**Type of property**
3000 - 33FF	Common properties that appear on multiple objects, such as display name and entry identifier
3400 - 35FF	Message store properties
3600 - 36FF	Container (folder and address book container) properties
3700 - 38FF	Attachment properties
3900 - 39FF	Address book properties
3A00 - 3BFF	Messaging user properties
3C00 - 3CFF	Distribution list properties
3D00 - 3DFF	Profile properties
3E00 - 3FFF	Status object properties

About Property Tags

A property tag is a 32-bit number that contains a unique property identifier in bits 16 through 31 and a property type in bits 0 through 15 as shown below:

{ewc msdncd, EWGraphic, groupx831 2 /a "MAPI_10.WMF"}

MAPI defines a set of property tags for use by clients and service providers. If necessary, new property tags can be created if the predefined ones are insufficient. The MAPI property tags are represented by constants stored in the MAPITAGS.H header file. These constants follow a naming convention for consistency and ease of use. All property tags begin with the prefix PR_.

A few macros are available to help manipulate the property tag data structure, among them PROP_TYPE, PROP_ID, and PROP_TAG. PROP_TYPE extracts the property type from the property tag; PROP_ID extracts the identifier. PROP_TAG builds the property tag from a property type and identifier.

About Required and Optional Properties

Every object has a set of required properties and a set of optional properties. Depending on the object, the service provider supplying the implementation, and the property, a client can have read/write or read-only access to a property in either set. A required read/write property is a property that must exist on an object before it can be successfully saved with the **IMAPIProp::SaveChanges** method. A required read-only property is a property that always exists on an object and is therefore always available by calling either **IMAPIProp::GetProps** or **IMAPIProp::OpenProperty**. Read-only properties are typically computed by the service provider supplying the object implementation.

Optional properties cannot be expected to be available or set to valid values. When a client or service provider calls **GetProps**, asking for an unavailable property, it succeeds with the warning MAPI_W_ERRORS_RETURNED. **OpenProperty**, however, fails with the error MAPI_E_NOT_FOUND. Clients and service providers must check that a requested property is returned before attempting to use it.

When an optional property is included as one of the columns in a table, some of the rows might have valid values for the column while others might not. Whether or not a valid value exists for a column depends on whether or not the object providing the information for the row sets the property. Depending on the implementation of the object, a non-existent property can be represented in the table as PR_NULL or an arbitrary value. Users of tables must be careful to differentiate between properties that are nonexistent and have meaningless values and properties that do exist and have valid values.

To find out exactly which properties are currently set for an object, clients and service providers can call **IMAPIProp::GetPropList. GetPropList** lets a caller find out what is available before an attempt to open a potentially nonexistent property is made. Because there is no standard set of properties that all objects of a specific type support, it is impossible to guess whether or not an object supports a particular property. Calling **GetPropList** eliminates the guess work.

About Property Errors

The methods of **IMAPIProp** can report partial success in one of two ways. **GetProps** returns the error MAPI_W_ERRORS_RETURNED and places individual error information with property data in the property value array that is passed back as output. The error information is made up of a PT_ERROR property type and an SCODE indicating the reason for the error. For example, if a client requests three properties and the third is unavailable, the service provider places PT_ERROR in the third property type in the property value array and MAPI_E_NOT_FOUND in the third property value.

SetProps, **DeleteProps**, **CopyTo**, and **CopyProps** do not return MAPI_W_ERRORS_RETURNED and do not place information in a property value array. Instead, these methods return S_OK for partial success and place error information in an **SPropProblemArray** data structure. Unlike the property value array in **GetProps** that contains data regardless of whether the method succeeded or failed, the problem problem array in these methods exists only if there are errors and only if the caller has registered interest in learning about the errors. Callers must specify a valid **SPropProblemArray** pointer to register for error information.

The **SPropProblemArray** structure contains an array of **SPropProblem** structures. The **SPropProblem** structure is defined as follows:

```
typedef struct _SPropProblem
{
    ULONG    ulIndex;            // which tag has the problem
    ULONG    ulPropTag;          // tag with a problem
    SCODE    scode;              // type of problem
} SPropProblem, FAR *LPSPropProblem;
```

When an error code is returned fom **SetProps**, **DeleteProps**, **CopyTo**, or **CopyProps**, this indicates failure rather than partial success. The property problem array, if available, is not valid. Clients should not try to access data held in the structure nor should they try to free the structure itself. The appropriate response is to call **IMAPIProp::GetLastError**.

GetLastError is similar to the function of the same name provided in the Win32 SDK. Both provide more detailed information about an error than is available with the return value. They both return information about the previous error that has occurred. The difference is that the Win32 **GetLastError** function reports on an error generated by the calling thread and the **IMAPIProp::GetLastError** method reports on an error generated by the current object. That is, if a client calls **DeleteProps** on a message and MAPI_E_NO_ACCESS is returned to indicate that the message is read-only, **GetLastError** returns data provided by the message.

About Defining New Properties

In spite of the wealth of properties supplied by MAPI for use by clients and service providers, MAPI enables new properties to be created if necessary. Some of the valid scenarios for defining new public properties include a client creating properties to support a new message class and a service provider creating new properties to expose unique messaging system features. It is typically not valid for a service provider to define new properties for an existing MAPI object or message class. One of the primary benefits of using MAPI is that standard identifiers and formats for a large number of messaging system elements are set up, enabling users to seamlessly mix and match service providers. Service providers that use nonstandard properties do not work as well with other service providers.

Clients can create content properties for new message classes in one of two ways:

- Using property identifiers within a designated range for message class-specific content properties.
- Using named properties.

Property identifiers must fall in predefined ranges. Assigning an identifier to a property in the appropriate range helps prevent collisions between properties defined by different vendors or users. MAPI defines two separate ranges for clients to use for new message class-specific content properties. The 0x6800 to 0x7BFF range is for transmittable properties and the 7C00 to 7FFF range is for nontransmittable properties. Users of properties in these ranges cannot make assumptions as to the behavior of the properties. Every client that creates a new message class has access to these ranges; a property with identifier xxxx can mean one behavior for one message class and another behavior for another message class. Creating public properties in the 0x6800 to 7FFF range is preferable to using named properties for custom message classes because not all service providers support named properties.

Named properties are used to guarantee a specific property is unique to a message class. Named property identifiers start in the 0x8000 range. Clients define one or more names and then call an object's **IMAPIProp::GetIDsFromNames** method to associate an identifier with each name. Named properties can be used by clients or service providers to define new properties for any object if the owner of the object supports named properties. Users of these properties call **GetIDsFromNames** and a related **IMAPIProp** method, **GetNamesFromIDs,** to map between a name and its identifier.

All properties, new or existing, must use the set of predefined property types. New property types cannot be added and existing types cannot be modified or deleted. Simple, small properties, such as single-character or 16-bit integer properties, can be stored in any appropriate type. For example, integers can be stored as ULONG and strings can be stored as PT_STRING8.

Use the PT_BINARY type to indicate a counted byte array. This property type is useful for extending the types of data that can be stored in an object. Bytes are transmitted in sequence and no assumptions are made about the meaning of the data. When a client application reads data out of such a property, the data is unchanged from how it was stored. The client must perform any necessary byte-swapping when moving data across CPUs.

About Object and Property Access

Access level, or the set of permissable operations, can be a characteristic of MAPI objects and of individual properties on those objects. Object access is determined by the container that holds the object, such as the folder that holds the message or the address book container that holds the distribution list. Therefore, two copies of the same object can have different access levels.

Object users can request the highest level of access for an object by setting the MAPI_BEST_ACCESS flag on the **OpenEntry** call. To determine the level of access actually assigned, object users can examine the PR_ACCESS property.

Container level access can be determined with the PR_ACCESS_LEVEL property. By retrieving PR_ACCESS_LEVEL, a caller can find out if the container is a read-only container or if it allows objects within it to be created, modified, or deleted. Container level access affects clients in terms of how they display their user interface. It also impacts the implementors of objects within containers, in terms of their user interface display and their general implementation.

Property access is determined by the property schema set up by MAPI for the object that owns the property. Property schemas specify the set of required and optional properties for an object and their access levels. A property's access level is global; an object's container has no bearing on whether its properties are read-only or read/write.

An object's read-only properties will always be available with a **GetProps** or **OpenProperty** call, but when **SetProps** is called to change their value or **DeleteProps** is called to delete them, the method either fails, returning MAPI_E_NO_ACCESS, or succeeds with no action taken. Whether or not these calls fail is up to the service provider implementing the object.

Property and object access can also be retrieved or set using one of the interfaces that inherits from **IMAPIProp**, **IPropData**. MAPI provides an implementation of **IPropData** that is based on data in memory. Service providers can use **IPropData** to implement **IMAPIProp** in certain circumstances, such as for the status object or if your provider is using a database that does not have built-in transactions. **IPropData** works exclusively in memory, making it unnecessary to lock and unlock data.

The **IPropData** interface provides the ability to retrieve and change the access for an object's properties. Service providers access a property data object by calling the MAPI API function, **CreateIProp**. There are four **IPropData** methods: **HrSetObjAccess**, **HrAddObjProps**, **HrSetPropAccess**, and **HrGetPropAccess**.

HrSetObjAccess establishes the access mode for an object. By default, objects have read/write access, enabling the addition, deletion, and modification of properties. To create an object with properties that will be read-only to clients, service providers can start out with a read/write object, add the necessary properties, and then call **HrSetObjAccess** to change the object's access to read-only.

HrAddObjProps adds object properties to an object's property set. That is, it adds properties of type PT_OBJECT, such as PR_CONTAINER_CONTENTS, to a MAPI object, such as a folder. Callers can specify a pointer to a property problem array for MAPI to use to report errors. If S_OK is returned and a valid **SPropProblemArray** pointer was specified, callers need to check for possible errors. Typically, the only problem that occurs is lack of memory. To add an object property, the target object must have read/write access. Any attempt to add an object type property to an object with read-only access will result in MAPI_E_NO_ACCESS being returned.

HrSetPropAccess is used to set individual properties to read-only or read/write access and to set the state of a property to clean or dirty. This is useful for determining when a particular property value changes. The bits in the *rgulAccess* parameter to this method are positional with respect to the *lpPropTagArray* parameter. This means that each entry in the flag array corresponds to an entry in the property tag array.

HrGetPropAccess is used to retrieve access rights for properties and for determining whether a property has been modified or deleted. A property tag array with entries for all of an object's properties is returned. If a property that once existed on the object has been deleted, its property tag entry will contain zero. If a property was modified, its access level will be IPROP_DIRTY rather than a value that indicates read-only or read/write access. If a property was neither modified nor deleted, its access level will read IPROP_READONLY or IPROP_READWRITE.

About Opening Properties

The **IMAPIProp** methods **GetProps** and **OpenProperty** and the API function **HrGetOneProp** are used to open one or more properties. Opening a property is synonymous to retrieving its value, its type, and identifier, the data in an **SPropValue** structure. **GetProps** is used to retrieve one or more properties that can be manipulated with **IMAPIProp** alone. This implies that the properties available with **GetProps** are small, such as integers and boolean values. **OpenProperty** is used to open larger properties that require another interface such as **IStream** or **IMAPITable** for access. **OpenProperty** is typically used to open large character string, binary, and object properties and can only open one property at a time. Callers pass in the identifier of the additional interface that is required as one of the input parameters.

HrGetOneProp also opens one property at a time. Callers that need several properties can either call **HrGetOneProp** or **OpenProperty** in a loop or make one call to **GetProps**. Calling **GetProps** once is more efficient. In fact, **HrGetOneProp** should only be used when the target object exists on the local machine. When the target object is not locally available, using **HrGetOneProp** can result in multiple remote procedure calls and lessened performance.

To open one or more properties with **GetProps**, a caller specifies the property tags for the properties to be opened, a flag that indicates Unicode support, the address for a property value array to hold the returned property values, and a count of the number of entries in the array. A caller can specify one or more specific property tags, NULL for the property tag array pointer, or the tag PR_NULL in one of the members of the array. A NULL property tag array pointer indicates that **GetProps** should return all of the property values supported by the object. The PR_NULL property tag is used to reserve a slot in the returned property value array for a property to be added after the **GetProps** call.

Setting the MAPI_UNICODE flag indicates that all character strings passed into the **GetProps** call are Unicode strings and that all character strings passed out of the call should be in Unicode format.

The count of the number of entries in the property value array is always the same as the number of tags passed in. This value is most useful when a caller asks for all properties on an object without knowing how many properties the object supports.

To open a property with GetProps

1. Allocate a property tag data structure to hold the number of properties to be opened.
2. Set each member in the property tag array to the identifier and type of one property to be opened. If the appropriate type is unavailable, set it to PT_UNSPECIFIED.
   - If neither the type or the identifier for a particular property is available, call **IMAPIProp::GetPropList** to access a property tag array of all of the properties supported by the object.
   - If the property is a named property and only the name is available, call **IMAPIProp::GetIDsFromNames** to access the associated identifier.
3. Set a numeric variable to the number of entries in the property tag array.
4. Call **IMAPIProp::GetProps** to open the property or properties.

If you cannot allocate memory, return MAPI_E_NOT_ENOUGH_MEMORY. If you cannot allocate for a specific property, return success and fail just the one property.

GetProps can return one of the following values:

S_OK
MAPI_W_ERRORS_RETURNED
MAPI_E_NOT_ENOUGH_MEMORY

S_OK indicates a successful retrieval of every property requested while MAPI_W_ERRORS_RETURNED indicates that there was a problem with one or more of the properties. Possibly memory could not be allocated or the property is unsupported. The property value array that **GetProps** returns has one entry for every requested property, regardless of whether or not it succeeded in retrieving the property. For failed properties, **GetProps** sets the property type to PT_ERROR and the property value to an status code that describes the error. For example, when memory cannot be allocated, its value in the returned array is MAPI_E_NOT_ENOUGH_MEMORY and when a property is not supported, its value is MAPI_E_NOT_FOUND. **GetProps** fails completely with the value MAPI_E_NOT_ENOUGH_MEMORY when it cannot allocate even partial memory.

Some of the common uses of **OpenProperty** include opening PR_BODY, the property that holds the body of a text-based message, PR_ATTACH_DATA_OBJ, the property that holds an OLE object or message attachment, and PR_CONTAINER_CONTENTS, the property that holds a message store or address book contents table. Depending on the property, a different interface is requested from **OpenProperty**. **IStream**, an interface that allows property data to be read and written as a stream of bytes, is typically used to access PR_BODY. Either **IMessage** or **IStreamTnef** can be used to access PR_ATTACH_DATA_OBJ. Embedded message attachments that are standard messages use **IMessage** whereas messages in the TNEF format use **IStreamTnef**. Because PR_CONTAINER_CONTENTS is a table object, it is accessed with **IMAPITable**.

The following code sample shows how a client creates a file attachment, storing the data in the message's PR_ATTACH_DATA_BIN property. The client begins by opening a stream for the file using the MAPI utility function **OpenStreamOnFile** and then calls **OpenProperty** to open a stream for the attachment property. The data in the file is copied directly from the file stream into the attachment stream using **IStream::CopyTo** after a call to **IStream::Stat** to determine the size of the file stream. Another way to determine stream size is to call **IStream::Seek** with the flag STREAM_SEEK_END. When the copy operation is finished, both streams are released.

```
LPSTREAM pStreamFile, pStreamAtt;
HRESULT hr;

hr = OpenStreamOnFile (MAPIAllocateBuffer, MAPIFreeBuffer,
                       STGM_READ, "myfile.doc", NULL, &pStreamFile);
if (HR_SUCCEEDED(hr))
{
    // Open the destination stream in the attachment object
    hr = pAttach->OpenProperty (PR_ATTACH_DATA_BIN,
                                &IID_IStream,
                                0,
                                MAPI_MODIFY | MAPI_CREATE,
                                (LPUNKNOWN *)&pStreamAtt);
    if (HR_SUCCEEDED(hr))
    {
        STATSTG StatInfo;
        pStreamFile->Stat (&StatInfo, STATFLAG_NONAME);
        hResult = pStreamFile->CopyTo (pStreamAtt, StatInfo.cbSize,
                                       NULL, NULL);
        pStreamAtt->Release();
    }
    pStreamFile->Release();
}
```

When **IStream** is used for property access, some service providers automatically send the size of the property back with the stream. Calling **OpenProperty** with the MAPI_DEFERRED_ERRORS flag can delay the opening of the property and the return of the stream size. If **IStream::Stat** is called to retrieve this size after **OpenProperty** with the MAPI_DEFERRED_ERRORS flag set, performance will be impacted because this sequence of calls forces an extra remote procedure call. To avoid the performance hit, clients can call any MAPI method between the calls to **OpenProperty** and to **Stat**.

Note Secure properties are not automatically available with other properties in a **GetProps**, **HrGetOneProp**, or **GetPropList** call. Secure properties must be explicitly requested by property identifiers.

About Setting Properties

To set property values, call **IMAPIProp::SetProps** or the API function **HrSetOneProp**. **SetProps** can be used to set one or more properties in an object whereas **HrSetOneProp** sets only one property at a time. Use **HrSetOneProp** only if the target object is local; this function can be expensive when used with remote objects.

The parameters for **SetProps** are similar to the parameters for **GetProps** with the caller supplying a property value array containing the new property values. Implementors of **SetProps** can enable callers to change not only the property value but also the property type. New property tags can be formed from an existing identifier and the new property type that is passed in. For example, address book providers can support changing the home telephone number property from PT_UNICODE to PT_MV_UNICODE when the user adds a second home telephone number.

About Copying Properties

There are two **IMAPIProp** methods for copying properties between objects. The **CopyTo** method copies or moves all of the properties of one object to another object; the **CopyProps** method copies or moves a selected set. Client applications primarily use **CopyTo** for moving and copying objects, such as moving a message from the Inbox to another folder. Typically it is appropriate for all of the message's properties, with the exception of status, to be moved with the message. **CopyProps** is used mainly for replying to and forwarding messages, where only some of the set of properties from the original message travel with the reply or forwarded copy.

The **CopyTo** and **CopyProps** methods have very similar parameter lists. Both methods include parameters for requesting a progress dialog box and a problem report and for specifying the destination object and an interface for access. Also, there is a set of flags for controlling the procedure. **CopyTo** has three additional parameters for excluding specific interfaces or properties from the operation.

Because the copy operation can be lengthy, service providers implementing **CopyTo** and **CopyProps** are encourage to support the display of a progress dialog box. Callers requesting the progress display can implement the **IMAPIProgress** interface and pass a pointer to that implementation to the copy method or ask to use an implementation provided by MAPI. To request the MAPI implementation, callers pass NULL for the **IMAPIProgress** pointer. All callers set the MAPI_DIALOG flag regardless of which progress implementation is to be used.

CopyTo and **CopyProps** can report global and individual errors, or errors that occur with one or more properties. These individual errors are placed in an **SPropProblemArray** structure. Callers can suppress error reporting at the property level by passing NULL for the property problem array structure parameter rather than a valid pointer.

By default, **CopyTo** copies all of an object's properties from the object to a destination object whereas **CopyProps** only copies a set of specified properties. Both methods have a property tag array parameter for holding all of the identifiers of the properties to be excluded, in the **CopyTo** case, or included, in the **CopyProps** case. **CopyTo** also enables callers to exclude interfaces from the copy operation. Callers to **CopyTo** can set one of these parameters, both of them, or neither. Callers to **CopyProps**, on the other hand, are required to specify at least one property in the property tag array.

Excluding properties on a **CopyTo** call can be useful. For example, some objects have properties that are specific to a single instance of the object, such as the date and time of message delivery. To avoid copying a message's delivery time when copying the message to a different folder, callers specify PR_MESSAGE_DELIVERY_TIME in the property tag exclude array to **CopyTo**.

The usefulness of the **CopyTo** feature for excluding interfaces is perhaps not as obvious as the usefulness of excluding properties. Callers can exclude an interface when they are copying to an object that has no knowledge of a whole set of properties. For example, if a client copies properties from a folder to an attachment, the only properties that the attachment will be able to work with are the generic properties accessible with any **IMAPIProp** implementation. By excluding **IMAPIFolder** from the copy operation, the attachment will not receive any of the more specific folder properties.

Note Callers that specify a base interface to be excluded cause the interfaces inherited from it to be excluded also. Therefore, do not ever exclude the **IUnknown** interface in the **CopyTo** call. Because **IUnknown** is at the top of the inheritance, excluding it will cause nothing to be copied.

The set of flags on the copy methods include:

MAPI_NO_REPLACE
MAPI_MOVE
MAPI_DIALOG
MAPI_DECLINE_OK

Callers specify MAPI_NO_REPLACE to disable the copy if the property already exists on the destination object and MAPI_MOVE to make the operation a move operation rather than a copy. MAPI_DIALOG is set to request a progress dialog box. MAPI_DECLINE_OK is a flag that is set by MAPI in its calls to message store provider **CopyTo** or **CopyProps** implementations. Clients should not use this flag. MAPI_DECLINE_OK indicates to a message store provider intending to call the support method **DoCopyTo** or **DoCopyProps** for its **CopyTo** or **CopyProps** implementation, that it should return MAPI_E_DECLINE_COPY instead.

Callers copying properties that are unique to the source object type must be sure that the destination object is of the same type. MAPI does not prevent clients and service providers from associating properties that typically belong to one type of object with another type of object. It is up to the caller to copy properties that make sense for the destination object and to the implementor of the destination object to potentially deal with foreign properties that it cannot access. To check that the source and destination object are the same type of object, a caller can either compare pointers to the two objects or call **IUnknown::QueryInterface**.

MAPI provides three API functions for copying properties in memory rather than from one object to another object. These functions are presently supported, but might not be supported in a future release. **PropCopyMore** copies a single property value from one location to another. **PropCopyMore** must be used with caution because it is possible, when copying one value at a time, to allocate many small blocks of memory and cause memory to fragment. Consider using the function **ScCopyProps**, if possible, to copy property values in bulk. **ScCopyProps** can copy property values that have been built from disjointed blocks of memory. It returns a new property array. If this array should be stored on disk, the **ScRelocProps** function can be used to adjust the pointers. **ScRelocProps** should be called twice; once to adjust the addresses before writing the data operation and then again during the read operation. The **ScRelocProps** function assumes that the property value array was originally allocated in a single allocation.

About Saving Property Changes

Many objects support a transaction model of processing whereby changes to the object are not made permanent until they are committed in a separate call at a later time. The **IMAPIProp::SetProps** method is used to change the value of a property and the **IMAPIProp::DeleteProps** method is used to delete a property. However, neither of these calls make permanent changes to the object; clients or service providers must call the **IMAPIProp::SaveChanges** method to perform the commit. With a transaction model in place, a property that has been set might not be available until after the call to save changes has successfully completed.

When a client application receives the error value MAPI_E_OBJECT_CHANGED from a **SaveChanges** call, this is a warning that another client has already called **SaveChanges** on the object. It is possible for multiple clients to open an object with read/write access at the same time. That is, multiple **OpenEntry** calls on the same object with the MAPI_MODIFY flag set can succeed, depending on the provider. The object that is returned from such an **OpenEntry** call is a snapshot of the storage data. Each subsequent attempt to change this data can overwrite the previous attempt.

Upon receiving MAPI_E_OBJECT_CHANGED from **SaveChanges**, a client can either make a copy of the object to hold the changes or make another call to **SaveChanges**, specifying FORCE_SAVE. Calling **SaveChanges** with the FORCE_SAVE flag overwrites the previous save and makes a client's changes permanent.

About Named Properties

MAPI provides a facility for assigning names to properties, for mapping these names to unique identifiers, and for making this mapping persistent. Persistent name to identifier mapping insures that property names remain valid across sessions.

Clients and service providers can define named properties for any object if the implementor of the object supports named properties. To define a named property, a client or service provider makes up a name and stores it in a **MAPINAMEID** data structure. Because names are made up of a 32-bit globally unique identifier, or GUID, and either a Unicode character string or numeric value, creators of named properties can create meaningful names without fear of duplication. Because names are unique, they can be used without regard to the value of their identifiers.

To support named properties, a service provider implements two **IMAPIProp** methods, **GetIDsFromNames** and **GetNamesFromIDs**, to translate between names and identifiers and allows its **IMAPIProp::GetProps** and **IMAPIProp::SetProps** methods to retrieve and modify properties with identifiers in the named property range. The range for named property identifiers is between 0x8000 and 0xFFFE.

Creating names for properties is one way for clients to define new properties for existing or custom message classes. Service providers can use named properties to expose unique features of their messaging systems. Yet another use for named properties is to provide an alternate way of referring to properties with identifiers below 0x8000.

About Support for Named Properties

Any object that implements the **IMAPIProp** interface can support named properties. Support for named properties is required for:

- Address book providers that allow entries from other providers to be copied into their containers.
- Message store providers that can be used to create arbitrary message types.

Named property support is optional for all other service providers. Service providers that do support named properties implement name-to-identifier mapping in two **IMAPIProp** methods, **GetNamesFromIDs** and **GetIDsFromNames**. Clients call **GetNamesFromIDs** to retrieve the corresponding names for one or more property identifiers in the over 0x8000 range and **GetIDsFromNames** to either create or retrieve the identifiers for one or more names.

Service providers that do not support named properties must:

- Fail calls to **IMAPIProp::SetProps** to set properties with identifiers of 0x8000 or greater by returning MAPI_E_UNEXPECTED_ID in the **SPropProblem** array.
- Return MAPI_E_NO_SUPPORT from the **GetNamesFromIDs** and **GetIDsFromNames** methods.

About Property Names and Property Sets

The name of every named property has two parts:

- A globally unique identifier, or GUID, that specifies a property set.
- A Unicode character string or 32-bit numeric value.

Names of named properties are described using a **MAPINAMEID** structure. This structure contains a property set member, a member for specifying the name in either numeric or string format, and a member for identifying which format is used. Because the property set is part of the property's name, it is not optional. MAPI has defined several property sets for use by clients and service providers, but if an existing property set is inappropriate, a new property set can be defined. Clients and service providers can define their own property sets by calling UUIDGEN.EXE or the **CoCreateGUID** function to retrieve a unique property set identifier. **CoCreateGUID** is in the 32-bit DLL, OLE32.DLL, and the 16-bit DLL, COMPOBJ.DLL.Typically these property sets are created for custom client applications.

MAPI's property sets are represented by the following constants:

PS_MAPI
PS_PUBLIC_STRINGS
PS_ROUTING_EMAIL_ADDRESSES
PS_ROUTING_ADDRTYPE
PS_ROUTING_SEARCH_KEY
PS_ROUTING_DISPLAY_NAME
PS_ROUTING_ENTRYID

The PS_MAPI property set is reserved; it is used by service providers to generate names for properties with identifiers below the named property range. The PS_PUBLIC_STRINGS property set is used by clients for named properties of IPM messages and is the default property set for the OLE Messaging Library Fields collection. One of the specific uses of the PS_PUBLIC_STRINGS property set is to map the summary properties of IPM.Document messages. Because named properties in the PS_PUBLIC_STRINGS property set appear in a client's user interface, non-visible messages such as those that belong to the IPC message class should avoid creating named properties with this property set. Instead, they should create properties in the message class-specific range, 0x6800 through 0x7FFF.

The other property sets hold named properties describing recipients that are typically members of a routing list. Containing the same type of information as the properties that are associated with recipient list properties, properties in these property sets are understood by gateways to require mapping for a target messaging system. Because there are five types of information for describing properties, MAPI has defined five different property sets. A client sending a message that must include an address and address type for its routing list members assigns a named property for each member in the PS_ROUTNG_EMAIL_ADDRESSES and PS_ROUTING_ADDRTYPE property sets. This insures that the address and address type remain viable when sent to a foreign messaging system.

About Mappings and Mapping Signatures

When a service provider supports named properties, each set of identifier and name pairs is referred to as a mapping. Service providers can support one mapping or several. That is, one message store provider, for example, can implement the **GetIDsFromNames** and **GetNamesFromIDs** methods for all of its message, folder, and message store objects to work with a single list of names and their corresponding identifiers. Another message store provider might have one list for every folder and the messages contained within it or implement a unique list for every message and every folder. Message store providers that use a unique mapping for every message must not allow named properties to appear in their folder contents tables, because for a given property name, the property identifier will differ from message to message. MAPI recommends that providers keep it simple and operate with a single list for all of their objects including tables.

For every mapping, service providers must supply a mapping signature. A mapping signature is a binary value, usually a GUID, that uniquely identifies a set of property identifiers and their corresponding names. Mapping signatures are stored in an object's PR_MAPPING_SIGNATURE property. Service providers must change the value for their PR_MAPPING_SIGNATURE property whenever a change is made to the mapping that it represents. For example, PR_MAPPING_SIGNATURE must be updated if a new identifier is assigned to a name or a new name and identifier pair is added.

Clients working with the named properties of objects use the objects' PR_MAPPING_SIGNATURE properties in comparison and copy operations. To compare named property identifiers belonging to two objects, clients not using mapping signatures must call **GetNamesFromIDs** on both objects to retrieve the names for each of the identifiers. Using the mapping signatures of objects can render this call unnecessary. When two objects have the same value for their PR_MAPPING_SIGNATURE properties, they use the same mapping. Identifiers that use the same mapping can be compared directly. Service providers that implement **IMAPIProp::CopyTo** and **CopyProps** can also take advantage of an object's mapping signature. When copying named properties between objects, service providers can avoid the conversion step when the source and destination objects have the same mapping signature.

About IMAPIProp::GetIDsFromNames

Clients can call the **IMAPIProp::GetIDsFromNames** method to:

- Create identifiers for new names.
- Retrieve identifiers for specific names.
- Retrieve identifiers for all named properties that are included in the object's mapping.

To create a new identifier and name pair, clients pass the MAPI_CREATE flag to **GetIDsFromNames** and a **MAPINAMEID** structure that describes the name. When clients do not pass this flag, the set of property identifiers that have already been associated with the specified list of names is returned. To retrieve identifiers for particular names, clients pass the names to **GetIDsFromNames** in an array of **MAPINAMEID** structures. To retrieve all identifiers in the object's mapping, clients do not specify any names.

Because property types are not returned from **GetIDsFromNames** along with the property identifiers, clients must combine the identifiers that are returned with an appropriate type to form property tags usable in other method calls.

About IMAPIProp::GetNamesFromIDs

Clients call the **IMAPIProp::GetNamesFromIDs** method to:

- Retrieve names for specific property identifiers in a specific property set.
- Retrieve names for specific property identifiers in any property set.
- Retrieve names for all named properties that are included in the object's mapping.

To retrieve all of the named properties for an object, clients must first call the object's **IMAPIProp::GetPropList** method and then pass the returned identifiers that are above the 0x8000 range to **GetNamesFromIDs**.

GetNamesFromIDs includes three parameters:

- A property set
- An array of property tags
- A flag that indicates the format for the returned names

Both the property set and property tag array can contain valid values or be set to NULL. The flag can be set to MAPI_NO_IDs to request that only names stored as Unicode strings be returned or MAPI_NO_STRINGS to request that only names stored as numeric identifiers be returned.

Service providers implement **GetNamesFromIDs** as described in the following table depending on the values for the property tag array and property set parameters.

Property tag array	**Property set**	**Result**
One or more valid identifiers	Valid GUID	Property set ignored; returns all names that map to the identifiers regardless of the property set
One or more valid identifiers	NULL	Returns all names that map to the identifiers
NULL	NULL	Returns names for all named properties
NULL	Valid GUID	Undefined

When **GetNamesFromIDs** receives a valid property set without a valid property tag array, service providers can choose one of the following paths to take in their **GetNamesFromIDs** implementations:

- Ignore the property set and return the names for the identifiers in the property tag array.
- Return the names for only the identifiers in the property tag array that belong to the specified property set.
- Fail the call, returning MAPI_E_INVALID_PARAMETER.

Clients using an implementation of **GetNamesFromIDs** that takes the middle path, returning names from the specified property set, should be aware of the approach used when the property set is PS_PUBLIC_STRINGS. **GetNamesFromIDs** returns all names that were ever created regardless of whether the object or any other object supported by the service provider actually stores a property under each of the identifiers associated with the public strings.

For example, a client could use code similar to the following code to retrieve the names for all of an object's named properties:

```
LPSPropTagArray FAR *    lppPropTags = NULL;
LPGUID                   lpPropSetGuid = NULL;
```

```
ULONG FAR *               lpcPropNames;
LPMAPINAMEID FAR * FAR * lpppPropNames;

lpMAPIProp->GetNamesFromIDs (lppPropTags,
                             lpPropSetGuid,
                             0,
                             lpcPropNames,
                             lpppPropNames);
```

To request all names from the PS_PUBLIC_STRINGS property set, a client would replace the NULL in the property set parameter to PS_PUBLIC_STRINGS as follows:

```
LPSPropTagArray FAR *    lppPropTags = NULL;
LPGUID                   lpPropSetGuid = &PS_PUBLIC_STRINGS;
ULONG FAR *               lpcPropNames;
LPMAPINAMEID FAR * FAR * lpppPropNames;

lpMAPIProp->GetNamesFromIDs (lppPropTags,
                             lpPropSetGuid,
                             0,
                             lpcPropNames,
                             lpppPropNames);
```

Handling Named Property Errors

When a request is made to **IMAPIProp::GetIDsFromNames** or **IMAPIProp::GetNamesFromIDs** that is too large for the implementor to handle, the error value MAPI_E_TOO_BIG is returned. Callers must divide their request into several requests, calling the appropriate method in a loop.

When a call results in partial success, such as when the request is for names that map to specific identifiers and one or more names can not be found, **GetNamesFromIDs** returns MAPI_W_ERRORS_RETURNED and places PT_ERROR in the property type for the missing property in the property tag array.

Sometimes a client makes a call to **GetNamesFromIDs** that results in no properties being returned, such as when there are no properties in a specified property set or when all named properties are of a type excluded by the flags. Clients can expect service providers to:

- Return S_OK.
- Set the contents of the property tag array pointer to a newly allocated property tag array with its **cValues** member set to zero.
- Set the contents of the **MAPINAMEID** structure array to NULL.
- Set the contents of the count of **MAPINAMEID** structures to zero.

About Transmitting and Copying Named Properties

Whenever named properties are sent, moved, or copied, the name remains constant but the identifier must change to adhere to the mapping of the destination object. The only exception to this rule is when the source and destination have the same mapping signature, making re-mapping unnecessary.

It is the responsibility of the transport provider to re-map the names of transmitted named properties to appropriate identifiers that work at the destination. The sending transport provider cannot know what the correct mapping is at the destination; it must transmit the names and rely on the receiving transport provider to map them to identifiers that work. MAPI's implementation of TNEF handles the re-mapping of named properties for transport providers. Transport providers can either handle the re-mapping manually or use the TNEF implementation.

A similar re-mapping of named properties must occur when these properties are copied between message stores. However, because message store providers can retrieve the name to identifier mapping of the destination, they can re-map the properties right away and not have to rely on the destination message store.

Tables

A MAPI table is an object that is used for looking at a summary view of the properties of other MAPI objects. MAPI tables are structured in a row and column format with each column representing a property and each row representing the object owning the property. One of the properties usually included in each row is an identifier that can be used to open and modify the object. Because rows contain property values, retrieving a row from a table is similar to getting a set of properties directly from the object represented by the row. Both operations result in the return of a property value array. The main difference is in the handling of long string and binary properties. For inclusion in a table, some service providers truncate these properties to 255 bytes. When retrieved directly from the object, the full value is always available.

Tables are usually implemented by the implementor of the objects presented in the rows. For example, a message store provider implements messages and also implements the tables that display information about those messages. An address book provider implements address book containers and the hierarchy table that shows their organization. MAPI implements several different tables, some for use by client applications, some by service providers, and some by both.

The following illustration shows one of the frequent uses of a table in MAPI: to display the contents of a folder. On the right is a display of two messages as might appear in a typical messaging client application. The display contains four pieces of information about each message: the sender, the recipient, the subject, and the message text. Each piece of information is represented by a message property.

On the left is a typical view of the contents table for these messages. The view represents how this user sees the underlying data of the entire table. The view has two rows, one row for each message. Each row has three columns, one column for each message property included in the table. The columns are as follows:

PR_SENDER_NAME	Sender name
PR_ORIGINAL_DELIVERY_TIME	Date and time when the message was sent
PR_SUBJECT	Message subject line

Notice that the set of properties displayed in the message are not the same as the set of columns displayed in the table. The implementor of the table, in this case a message store provider, supplies a default set of columns in a default order. The client can modify this column set, requesting additional columns or rejecting default ones, and ask that they be ordered in a specific way. The client can also order the rows, sorting them according to the value of one or more columns.

{ewc msdncd, EWGraphic, groupx832 0 /a "MAPI_54.WMF"}

All tables implement the **IMAPITable : IUnknown** interface. IMAPITable gives clients and service providers a read-only view of the underlying data of a table. It allows them to view and change the presentation of the data in a table's rows and columns. Multiple users can access the same data concurrently through **IMAPITable**.

Types of Tables

There are many different types of tables, each differentiated by the information that it presents. Tables enable client applications and service providers to readily access and manipulate the important properties of many types of objects. Some tables, such as contents tables, provide an alternate way of accessing sets of properties for MAPI objects. A client can retrieve the subject of a message, its PR_SUBJECT property, either directly from the object by calling its **GetProps** method or through the contents table. Other types of tables, such as attachment tables, provide the only way to access the set of properties for an object. A client can access the PR_ATTACH_METHOD property, for example, only by retrieving the attachment table which includes it as one of its columns.

A table view can be static or dynamic. With a static table view, changes to the underlying data do not cause the view to be updated. Once the view has been instantiated, it does not change. Users of static tables can be informed of changes to table data through notifications. A dynamic table view always reflects the underlying data.

All tables have a default column set, or a set of columns that a table user expects to find in a table that has not yet been affected by a **SetColumns** call. Static or dynamic additions to a default column set can be made. Some table implementors include additional optional columns in every table view they create. Dynamic additions of columns can occur following a client request; not all table implementors support this type of column set modification.

The following table lists the MAPI tables and includes the typical implementor and the typical user for each.

Table	**Implementors**	**Users**
Attachment	Message store providers	Clients and transport providers
Contents	Message store and address book providers	Clients
Display	MAPI and service providers	MAPI and service providers
Hierarchy	Message store and address book providers	Clients
Message service	MAPI	Clients
Message store	MAPI	Clients
One-off	Address book providers	MAPI
Outgoing queue	Message store providers	MAPI spooler
Profile	MAPI	Clients
Provider	MAPI	Clients
Receive folder	Message store providers	Clients
Recipient	Message store providers	Clients and transport providers
Status	MAPI	Clients

About Attachment Tables

An attachment table describes the attachment objects that are associated with a submitted message or a message under composition. Attachment tables are implemented by message store providers and used by client applications, which call the **IMessage::GetAttachmentTable** method to retrieve a pointer to the table object. Some message store providers also support access through the **IMAPIProp::OpenProperty** method, where the caller specifies the PR_MESSAGE_ATTACHMENTS property.

The following properties make up the required column set in attachment tables:

PR_ATTACH_NUM
PR_INSTANCE_KEY
PR_RECORD_KEY
PR_RENDERING_POSITION

All of the properties except for PR_INSTANCE_KEY are attachment object properties. PR_INSTANCE_KEY is specific to tables and uniquely identifies a particular row in a table.

PR_ATTACH_NUM is a nontransmittable property that contains a value for uniquely identifying an attachment within a message. PR_ATTACH_NUM is often used as an index into the rows of the table. While an attachment table is open, the value of PR_ATTACH_NUM remains constant.

PR_RECORD_KEY is a common property for many objects and is used in attachment tables to identify an attachment. Unlike PR_ATTACH_NUM, PR_RECORD_KEY has the same scope as a long term entry identifier; it remains available and valid even after the message is closed and reopened.

PR_RENDERING_POSITION is an offset in characters that identifies where a client should display the attachment within a message. The first character of PR_BODY has offset 0. Not all attachments use PR_RENDERING_POSITION. Applications may assign a value of 0xFFFFFFFF to PR_RENDERING_POSITION, indicating that the attachment should not be rendered within the message body. When sorting the attachment table by rendering position, message stores should treat this as a signed value (PT_LONG). That is, an attachment whose rendering position is 0xFFFFFFFF should sort before an attachment whose rendering position is 1.

Message store providers are free to add other columns to their attachment tables. The following properties are often added because they are easy to compute or retrieve:

PR_ATTACH_ENCODING	PR_ATTACH_EXTENSION
PR_ATTACH_FILENAME	PR_ATTACH_LONG_FILENAME
PR_ATTACH_PATHNAME	PR_ATTACH_LONG_PATHNAME
PR_ATTACH_METHOD	PR_ATTACH_TAG
PR_CREATION_TIME	PR_ATTACH_TRANSPORT_NAME
PR_DISPLAY_NAME	PR_LAST_MODIFICATION_TIME

The attachment table is dynamic, so it can change while it is open. That is, if a client creates or deletes an attachment or modifies one of the properties in the column set, the changes will be reflected in the table. However, only attachments that have been saved are included in the table.

Message store providers are not required to support sorting. If sorting is not supported, the table must be presented sorted on PR_RENDERING_POSITION.

About Contents Tables

A contents table lists summary information about messaging user or distribution list objects in address book containers or about messages in folders. Address book providers implement contents tables for each of their containers. Both message store and remote transport providers implement contents tables for folders.

Folders support two types of contents tables:

- Standard
- Associated

Standard contents tables contain standard messages, or messages that are transmitted and are visible to the user. Associated contents tables contain hidden information with a specific purpose for the client. For example, some clients might want to store an alternate representation of a standard message as associated information. To retrieve a contents table with only this hidden information, clients specify the MAPI_ASSOCIATED flag on the **GetContentsTable** call.

There are two methods that a client or provider can call on a folder or container to access its contents table and table implementors must support them both:

- **IMAPIContainer::GetContentsTable**
- **IMAPIProp::OpenProperty** with PR_CONTAINER_CONTENTS or PR_FOLDER_ASSOCIATED_CONTENTS (folders only) specified as the property tag and IID_IMAPITable as the interface identifier

The call to **IMAPIProp::OpenProperty** involves accessing the contents table by opening its corresponding property, PR_CONTAINER_CONTENTS for standard contents tables, and PR_FOLDER_ASSOCIATED_CONTENTS for associated contents tables. Although neither or these properties can be retrieved through a folder or container's **IMAPIProp::GetProps** method, they are included in the property tag array that is returned by the **IMAPIProp::GetPropList** method.

PR_CONTAINER_CONTENTS can also be used to include or exclude a contents table from a copy operation. If a client specifies PR_CONTAINER_CONTENTS in the *lpExcludeProps* parameter for **IMAPIProp::CopyTo** in a copy operation, the new folder or container will not support the contents table of the original folder or container.

GetContentsTable accepts as input several flags that specify preferences. The MAPI_DEFERRED_ERRORS flag indicates to the service provider that any errors encountered during the table creation do not need to be reported until some later time.

All contents tables can support categorization, or sorting by categories. Whereas many folder contents tables can sort by categories, few address book container tables can. Address book providers usually do not support categorization on the contents tables of their containers.

Address book container and standard folder contents tables have a lengthy list of required columns, or columns that clients can expect to be available through the **IMAPITable** pointer returned from **GetContentsTable** or **OpenProperty**. Associated contents tables do not have a required column set. Clients must define their own column set for these contents tables by calling **IMAPITable::SetColumns** if the default one returned by the provider is inappropriate.

The contents tables implemented by address book, message store, and remote transport providers have different required column sets. The following table lists the required columns for each of the types of contents tables.

Required column	**Type of contents table**
PR_ADDRTYPE	Address book container tables
PR_DISPLAY_NAME	Address book container tables
PR_DISPLAY_CC	Message store folder tables

PR_DISPLAY_TO	All folder contents tables
PR_DISPLAY_TYPE	Address book container tables
PR_ENTRYID	All contents tables
PR_HASATTACH	All folder contents tables
PR_INSTANCE_KEY	All contents tables
PR_LAST_MODIFICATION_TIME	Message store folder tables
PR_MAPPING_SIGNATURE	Message store folder tables
PR_MESSAGE_CLASS	All folder contents tables
PR_MESSAGE_DOWNLOAD_TIME	Remote transport folder tables
PR_MESSAGE_FLAGS	All folder contents tables
PR_MESSAGE_SIZE	All folder contents tables
PR_MSG_STATUS	All folder contents tables
PR_OBJECT-TYPE	All contents tables
PR_PARENT_ENTRYID	Message store folder tables
PR_RECORD_KEY	Address book container and message store folder tables
PR_SENT_REPRESENTING_NAME	Remote transport folder tables
PR_STORE_ENTRYID	Message store folder tables
PR_STORE_RECORD_KEY	Message store folder tables

The entry identifier available with each row can either be a short- or long-term entry identifier, depending on the table. Short-term entry identifiers are typically used in situations where performance is an issue. Either type of entry identifier can be used to access the corresponding object.

Contents tables also have a set of columns that are optional, but they are commonly included by service providers in their implementations. The following table lists these optional columns.

Optional column	**Type of contents table**
PR_CLIENT_SUBMIT_TIME	Message store folder tables
PR_CONVERSATION_INDEX	Message store folder tables
PR_CONVERSATION_KEY	Message store folder tables
PR_EMAIL_ADDRESS	Address book container tables
PR_IMPORTANCE	All folder contents tables
PR_MESSAGE_DELIVERY_TIME	All folder contents tables
PR_NORMALIZED_SUBJECT	All folder contents tables
PR_PRIORITY	All folder contents tables
PR_SEARCH_KEY	Address book container tables
PR_SEND_RICH_INFO_	Address book container tables
PR_SENDER_NAME	All folder contents tables
PR_SENSITIVITY	All folder contents tables
PR_SUBJECT	All folder contents tables
PR_TRANSMITTABLE_DISPLAY_NAME	Address book container tables

Message store providers must also include PR_PARENT_DISPLAY for search-result folders contents tables.

This list of properties does not include all the properties that can be included in a contents table. Clients can request other properties; it is up to the service provider as to whether the table can include them. Named properties may be added to the column set of a folder contents table only if all messages in the folder have the same mapping signature, that is, the same mapping of property names to property identifiers. Folder contents tables should support adding message class specific properties to the column set, at least if they support the creation of arbitrary messages in the folder.

About Display Tables

A display table describes how to show a specific type of dialog box – one having one or more tabbed property pages dedicated to displaying and possibly editing properties using an implementation of the **IMAPIProp** interface. The rows in a display table represent the controls, or user interface objects, that appear in the dialog box. MAPI defines many types of controls, some with static values and some with values that a user can change. Most controls can be associated with properties maintained with the **IMAPIProp** implementation. When a user changes the value of a modifiable control, the corresponding property is updated.

Creating a display table is similar to writing a program with a scripting language. Service providers can create a display table using one of the following techniques:

- Calling the **BuildDisplayTable**function.
- Including custom code that populates the display table directly using a table data object.

The **BuildDisplayTable** function combines information from display table structures with visual elements from a dialog box resource to build display table rows. The function returns a pointer to an **IMAPITable** interface implementation, and, if requested, a pointer to an **ITableData** interface implementation.

Using **BuildDisplayTable** to create a display table is straightforward and makes maintenance easier when visual elements of the display change. However, service providers that do not need to use **BuildDisplayTable** can create a display table with custom code that uses the methods of **ITableData**. For example, service providers that have an existing template structure for their property pages might prefer custom code over **BuildDisplayTable** for display table creation.

Associated with every display table is an **IMAPIProp** interface implementation. Service providers supply this implementation to maintain the property data that is presented in the dialog box. There are a variety of ways service providers can implement the property interface for their display table; these include:

- Supplying a standard **IMAPIProp** implementation.
- Supplying a wrapped **IMAPIProp** implementation that includes special processing before making the standard calls.
- Supplying an **IPropData** implementation.

The type of implementation depends on the characteristics of the data to be displayed and the responsible service provider. For example, if there is an implicit relationship between the data in two edit controls and one of the controls changes, the **IMAPIProp** implementation must change the value of the other control appropriately.

About Display Table Columns

Display tables have the following properties in their required column set:

PR_XPOS	PR_YPOS
PR_DELTAX	PR_DELTAY
PR_CONTROL_TYPE	PR_CONTROL_FLAGS
PR_CONTROL_STRUCTURE	PR_CONTROL_ID

PR_XPOS and PR_YPOS specify the X and Y coordinates of the upper left corner of the control. The horizontal units are 1/4 of the dialog base width unit; the vertical units are 1/8 of the dialog base height unit. Windows computes the current dialog base units from the height and width of the current system font. The coordinates are relative to the origin of the property page area. The size of property pages is limited to approximately 200 by 180 dialog units.

PR_DELTAX and PR_DELTAY are the width and height of the control. These are ULONG values. The width units are 1/4 of the dialog base width unit; the height units are 1/8 of the dialog base height unit. The coordinates are relative to the origin of the control.

The other four properties describe various characteristics of the control. PR_CONTROL_TYPE indicates the type of control. MAPI defines twelve types of controls, each with a different set of attributes. These attributes are desribed in the flags property, PR_CONTROL_FLAGS. Examples of attributes include whether or not a control is editable or required.

The control structure, PR_CONTROL_STRUCTURE, contains information relevant to the particular type of control. Each type of control is described with a different structure. For example, edit controls are described with the DTBLEDIT structure. DTBLEDIT structures contain members that list the number of and specific types of characters that can be placed on the control and a property tag that identifies the property whose value is to be displayed in the control. PR_CONTROL_STRUCTURE is stored as a binary property.

The control identifier, PR_CONTROL_ID, uniquely identifies the control in the dialog box described by the display table. PR_CONTROL_ID is set from the values placed in the *lpbNotif* and *cbNotif* members of the **DTCTL** structure that is used by **BuildDisplayTable** to create the display table. Because MAPI sometimes combines display tables, the PR_CONTROL_ID should always be unique. Typically, providers assign a **GUID** structure to PR_CONTROL_ID to ensure its uniqueness. The PR_CONTROL_ID property is included in the **TABLE_NOTIFICATION** structure when a display table notification is generated.

To illustrate how the rows and columns of a display table map to fields on a dialog box, consider the following example. This dialog box is typical of the type that an address book provider might display to allow users to define new Internet recipients. There are only four controls: two labels and two edit controls.

{ewc msdncd, EWGraphic, groupx832 1 /a "MAPI.BMP"}

The corresponding display table contains four rows, one for each control. The values for the columns that indicate position would be as follows:

Control	**XPOS**	**YPOS**	**DELTAX**	**DELTAY**
Label (name)	14	18	49	8
Edit (name)	76	16	89	12
Label (address)	14	42	50	8
Edit (address)	76	40	89	12
Checkbox	14	64	90	12

This next table suggests appropriate values for the control type, flags, and structure columns for each of the rows. Notice that the value for the label controls appears in memory directly following the structure.

Control	Type	Flags	Structure
Label (name)	DTCT_LABEL	0	{sizeof(DTBLLABEL), 0} "Display name:"
Edit (name)	DTCT_EDIT	DT_EDITABLE | DT_REQUIRED	{sizeof(DTBLEDIT), 0, 80, PR_DISPLAY_NAME}
Label (address)	DTCT_LABEL	0	{sizeof(DTBLLABEL), 0} "E-mail address:"
Edit (address)	DTCT_EDIT	DT_EDITABLE | DT_REQUIRED	{sizeof(DTBLEDIT), 0, 80, PR_EMAIL_ADDRESS}
Checkbox	DTCT_CHECKBOX	DT_EDITABLE	PR_SEND_RICH_INFO

Note The **OK**, **Cancel**, and **Help** buttons are not included in the display table. The user interface can add context to a dialog box by adding controls not in the display table.

About Display Table Scenarios

Display tables are used to show dialog boxes:

- By address book providers to view and edit detailed information about messaging users.
- By address book providers to view and edit advanced search dialog boxes.
- By address book providers to view and edit templates for new users.
- By transport providers to view and edit message option information.
- By all providers to view and edit configuration information.

In some of these scenarios, a service provider creates a display table and passes it and a pointer to an **IMAPIProp** interface implementation to MAPI. In other scenarios, the display table is obtained from the **IMAPIProp** implementation when MAPI calls its **OpenProp** method and requests PR_DETAILS_TABLE.

MAPI is responsible for displaying the dialog box to the user with property values made available either through the **IMAPIProp** implementation or the display table. As the user works with the dialog box, changing values in the controls, MAPI calls **IMAPIProp::SetProps** to save a changed control if the control's DT_SET_IMMEDIATE flag is set. For controls without the DT_SET_IMMEDIATE flag set, changes to properties are saved when the user dismisses the dialog box by selecting the **OK** or **Apply Now** button. When either of these buttons or the **Cancel** button is selected, MAPI removes the dialog box from view.

For example, in the first scenario, a user of a client application selects an entry from the address book and requests a detailed display. Processing proceeds as follows:

1. The client calls **IAddrBook::Details**, implemented by MAPI.
2. MAPI calls the address book provider's **IABLogon::OpenEntry** method to open the messaging user object that represents the selected entry.
3. MAPI calls the messaging user's **IMAPIProp::OpenProperty** method to retrieve the PR_DETAILS_TABLE property, the display table for the details dialog box.
4. MAPI displays the dialog box, handling the user's interaction with the information, and removes it when the user has finished.

In another more complicated scenario, the user of a client application requests the message options dialog box, a collection of properties supported by the transport provider. To make these properties available, a transport provider must return an **OPTIONDATA** structure with information about its supported options when MAPI calls its **IXPLogon::RegisterOptions** method and implement a function that conforms to the **OPTIONCALLBACK** prototype. The **OPTIONCALLBACK** function handles the data stored in the **OPTIONDATA** structure.

The client calls **IMAPISession::MessageOptions** to display a message options dialog box. MAPI calls the transport provider's **OPTIONCALLBACK** function which returns an **IMAPIProp** interface implementation. The **OpenProperty** method of this implementation supports the PR_DETAILS_TABLE property. When the **OPTIONCALLBACK** function returns, MAPI calls **OpenProperty** to open PR_DETAILS_TABLE and retrieve the display table describing the message options dialog box.

About Display Table Structures

There are two display table structures that relate directly to the **BuildDisplayTable** function; they have no meaning outside the context of **BuildDisplayTable**. Therefore, these structures are only used by service providers that call **BuildDisplayTable** to create their display tables.

There are three types of display table structures:

- A structure to describe a single tabbed property page (**DTPAGE**)
- A structure to describe a single control (**DTCTL**)
- A structure to describe the data of a single control, one of the controls in the display table (such as **DTBLEDIT**, **DTBLLBX**)

When **BuildDisplayTable** is called to create a display table, the service provider passes one or more **DTPAGE** structures as an input parameter. The **DTPAGE** structures contain an array of **DTCTL** structures and a count of the number of **DTCTL** structures in the array. There is one structure for every control to appear in the dialog box. **DTPAGE** structures also have a character string that represents the name of a corresponding help file and dialog resource.

Each **DTCTL** structure in a **DTPAGE** structure contains data that is used to set properties for the control:

- Control type for setting PR_CONTROL_TYPE
- Control flags for setting PR_CONTROL_FLAGS
- Notification data for setting PR_CONTROL_ID
- Control structure for setting PR_CONTROL_STRUCTURE

DTCTL structures also contain a resource identifier and, for edit and combo box controls, a character filter.

The control structure member of a **DTCTL** structure describes the data that is unique for the type of control. MAPI defines a different structure for each control type. For example, the data of an edit control is represented by a **DTBLEDIT** structure; the data of a list box by a **DTBLLBX** structure.

The relationship between these three types of display table structures is illustrated in the following diagram. The dialog box described by this display table has two controls: a label and an edit control. The **DTBLLBX** structure has a label offset member, as do several of the control structures, that describes where the character string for the label begins. Label character strings are typically placed in memory immediately following the structure.

{ewc msdncd, EWGraphic, groupx832 2 /a "MAPI.BMP"}

About the DTCTL Structure

The **DTCTL** structure describes one control of any type. Most of its members are used to set properties on the control. Every **DTCTL** structure contains the following members:

- A constant that represents the type, such as DTCT_EDIT or DTCT_CHECKBOX, and corresponds to the control's PR_CONTROL_TYPE property.
- A set of flags that describe the control's features, such as DT_SET_IMMEDIATE or DT_EDITABLE, and corresponds to the control's PR_CONTROL_FLAGS property.
- A structure that holds the data for the control, such as **DTBLEDIT** or **DTBLCHECKBOX**, and corresponds to the PR_CONTROL_STRUCTURE property.

The **lpbNotif** and **cbNotif** members relate to notification data. The **lpbNotif** member points to a structure made up of a GUID to represent the service provider and an identifier for the control. The **cbNotif** member holds the number of bytes in the structure pointed to by **lpbNotif**. These members correspond to the control's PR_CONTROL_ID property and are used to notify the user interface when the control needs updating.

The **lpszFilter** member holds a character string that describes which characters can be entered into an edit or combo box control. For other types of controls, the **lpszFilter** member can be NULL. For edit and combo box controls, it should be a regular expression that applies to a single character at a time. The same filter is applied to all characters in the control.

The format of the filter string is as follows:

Symbol	**Description**	**Example**
*****	Any character is allowed.	"*"
[]	Defines a set of characters.	**[**0123456789**]**
-	Indicates a range of characters.	[a**-**z]
~	Indicates that these characters are not allowed.	[~0-9]
\	Used to quote any of the above symbols.	[\-\\\[\]] means -, \, [, and] characters are allowed

The **ulItemID** member is a value that identifies the control in the dialog box resource. For tabbed pages, controls of type DTCT_PAGE, the **ulItemID** member is optionally used to load the component name for the page from a string resource.

The **ctl** member is a union of structures that relate to a particular type of control. If the **DTCTL** structure is describing an edit control, for example, the **ctl** member will point to an **DTBLEDIT** structure. This structure corresponds to the control's PR_CONTROL_STRUCTURE property. The union has as its first member a variable of type LPVOID to permit compile time initialization of the **DTCTL** structure.

About Control Flags

MAPI specifies several flags that are used to describe the attributes of one or more controls. These flags are grouped together and stored in the **ulCtlFlags** member of the control's **DTCTL** structure and in its PR_CONTROL_FLAGS property. Most of the control flags apply to all of the controls that allow user input; a few apply only to the edit control. Controls that do not allow user input, such as a button or a label, set 0 for their control flags.

Many of the flag values are self-explanatory. For example, when DT_REQUIRED is set for a control, it must contain a value before the dialog box is allowed to be dismissed. Either the service provider can supply a value through its **IMAPIProp** implementation or the user can enter one. DT_EDITABLE indicates that the value for the control can be modified. DT_MULTILINE allows the value for an edit control to span multiple lines.

Some control flags are not so obvious in their meaning. When a control sets the DT_SET_IMMEDIATE flag, any changes to its value take affect as soon as the user moves to a new control. MAPI makes a single call to the property interface's **IMAPIProp::SetProps** method for the control's property. This is different from the default behavior, which is to postpone having changes to control values take effect until after the user selects the **OK** button or dismisses the dialog box. The DT_SET_IMMEDIATE flag is often used in combination with display table notifications.

The following table lists the types of controls and all of the flag values that can be set for each type.

Control	**Valid control flags**
Button	Must be zero
Check box	DT_EDITABLE, DT_SET_IMMEDIATE
Combo box	DT_EDITABLE, DT_REQUIRED, DT_SET_IMMEDIATE
Drop-down list box	DT_EDITABLE, DT_SET_IMMEDIATE
Edit	DT_ACCEPT_DBCS, DT_MULTILINE, DT_EDITABLE, DT_PASSWORD_EDIT, DT_REQUIRED, DT_SET_IMMEDIATE
Group box	Must be zero
Label	Must be zero
List box	Must be zero
Multivalue drop-down list box	Must be zero
Multivalue list box	Must be zero
Tabbed page	Must be zero
Radio button	Must be zero

About Display Table Notifications

Notifications on a display table are sent by the service provider responsible for creating the display table to MAPI. MAPI registers for these notifications by calling a display table's **IMAPITable::Advise** method and specifying the table modified event.

Each display table notification contains a **TABLE_NOTIFICATION** structure. Only the **ulTableEvent** and the **propIndex** members of this structure are significant; the other members are ignored. The **ulTableEvent** member is set to TABLE_ROW_MODIFIED and the **propIndex** member is set to the value of the PR_CONTROL_ID column in the corresponding row. MAPI responds to the notification by calling the **IMAPIProp::GetProps** method for the property displayed in the control and by displaying the new value.

Display table notifications can be used by a service provider to coordinate changes to related controls on the dialog box. For example, if the property interface implementation needs to refresh one or more fields on the dialog box, perhaps in response to another control that has set the DT_SET_IMMEDIATE flag in its PR_CONTROL_FLAGS property, it can generate a display table notification. A display table notification can alert the property interface implementation that the value of one or more controls needs to be re-read due to a change being made or an external event occurring.

Service providers can issue display table notifications in one of two ways:

- With a call to **ITableData::HrNotify**, if the display table was built with a table data object.
- With its own code, if the display table was built with the provider's **IMAPITable** implementation.

MAPI responds to display table notifications when necessary by re-reading a control's value from the property interface implementation. The following table describes the details surrounding how MAPI handles notifications for specific types of controls.

Control	Action
Button	Calls **IMAPIProp::OpenProperty** to retrieve the control object via the property represented by the **ulPRControl** member of the **DTBLBUTTON** structure if the call had failed previously. Calls the control object's **IMAPIControl::GetState** to determine whether the button should be enabled and enables or disables the button accordingly.
Check box	Rereads the value for the **ulPRPropertyName** member .
Combo box	Reopens the table associated with the **ulPRTableName** member of the **DTBLCOMBOBOX** structure. Rereads all of the rows including the value for the **ulPRPropertyName** member.
Drop-down list box	Reopens the table associated with the **ulPRTableName** member of the **DTBLDDLBX** structure and rereads all of the rows. Calls **IMAPIProp::GetProps** to retrieve the values for the properties stored in the **ulPRDisplayProperty** and the

	ulPRSetProperty members.
Edit	Rereads the property and redisplays.
Group box	Ignores the notification.
Label	Ignores the notification.
Multiple selection list box	If one of the columns is an entry identifier, refreshes the list box. The corresponding object is not closed or reloaded.
Single selection list box	Reads the set property, trying to identify it.
Multivalued list box	Rereads the property and re-populates the list box.
Tabbed page	There are no notifications for this control; everything is static.
Radio button	Rereads the property that is associated with the button and is stored in the **ulPropTag** member of the **DTBLRADIOBUTTON** structure and makes the appropriate selection with the controls.

About Types of Controls

There are many different types of controls, none unique to MAPI. However, MAPI defines its own structures that are used in conjunction with **BuildDisplayTable** to describe the unique set of data involved with each control.

The following table describes the structures that describe each type of control.

Control structure	Description
DTBLBUTTON	Describes a button control.
DTBLCHECKBOX	Describes a check box control.
DTBLCOMBOBOX	Describes a combo box control.
DTBLDDLBX	Describes a drop down list box control.
DTBLEDIT	Describes an edit control.
DTBLGROUPBOX	Describes a group box control.
DTBLLABEL	Describes a label control.
DTBLLBX	Describes a list box control.
DTBLMVDDLBOX	Describes a multi-value drop down list box control.
DTBLMVLISTBOX	Describes a multi-value list box control.
DTBLPAGE	Describes a tabbed page control.
DTBLRADIOBUTTON	Describes a radio button control.

About Button Controls

Button controls allow users to begin an operation. Typically clicking a button causes a modal dialog box to be displayed or a programmatic task to be invoked. Service providers can implement anything through a button control. If the button is supposed to perform a task based on the values of other controls, those controls must have set the DT_SET_IMMEDIATE flag.

The control structure that describes a button control, **DTBLBUTTON**, has three members:

- A label offset
- A flags value
- A property tag for a property of type PT_OBJECT, the **ulPRControl** member

The label offset is the position in memory of the character string that is displayed on the button. Service providers can add an ampersand character (&) to indicate a Windows accelerator in the button label. Typing an accelerator has the same effect as pushing the button.

The flags value indicates whether or not the button label is in Unicode or ANSI format.

The property tag describes an object property that, when opened with the **IMAPIProp::OpenProperty** method, returns a pointer to a control object. Implementing a control object – an object that supports the **IMAPIControl** interface – is a way to extend the MAPI feature set and define the operation or task that occurs when the button is clicked. **IMAPIControl** supplies two methods for manipulating buttons: **GetState** to disable or enable buttons and **Activate** to handle button presses.

About Check Box Controls

A check box is a control that reflects one of two states: enabled or disabled. A check box control is described by the **DTBLCHECKBOX** which has three members:

- A label offset
- A flags value
- A property tag for a property of type PT_BOOLEAN, the **ulPRPropertyName** member

The label offset is the position in memory of the character string that is displayed with the check box and the flags value indicates whether or not the check box label is in Unicode or ANSI format.

The property tag describes a boolean property whose value is manipulated by changing the state of the check box. When the check box is first displayed, MAPI calls the **IMAPIProp** implementation's **GetProps** method to retrieve a set of default properties. If one of the properties maps to the property tag in the **DTBLCHECKBOX** structure, the value for that property is displayed as the check box's initial value.

Check box controls can be modifiable, allowing a user to change their states. Modifiable check boxes set the DT_EDITABLE flag in the **ulCtlFlags** member of their **DTCTL** structure and in their PR_CONTROL_FLAGS property. When a check box changes its state, MAPI calls **IMAPIProp::SetProps** to set the property identified in the property tag member of the **DTBLCHECKBOX** structure to the new state.

For example, an address book provider might include a modifiable check box control in its configuration dialog box to manipulate the setting of a recipient's PR_SEND_RICH_INFO property. When the user selects the check box, MAPI sets this property to TRUE. When the check box is unselected, the property is set to FALSE.

About Combo Box Controls

A combo box control consists of a list box and a selection field. The list box presents the information from which a user can select and the selection field displays the current selection. The selection field is an edit control that can also be used to enter text not already in the list.

The following dialog box includes an example of a combo box.

{ewc msdncd, EWGraphic, groupx832 3 /a "MAPI.BMP"}

Combo box controls share some characteristics with edit controls and some characteristics with list box controls. They are described with a **DTBLCOMBOBOX** structure which contains the following members:

- A character filter
- A flags value
- A character count
- A property tag for a property of type PT_TSTRING, the **ulPRPropertyName** member
- A property tag for a property of type PT_OBJECT, the **ulPRTableName** member

The character filter is a numeric value that is an offset from the beginning of the **DTBLCOMBOBOX** structure to a character string that describes restrictions, if any, to the characters that can be entered into the combo box's edit control.

The flags value indicates whether or not the information displayed in the list box and entered in the edit control is in Unicode or ANSI format.

The character count indicates the maximum number of characters that can be entered into the combo box's edit control.

The two property tag members work together to coordinate the list box display with the edit control. When MAPI first displays the combo box, it calls the **IMAPIProp** implementation's **OpenProperty** method to retrieve the table represented by the **ulPRTableName** member of the **DTBLCOMBOBOX** structure. This table has one column, a column that contains values for the property represented by the **ulPRPropertyName** member. Therefore, this column must be of the same type as the **ulPRPropertyName** property and both columns must be character strings.

The values for the column are displayed in the list box portion of the combo box. Therefore, PR_NULL is not a valid property tag for **ulPRPropertyName**. When a user either selects one of the rows or enters new data into the edit box, the **ulPRPropertyName** property is set to the selected or entered value.

To display an initial value for the edit control, MAPI calls **IMAPIProp::GetProps** to retrieve the property values for the display table. If one of the retrieved properties matches the property represented by the **ulPRPropertyName** member, its value becomes the initial value.

About Drop-Down List Box Controls

A drop-down list box control is a list box that is displayed as a single item until the user elects to expand it. The following illustration provides an example of a drop-down list box.

{ewc msdncd, EWGraphic, groupx832 4 /a "MAPI.BMP"}

A drop-down list box is described with a **DTBLDDLBX** structure. **DTBLDDLBX** structures contain the following members:

- A flags value which is reserved and must be zero
- A property tag for a property of type PT_OBJECT, the **ulPRTableName** member
- A property tag for a property of type PT_TSTRING, the **ulPRDisplayProperty** member
- A property tag for any type of property, the **ulPRSetProperty** member

The three properties identified by the property tags work together to display the information in the list box and set a related property. The **ulPRTableName** member is a table object that is accessed through a call to **IMAPIProp::OpenProperty**. The table has two columns: one column for the property identified by the **ulPRDisplayProperty** member and the other for the property identified by the **ulPRSetProperty** member.

The **ulPRDisplayProperty** property drives the list box display. When a user selects one of the values from the display, MAPI calls **IMAPIProp::SetProps** to set the corresponding property as identified by the **ulPRSetProperty** member, meaning the property in the same row as the selected display property. The **ulPRSetProperty** member cannot be set to PR_NULL.

An initial value is displayed in the list box if MAPI has retrieved the property represented by the **ulPRSetProperty** member through a call to **IMAPIProp::GetProps** and located a row in the table with the value for the **ulPRSetProperty** member. The initial displayed value is the contents of the **ulPRDisplayProperty** column from that row that matches the property in the **ulPRDisplayProperty** member of the structure. The value returned by **GetProps** for the property identified by the **ulPRDisplayProperty** member becomes the initial value that is shown when the list box is first displayed.

About Edit Controls

Edit controls are areas that contain alphanumeric information. Almost all dialog boxes have at least one edit control. Edit controls can be modifiable by a user or read-only.

Edit controls can also be single line or multiline. Multiline edit controls typically have a scroll bar associated with them as is shown in the following illustration.

{ewc msdncd, EWGraphic, groupx832 5 /a "MAPI.BMP"}

The **DTBLEDIT** structure is used to describe an edit control. **DTBLEDIT** structures have the following members:

- A character filter
- A flags value
- A character count
- A property tag for a property of type PT_TSTRING, the **ulPropTag** member

The character filter is a numeric value that is an offset from the beginning of the **DTBLEDIT** structure to a character string that describes restrictions, if any, to the characters that can be entered into the edit control.

The flags value indicates whether or not the information entered in the edit control is in Unicode or ANSI format.

The character count indicates the maximum number of characters that can be entered into the edit control.

The **ulPropTag** member identifies the string property whose data is displayed and edited in the edit control.

About Group Box Controls

Group box controls are used to visually associate other controls in the dialog box. The highlighting technique involves surrounding the other controls by a box, as is illustrated in the following dialog box.

{ewc msdncd, EWGraphic, groupx832 6 /a "MAPI.BMP"}

Group box controls are described with the **DTBLGROUPBOX** structure which has the following two members:

- A label offset
- A flags value

The label offset is the position in memory of the character string that accompanies the group box. If displayed, the label appears on the top, left-hand side of the box.

The flags value indicates whether or not the group box label is in Unicode or ANSI format.

About Label Controls

Label controls are text displayed with another type of control to add meaning to that control. For example, most edit controls are placed next to labels to inform the user of the type of information to be entered. Some controls, such as group boxes and radio buttons, hold their own labels.

Label controls are described with the **DTBLLABEL** structure which has the following two members:

- A label offset
- A flags value

The label offset is the position in memory of the character string that is displayed as the label, the value of the control. The label can include a Windows accelerator, identified as the character following the ampersand (&). Typing the accelerator character puts the focus in the first non-label, non-button control following this label in the display table.

The flags value indicates whether or not the label is in Unicode or ANSI format.

There is no support for multiline labels. Showing multiple lines requires multiple labels.

It is not possible to use a label as a read-only edit control. The distinction is that an edit control can be selected and copied while a label cannot.

About List Box Controls

List box controls are used to show multiple items and allow a user to select one or more of the items. List box controls are described with the **DTBLLBX** structure which contains the following three members:

- A flags value
- A property tag for a property of any type, the **ulPRSetProperty** member
- A property tag for a property of type PT_OBJECT, the **ulPRTableName** member

The flags value indicates whether or not a horizontal or vertical scroll bar should be displayed with the list box. The default is to be have types of scroll bars appear if necessary. Service providers can set MAPI_NO_HBAR to suppress a horizontal scroll bar and MAPI_NO_VBAR to suppress a vertical scroll bar.

The two property tag members work together to display values in the list box and set corresponding properties when an item in the list box is selected. When MAPI first displays the list box, it calls the **IMAPIProp** implementation's **OpenProperty** method to retrieve the table identified in the **ulPRTableName** member. The number of columns in the table depend on the value of the **ulPRSetProperty** member. If **ulPRSetProperty** is set to PR_NULL, the list box is a multiple selection list box based on an object that contains recipients, such as an address book container, a recipient table for a message, or a distribution list contents table.

A table for a multiple selection list box must include the following columns:

PR_DISPLAY_NAME
PR_ENTRYID
PR_INSTANCE_KEY

PR_DISPLAY_TYPE and a maximum of five other multivalued string properties can also be displayed with the three required columns.

If the **ulPRSetProperty** member is not set to PR_NULL, the list box is a single selection list box. The initial value of **ulPRSetProperty** determines the initially selected row. When a user selects one of the rows, the **ulPRSetProperty** is set to the selected value and this value is written back to the property interface implementation with a call to **IMAPIProp::SetProps**.

About Multivalue List Box Controls

Multivalue list box controls are read-only lists of items. There are two types of multivalue list boxes: standard and drop-down. Whereas standard list boxes are displayed immediately, drop-down dialog boxes are displayed when a user clicks on a scroll bar.

The two structures have only two members: **ulMVPropTag** and a flags value. The flags value that is reserved and must be zero.

The properties that are displayed must be of type PT_MV_TSTRING and the data comes from the **ulMVPropTag** member of either the **DTBLMVLISTBOX** structure, for multivalue list boxes, or **DTBLMVDDLBOX** structures, for multivalue drop-down list boxes. There is no need to read from the property interface. Also, because users are not allowed to make selections from these types of list boxes, data is also not written to the property interface.

About Radio Button Controls

Radio button controls are buttons that are associated with a group. Only one button in the group can be checked; setting a button causes all of the other buttons in the group to be unset. The following dialog box includes an example of a group of three radio button controls.

{ewc msdncd, EWGraphic, groupx832 7 /a "MAPI.BMP"}

Radio buttons are described with a **DTBLRADIOBUTTON** structure. **DTBLRADIOBUTTON** structures contain the following members:

- A label offset
- A flags value
- The number of buttons in the group
- A property tag for a property of type PT_LONG, the **ulPropTag** member
- A return value identifying the selected button

The label offset is the position in memory of the character string that can be displayed with the radio button and the flags value indicates whether or not this label is in Unicode or ANSI format.

The button count is the number of radio buttons in the group. The structures for the other radio buttons in the group must be in subsequent rows in the display table. Each of these structures should have the same value for its button count.

About Tabbed Page Controls

Tabbed page controls are used to separate several related property sheets, dialog boxes that show configuration, message, or recipient options. By clicking the tab, the user can switch from one sheet to another.

The following dialog box includes four tabbed page controls: one control for each type of property relating to a particular address type.

{ewc msdncd, EWGraphic, groupx832 8 /a "MAPI.BMP"}

Tabbed page controls are described with the **DTBLPAGE** structure which contains the following members:

- A label offset
- A flags value
- A component string
- A context identifier

The label offset is the position in memory of the character string that specifies the text to appear on the tab and the flags value indicates whether or not this label is in Unicode or ANSI format.

The component string and context identifier provide information about whether or not extended help is available for the tabbed page and if it is available, how to access it. The component string maps to the help file; the context identifier maps to the initial help topic. If the context identifier is zero and the component string is NULL, extended help is not available.

About Hierarchy Tables

A hierarchy table displays information about the folders in a message store or the containers in an address book container. Each row of a hierarchy table contains a set of columns containing information about one folder or address book container. Hierarchy tables are used by message store providers to show a tree of folders and subfolders and by address book providers to show a tree of containers within the address book. Containers that cannot hold subcontainers, as indicated by the absence of the AB_SUBCONTAINERS flag in their PR_CONTAINER_FLAGS property, do not implement a hierarchy table.

Your address book provider can support table notifications on its hierarchy tables to alert client applications to changes affecting the containers if those containers support change.

There are two methods that a client or provider can call on a folder or container to access its hierarchy table and table implementors must support them both:

- The **IMAPIContainer::GetHierarchyTable** method.
- The **IMAPIProp::OpenProperty** method passing PR_CONTAINER_HIERARCHY as the property tag and IID_IMAPITable as the interface identifier.

The call to **IMAPIProp::OpenProperty** involves accessing the hierarchy table by opening its corresponding property, PR_CONTAINER_HIERARCHY. Although PR_CONTAINER_HIERARCHY cannot be retrieved through a folder or container's **IMAPIProp::GetProps** method, it is included in the property tag array that is returned by the **IMAPIProp::GetPropList** method.

PR_CONTAINER_HIERARCHY can also be used to include or exclude a hierarchy table from a copy operation. If a client specifies PR_CONTAINER_HIERARCHY in the *lpExcludeProps* parameter for **IMAPIProp::CopyTo** in a copy operation, the new folder or container will not support the hierarchy table of the original folder or container.

The following properties make up the required column set in hierarchy tables:

PR_COMMENT	PR_DEPTH
PR_DISPLAY_NAME	PR_DISPLAY_TYPE
PR_ENTRYID	PR_OBJECT_TYPE
PR_STATUS	

PR_DISPLAY_NAME contains the name for the container or folder that should appear in the display of the hierarchy.

PR_ENTRYID is the entry identifier associated with this container or folder. It is expected to be a long-term entry identifier. Clients and MAPI can pass this entry identifier to **OpenEntry** to open the container or folder and view its contents by calling **IMAPIContainer::GetContentsTable**.

PR_DEPTH is the hierarchy level; it indicates the level of indentation for this container or folder with zero being the top level.

PR_OBJECT_TYPE is always set to MAPI_ABCONT for address book hierarchy tables and MAPI_FOLDER for folder hierarchy tables.

PR_DISPLAY_TYPE identifies the type of container or folder in the row. It is mainly used for display purposes, to differentiate visually between types of containers or folders. It is up to the address book provider to supply icons to represent display type; MAPI does not supply defaults.

PR_CONTAINER_FLAGS indicates various attributes about this container in the hierarchy and is used to distinguish one container from another.

In addition to these required columns, address book hierarchy tables must include the following properties:

PR_CONTAINER_FLAGS
PR_INSTANCE_KEY

An optional property for address book hierarchy tables is the PR_AB_PROVIDER_ID property.

Message-store hierarchy tables include these properties in their required column set:

PR_FOLDER_TYPE
PR_SUBFOLDERS

An address book provider's **IMAPITable** implementation for the hierarchy table of its containers must support the following methods because they are required by MAPI's integrated address book:

QueryColumns	**QueryPosition**
SeekRow	**SeekRowApprox**
FindRow	**Restrict**
CreateBookmark	**FreeBookmark**
QueryRows	

About Message Service Tables

A message service table lists information about the message services in the current profile. There is one message service table for every MAPI session. The message service table is a static table, meaning that once it has been created, it will not reflect any changes in underlying data of the table. The message service table is implemented by MAPI and used by special purpose client applications that provide configuration support. Clients retrieve a message-service table object by calling the **IMsgServiceAdmin::GetMsgServiceTable** method.

The following properties make up the required column set in message service tables:

PR_DISPLAY_NAME	PR_INSTANCE_KEY
PR_RESOURCE_FLAGS	PR_SERVICE_DLL_NAME
PR_SERVICE_ENTRY_NAME	PR_SERVICE_NAME
PR_SERVICE_SUPPORT_FILES	PR_SERVICE_UID

PR_DISPLAY_NAME is the displayable name for the message service and the default sort key column.

PR_INSTANCE_KEY serves as the index column for the table, uniquely identifying a row.

PR_RESOURCE_FLAGS describes the message service's capabilities.

PR_SERVICE_DLL_NAME is the name of the DLL file that contains the message service implementation.

PR_SERVICE_ENTRY_NAME is the name of the message service's entry point function that conforms to the MSGSERVICEENTRY prototype.

PR_SERVICE_NAME is a required entry in the [Services] section in MAPISVC.INF. The value for this property will never be changed or localized. PR_SERVICE_NAME can be used to programmatically identify the message service.

PR_SERVICE_SUPPORT_FILES is a list of files that must be installed with the message service.

PR_SERVICE_UID is a unique identifier for the message service.

About Message Store Tables

The message store table lists information about message store providers in the current profile. Clients can use this table, for example, to locate all instances of a particular provider or to locate a specific message store. Clients access the message store table by calling the **IMAPISession::GetMsgStoresTable** method.

MAPI implements one message store table for each session. The message store table is dynamic; it always reflects what is in the current profile. If the user of a client application edits the profile, changing the default message store, for example, the values of the PR_DEFAULT_STORE properties for the affected message stores are immediately updated.

The following properties make up the required column set in message stores tables:

PR_DEFAULT_STORE	PR_DISPLAY_NAME
PR_ENTRYID	PR_INSTANCE_KEY
PR_MDB_PROVIDER	PR_OBJECT_TYPE
PR_PROVIDER_DISPLAY	PR_RECORD_KEY
PR_RESOURCE_FLAGS	PR_RESOURCE_TYPE

About One-Off Tables

A one-off table lists the templates that an address book provider supports for creating new recipients. One-off tables are implemented by address book providers, individual address book containers, and by MAPI, and can be persistent or temporary.

Note Do not confuse the templates in one-off tables with template identifiers; while their purposes are similar, their code constructs are nothing alike. Templates are used to create recipients of a particular type while template identifiers are used to bind the data of one recipient belonging to a host provider with code to support another recipient belonging to a foreign provider.

Clients create new recipients under two conditions:

- To add to the recipient list of an outgoing message
- To add to one of the containers in the address book

In both scenarios, an address book provider is asked to return a one-off table. Address book providers can implement either a single one-off table to be used in both scenarios or a unique one-off table for each scenario.

When the recipient will be included with an outgoing message, MAPI calls the address book provider's **IABLogon::GetOneOffTable** method to retrieve its one-off table. The one-off table includes templates which allow a user to enter information resulting in the creation of a recipient with a valid address. MAPI registers for notifications on this table, keeping it open so that changes can be reflected back to the user. MAPI releases the table only when its subsystem or address book status objects' **IMAPIStatus::ValidateState** method is called.

When the recipient will be added to a container, MAPI makes a different call, invoking the container's **IMAPIProp::OpenProperty** method to retrieve its PR_CREATE_TEMPLATES property. The set of templates included in this one-off table represents the types of recipients that can be added to the container. For example, mail servers often expose one container for every gateway that is installed so that each container only holds addresses specific to the corresponding gateway.

MAPI provides a one-off table that includes its own templates as well as templates from each of the address book providers in the session. MAPI provides a generic template that can be used to create a new recipient for any address type, assuming that the user knows its format. Address book providers use this one-off table by calling **IMAPISupport::GetOneOffTable**. Each of the templates included in MAPI's one-off table results in the creation of recipients with valid recipient addresses.

Address book providers typically supply one template for every address type they support. However, support for templates is not required. Address book providers that do not allow the creation of new addresses can return MAPI_E_NO_SUPPORT when MAPI calls to request a one-off table. Address book providers that do allow new address creation but do not supply any templates can call **IMAPISupport::GetOneOffTable** to use the templates listed in MAPI's one-off table.

The following properties make up the required column set in one-off tables:

PR_ADDRTYPE	PR_DEPTH
PR_DISPLAY_NAME	PR_DISPLAY_TYPE
PR_ENTRYID	PR_INSTANCE_KEY
PR_SELECTABLE	

The PR_ADDRTYPE column indicates the type of address that can be associated with the new recipient created with the template.

The PR_DISPLAY_NAME and PR_DISPLAY_TYPE columns associate data with the new recipient. PR_DISPLAY_NAME contains a character string that identifies the new recipient and PR_DISPLAY_TYPE contains a constant that identifies the type of icon to be displayed with the row.

Templates for messaging users have their PR_DISPLAY_TYPE column set to DT_MAILUSER; templates for distribution lists have their PR_DISPLAY_TYPE column set to DT_DISTLIST.

The PR_ENTRYID column is the entry identifier of the template to be used to create a new recipient. This entry identifier can be passed to future **IAddrBook::NewEntry**, **IAddrBook::OpenEntry**, and **IABContainer::CreateEntry** calls. Containers set the PR_ENTRYID column of their row for the default messaging user template to PR_DEF_CREATE_MAILUSER and the PR_ENTRYID column of their row for the default distribution list template to PR_DEF_CREATE_DL.

The PR_DEPTH column is used to support the hierarchical display of the entries in a one-off table by indicating the level of indention for the template. Although one-off tables can be displayed either as a flat list or a hierarchical display, the latter is preferable and address book providers should support it by setting the PR_DEPTH column for each row appropriately. PR_DEPTH is zero-based; rows with a value of 0 in their PR_DEPTH column are not indented. The higher the value of PR_DEPTH, the more the row is indented. For example, rows with PR_DEPTH set to 1 are indented one tab while rows with PR_DEPTH set to 3 are indented three tabs.

The PR_SELECTABLE column is used to indicate whether or not a row in the table represents a template that can be selected and used to create a new recipient. Although most rows in a one-off table do represent templates, providers can include non-template rows. For example, a provider might want to organize the one-off table by template type, including a category row that appears in the display but is not used for recipient creation. The following dialog box illustrates a one-off table with five rows, four of which represent templates and have their PR_SELECTABLE column set to TRUE. The one non-template row, indicated by the text "Special Address Types," has its PR_SELECTABLE column set to FALSE.

{ewc msdncd, EWGraphic, groupx832 9 /a "MAPI.BMP"}

About Outgoing Queue Tables

An outgoing queue table lists all of the outgoing messages for a message store. Message store providers implement outgoing queue tables for the MAPI spooler to use. Stores that do not support the sending or receiving of messages need not implement this table.

To retrieve a pointer to an outgoing queue table, the MAPI spooler calls the **IMsgStore::GetOutgoingQueue** method.

There is a requirement that messages be preprocessed and submitted to the transport provider in the same order as they were sent by the client application. The MAPI spooler is designed to accept messages from the message store in ascending order of submission time. Because of this requirement, there can be some delay before some messages appear in the outgoing queue table.

Message stores should either allow sorting on the outgoing queue table so that the MAPI spooler can sort the messages by submission time, or the default sort order should be by ascending submission time.

The outgoing queue table must send notifications when the contents of the queue changes.

The following properties make up the required column set in outgoing queue tables:

PR_CLIENT_SUBMIT_TIME	PR_DISPLAY_BCC
PR_DISPLAY_CC	PR_DISPLAY_TO
PR_ENTRYID	PR_MESSAGE_FLAGS
PR_MESSAGE_SIZE	PR_PRIORITY
PR_SENDER_NAME	PR_SUBJECT
PR_SUBMIT_FLAGS	

About Profile Tables

A profile table lists information about all profiles associated with a particular client application. MAPI implements the profile table for use by clients. A pointer to the table is returned when clients call the **IProfAdmin::GetProfileTable** method.

There is one profile table for every session. The profile table is a static table, meaning that if profiles are added or deleted while the table is open, the table will not change. Profiles that have been marked for deletion are not included in the profile table.

As with most table implementations, if **GetProfileTable** is called and there are no profiles available to the client, a table object is returned with zero rows in the table.

The following properties make up the required column set in profile tables:

PR_DEFAULT_PROFILE
PR_DISPLAY_NAME

About Provider Tables

A provider table lists information about service providers for client use. There are two different provider tables. The **IMsgServiceAdmin::GetProviderTable** method creates a MAPI table object that holds all of the providers for the current profile. The **IProviderAdmin::GetProviderTable** method creates a table that stores all of the service providers for a message service.

The following properties make up the required column set in provider tables:

PR_INSTANCE_KEY	PR_DISPLAY_NAME
PR_PROVIDER_DISPLAY	PR_PROVIDER_DLL_NAME
PR_PROVIDER_ORDINAL	PR_PROVIDER_UID
PR_RESOURCE_FLAGS	PR_RESOURCE_TYPE
PR_SERVICE_NAME	PR_SERVICE_UID

The provider table can be used to display the current transport order or to change it. To display the current order, build a restriction to retrieve only those rows with the PR_RESOURCE_TYPE property set to MAPI_TRANSPORT_PROVIDER. Then use PR_PROVIDER_ORDINAL as a sort key to sort the table and retrieve all of the rows with either the **IMAPITable::QueryRows** method or the API function **HrQueryAllRows**.

To change the transport order, apply the same restriction and retrieve the rows. Then create an array of values from the PR_PROVIDER_UID property that represents the unique identifiers for the tranport providers. When the identifiers are in the desired order, pass them to the **IMsgServiceAdmin::MsgServiceTransportOrder** method.

Once a provider table has been returned to a client, it will not reflect changes to the environment, such as the addition or deletion of a provider.

About Receive Folder Tables

A receive folder table lists information for all of the folders designated as receive folders for a message store. A receive folder is a folder where incoming messages of a particular message class are placed. Message store providers implement receive folder tables and client applications use them by making a call to the **IMsgStore::GetReceiveFolderTable** method.

The following properties make up the required column set in receive folder tables:

PR_ENTRYID
PR_MESSAGE_CLASS
PR_RECORD_KEY

About Recipient Tables

The recipient table lists information about all of the recipients for a message. Message store providers implement recipient tables and client applications use them. Clients access a recipient table by making a call to the **IMessage::GetRecipientTable** method, or if the message store provider supports it, to the **IMAPIProp::OpenProperty** method. Clients access recipient tables with **OpenProperty** by specifying PR_MESSAGE_RECIPIENTS for the property tag and IID_IMAPITable for the interface identifier. Changes to a recipient table can be made by calling the **IMessage::ModifyRecipients** method.

Recipient tables have a different column set depending on whether the message has been submitted. The following properties make up the required column set in recipient tables:

PR_DISPLAY_NAME
PR_RECIPIENT_TYPE
PR_ROWID

The optional properties are:

PR_DISPLAY_TYPE
PR_ENTRYID
PR_SPOOLER_STATUS
PR_OBJECT_TYPE

Submitted messages should include these additional properties in their required column set:

PR_ADDRTYPE
PR_RESPONSIBILITY

Depending on a provider's implementation, additional columns, such as PR_SENDER_NAME and ENTRYID, might be in the table.

Any message store provider that supports message transmission, as indicated by the STORE_SUBMIT_OK bit being set in the provider's PR_STORE_SUPPORT_MASK property, should offer support for a particular set of restrictions in a recipient table implementation. The And, Or, Exists, and Property restrictions are among the types of restrictions that should be available to recipient table users. Only the equal and not equal operators need to be supported on the Property restriction. These restrictions must work with the following properties:

PR_ADDRTYPE
PR_EMAIL_ADDRESS
PR_RECIPIENT_TYPE
PR_RESPONSIBILITY
PR_SPOOLER_STATUS
PR_TRANSPORT_STATUS

About Status Tables

The status table lists information relating to the state of the current session. There is one status table for every session that includes information provided by MAPI and by service providers. MAPI provides data for three rows: a row for the MAPI subsystem, a row for the MAPI spooler, and a row for the integrated address book. Because transport providers are required to supply status information to the status table, there is one row for every active transport provider. Address book and message store providers can choose whether or not to support the status table.

Because each row is provided by a different resource, the set of columns can vary from row to row. There is a set of columns that every status object is required to supply and a set of columns that MAPI supplies. A service provider can add to these sets to expose provider-specific properties. For example, message store providers might add PR_STORE_RECORD_KEY to supply clients with the identifier of their message store. Clients must have advance knowledge of the existence of this extra information to be able to use it.

The following table lists the properties that must be in every status table row. The implementor of the status object provides some of the properties; others are computed by MAPI.

Properties provided by status object	Properties provided by MAPI
PR_DISPLAY_NAME	PR_PROVIDER_DLL_NAME
PR_STATUS_CODE	PR_RESOURCE_FLAGS
PR_RESOURCE_METHODS	PR_RESOURCE_TYPE

If the status object provides an identity, it should set PR_IDENTITY_DISPLAY, PR_IDENTITY_ENTRYID, and PR_IDENTITY_SEARCH_KEY, and include these properties in the table.

Four properties are computed by MAPI for each status table row:

PR_ENTRYID	PR_INSTANCE_KEY
PR_OBJECT_TYPE	PR_ROWID

MAPI assigns an entry identifier to the status row to enable clients to open the corresponding status object. A row identifier is also assigned to identify the row in the table as is an instance key to identify the data in the status object. The PR_OBJECT_TYPE property is set to MAPI_STATUS.

To access the status table, clients call the **IMAPISession::GetStatusTable** method. This call should not be made immediately upon startup. This is because **GetStatusTable** has to wait for the MAPI spooler to initialize the transport providers, an operation that is postponed until after the client has finished its logon. **GetStatusTable** is a relatively fast call after the MAPI spooler has completed its start up processing.

Status table information can be used in a variety of ways, such as to access a status object, to determine whether a client is running in a connected or offline mode, and to monitor a provider's state. For example, clients can open a specific service provider's status object by passing the value of the PR_ENTRYID property to the **IMAPISession::OpenEntry** method. The status object supports the **IMAPIStatus** interface, an interface that contains methods to change a service provider password, flush the message queue, display a configuration property sheet, or confirm status with a provider directly. Status table information can also be used to build a dialog box to inform clients of progress during a lengthy operation.

Service providers who do support the status table use the **IMAPISupport::ModifyStatusRow** method to create and update their row. Whenever a change occurs to their row, all advise sink objects registered to receive status table notifications must be notified. Service providers can call the **IMAPISupport::Notify** method if they are using MAPI's notification utility or call each advise sink's **IMAPIAdviseSink::OnNotify** method directly.

Common Table Operations

Working with a MAPI table is a little like working with a relational database table. In every row, the same columns appear in the same order. A user can limit the number of rows and columns in the view and specify their order. Rows can be retrieved one at a time or in groups. A cursor that keeps track of the current position can be moved to a specific place in the table.

To work with tables, client applications use the read-only interface, **IMAPITable : IUnknown**, whereas service providers, depending on whether they own the data that the table is based on, can use either **IMAPITable** or **ITableData : IUnknown**. The operations defined in these interfaces can be divided into three groups: operations that affect single columns, operations that affect single rows, and operations that affect the entire table. Column operations include specifying the properties to be included in the column set and the order in which they should be included. Row operations include data retrieval and the maintenance operations: adding, deleting, and modifying a single row or rows. Operations of a global nature include event notification, searching, and sorting.

Defining Column Sets

The column set of a table is the group of properties that are displayed in the table view. Table implementors have a default column set that they return to a user if the user has made no attempt to change it. A user can request that the default set be changed to include other columns, if the table implementor supports them, or different columns. A table's current column set is either the set of columns established when the user made a call to change the default set, or if no call was made, it is the default set.

To change a column set, a client or service provider calls the **IMAPITable::SetColumns** method or the API function, **HrAddColumnsEx**. When a column set is altered with **SetColumns**, columns from the default set can be removed and new columns added, and the order in which these columns appear can be changed. **HrAddColumnsEx** operates slightly differently in that it always adds columns to the beginning of the column set and never removes columns. Whatever was in the column set before the call will still be there after the call, positioned after the newly added columns. Whereas **SetColumns** is used on most tables to provide their users with a customized view of the data, **HrAddColumnsEx** is used mostly to add to the column set of recipient tables.

SetColumns specifies the columns that are returned with each row and the order of these columns within the row. The success of this operation is apparent only after a table user makes a separate call to retrieve the table data. It is then that the table implementor reports an error if not all of the columns in the property tag array passed to **SetColumns** are supported. No error is returned from **SetColumns** due to unsupported columns.

SetColumns is typically a synchronous operation. However, callers can request that it operate asynchronously by setting the *ulFlags* parameter to TBL_ASYNC, but service providers are not required to honor the request. If a service provider that does not support asynchronous column set definition receives a **SetColumns** call with the TBL_ASYNC flag set, the provider should perform the operation synchronously. With asynchronous column modification, **SetColumns** will return before the operation completes and the caller must use the **IMAPITable::GetStatus** method to determine when completion occurs.

Another flag, TBL_BATCH, allows callers to specify that the table implementor can defer evaluating the results of the operation until a later time. Whenever possible, callers should set this flag because batched operation improves performance.

It is often convenient for callers to reserve some columns in the retrieved row set for values to be added later. Callers do this by placing PR_NULL at the desired positions in the property tag array passed to **SetColumns**; the table will then pass back PR_NULL at those positions in all rows retrieved with **QueryRows**.

Working with Large Columns

Columns with string or binary property data can be large, possibly many thousands of bytes long. Because including one or more columns with hundreds of bytes in a view is often impractical, MAPI enables table implementors to truncate the value, most often to 255 bytes and less often to 510 bytes. Table implementors should whenever possible include the full value of a property in a table column. The recommended alternative is to include only the first 255 bytes.

Clients cannot know ahead of time whether or not a table they are using truncates large columns. They should assume that a column represents a truncated property if the length of the column is either 255 or 510 bytes. Clients can retrieve the full value of a truncated column if necessary from the object directly by callings the object's **IMAPIProp::GetProps** method.

Clients building restrictions with large properties should be aware that it is up to the table implementor as to how these restrictions operate. Some table implementors allow restrictions that are built with a truncated column to be based on the truncated size while others base it on the entire value.

Using Row and Row Set Data Structures

A row represents a MAPI object of a particular type and is defined using the **SRow** data structure. The **SRow** structure contains an array of property values, or **SPropValue** structures, that describe the column set and a count of the number of structures in the array. Rows are sometimes combined into row sets, or arrays of **SRow** structures. Row sets are defined using the **SRowSet** data structure.

The following illustration shows the relationship between an **SRow** and an **SRowSet** data structure.

{ewc msdncd, EWGraphic, groupx832 10 /a "MAPI_17.WMF"}

Table Positioning

The current position within a table is always indicated by a cursor. There is one cursor for each view of a table; its value is set by the table's implementor. When a client or service provider using the table makes a call to change the position of the table, the value of the cursor is reset. Clients and service providers change the position of a table by:

- Using a bookmark, a resource that indicates a particular location in a table.
- Using a fractional value.
- Setting a restriction.
- Retrieving rows.

Setting a bookmark makes it possible to return to the associated position at a later time, a feature that can significantly improve the performance of table operations. MAPI defines three standard bookmarks:

BOOKMARK_CURRENT	Points to the present row in a table.
BOOKMARK_BEGINNING	Points to the first row in a table.
BOOKMARK_END	Points to the last row in a table.

Table implementors are required to support these standard bookmarks but can support others. However, there are not unlimited numbers of bookmark resources available, and clients and service providers must free them as soon as possible.

To establish a bookmark at the current position, call **IMAPITable::CreateBookmark**. Occasionally there will be insufficient memory available to allocate the new bookmark, causing **CreateBookmark** to return the MAPI_E_UNABLE_TO_COMPLETE error value. To free a bookmark, call **IMAPITable::FreeBookmark**.

To move the cursor to the position identified by an established bookmark, call **IMAPITable::SeekRow**. **SeekRow** establishes a new value for the BOOKMARK_CURRENT position. **SeekRow** can be used, for example, to position a table ten rows from the current position or to start over at the beginning. Clients or service providers can seek to the current, beginning, or end of a table, or any other position that is associated with a predefined bookmark. They can move in either a forward or backward direction and limit the operation to a specified number of rows. As a rule, callers should seek through no more than 50 rows with **SeekRow**; **IMAPITable::SeekRowApprox** should be used with larger numbers of rows.

To move the cursor to an approximate position based on a fraction, call **SeekRowApprox.** **SeekRowApprox** moves to the row that represents a particular percentage of rows in relation to the beginning of the table. This percentage is specified by the caller in the *ulNumerator* and *ulDenominator* parameters. With **SeekRowApprox**, a client can move to the row that is one third of the rows from the beginning of the table.

To determine a table's fractional position, call **QueryPosition**. **QueryPosition** informs the table's user of an approximate position by setting a fractional value based on the number of rows in the table and the row number. Service providers are not required to make this value accurate; it is meant to be an approximation. In fact, because accurate implementations can be expensive to invoke, especially on categorized tables, it is recommended that service providers avoid complete accuracy.

Clients and service providers use the **SeekRowApprox** and **QueryPosition** methods to implement a scroll bar and inform a user of the current position.

To position a table to the first row that matches the criteria established in a restriction, call the **FindRow** method. Starting with the row represented by a particular bookmark, **FindRow** searches in either a forward or backward direction to locate a row that matches the criteria specified in the restriction. **FindRow** can be useful for implementing a scroll bar that is based on character strings, rather than fractional values. For example, a client can call MAPI's implementation of **FindRow** when searching through the integrated address book to enable a user, by entering one or more characters, to locate the first name that begins with the specified characters.

Retrieving Rows

To retrieve a row or rows, client applications or service providers obtain the underlying data of the columns specified for the table view and modify the current position. Clients and service providers can retrieve a single row, a group of rows, or all of the rows in a forward or backward direction starting from the beginning of the table, the end of the table, or a position in the middle.

Often one of the columns in a returned row holds the PR_ENTRYID property, an entry identifier that can be used to open the object that corresponds to the row. This entry identifier is a short term entry identifier, one that that does not persist past the lifetime of the table. However, because some service providers support only one type of entry identifier, entry identifiers returned from their tables will have long term longevity.

Clients and service providers can make one of the following calls to retrieve rows:

IMAPITable::QueryRows	Retrieves a specified number of rows in a table.
HrQueryAllRows	Retrieves all of the rows in a table.
ITableData::HrQueryRow	Retrieves a row in a table identified by the value of a particular column.

Using ITableData::HrQueryRow

To retrieve a row that has a value that matches the value of an index column, service providers and client applications call the **ITableData::HrQueryRow** method. Most tables use the PR_INSTANCE_KEY property as their index column. The returned row contains columns for all of the properties that belong to the object associated with the row.

Using IMAPITable::QueryRows

To retrieve any number of rows in either a forward or a backward direction, service providers and clients call the **IMAPITable::QueryRows** method. **QueryRows** retrieves rows beginning with the current cursor position. The actual rows and columns that are returned and the order in which they are returned depend on whether a successful call to the **IMAPITable** method **SetColumns**, **Restrict**, or **SortTable** has been made prior to the **QueryRows** call. If none of these calls has been made, **QueryRows** returns all of the rows in the table. Each row contains the default column set in default order.

IMAPITable method	Effect on output of QueryRows
SetColumns	The particular columns in each row and the order of those columns match the set of columns specified in the call to **SetColumns**.
Restrict	Only the rows that match the criteria specified in the restriction are returned.
SortTable	The returned rows are in the order specified in the **SSortOrderSet** structure passed to **SortTable**.

It is possible that a client or service provider calling **SetColumns** requests an unsupported column to be included in the column set. When this occurs, **QueryRows** places the special property type PT_ERROR in the property tag and the error value MAPI_E_NOT_FOUND in the property value for the unsupported column.

With **IMAPITable::QueryRows**, the caller can specify the number of rows to retrieve. Rows are always returned in the current sorted order. A positive row count number means that rows are returned from the current position. A negative row count number means to back up from the current position the given number of rows and to start reading the rows in a forward direction. If the TBL_NOADVANCE flag is set, the cursor is located at the first row returned. If the TBL_NOADVANCE flag is not set, the cursor position ends up at the starting position.

Implementors of **QueryRows** can treat the row count as a request rather than a requirement. It is acceptable for **QueryRows** to return anywhere from zero rows if there are no rows in the direction of the query to the number requested.

When implementing **QueryRows**, return only the rows that the user will see when rows are requested from a categorized table view. This allows the caller to make valid assumptions about the scope of the data and avoid extra work.

When calling **QueryRows**, be aware that the timing of asynchronous operations such as notifications and calls to the **IMAPITable** methods **SetColumns**, **Restrict**, or **SortTable** can affect the set of rows that are returned. The returned set of rows might not accurately represent the underlying data. That is, if an asynchronous operation such as **Restrict** is pending, **QueryRows** returns the MAPI_E_BUSY value. This is not the case if table notifications are pending. For example, if a message has been deleted from a folder, but a notification causing the folder's contents table to be updated has not yet been processed, the message will still appear in the table view. Table users should wait for notifications to arrive before updating their view of the data.

Using HrQueryAllRows

HrQueryAllRows is a helper function that combines the functionality of the **IMAPITable** methods **SetColumns**, **Restrict**, **SortTable**, and **QueryRows**. With one call, a client application or service provider can retrieve all of the rows in a table that have a particular set of columns, match a particular set of criteria, and are sorted in a particular order. Specifying a row and column preference is optional, however, and if no preferences are indicated, **HrQueryAllRows** returns all of the underlying data for the table. The table's default column set and default sort order are used. **HrQueryAllRows** always begins processing at the beginning of the table.

With **HrQueryAllRows**, the caller can specify a maximum number of rows to retrieve. **HrQueryAllRows** always attempts to return all of the rows that it can. When no maximum number is specified, this is all of the rows in the table. When a maximum number is specified, only that number is returned. Setting a maximum limit allows **HrQueryAllRows** to fail when the specified number has been reached, saving memory and processing time.

Memory Issues in Row Retrieval

An important issue connected with retrieving rows from a table is memory usage. Lack of available memory can cause the **IMAPITable::QueryRows** method to fail to return the requested number of rows and the **HrQueryAllRows** function to fail to return the specified maximum number or rows. Deciding how to retrieve rows from a table depends on whether or not the table can be expected to fit in memory, and if it cannot, if failure is acceptable. Since it is not always easy to determine the amount of data that will fit into memory at one time, MAPI provides some basic guidelines for a client application or service provider to follow. Bear in mind that there are always exceptions, based on the particular table implementation and how the underlying data is stored.

The following guidelines can be used to evaluate table memory usage:

- Clients that can tolerate occasional working set memory usage in the megabyte range and are written for 32-bit platforms can assume they will have no problems reading an entire table into memory. Clients written to run on 16-bit platforms should be more conservative.
- Restrictions have an affect on a table's usage of memory. A severely restricted table with an extensive number of rows, such as a contents table, can be expected to fit into memory while an unrestricted large table usually cannot. However, large unrestricted tables might fit in memory if the table user has some extended knowledge about the table or control over its data.
- Several of the tables owned by MAPI such as the status, profile, message service, provider, and message store tables, will usually fit in memory. These are generally small tables. However, there are exceptions. For example, a server-based profile provider might generate a larger profile table that will not be able to fit.

To retrieve all of the rows from a table that will fit into memory with no problems, call **HrQueryAllRows**, setting the maximum number of rows to zero.

To retrieve all of the rows from a table that might or might not fit into memory, generating an error, call **HrQueryAllRows** specifying a maximum number of rows. The maximum number of rows should be set to a number greater than the minimum number of rows that are needed. That is, if a client must access at least 50 rows from a 300 row table, the maximum number of rows should be set to at least 51.

To retrieve all of the rows from a table that is not expected to fit into memory, call **QueryRows** in a loop with a relatively small row count, as the following code sample illustrates:

```
HRESULT      hr;
LPSRowSet    pRows = NULL;
LONG         irow;
LONG              cAsk = 50;                  // adjust this value

while ((hr = pTable->QueryRows(cAsk, 0, &pRows)) == hrSuccess
        && pRows->cRows != 0)
{
    for (irow = 0; irow < prows->cRows; ++irow)
    {
        // process the row...
    }
    FreeProws(pRows);
    pRows = NULL;
}
if (hr)
{
    // handle the error...
}
```

When this loop completes and all the rows in the table have been processed and *cRows* is zero, the position of the cursor will usually be at the bottom of the table.

Determining a Table's End

A common error is to assume that the end of the table has been reached when:

- **QueryRows** has been called in a loop, with the end of the loop determined by the row count returned by **IMAPITable::GetRowCount**. The count that **GetRowCount** returns does not always represent the exact number of rows in the table; it is an approximate count.
- **QueryRows** has been called with a fixed number of rows and assume that if less rows are returned that the end of the table has been reached. It is not until **QueryRows** returns a row set with a row count equal to zero that there are no more rows to retrieve.

Note The only time that a caller can assume that the cursor is positioned at the end of the table for a positive row count or at the beginning of the table for a negative row count is when the value S_OK and zero rows are returned. The value MAPI_E_NOT_FOUND is never returned.

Tips for Better Table Performance

Because many of the table operations can be time-consuming and there is no way to show progress, table users should be aware of a few techniques that can improve performance. The first technique involves the ordering of **IMAPITable** calls. A client or service provider that is planning to define a column set, build a restriction, and define a sort order before retrieving rows should perform these tasks in exactly this order. First, the call to define a column set should be made. Specifying the restriction should be next followed by the call to sort the table. Performing these tasks in this order limits the number of rows and columns that will be sorted, thereby improving performance.

The second technique involves delaying the work of a particular method until a later time. Setting the TBL_BATCH flag on a method allows the table implementor to collect several calls before acting on any one of them. Rather than make potentially many calls to a remote server, a table implementor can make one, performing all of the operations at one time. The results of the operations are not evaluated until they are needed. For example, when a client calls **IMAPITable::Restrict** to specify a restriction with the TBL_BATCH flag set, the restriction need not go into effect until the client calls **IMAPITable::QueryRows** to retrieve the data. This allows the table implementator to combine the work of two calls into one. Table users that take advantage of the TBL_BATCH flag should be aware that error handling under these conditions can be more complex.

Service providers implementing tables can lessen the time it takes to create a view by caching copies of commonly used object properties. Keeping a copy of these properties in memory saves having to retrieve them from the object each time the view must be rebuilt.

Sorting and Categorization

Sorting a table places rows in an order that makes sense for its viewer. For example, one viewer might prefer to see a contents table's messages in alphabetic order by subject so that all of the threads of a conversation are together while another viewer might want the messages to appear in order by the sender.

There are two types of sorting: standard and categorized. With standard sorting, all rows are displayed in a flat list using one or more columns as a sort key. When a table is sorted by category, again one or more columns are used as a sort key, but all rows with columns that have values that match the sort key, or category, are displayed as a group. This group contains two kinds of rows: heading rows and leaf rows. A heading row contains the column or columns designated as the category and a column for each of four computed properties: PR_CONTENT_UNREAD, PR_INSTANCE_KEY, PR_DEPTH, and PR_ROW_TYPE. A leaf row contains all of the columns in the column set minus the category columns.

Categorized tables are primarily supported by message store providers in their contents table implementations. Some address book providers will also support categorization in their contents tables.

Defining Sort Orders

A newly instantiated table is not necessarily sorted in any particular order. Client applications and service providers use two data structures to define a sort order: **SSortOrder** for specifying a sort key and **SSortOrderSet** for combining sort keys and specifying other information.

The **SSortOrder** data structure is defined as follows:

```
typedef struct _SSortOrder
{
  ULONG  ulPropTag;
  ULONG  ulOrder;
} SSortOrder, FAR * LPSSortOrder;
```

The **ulPropTag** member is the property tag for a sort key column. All table implementations allow columns in the current view to be used as a sort key. Support for sort keys defined with available columns that are not in the current view, although valuable, is optional. Available columns are those columns that are returned from **IMAPITable::QueryColumns** when the TBL_ALL_COLUMNS flag is set.

The **ulOrder** member indicates both directional order and categorization information. Rows can be sorted in either an ascending or descending sequence with all NULL entries placed last. The **ulOrder** member can be set to TABLE_ASCEND, TABLE_DESCEND, or TABLE_SORT_COMBINE, for categorized tables. The TABLE_SORT_COMBINE value indicates that the column specified in **ulPropTag** should be combined with the previous category column to form a composite category. That is, instead of categorizing on unique values of individual columns, TABLE_SORT_COMBINE allows categorization on unique values of a combination of columns. A single category could be defined, for example, to group messages received from a particular sender on a particular subject.

The **SSortOrderSet** structure is used to collect all of the sort key columns and is defined as follows:

```
typedef struct _SSortOrderSet
{
  ULONG       cSorts;
  ULONG       cCategories;
  ULONG       cExpanded;
  SSortOrder  aSort[];
} SSortOrderSet, FAR * LPSSortOrderSet;
```

The **CSorts** member specifies the number of columns in the sort key, which is the number of entries in the **SSortOrder** structure array. The **SSortOrder** structure array can contain both categorized columns and standard sort key columns.

The **cCategories** member specifies the number of columns that are designated as category columns. When there are more columns than there are categories, the first columns are used for the categorization. For example, consider the situation where **cCategories** is set to two and there are three columns defined as sort keys in the **SSortOrderSet** structure: sender name, subject, and date received. Sender name is the top level category grouping; subject is the secondary grouping. Every leaf row that has columns with values that match the two category columns is ordered by date received., so that the expanded display for such a grouping appears as shown in the following illustration:

{ewc msdncd, EWGraphic, groupx832 11 /a "MAPI_58.WMF"}

The **cExpanded** member specifies the number of categories that are initially expanded. When there are multiple categories, the table implementation begins with the first column to be designated as a category and continues in sequential order with the subsequent category columns until the number of **cCategories** has been exceeded. If there are more category columns than there are expanded columns, the category columns are collapsed. If **cExpanded** is equal to zero, only the top level heading row is available to the table user for display. If **cExpanded** is equal to one less than the number of categories, then all of the heading rows but none of the leaf rows are available. If **cExpanded** is equal to the number of categories, then the table is fully expanded.

Using SortTable

To perform the actual sorting of a table, client applications and service providers call the **IMAPITable::SortTable** method. **SortTable** accepts two input parameters: a pointer to the **SSortOrderSet** data structure containing the sort criteria and a set of flags. The flags can request that the sort operation be performed asynchronously and that evaluation of the results be deferred until they are needed.

Table sorting is not a feature that service providers are required to implement. Service providers can return the MAPI_E_NO_SUPPORT value if they do not support any level of sorting and the MAPI_E_TOO_COMPLEX value if a sort request is too difficult. For example, a service provider can return MAPI_E_TOO_COMPLEX if one of the columns specified in the **SSortOrder** array cannot be used as a sort key or if the value of the **cCategories** member is greater than zero and categorization is unsupported. Whenever **SortTable** fails, the sort order that was in effect before the failure is still in effect.

Properties for Categorized Tables

To help with categorization, a table implementor computes several properties:

PR_ROW_TYPE
PR_INSTANCE_KEY
PR_DEPTH
PR_CONTENT_UNREAD

The PR_ROW_TYPE property is a 32-bit value that indicates whether a row is a leaf row (TBL_LEAF_ROW), an expanded heading row (TBL_EXPANDED_CATEGORY), or a collapsed heading row (TBL_COLLAPSED_CATEGORY).

The PR_INSTANCE_KEY property is a binary value that uniquely identifies a row in a table view. PR_INSTANCE_KEY is a required column in most tables. If a row is included in two views, there will be two different instance keys. The instance key of a row may differ each time the table is opened, but remains constant while the table is open. Rows added while a table is in use do not reuse an instance key that was previously used.

Although uncommon, some service providers support categorization on multivalue columns. The expansion of a multivalue column logically occurs before categorization and every expanded row has a unique instance key.

The PR_DEPTH property indicates the level of a row in a categorized display. PR_DEPTH begins at 0 for the top-level heading rows, adding 1 for each subsequent heading row, and ending with the number of categories as the value of PR_DEPTH for each leaf row. For example, if there are three categories and five rows that fit a particular grouping of those three categories, the first category as displayed in the top level heading row has a PR_DEPTH of 0, the next heading row a PR_DEPTH of 1, and the third last heading row a PR_DEPTH of 2. All five of the leaf rows have their PR_DEPTH property set to 3.

The PR_CONTENT_UNREAD property is valid for contents tables only. It is a count of the number of unread messages in a category. Some client applications display the heading row of a category differently depending on the value of PR_CONTENT_UNREAD. For example, a client can display a category that includes unread messages in bold. PR_CONTENT_UNREAD cannot be used as a category and an attempt to do so results in the MAPI_E_INVALID_PARAMETER value being returned from the **IMAPITable::SortTable** method.

States of Categorized Tables

Categories have two states: collapsed and expanded. When categories are collapsed, only the heading rows are returned from **IMAPITable::QueryRows**. When categories are expanded, both heading and leaf rows are returned. Categories can be expanded or collapsed on an individual basis. That is, not all categories in a table must be one state or the other; some categories can be collapsed while others expanded.

The user of a categorized table decides how it is displayed. One common option is to use a control provided in the Win32 SDK called the treeview control. Treeview controls are listboxes that support information in a tree-like structure. Heading rows in the expanded state are marked with a minus sign and heading rows in the collapsed state are marked with a plus sign. Expanded categories are displayed with the leaf rows indented under the heading rows.

The following illustration shows a categorized contents table with an expanded category to illustrate the difference between how an expanded and collapsed category can appear to the user. The table is sorted by sender name, the PR_SENDER_NAME property. Heading rows contain the sender name and the computed properties for the row. The category where sender name is equal to Bill Lee is expanded to include five leaf rows. All of the leaf rows are visible and there is a minus sign next to the heading row, indicating that no further expansion of the catgory is possible.

{ewc msdncd, EWGraphic, groupx832 12 /a "MAPI_59.WMF"}

In the following illustration, the category is collapsed with none of the leaf rows available to include in the display. The heading row has a plus sign indicating that the category can be expanded further.

{ewc msdncd, EWGraphic, groupx832 13 /a "MAPI_57.WMF"}

Collapsing and Expanding Categories

To collapse and expand a category, a client application or service provider uses the following **IMAPITable** methods:

[GetCollapseState](#)
[SetCollapseState](#)
[ExpandRow](#)
[CollapseRow](#)

GetCollapseState saves the data necessary for **SetCollapseState** to use to rebuild the expanded and collapsed view categories of a categorized table. Although the format of the data and how to restore the state are entirely up to the table implementor, data that is typically saved includes:

- The sort keys (standard columns and category columns).
- Information about the row that the instance key represents.
- Information to restore the collapsed and expanded categories of the table.

A call to **GetCollapseState** must always precede a call to **SetCollapseState**. Callers pass a pointer to an instance key that represents the row at which the state will be rebuilt to **GetCollapseState**. **GetCollapseState** saves the necessary information and passes it back to the caller. When **SetCollapseState** is called, the information is passed in as input and a bookmark that identifies the same row as was represented by the instance key passed to **GetCollapseState** is returned as output. If this row no longer exists, service providers should return the bookmark BOOKMARK_BEGINNING. Callers are responsible for freeing the bookmark by calling **IMAPITable::FreeBookmark**.

SetCollapseState implementors are responsible for verifying that the sort order and restrictions are exactly the same as they were at the time of the **GetCollapseState**. If a change has been made, ideally **SetCollapseState** should not be called because the results can be unpredictable. This can happen if, for example, a client calls **GetCollapseState** and then **SortTable** to change the sort key before calling **SetCollapseState**. However, to be safe, **SetCollapseState** should check that the saved data is still valid before proceeding with the restoration.

To alter the display of a category, table users call the method **IMAPITable::ExpandRow** or **IMAPITable::CollapseRow**. When a category is to be expanded with all of its leaf rows available, a table user calls **ExpandRow**. Conversely, when a category is to be collapsed, **CollapseRow** is called to remove the leaf rows from view.

Users calling **ExpandRow** pass the instance key of the row to undergo the expansion, the number of bytes in the instance key, and the maximum number of rows to be returned. **ExpandRow** passes back an **SRowSet** structure with the additional rows. The rows in the **SRowSet** might or might not equal the number of rows that were actually added to the table, the entire set of rows for the category. Errors can occur, such as when insufficient memory prohibits the return of all of the rows in the category, or when the number of rows in the category exceeds the maximum number that can be returned. In either case, BOOKMARK_CURRENT should be positioned to the last row returned. This allows the caller to immediately call **IMAPITable::QueryRows** to retrieve the rest of the rows in the category.

CollapseRow requires an instance key identifying the category row to be collapsed and the number of bytes in the key. It returns the number of rows that are removed from the table view. When a row that is defined by a bookmark is collapsed out of view, the bookmark is moved to point to the next visible row.

Table notifications are not sent when a table is expanded or collapsed. A client maintaining the data can update its copy of the leaf rows when making the **ExpandRow** or **CollapseRow** call.

About Restrictions

A restriction is a way to limit the number of rows in a view to only those rows with values for columns that match specific criteria. There are many different opportunities for using restrictions with tables. Client applications can use restrictions, for example, to filter a contents table for messages sent by a particular person, to search for rows that either do not support a property or have set a property to a specific value, or to look for duplicate recipients within a message.

The **IMAPITable::Restrict** and **IMAPITable::FindRow** methods are used to set restrictions on a table. **Restrict** applies the restriction to the table without retrieving any rows. To retrieve only those rows that meet the restriction, a subsequent call to **IMAPITable::QueryRows** or a similar method is required. **FindRow** applies the restriction and retrieves the first row in the table that matches the criteria. **FindRow** applies a temporary restriction, in existence only for the duration of the call, whereas **Restrict** applies a more permanent restriction.

Some clients can build a restriction using columns that are not in the current column set. Supporting such a restriction is optional and table implementors that do support it add value, particularly for contents tables. Table implementors that do not support it can return the MAPI_E_TOO_COMPLEX value from a **Restrict** call or the MAPI_E_NOT_FOUND value from a **FindRow** call.

Clients should be aware that, even if the provider does support restrictions on columns not in the current column set, they will get better performance overall by specifying the columns they intend to use in their restrictions with **SetColumns**.

Types of Restrictions

There are many types of restrictions, some that focus on specific columns. All table implementations are expected to support restrictions on the columns in the current column set. However, to add value, table implementors can also support restrictions based on object properties that are not currently in the table view.

Some restrictions can be combined using a restriction that performs a logical AND, OR, or NOT operation. For example, most Property restrictions must be joined with Exists restrictions using And restrictions. There are a few exceptions, such as when the property used in the Property restriction is the PR_ANR property or when it is a required column in a table. Clients building restrictions to limit their view should use Exists restrictions with their Property restrictions because MAPI does not specify how service providers should evaluate Property restrictions when a property does not exist. It is reasonable and recommended that service providers fail the restriction, but there are no requirements.

A restriction is defined using the **SRestriction** data structure which contains a union of more specialized restriction structures and an indicator of the type of structure included in the union.

Each of the specialized restriction structures in the union represents a different type of restriction. The types of restrictions and their associated data structures are listed in the following table.

Type of restriction	Associated data structure	Description
Compare Property	**SComparePropsRestriction**	Compares two properties of the same type.
And	**SAndRestriction**	Performs a logical And operation on two or more restrictions.
Or	**SOrRestriction**	Performs a logical Or operation on two or more restrictions.
Not	**SNotRestriction**	Performs a logical Not operation on two or more restrictions.
Content	**SContentRestriction**	Locates specified data.
Property	**SPropertyRestriction**	Specifies a particular property value as criteria for matching. Can be used, for example, to search for a particular type of attachment.
Bitmask	**SBitMaskRestriction**	Applies a bitmask to a PT_LONG property, typically to determine if particular flags are set.
Size	**SSizeRestriction**	Tests the size of a property using standard relational operators.
Exists	**SExistRestriction**	Tests whether or not an object has a value for a property.
Subobject	**SSubRestriction**	Used for searching through subobjects, or objects that cannot be accessed with an entry identifier (recipients and attachments). Can be used, for example, to look for messages

		for a particular recipient.
Comment	**SCommentRestriction**	Associates an object with a set of named properties.

Some restrictions use regular expressions and MAPI supports a limited form of regular expression notation in the style that is used in the **grep** utility.

The Comment restriction is used by clients that save restrictions on disk, to keep application-specific information with the restriction. For example, a client saving the name of a named property used in a Property restriction can do so with a Comment restriction. Saving the name is not possible in a Property restriction; the **SPropertyRestriction** data structure holds only the property tag. Comment restrictions are ignored by **Restrict** in that they have no effect on the rows returned by **QueryRows** after a **Restrict** call has been made.

About Content Restrictions

Content restrictions are used to search for one or more characters in text. The **SContentRestriction** structure includes a property tag, a property value, and a comparison indicator. This indicator, called the fuzzy level, describes the degree of exactness or looseness that is applied when searching the text.

The fuzzy level can be defined with six different values, three of which are mutually exclusive. Although table implementors are not required to support all of these values, it is recommended that they do. Table implementors can return the MAPI_E_TOO_COMPLEX value from an **IMAPITable::Restrict** call that sets a value that is not supported. The mutually exclusive values are as follows:

Fuzzy level value	Description
FL_SUBSTRING	Specified property value must be included in the column.
FL_PREFIX	Specified property value must appear at the beginning of the column.
FL_FULLSTRING	Value of the column must match the specified property value exactly.

Three other fuzzy level values can also be set; callers can either set none of the values, some of the values, or all of the values. These values are meant to be suggestions to the table implementor; exact interpretation of them is totally implementation-dependent.

Fuzzy level value	Description
FL_IGNORECASE	Differences in upper and lower case are ignored.
FL_IGNORENONSPACE	Differences in non-spacing characters, such as accents and diacritics, are ignored.
FL_LOOSE	Less strict rules for matching are used.

About Address Book Restrictions

Address book providers are required to support three types of restrictions on the contents tables of their containers:

- Ambiguous name property restrictions
- Instance key property restrictions
- Prefixed display name content restrictions

Ambiguous name restrictions are property restrictions using the PR_ANR property to match recipient names with entries in address book containers. The PR_ANR property restriction is a "best guess" type of search whereby address book providers can choose the matching property that works best for their container. For example, one address book provider might implement the PR_ANR restriction by matching recipient names against the PR_ACCOUNT property of each container entry whereas another provider might use PR_DISPLAY_NAME.

MAPI recommends that implementations of the PR_ANR restriction strike a balance between adequate performance and user satisfaction. User satisfaction can be compromised when an address book provider implements the restriction in such a way that too few or too many matches are found. Some address book providers support what is known as a distinguished, or common, name that is not displayable in a dialog box but can match an ambiguous name restriction.

A typical implementation might be to parse the recipient's display name into words, matching any entry that contains all of the words. Attention to details such as sensitivity to word position, whether or not non-consecutive words are matched, and the choice of separator characters can vary. For example, if the name to be resolved is "Bill L," a typical PR_ANR restriction would select the following entries as matching:

Billy Larson
Bill Lee
Bill Logan Jr.
Sam Bill Lee

Instance key restrictions, or PR_INSTANCE_KEY property restrictions, are used in the implementation of list boxes that are used in client applications for viewing MAPI tables. Some list box implementations allow users to make multiple selections, scroll up or down, and return to the first item selected. To implement this behavior, clients call **IMAPITable::FindRow**, passing a property restriction on the PR_INSTANCE_KEY property to the method. Address book providers are required to support this restriction.

Another feature of list boxes used for table viewing is the ability to position the cursor based on a set of prefix characters. As the user starts typing prefix characters, the client moves the cursor to the first item that begins with these characters. Clients implement this feature with a content restriction based on the PR_DISPLAY_NAME property and the FL_PREFIX fuzzy level.

Building a Restriction

To build a restriction, a client application creates a hierarchy of one or more **SRestriction** structures of various types and passes a pointer to the hierarchy to the **IMAPITable::Restrict** or **IMAPITable::FindRow** method. The illustration and the code sample that follow illustrate how a typical restriction is implemented with linked restriction structures of different types. A user of a client application is trying to find all messages that contain the word "volleyball" in the subject line and were sent to Sue from Sam. First, a generic **SRestriction** structure is allocated. This structure becomes the basis for other calls to the **MAPIAllocateMore** function to create linked **SRestriction** and **SPropValue** structures that can be freed with a single call to **MAPIFreeBuffer**. Because the criteria to apply to the set of messages is in three parts, the top level restriction structure is an And restriction. The **SAndRestriction** structure's **cRes** member is set to 3 to indicate the three restrictions to evaluate and its **lpRes** member is set to a three member array of **SRestriction** structures.

To search for messages that are sent to a particular recipient, it is necessary to search the recipient table for each message rather than the message itself. A Subobject restriction is used to perform the recipient table search. Therefore, the first member of the array points to a **SSubRestriction** structure with its **ulSubObject** member set to PR_MESSAGE_RECIPIENTS. Then, to specify what to look for in the recipient table, a Content restriction is used.

The second and third members of the array are more straightforward. They both point to Content restriction structures, one to search for messages that have a PR_SENDER_NAME property set to "Sam" and another that has a PR_SUBJECT property set to "volleyball."

{ewc msdncd, EWGraphic, groupx832 14 /a "MAPI_61.WMF"}

Sample Restriction Code

The following sample code shows how to create a restriction that filters out all messages that do not contain the word "volleyball" in the subject line and were not sent to Sue from Sam. A tree of **SRestriction** structures is required, with the top node being an And restriction implemented with an **SAndRestriction** structure. The three restrictions that are joined by the And operation are a Subobject restriction that searches for messages sent to Sue, a Content restriction that searches for messages from Sam, and another And restriction that searches for messages that have a subject containing "volleyball." Because subject is not a required property, an Exists restriction must be included.

This code uses dynamic allocation and initialization; it is possible to allocate and initialize statically as well. In the interest of brevity, the error checking that must occur following the allocation calls is not included in the sample.

```
HRESULT BuildRestriction (LPTSTR pszSent, LPTSTR pszFrom,
                                LPTSTR pszSubjectText);
{
    LPSRestriction pRest, pAndRes, pObjRes, pSubjAndRes;
    LPSPropValue pRecip, pSender, pSubject;
    HRESULT hResult;
    ULONG ulResCount = 3, ulSubjCount = 2

    // Allocate and build And restriction to join criteria
    hResult = MAPIAllocateMore (sizeof(SRestriction)*ulResCount, pRest,
            (LPVOID *)&pAndRes);
    pRest->rt = RES_AND;
    pRest->res.resAnd.cRes = ulResCount;
    pRest->res.resAnd.lpRes = pAndRes;

    // Allocate and build Subobject restriction to search recipient list
    hResult = MAPIAllocateMore (sizeof(SRestriction), pRest,
            (LPVOID *)&pObjRes);
    pAndRes[0].rt = RES_SUBRESTRICTION;
    pAndRes[0].res.resSub.ulSubObject = PR_MESSAGE_RECIPIENTS;
    pAndRes[0].res.resSub.lpRes = pObjRes;

    // Allocate and build Content restriction to look for recipient
    hResult = MAPIAllocateMore (sizeof(SPropValue), pRest,
            (LPVOID *)&pRecip);
    pObjRes->rt = RES_CONTENT;
    pObjRes->res.resContent.ulFuzzyLevel =
            FL_FULLSTRING | FL_IGNORECASE;
    pObjRes->res.resContent.ulPropTag = pRecip->ulPropTag =
            PR_DISPLAY_NAME;
    pObjRes->res.resContent.lpProp = pRecip;
    pRecip->Value.LPSZ = pszSent;              // pszSent set to Sue


    // Allocate and build Content restriction to look for sender
    hResult = MAPIAllocateMore (sizeof(SPropValue), pRest,
            (LPVOID *)&pSend);
    pAndRes[1].rt = RES_CONTENT;
    pAndRes[1].res.resContent.ulFuzzyLevel =
            FL_FULLSTRING | FL_IGNORECASE;
    pAndRes[1].res.resContent.ulPropTag = pSend->ulPropTag =
            PR_SENDER_NAME;
    pAndRes[1].res.resContent.lpProp = pSend;
    pSend->Value.LPSZ = pszName;                    // pszName set to Sam

    // Allocate and build And restriction to look for subject
    hResult = MAPIAllocateMore (sizeof(SRestriction)*ulSubjCount, pRest,
            (LPVOID *)&pSubjAndRes);
    pRest->rt = RES_AND;
    pRest->res.resAnd.cRes = ulResCount;
    pRest->res.resAnd.lpRes = pAndRes;

    // Create an And restriction to search for subject
    hResult = MAPIAllocateMore (sizeof(SPropValue), pRest,
            (LPVOID *)&pSubjAndRes);
    pAndRes[2].rt = RES_AND;
    pAndRes[2].res.resAnd.cRes = ulSubjCount;
    pAndRes[2].res.resAnd.lpRes = pSubjAndRes;

    // Exists restriction to check that PR_SUBJECT exists
    hResult = MAPIAllocateMore (sizeof(SPropValue), pRest,
            (LPVOID *)&pSubj);
    pSubjAndRes[0].rt = RES_EXIST;
    pSubjAndRes[0].res.resExist.ulReserved1 = 0;
    pSubjAndRes[0].res.resExist.ulReserved2 = 0;
    pSubjAndRes[0].res.resExist.ulPropTag = PR_SUBJECT;

    // Content restriction to check for "volleyball" in subject
    hResult = MAPIAllocateMore (sizeof(SPropValue), pRest,
            (LPVOID *)&pSubj);
    pSubjAndRes[1].res.resContent.ulFuzzyLevel =
            FL_SUBSTRING | FL_IGNORECASE;
    pSubjAndRes[1].res.resContent.ulPropTag = pSubj->ulPropTag =
            PR_SUBJECT;
    pSubjAndRes[1].res.resContent.lpProp = pSubj;
    pSubj->Value.LPSZ = pszSubjectText;

    return hResult;
}
```

Advanced Table Operations

Advanced table operations are relevant to a minority of MAPI-compliant client applications and service providers. Some advanced operations are more complex to implement; others are no more complex, but are of interest to a small minority of MAPI components. For example, many clients and service providers work with complex asynchronous table notifications but only a handful of clients and service providers work with multivalued columns.

About Table Notifications

Service providers send asynchronous notifications to clients that have registered to receive them when a change occurs to data in a table. To register for table notifications, a client creates an advise sink object and passes a pointer to it in a call to the **IMAPITable::Advise** method. This advise sink object can be implemented by the client or by MAPI through the API function **HrAllocAdviseSink**.

Clients often rely on table notifications to learn of changes to objects rather than registering for notifications directly with the object. When notification arrive, clients can determine whether or not to make another call to reload the table.

Typical changes that cause notifications to be sent include the addition, deletion, or modification of a row and any critical error. Service providers can implement the registration and generation of notifications themselves or delegate to the following support methods provided by MAPI:

IMAPISupport::Subscribe

IMAPISupport::Unsubscribe

IMAPISupport::Notify

When providers generate table notifications, they construct a **NOTIFICATION** structure that includes a TABLE_NOTIFICATION structure in its **info** member. The **TABLE_NOTIFICATION** describes the change that has occurred to the table. One of its members, **ulTableEvent**, specifies the type of change that has occurred. There are several different events that can occur and table users that have registered for notifications should be prepared to handle any of them. The list of possible events is as follows:

TABLE_CHANGED	TABLE_RELOAD
TABLE_ERROR	TABLE_SORT_DONE
TABLE_ROW_ADDED	TABLE_ROW_DELETED
TABLE_ROW_MODIFIED	

TABLE_NOTIFICATION structures also include an HRESULT datatype for reporting an error value, the instance key of the affected row, the instance key of the row just prior to the affected row, and an **SRow** structure containing the property data for the affected row. If the affected row is the first row in the table, **propPrior** must be set to PR_NULL and not zero. Zero is not a valid property tag. The properties that are included in the **SRow** structure are ordered using the column set that was established from the previous **SetColumns** call, if one occurred. If no **SetColumns** call was made prior to the notification being sent, the columns are ordered in a default order decided by the table implementor.

Service providers that send batched notifications must order them so that they can be interpreted from the first notification to the last. This ordering is especially necessary when a notification batch contains a series of events such as TABLE_ROW_ADDED and one event refers to a prior row that was added in another event in the same batch.

About Asynchronous Table Operations

Table notifications to registered client applications and service providers are always asynchronous as can be a few of the table methods. When a method is asynchronous, it can return before the operation being performed is complete. The three asynchronous table methods are the **IMAPITable** methods **SetColumns**, **Restrict**, and **SortTable**.

As an advise sink receiving table notifications, there are a few problems to be aware of that can occur due to the asynchronous nature of event notification. First, the data passed in the **TABLE_NOTIFICATION** structure might not be the most current information about the state of the table. For example, an advise sink might receive a TABLE_ROW_MODIFIED notification on a row that has since been deleted. The corresponding TABLE_ROW_DELETED notification is pending.

Second, the columns that are returned in a notification and order of their return might not match the order and selection defined in the most recent call to **SetColumns**. MAPI requires that the column set match the set from the last **SetColumns** call that was in effect at the time that the notification is generated. However, if a client makes a second **SetColumns** call after the notification is sent but before the **IMAPIAdviseSink::OnNotify** call is made, the column set will not match the client's current view.

Third, table notifications are only sent for rows that are part of the view. That is, if a row is excluded due to a restriction or because the table is in a collapsed state, no notification will be sent if that row changes.

To work with asynchronous table operations, table users can call one of the following three **IMAPITable** methods:

GetStatus
Abort
WaitForCompletion

GetStatus returns information about the table type and if an operation is in progress or an error has occurred from a completed operation. For example, if a client needs to cancel a sort operation because it is taking too much time, the client can first call **GetStatus** to determine if, in fact, a sort operation is presently processing. Then the client can call **Abort** to stop it. To suspend activity until an asynchronous task has completed, a client or service provider can call **WaitForCompletion**. Calling **WaitForCompletion** allows the task to complete without interruption.

Working with Multivalued Columns

A multivalued column contains the data of a multivalued property, or property that has an array of values of the base type rather than a single value. Because none of the tables include multivalued properties in their default column sets, multivalued properties are included in a table only if the user of the table requests it.

Multivalued columns can be displayed in tables in one of two ways:

- In a single row, with all of the property values appearing in the single column instance. This is the default.
- In a series of rows, with one row for each of the property values. Each unique value appears in the column in its own row with there being as many rows as there are values in the multivalued property. Each row has a unique value for the PR_INSTANCE_KEY property, but the same values for the other columns.

  If a row contains more than one column with multiple values, for example, two columns with *M* and *N* values respectively, then *M*N* instances of the row appear in the table.

A table user requests the non-default type of display by calling the **IMAPITable::SetColumns** method with the MVI_FLAG flag set in the property type of the multivalued column. The MVI_FLAG flag is a constant defined as the result of combining the MV_FLAG and MV_INSTANCE flags with a logical OR operation. In addition to being used in **SetColumns**, MVI_FLAG can also be passed to **IMAPITable::SortTable** in the *lpSortCriteria* parameter and **IMAPITable::Restrict** in the **ulPropTag** member of the *lpRestriction* parameter. When passed the MVI_FLAG, **SortTable** performs similarly to **SetColumns**, adding one row for each value in the multivalued column and sorting on the single values in the instances and rows are added for each value. **Restrict**, however, does not expand the multivalued column into additional computed rows. A multivalued column with the MVI_FLAG set instructs the service provider to use that column in restricting the table. If there is a property value in the restriction, it must be a single value property tag identical to the one that would be returned by **IMAPITable::QueryRows** for the column.

Table implementors are only required to support the default type of display and can return the MAPI_E_TOO_COMPLEX value when a caller requests the other alternative. The ability to support both types of display is most important for message store providers implementing folder contents tables.

Working with Unicode Columns

Character strings in tables can be represented using standard 8-byte characters, which are property type PT_STRING8, or 16-byte Unicode characters, which are property type PT_UNICODE. Table implementors are free to choose whether or not their tables support Unicode strings. Because Unicode adds value for both clients and service providers by extending the feature set, supporting Unicode wherever possible is recommended. Although all of the methods in **IMAPITable** support Unicode, most of the MAPI table object implementations do not.

Many table methods accept a flag that dictates whether or not string property values are expected to be Unicode. On input, specifying the MAPI_UNICODE flag indicates to the table implementor that all string property values passed in with the call are Unicode strings and have property types of PT_UNICODE. On output, this flag indicates that all returned string property values should be Unicode strings, if possible. Whether the flag has a meaning for input or output depends on the method. Table implementors that do not support Unicode and are passed the MAPI_UNICODE flag return the MAPI_E_BAD_CHAR_WIDTH value.

Using Table Data Objects

A table data object is a utility object implemented by MAPI to help service providers and client applications implement table objects and perform table maintenance. To obtain a table data object, service providers and clients call the API function **CreateTable**.

Table data objects support the **ITableData : IUnknown** interface, which has methods for:

- Adding data to rows.
- Deleting data from rows.
- Changing data in rows.
- Creating a table object.
- Retrieving rows in a special order.
- Retrieving property values.
- Generating table notifications.

A table data object holds all of the data and any associated restrictions in memory, making it unsuitable for use with very large tables. Large restrictions and complex operations such as categorization are unsupported.

Table data objects identify rows using an index column, a property that is guaranteed to have a unique value for each row. Most table implementors use the PR_INSTANCE_KEY property as the index column. Multivalued properties cannot be used as an index column.

To create a table object, table data object users call the **HrGetView** method. Callers can specify a sort order and a callback function to be invoked when the view is released. Frequently, message store or address book providers call **HrGetView** to generate a table object for a client. When the client is finished using the table and calls its **Release** method, the callback function defined in the **HrGetView** parameter list is called.

To change one row in a table, table data object users call the **HrModifyRow** method, and to change multiple rows they call the **HrModifyRows** method. Callers specify the row or rows to be changed by passing in one or more index columns.

To delete one row in a table, table data object users call the **HrDeleteRow** method, and to delete multiple rows they call the **HrDeleteRows** method. As with the modify row methods, callers specify one or more index columns indicating the row or rows to delete.

Table data objects generate a single notification regardless of the number of rows affected by a change or deletion. If a target row in an operation does not exist, a row is added.

To retrieve rows from a table in the order that they were inserted, table data object users call the **HrEnumRow** method. When a row is inserted, MAPI assigns it a sequential number. **HrEnumRow** uses this number to determine the order for retrieval.

To insert a row at a particular position, table data object users call the **HrInsertRow** method. The position is specified by callers with a sequential number, 0 being the first row in the table. For example, if the value 50 is passed to **HrInsertRow**, MAPI inserts the new row as the 51st row in the table.

To generate a notification to all clients and service providers that hold pointers to table objects created with **HrGetView**, table data object users call the **HrNotify** method. Callers pass in a set of properties; MAPI generates table modified notifications for each of the rows with columns matching these properties. Notifications are sent regardless of whether the data has actually changed. Users of display tables, or tables whose rows represent information about user interface controls, reload the data associated with the affected control upon receiving the table modified notification.

MAPI Form Architecture

This section provides an overview of the MAPI form architecture. After reading this section you will have an understanding of what MAPI forms are and how they interact with other components of the MAPI subsystem. The purpose of this section is to give you the conceptual knowledge you need to implement your own MAPI form servers.

Note Because the MAPI form architecture is based on the OLE component object model, developing form server applications requires knowledge of OLE programming. For more information on OLE, see the *OLE Programmer's Reference* in the *Win32 Software Development Kit*.

About MAPI Form Components

The relationship between the MAPI components involved in using forms is shown in this diagram. Following is a brief description of the components involved; more detailed descriptions are provided later in this section.

{ewc msdncd, EWGraphic, groupx840 0 /a "MAPI.WMF"}

In the diagram, notice that the form manager plays a role that is similar to other MAPI service providers, although it is not a service provider itself. That is, the form manager is a replaceable DLL that implements a portion of the MAPI interfaces. Although developers can implement their own form manager, most environments will use the form manager provided by Microsoft due to the form manager's complexity.

The components shown in the diagram and their relationship to other components are:

- Messaging client: An application that is able to use form objects. The messaging client uses the MAPI form interfaces to communicate with the form manager in order to load messages into form objects.
- MAPI form interfaces: A defined standard for communication between MAPI components that are related to forms.
- Form manager: The DLL that messaging clients use to handle installation of forms in form libraries, loading of form servers, and initial communication between messaging clients and form servers.
- Form libraries: Permanent storage for the executable files associated with form servers.
- Form servers: Executable files which implement a form. Form servers create form objects and user interfaces to deal with specific messages. This executable is also an OLE server and adheres to the normal OLE conventions.
- Form objects: Run-time objects created by form servers that correspond to specific messages. Form objects run in the same process context as their form server.

About Form Servers

The user's perception of a form is usually a property sheet for a message or a data-entry form that enables users to enter structured information. However, it can be any user interface that is associated with a message class. From a programmer's point of view, a form consists of:

- A type of MAPI message with it's own message class and OLE identifier.
- The executable file that implements the form server.
- A collection of MAPI properties, custom or otherwise, that the form server uses. Some or all of these may be available to messaging clients for use.
- The configuration file that describes the form and is used by the form manager.

Because forms are **IMessage** objects, they exhibit properties and behavior that is consistent with MAPI message objects. However, because forms can have custom properties, controls, and a display rendering that is application-specific, the MAPI interfaces that forms are generic enough to permit any sort of interface that is needed. The actual definition of a form is stored in a form library, which is discussed later in this section.

Note More accurately, all messages are instances of MAPI forms. However, it is usually easier to think of custom forms as special cases of messages, since forms for composing and reading normal e-mail messages are the most commonly used forms. The fact that all messages are really just forms gives custom forms the same status as any other message in the MAPI system.

Every form has a set of properties, some of which are visible within the form's user interface. Usually, properties are matched to fields within the form's user interface. For example, a purchase order form might have the fields Item, Description, Price, Tax, and Subtotal. These fields are simply visual renderings of form properties of the same names. Clients ascertain which properties are supported by a particular message class through the **IMAPIFormInfo::CalcFormPropSet** method, which is implemented by the MAPI form manager.

Like basic messages, MAPI forms can contain all the standard message properties such as the sender, the intended recipient, and when the message was sent. Forms can also contain any number of custom properties that are specific to the form. For example a "Bug Report" form might contain custom properties for Bug Type, Bug Severity, and Product Version.

To create a form you must implement a form server. The form server is the executable file that is loaded when a messaging client needs to display a message that is the type supported by the form server. The form server in turn creates form objects as necessary to display specific messages and handle user interactions with those messages.

Every form server has a configuration file associated with it. This file contains information that describes the form server for the benefit of the form manager. The form manager uses this information when installing the form server into a form library.

For details on creating the parts of a form, see Developing MAPI Form Servers.

About MAPI Message Classes

Message classes are an important concept within MAPI, and of particular importance to forms. A message class is a string property that is assigned to all MAPI messages and identifies the type of message. The property associated with the message class string is PR_MESSAGE_CLASS. Programmers often use the term "message class" to refer to the set of messages of a particular class. For example, the string "IPM.Note" can be used to refer to the set of all messages where PR_MESSAGE_CLASS is equal to "IPM.Note." For information on the format of a message class string, see PR_MESSAGE_CLASS.

Each message class is associated with a form server that MAPI activates whenever a message of that class must be rendered or handled. Form servers adhere to the OLE component object model (COM). Form servers run as standalone executables, not as in-proc servers. For more information, see the *OLE Programmer's Reference*.

Message classes can be perceived as forming a hierarchy, and one message class can be a superclass of another message. For instance, IPM.Note is a superclass of IPM.Note.Secure and IPM.Note.Ink. A message class is considered a superclass of another if the message class string of the first appears as a period-delimited prefix of the second. For example, IPM.Note is a superclass of IPM.Note.Signed, but is not a superclass of IPM.Phone or IPM.Noteworthy. A message class string that is an extension of another message class string is referred to as a subclass.

Messages of a particular class contain the same set of properties. A superclass contains a set of properties common to all of its subclasses. Subclasses may define additional properties beyond the common ones defined by the superclass, or they may interpret the superclass's properties differently.

Each form server is identified by a unique OLE class identifier often written as just CLSID. There is always a one-to-one mapping between a class identifier and it's message class. This does not mean, however, that a form server can only work with messages of one message class. If no form server is available to service a message of a particular class, the form manager being used should attempt to find a form server for a message class higher in the message class hierarchy; the default form manager supplied with the MAPI SDK does this. Such a form server will probably be able to render only a subset of the message's properties (the ones supported by the superclass), but it will be better than nothing. What happens when no matching form server is found at all is an implementation detail specific to the form manager being used; the default form manager does not open messages when this happens.

About Form Verbs

A form's user interface typically offers menu items or controls that enable users to take some kind of action with the form. It is the form server's job to handle these user actions. This interface is implemented using standard Win32 APIs; writing one is just like writing other interfaces for regular Win32 programs.

Often, user actions are associated with verbs. A verb is the name for an action that is specific to a certain message class. For example, *Reply* is a verb that is implemented by many form servers, each of which may have a different interpretation of that verb. Verbs are sometimes referred to as commands.

Note Not all menu items and controls on a form correspond to a verb. For example, a Cancel button does not correspond to a Cancel verb within the form server. Usually, verbs are associated with actions that are specific to a particular message class or a set of message classes. Although different message classes can support different sets of verbs, all support at least the Open verb, which displays the form's user interface and loads it with the message's property values.

Verbs may take no parameters. Forms that export commands with variable parameters must use the OLE Automation mechanisms.

Clients can determine which verbs are supported by a particular message class through the **IMAPIFormInfo::CalcVerbSet** method, which is implemented by the MAPI form manager. The form manager gets this information from the form's configuration file. The verb set returned by this method is used by the client to show the user which commands can be executed on a message. For example, a client might enable users to click the right mouse button over a message to display verbs applicable to that message.

About Form Objects

Form objects are created dynamically by form servers in order to display specific messages and allow users to interact with them. A form object is, therefore, usually an instantiation of the **IMAPIForm**-derived class implemented by the form server. When a client application opens a message, the form server for that message class creates a form object to handle the message. The form object then creates its interface, and displays the properties of the message in it. The form object and its interface persists until the user closes it. The form object handles any changes to the values of the message's properties.

Additionally, the MAPI form interfaces define a mechanism by which one form object can load and display a series of messages. This is an efficiency mechanism, as it avoids needless destruction and creation of message objects and their interfaces. When requested by the messaging client to load a different message, the form object should save any changes to the current message's properties.

About the Form Manager

A form manager is an object that implements the **IMAPIFormMgr** interface. Most organizations will use the form manager supplied with MAPI, referred to as the default form manager. However, an organization can replace the default form manager with a custom form manager if desired. The form manager takes care of locating forms within form libraries, loading forms in response to user requests, and installing forms into a user's local form library, folder form library, or personal form library.

When a user wants to interact with a message, the form which implements the message's message class must be activated. Activation means that the form transitions from an idle state to a running state so the user can interact with it.

To interact with a message, an instance of the form server for the message's message class must be created and activated to display the message and carry out the requested operation on the message. As described in the topic About Form Libraries, a form's implementation can exist in several different locations (form libraries) and there is no guarantee that a form or its server will be locally available or in a running state when a user wants to interact with it. The form manager takes care of the details of locating and activating the form.

Clients use services provided by the form manager to find and activate forms. The **IMAPIFormMgr** interface is implemented by the form manager and is called by clients to access its services. The form manager is an essential component because it hides almost all the details of finding and activating forms from messaging clients.

When loading form servers, the default form manager loads the form from the first form library in which an implementation for the form's message class is found. The default form manager searches the form libraries in the following order:

1. The user's local form library. This form library is searched first because it provides the fastest access to a form's implementation if the implementation is installed in the local form library.
2. The folder form library of the message's container – the folder that the message being loaded is stored in.
3. The user's personal form library.

A custom form manager can search the available form libraries in any order, or can implement other form libraries such as an organization-wide form library. For more details on form libraries, see About Form Libraries.

About Capabilities Not Supported by Form Managers

For performance reasons, the following capabilities aren't supported by the default form manager, but may be supported by custom form managers:

- A hierarchy that enables forms to be grouped or categorized within a form library. A form library is essentially a flat-file database of forms.
- Access control for categories of forms, corresponding to message classes or superclasses.
- Support for multiple language versions of the same form in a single form library.

Those are entirely implementation issues. There is nothing to prevent a custom form manager from implementing those features.

The MAPI form architecture does not support multiple form managers running concurrently. Although MAPI supports multiple concurrent message store providers, transport providers, and address book providers, only a single form manager is supported.

Because only one form manager may be running at once, if you implement a custom form manager you will have to re-implement any functionality from the default form manager that you need. Since your custom form manager will entirely replace the default form manager, capabilities of the default form manager will be unavailable unless they are duplicated in your custom form manager.

About Disk Instances and Cache Tables

To activate a form, its executable files must be available on the user's computer. If they are not available, they must be copied from the form library to the local disk. To do this, the default form manager creates a subdirectory within the user's Windows directory to contain the form's executable files (.EXEs, .HLPs). This directory is referred to as the disk instance of the form.

The default form manager maintains a table of all disk instances so that if a disk instance already exists it can be used without having to copy files from the form library to the user's disk. The table of disk instances is managed as a least frequently used cache. If a new disk instance is needed, it is copied to the user's computer, replacing the least frequently used disk instance. The disk instance cache table is then updated to reflect the latest configuration. The size of the disk cache is a user-configurable option, enabling users to balance speed with available disk capacity.

In addition to the disk instance cache, the default form manager maintains a running instance table that lists all running instances of form servers on the user's computer. This uses MAPI's ability to keep idle form instances running in an invisible state until a form of that form server's message class is activated. In other words, form servers can be cached in RAM to minimize the number of times a form's executable must be located within a form library and loaded into memory from disk or over the network. Like the disk instance cache, the running instance cache behaves in a least frequently used fashion so that a running form instance can be purged from the cache to make room for another form instance. This cache is searched for a running instance of a form server before the form libraries are searched for the form server.

Note The default form manager displays a progress indicator when installing a form on a user's workstation, enabling the user to cancel the operation. This is especially useful if the user's connection to the form server's executable file is over a low bandwidth network.

About Platform Independence

Form libraries are independent of the platform on which the forms in them are used. This means that a single form definition in a form library can include versions of the form for multiple platforms such as Windows for Workgroups, Windows NT, and Macintosh. However, if a form does not support the platform that a user is using, the default form manager will attempt to find a form server for a superclass of the message being opened to view the form instead, provided that any are available which do support the user's platform.

About Form Libraries

To help organize form servers and make them easily accessible to client applications, the MAPI form architecture includes form libraries, which support the installation, administration, replication, and use of form servers. Form libraries are repositories of form servers and configuration information about them that the form manager uses when loading form servers.

Note Early in the history of MAPI, form libraries were called "registries." Because of this, the term "registry" has persisted in some form-related MAPI interfaces, properties, configuration file entries, and so forth. If your application's interface requires use of one of these terms, you should use the term "library" exclusively.

About Form Storage

Although it is not necessary to know all the details of how forms are physically stored, it is useful to understand a few of the main concepts. Therefore, before describing the three types of form libraries supported by the default form manager, this topic gives an overview of how forms are stored.

Form definitions can be physically stored within folders in one or more MAPI message stores. Every MAPI folder can be thought of as having two areas for storing message objects: the standard part and the associated part. The standard part of the folder includes the messages and folders that users manipulate.

The associated part includes hidden message objects that are associated with the folder, including form definitions, views, rule templates, reply templates, and so on. This alternate part is called the folder associated contents table, and the set of messages in the associated contents table is referred to as the folder-associated information. The hidden messages are an integral part of the folder and are copied along with the standard folder contents when the folder is copied. Although physically stored as messages, information in a folder's associated contents table behaves more like properties than like viewable messages. Any folder object that supports an associated contents table is capable of storing custom forms. The **IMAPIContainer::GetContentsTable** method can return either the standard contents or the associated contents of the folder, depending on the value of the *ulflags* parameter of the method.

A form library consists of form definitions stored in a folder's associated contents table. The form definition includes the form's properties, the actions the form supports, and even the form server executable file, which is stored as one or more message attachments.

Additionally, forms can be stored in any file or location that the form manager being used supports. The default form manager stores form servers in MAPI folders, but a custom form manager could implement its own storage for form servers.

A form can have multiple user interfaces that are bound to its message class. For example, a form can have separate Compose and Read interfaces. The form takes care of invoking the proper interface for different user requests, depending on which of the form's verbs is being called. For example, if your form server has separate composing and reading interfaces, the Compose interface can be opened automatically when the user creates a new message of the form's message class and the Read interface can be opened automatically when the user opens an existing message of the form's message class.

Most of the information stored within a form definition is available by invoking the **IMAPIFormInfo::IMAPIProp** method on an **IMAPIFormInfo** object. The **IMAPIFormInfo** interface simplifies access to form information by calling all the MAPI folder and message methods needed to retrieve the information. An **IMAPIFormInfo** object can be obtained by calling the **IMAPIFormContainer::ResolveMessageClass** method.

Each of the three types of form library is described in the following sections.

About Local Form Libraries

Local form libraries are stored directly on a user's machine, usually in a file called FRMCACHE.DAT in the %WINDOWS%\FORMS directory. Client applications can access forms in the local form library without accessing any network resources. Local form libraries are an exception to the rule that forms are stored in associated contents tables, since the local form library is just a file on disk that is not part of any MAPI folder heirarchy.

About Folder Form Libraries

In some cases, you might want to associate one or more forms with a specific folder. For example, employees in your organization could all have a Progress Report folder in their personal message store for creating and storing progress reports. Because the progress report is specific to each user's Progress Report folder, it might not be appropriate to store the progress report form in the system wide form library. However, a copy of the progress report form can be kept in the associated contents table of each user's Progress Report folder. This restricts the user from using progress report forms outside of the designated folder.

Conceptually, there is one folder form library for every folder in a message store, even if no form servers are installed in it. Folder form libraries are implemented like other form libraries − they are stored as associated contents tables in the alternate part of the folder. Because folder form libraries are contained within the folder, they are copied along with their parent folder in copy operations.

About Personal Form Libraries

As its name suggests, personal form libraries contain forms of interest to a particular user. A user's personal form library is the form library associated with the default message store identified in the user's profile; each profile installed on a workstation can use a separate default store, and therefore, a separate personal form library. A personal form library can contain copies of forms that are also contained in other form libraries in addition to other forms.

A personal form library is implemented in the associated contents table of the root folder in a user's default message store − whether that resides on a server or locally on the user's workstation is immaterial. If the user's default message store is a stored on the user's workstation, personal form libraries offer enhanced performance by enabling applications to access forms locally rather than over the network. It also makes forms available to users working off-line, which can occur when users want to take their forms with them on portable computers and are without access to a network.

The properties and underlying implementation of personal form library entries include a "Container ID" property that identifies a master container that the local entry must be synchronized with. This can be the ID of an arbitrary folder that contains forms. This is useful if you are using a custom form manager that supports some sort of organization-wide form library; the form manager would take care of synchronizing the forms stored in the personal form library and the organization-wide form library. This would probably happen when the form manager was loaded, but could theoretically happen at any time.

About Forms and MAPI Client Applications

Since all MAPI messages are instances of forms, client applications display and interact with form objects just as they do with other message objects. Client applications that use the MAPI form interfaces can provide additional interaction with form objects. Client applications can use any special verbs defined by a form to provide users with the full range of interactions intended by the form's designer. Clients can also use any custom properties that the form server exposes to help users sort, preview, or otherwise manage forms in the client's interface.

About MAPI Form Interfaces

MAPI defines the following interfaces relating to forms:

Interface name	Description
IMAPIForm	Manipulates form objects and handles form object commands.
IMAPIFormAdviseSink	Determines if the form object can handle the next message and changes the next or previous state of the form object.
IMAPIFormContainer	Supports installation, de-installation, and resolution of form servers against a specific form container.
IMAPIFormFactory	Supports the use of configurable run-time form servers.
IMAPIFormInfo	Enables client applications to work with properties that are specific to a message class.
IMAPIFormMgr	Enables client applications to get information about form servers, activates form servers, and installs form servers in the messaging system.
IMAPIMessageSite	Used to manipulate messages associated with form objects.
IMAPIViewAdviseSink	Notifies client applications that an event has occurred in the form object.
IMAPIViewContext	Used to respond to Next, Previous, and Delete commands in the form object.
IPersistMessage	Used to save, initialize, and load form objects to and from message storage.

For more information on the methods of the MAPI form interfaces, see MAPI Interfaces. You do not have to implement all of the MAPI form interfaces in order to create a custom form. A form itself requires only that you implement the **IPersistMessage**, **IMAPIForm**, and **IMAPIFormAdviseSink** interfaces. Additionally, it is also a good idea to implement **IMAPIFormFactory** and **IMAPIFormInfo**. **IMAPIFormFactory** is useful for OLE compliance, and **IMAPIFormInfo** allows well written client applications to make better use of your forms.

Note Strictly speaking, **IMAPIFormAdviseSink** is an optional interface. However, it is strongly recommended that you implement it in your form servers. This interface is critical to efficient interaction between messaging clients and form servers, especially when several messages of your form server's message class are being dealt with.

MAPI Component Basics

The following topics apply to developers of client applications and service providers. These topics are arranged in order of importance. That is, topics that are critical to all programmers appear first and topics that are either of minimal importance or do not apply to all programmers appear last.

About Entry Identifiers

An entry identifier is a binary data structure called an **ENTRYID** that is used to uniquely identify and open a message store or address book provider object. Message store providers assign entry identifiers to message stores, folders, and messages; address book providers assign them to address book containers, distribution lists, and messaging users. Entry identifiers are also used for accessing an object represented by a row in a table, such as the status object implemented by a transport provider and represented by a row in the status table. Objects store their entry identifiers in their PR_ENTRYID properties.

Whereas service providers create, assign, and examine entry identifiers, client applications use them only as tools for accessing the objects they represent. They retrieve them from an object's **IMAPIProp::GetProps** method or a table's **IMAPITable::QueryColumns** method. They compare them with the **CompareEntryIDs** method and pass them to the **OpenEntry** method to open the corresponding objects. **CompareEntryIDs** and **OpenEntry** are available with several MAPI objects. To clients, entry identifiers are opaque pieces of binary data and have nothing to do with the underlying messaging system.

However, clients should always pass naturally aligned entry identifiers in their calls to service providers because although service providers should handle entry identifiers that are arbitrarily aligned, this is not always the case. A naturally aligned memory address allows the computer to access any data type it supports at that address without generating an alignment fault. The natural alignment factor is typically the same alignment factor used by the system memory allocator and is usually 8 bytes.

Entry identifiers come in two types: short-term and long-term. Short-term entry identifiers are faster to construct, but their uniqueness is guaranteed only over the life of the current session on the current workstation. Long-term entry identifiers have a more prolonged lifespan. Short-term entry identifiers are used primarily for rows in tables and entries in dialog boxes whereas long-term entry identifiers are used for many objects such as messages, folders, and distribution lists.

Constructing Entry Identifiers

Entry identifiers are constructed with the **ENTRYID** data structure. The **ENTRYID** data structure is made up of a flag that describes the attributes of the entry identifier and the actual entry identifier. The **ENTRYID** structure is defined as follows where MAPI_DIM is defined in the MAPIDEFS.H header file.

```
typedef struct
{
    BYTE        abFlags[4];
    BYTE        ab[MAPI_DIM];
}  ENTRYID, FAR *LPENTRYID;
```

The first byte of the 4-byte **abFlags** member describes the type and use of the entry identifier and can have the following values:

MAPI_NOTRESERVED	MAPI_NOTRECIP
MAPI_NOW	MAPI_SHORTTERM
MAPI_THISSESSION	

MAPI_NOTRESERVED, when set, indicates that the entry identifier can be used by other service providers for other objects. The MAPI_NOTRECIP value indicates whether or not the entry identifier can be applied to a message recipient. When MAPI_NOW is set, the entry identifier can only be used at the present time, and when MAPI_THIS_SESSION is set, the entry identifier can only be used for the current session. MAPI_SHORTTERM identifies the entry identifier as a short-term identifier.

Clients expect to be able to check this first **abFlags** byte to determine the entry identifier's longevity. A zero in **abFlags[0]** indicates a long-term identifier and a nonzero indicates a short-term identifier. The last three bytes of the **abFlags** member must be zero.

The **ab** member of entry identifiers created by address book and message store providers is made up of two pieces: a 16-byte **MAPIUID** that identifies the service provider and a piece to identify the object. A **MAPIUID** is structure that contains a globally unique identifier, or GUID. A GUID is a byte order independent identifier which can be created using the Microsoft utility UUIDGEN.EXE.

A service provider registers its **MAPIUID** with MAPI during the logon process in a call to **IMAPISupport::SetProviderUID**. When a client calls an **OpenEntry** method to access an object, MAPI uses the **MAPIUID** to determine which service provider can provide that access. Service providers should use the same **MAPIUID** for all versions of their DLL. This enables clients with the newer version to respond to messages sent and saved with the older verion. Service providers that are using a profile that must work on two different processors, each with a different byte order, use a **MAPIUID** to differentiate between objects.

The rest of the **ab** member after the 16-byte **MAPIUID** contains service provider-specific binary data for identifying particular objects. A service provider typically includes the following in this part of their entry identifiers:

- Version information
- Location information

Because it is common for a service provider to change the format of its entry identifiers from version to version, storing version information makes it possible to quickly determine how to decipher any entry identifier.

Location information is data that gives a service provider an indicator of how to locate the object represented by the entry identifier. For example, a service provider can store the disk offset for the last place in a data file that the object was stored. Because this type of information can change over time, service providers should provide multiple ways for locating objects in their entry identifiers.

About Short-Term Entry Identifiers

A short-term entry identifier is assigned by a service provider to an object when the identifier must be constructed quickly and does not need to last over time or distance. The uniqueness of a short-term entry identifier is guaranteed only over the life of the current session on the current workstation. Typically a short-term entry identifier is valid only until the object that it represents is released.

Short-term entry identifiers are assigned to rows in tables and to entries in dialog boxes, where it is necessary to provide data quickly for browsing. For example, message store providers assign short-term entry identifiers to rows of messages in a contents table and to recipients in a recipients table. Clients can use these short-term entry identifiers to open the objects represented by the table rows. However, unlike long-term entry identifiers that can be used with any of the **OpenEntry** methods, short-term entry identifiers should be used with the container's **OpenEntry** method.

The most common ways to implement short-term entry identifiers include:

- Making the short-term entry identifiers the same as the long-term identifiers, leaving all of the flags unset.
- Making the short-term entry identifiers different from the long-term identifiers, setting all of the flags.

Clients can identify a short-term entry identifier of the second type by examining its **abFlags** member as follows:

```
abFlags[0] = 0xFF;
```

Some service providers create short-term entry identifiers that have greater validity by clearing one or more flags. For example, the following **abFlags** members represent short-term entry identifiers that can be used for multiple days or for multiple sessions:

```
abFlags[0] = 0xFF & ~MAPI_NOW;
abFlags[0] = 0xFF & ~MAPI_THISSESSION;
```

Clients quickly acquire, use, and discard short-term entry identifiers. For the most part, they can be used in the same manner as long-term entry identifiers. They can be retrieved from a table, passed to **OpenEntry**, and compared with **CompareEntryIDs**. The one exception is that they are never returned from **IMAPIProp::GetProps**. The properties returned from **GetProps** are always long-term entry identifiers.

About Long-Term Entry Identifiers

A long-term entry identifier is assigned by a service provider to an object when an object requires an identifier with a prolonged lifespan. Long-term entry identifiers are always valid for weeks or months and can be valid on other workstations, depending on the provider. The long-term identifiers created by address book providers for custom recipients are universally valid.

Long-term entry identifiers are assigned to message stores, folders, messages, address book containers, messaging users, and distribution lists. When client applications call the **IMAPIProp::GetProps** method of these objects, it is always a long-term entry identifier that is returned.

Because long-term entry identifiers must be unique across all message stores in the active profile, when a message or folder is copied from one message store to another, it must be assigned a new entry identifier. However, when a message store object is moved, whether or not the original entry identifier remains valid is up to the message store provider implementing the move. Some service providers assign new entry identifiers to moved objects; others do not. Therefore, clients must be prepared for either scenario depending on the provider.

Typically, message store providers implement the following behavior when moving folders:

- When moving a folder from one message store to another store of a different type, the entry identifier is guaranteed to change.
- When moving a folder from one message store to another store of the same type, the entry identifier almost always changes.
- When moving a folder to another location within the same message store, the entry identifier might or might not change, depending on the message store provider.

Note Renaming a folder without changing its parent usually does not cause the entry identifier to change.

About Record and Search Keys

Service providers assign two types of binary comparable identifiers to their objects: record keys and search keys.

A record key is a required property of all message store and address book objects that uniquely identifies an object and is used for comparison. Record key properties have property tags that end in RECORD_KEY, such as PR_RECORD_KEY and PR_STORE_RECORD_KEY. Because a record key identifies an object and not its data, every instance of an object has a unique record key. The scope of a record key is the provider resource in which the object is contained. For message store providers, this is the message store. For address book providers, this is the set of top-level containers used by MAPI in its integrated view.

Record keys can be duplicated in another resource. For example, different messages in two different message stores can have the same record key. This is different from long-term entry identifiers; because they contain a reference to the service provider, they have a wider scope. A message store's record key is similar in scope to a long-term entry identifier; it should be unique across all message store providers. To ensure this uniqueness, message store providers typically set their record key to a value that is the combination of their PR_MDB_PROVIDER property and an identifier that is unique to the message store.

A search key is used to compare the data in a message or an address book entry to data in other messages or address book entries. Folders and message stores do not have search keys. Each search key has a property tag that ends in SEARCH_KEY, such as PR_SEARCH_KEY. Because a search key represents an object's data and not the object itself, two different objects with the same data can have the same search key. When an object is copied, for example, both the original object and its copy have the same data and the same search. key.

The search key of a message is a required property, and message store providers are required to furnish it at message creation time. It can be modified at any time. The search key of an address book entry is computed from the address type (PR_ADDRTYPE) and address (PR_EMAIL_ADDRESS). If the address book entry is writeable, its search key might not be available until the address type and address have been set using **IMAPIProp::SetProps** and the entry saved using **IMAPIProp::SaveChanges**. When these address properties change, it is possible for the corresponding search key to be out of sync with the new values until the changes have been committed with a **SaveChanges** call.

The value of an object's record key can be the same as or different from the value of its search key, depending on the service provider. Some service providers use the same value for an object's search key, record key, and its entry identifier. Other service providers assign unique values for each of its objects' identifiers.

About Session Identity

Most MAPI sessions have one message service that provides the identity for the session. A message service provides identity through an object, typically an address book object. When a service provider supplies an object that can be used for session identity, it inserts the following entry in its section in the MAPISVC.INF configuration file:

```
PR_RESOURCE_FLAGS=STATUS_PRIMARY_IDENTITY
```

Service providers that cannot supply an identity insert this entry in MAPISVC.INF:

```
PR_RESOURCE_FLAGS=STATUS_NO_PRIMARY_IDENTITY
```

Multiple service providers within a message service and multiple message services within a profile can provide identity, as indicated by the presence of this flag in MAPISVC.INF. Message services publish whether or not they can supply an identity through one of their providers by including one of the following two entries in their MAPISVC.INF sections:

```
PR_RESOURCE_FLAGS=SERVICE_PRIMARY_IDENTITY
PR_RESOURCE_FLAGS=SERVICE_NO_PRIMARY_IDENTITY
```

Although it is possible for multiple providers and message services to supply primary identity, only a single provider and a single message service is selected to represent each session. Selection occurs when a profile is created or when a client calls **IMsgServiceAdmin::SetPrimaryIdentity**.

When a profile is created, MAPI designates the first message service to be configured that includes a provider with the STATUS_PRIMARY_IDENTITY flag set in its PR_RESOURCE_FLAGS property is chosen to supply the primary identity. Within the designated message service, the first provider to be configured that with this resource flag set is chosen to provide the identity for the service. The STATUS_PRIMARY_IDENTITY flag is cleared for all other providers in the profile. If at any time the provider supplying primary identity is removed from the profile, the next provider to be configured that can supply identity takes over the role.

When a client calls **IMsgServiceAdmin::SetPrimaryIdentity**, it specifies the **MAPIUID** for a service provider within the target service. The service provider represented by the MAPIUID is designated to supply the primary identity for the message service and for the session, and all of the other providers in the service will share this identity.

When a message service is selected to provide the primary identity for a session, all service providers within the message service share the same identity. Every provider in the service updates its row in the status table to include three properties, set to the display name, search key, and entry identifier of the primary identity object. These three properties are:

PR_IDENTITY_DISPLAY
PR_IDENTITY_SEARCH_KEY
PR_IDENTITY_ENTRYID

To retrieve the entry identifier for the primary identity, clients call the **IMAPISession::QueryIdentity** method. **QueryIdentity** searches the status table for the row that contains the value STATUS_PRIMARY_IDENTITY in its PR_RESOURCE_FLAGS column and returns the corresponding PR_IDENTITY_ENTRYID as the entry identifier for the primary identity.

About Managing Memory

Knowing how and when to allocate and free memory is an important part of programming with MAPI. MAPI provides both functions and macros that your client or service provider can use to manage memory in a consistent way. The three functions are:

MAPIAllocateBuffer
MAPIAllocateMore
MAPIFreeBuffer

When clients and service providers use these functions, the issue of who "owns" (that is, knows how to release) a particular block of memory disappears. A client making a service provider method call need not pass a buffer large enough to hold a return value of any size. The service provider can simply allocate the appropriate amount of memory and the client can later release it at will, independent of the service provider.

The memory macros are used to allocate structures or arrays of structures of a specific size. Clients and service providers should use these macros rather than allocate the memory manually. For example, if a client needs to perform name resolution processing on a recipient list with three entries, the **SizedADRLIST** macro can be used to create an **ADRLIST** structure to pass to **IAddrBook::ResolveName** with the correct number of **ADRENTRY** members. All of the memory macros are defined in the MAPIDEFS.H header file.

MAPI also supports the use of the OLE interface, **IMalloc**, for memory management. Service providers are given an **IMalloc** interface pointer by MAPI at initialization time and can also retrieve one through the **MAPIGetDefaultMalloc** function. The main advantage to using the **IMalloc** methods for managing memory over the MAPI functions is that with the OLE methods it is possible to reallocate an existing buffer. The MAPI memory functions do not support reallocation.

About Memory Management Functions

MAPIAllocateBuffer, **MAPIAllocateMore**, and **MAPIFreeBuffer** are the three functions that enable client applications and service providers to exchange blocks of memory. **MAPIAllocateBuffer** is called to initially allocate memory for a buffer; **MAPIAllocateMore** can allocate a subsequent block of memory that is linked to the initial block at a later time. Assume when calling these allocators that the returned buffer is appropriately aligned for the CPU architecture.

The chaining of allocations supported by **MAPIAllocateMore** simplifies the handling of complex memory objects for clients. An array of property values, for example, can consist of numerous blocks of memory linked together by pointers that can be released with a single call to the third function, **MAPIFreeBuffer**. Whenever a block of memory is returned from any MAPI method, such as **IMAPIProp::GetProps**, **MAPIFreeBuffer** must be used to release it. **MAPIFreeBuffer** releases the initial block and any subsequent blocks.

Whenever possible, allocate all of the necessary memory with a single call to **MAPIAllocateBuffer**. **MAPIAllocateMore** exists merely as a convenience. However, if **MAPIAllocateMore** is used to allocate additional memory for a buffer, this memory should always be freed with the initial buffer. Ignore any errors returned from **MAPIFreeBuffer**. The function almost always succeeds, and in the rare case that an error is returned, there is little that the caller can do about it.

Note Clients call the **MAPIAllocateBuffer** and **MAPIAllocateMore** functions directly, while service providers must make an indirect call, retrieving the function pointers through a call to **IMAPISupport::GetMemAllocRoutines**.

About Managing Memory for ADRLIST and SRowSet Structures

The recommendation of allocating all memory for a buffer whenever possible with a single **MAPIAllocateBuffer** call does not apply when using the address list, or **ADRLIST**, and row set, or **SRowSet**, data structures. These two structures are exceptions to the standard rules for allocating and releasing memory. They contain multiple levels of structures and are designed to enable individual members to be added or removed. Therefore, each property is a separate allocation. Where most structures are freed with one call to **MAPIFreeBuffer**, each individual entry in an **ADRLIST** or **SRowSet** structure must be freed with its own call to **MAPIFreeBuffer** or a single call to either **FreePRows** or **FreePadrlist**. **FreePRows** and **FreePadrlist** are API functions provided by MAPI for simplifying the freeing of these data structures. **FreePadrlist** frees the memory for the **ADRLIST** structure plus all associated memory for the structure members; **FreePRows** does the same for the **SRowSet** structure.

The following diagram shows the layout of an **ADRLIST** data structure, indicating the separate memory allocations required. The gray boxes show memory that can be allocated and released with one call.

{ewc msdncd, EWGraphic, groupx834 0 /a "MAPI_52.WMF"}

About Allocating and Freeing Memory

In addition to specifying how to allocate and free memory, MAPI defines a model for knowing when memory passed between public interface method and API function calls should be freed. The model applies only to memory allocated for parameters that are not pointers to interfaces, such as strings and pointers to structures. Interface pointers use the reference counting mechanism implemented through **IUnknown**. When allocating and freeing non-MAPI related memory internally within a client application or service provider, use whatever mechanism makes sense.

The model defines parameters as one of three types. They can be input parameters, set by the caller with information to be used by the called function or method, output parameters, set by the called function or method and returned to the caller, or input-output parameters, a combination of the two types. Output parameters are frequently pointers to data or pointers to pointers to data. Although the called function is responsible for allocating the data for output parameters, the caller allocates the memory for the pointer.

The rules for allocating and releasing memory for these types of parameters are explained in the following table.

Type	Memory allocation	Memory release
Input	Caller is responsible and can use any mechanism.	Caller is responsible and can use any mechanism.
Output	Called function is responsible and must use **MAPIAllocateBuffer**.	Caller is responsible and must use **MAPIFreeBuffer**.
Input-output	Caller is responsible for the initial allocation and called function can reallocate if necessary using **MAPIAllocateBuffer**.	Called function is responsible for initial freeing if reallocation is necessary. Caller must free the final return value.

During failure conditions, implementors of interface methods need to pay attention to output and input-output parameters since the caller generally has no way to clean them up. If an error is returned, then each output or input-output parameter must either be left at the value initialized by the caller or set to a value that can be cleaned up without any action on the part of the caller. For example, an output pointer-parameter of `void ** ppv` must be left as it was on input or can be set to NULL (`*ppv = NULL`).

About Structured Storage

Structured storage refers to the hierarchical organization of storage introduced with OLE. Storage is organized into two or three types of objects:

- Stream objects
- Lock bytes objects
- Storage objects

Stream and lock bytes are lower level objects that directly access the data. Stream objects implement the **IStream** interface which defines methods for reading, writing, positioning, and copying bytes of data. Lock bytes objects implement another OLE interface, **ILockBytes**, to access data with a byte array. Byte arrays are typically used to provide customized access to underlying storage.

Storage objects are layered on top of the stream or lock byte objects; they can contain one or more of these objects as well as other storage objects. Storage objects implement the **IStorage** interface which defines methods for creating, accessing, and maintaining nested objects.

Because **IStream**, **ILockBytes**, and **IStorage** are OLE interfaces rather than MAPI interfaces, their methods return OLE error values rather than MAPI values. Clients and service providers calling methods in these interfaces must use the API function **MapStorageSCode** to translate these values into MAPI error values.

Clients and service providers use structured storage for working with properties that are too large to maintain with the **IMAPIProp** methods, typically large string and binary properties. One of the common ways that clients or service providers access them is by specifying **IStream** or **IStorage** as the interface identifier in a call to the **IMAPIProp::OpenProperty** method. For example, clients call **OpenProperty** with PR_ATTACH_DATA_BIN as the property tag and IID_IStream as the interface identifier to access a binary attachment in a message.

Clients and service providers can implement their own stream and storage objects or call API functions to access implementations supplied by MAPI or OLE. Because the supplied implementations serve most purposes, clients and service providers rarely need to create their own.

When a client calls **OpenProperty** on a MAPI object to access one of its properties through a storage object, the service provider will typically open the storage object in direct mode. However, this is typical rather than required behavior. Clients should assume that all storage objects opened or created by service providers are transacted and require a call to **IStorage::Commit**. They should also remember that changes to storage objects will not be made permanent until they call **IMAPIProp::SaveChanges** after the final **Commit** to save the MAPI object.

MAPI and OLE provide several API functions for defining or accessing storage and stream objects. The commonly used functions are described in the following table.

Function	Description
CreateDocfile	Creates a general purpose storage object.
HrIStorageFromStream	Creates a storage object to access a stream or lock bytes object.
OpenIMsgOnIStg	Creates a message object to access a storage object.
OpenStreamOnFile	Creates a stream object to access a file.
WrapCompressedRTFStream	Creates a stream object that contains the compressed or uncompressed version of a

stream holding the rich text of a message.

CreateDocfile is used frequently in OLE and in MAPI to create a storage object. For more information on this function, refer to the OLE documentation.

To create a storage object that accesses an existing stream or lock bytes object, clients call **HrIStorageFromStream**.

To create a message object that accesses an existing storage object, service providers and clients call **OpenIMsgOnIStg**. The message object that is created differs from the typical message objects created by message store providers in that it does not support all of the **IMessage** interface methods. For example, service providers and clients cannot call **IMessage::SubmitMessage** on this message.

An optional input parameter to **OpenIMsgOnIStg** is a callback function that conforms to the **MSGCALLRELEASE** prototype. This function is called by the new message object when its reference count reaches zero. Implementing a **MSGCALLRELEASE** function can be useful for performing some final processing before the new message is completely removed.

OpenStreamOnFile is commonly used for storing file attachments because it creates a stream that reads from and writes to a file. **OpenProperty** with PR_ATTACH_DATA_BIN as the property tag creates a stream for storing binary attachment data.

To compress or uncompress a stream containing the Rich Text Format message text, clients call **WrapCompressedRTFStream**. **WrapCompressedRTFStream** creates a stream that wraps the RTF stream. The wrapper stream does not implement all of the **IStream** methods; the following methods are excluded:

Seek
SetSize
Revert
LockRegion
UnlockRegion
Stat
Clone

Because the stream objects created by **WrapCompressedRTFStream** do not support either **SetSize** or **Stat**, there is not an easy way to either extend or retrieve their size. The best strategy is to pick a reasonable buffer size and read or write in a loop.

Note OLE has a storage object implementation based on a byte array that returns an **IEnumSTATSTG** object from the **EnumElements** method that is problematic.In particular, the **QueryInterface** method does not work correctly. Service providers that implement their own storage objects using OLE's implementation should create a thin wrapper for the **IEnumSTATSTG** object that forwards calls on to the underlying **IEnumSTATSTG** methods but implements its own **AddRef**, **Release**, **QueryInterface**, and **Clone** methods.

About Error Handling

Success, warning, and error values are returned using a 32-bit number known as a result handle, or HRESULT. An HRESULT is really not a handle to anything; it is merely a 32-bit value with several fields encoded in the value. A zero result indicates success and a nonzero result indicates failure.

HRESULT values work differently depending on the platform your client or service provider is using. On 16-bit platforms, an HRESULT is generated from a 32-bit value known as a status code, or SCODE. On 32-bit platforms, an HRESULT is the same as an SCODE; they are synonymous data types. MAPI on 32-bit platforms works solely with HRESULT values.

SCODES on 16-bit platforms are divided into four fields: a severity code, a context field, a facility field, and an error code. The format of an SCODE on a 16-bit platform is shown below; the numbers indicate bit positions.

{ewc msdncd, EWGraphic, groupx834 1 /a "MAPI_48.WMF"}

HRESULTs on 32-bit platforms have the following format.

{ewc msdncd, EWGraphic, groupx834 2 /a "MAPI_49.WMF"}

The severity code in the 16-bit SCODE and the high order bit in the HRESULT indicates whether the return value represents success or failure. If set to zero, the value indicates success. If set to 1, it indicates failure.

In the 16-bit version of the SCODE, the context field is reserved as are the R, C, N, and r bits in the HRESULT.

The facility field in both versions indicates the area of responsibility for the error. There are several facilities, but the vast majority of MAPI errors use FACILITY_ITF to represent interface errors. The most common facilities that are currently used are: FACILITY_NULL, FACILITY_ITF, FACILITY_DISPATCH, FACILITY_RPC, and FACILITY_STORAGE. If new facilities are necessary, Microsoft allocates them because they need to be unique. The following table describes the various facility fields.

Facility	**Description**
FACILITY_NULL	For broadly applicable common status codes such as S_OK or E_OUTOF_MEMORY; the value is zero.
FACILITY_ITF	For most status codes returned from interface methods; the value is defined by the interface. That is, two SCODE or HRESULT values with exactly the same 32-bit value returned from two different interfaces might have different meanings.
FACILITY_DISPATCH	For late binding **IDispatch** interface errors.
FACILITY_RPC	For status codes returned from remote procedure calls.
FACILITY_STORAGE	For status codes returned from **IStorage** or **IStream** method calls relating to structured storage. Status codes with code (lower 16 bits) values in the range of MS-DOS error codes (that is, less than 256) have the same meaning as the corresponding MS-DOS errors.

The code field is a unique number that is assigned to represent the error or warning.

Return Value Naming Convention

The MAPICODE.H header file contains many of the values that a client or service provider might return from an interface method implementation or might see returned from a call. These values are SCODE values rather than HRESULT values. The OLE SCODE value S_OK is used to represent success in MAPI calls.

The codes to represent warning and failure conditions follow a different naming convention that begins with the prefix MAPI, an underscore, and either a W or an E to indicate the type of code. The rest of the code is a short character string to describe the condition. Each word in the string is separated by an underscore. For example, the error value MAPI_E_TOO_COMPLEX indicates that the implementation could not handle whatever was being requested in the call. The warning value MAPI_W_PARTIAL_COMPLETION indicates that the call succeeded, but that there were problems. Only part of the operation was completed successfully.

Using Macros for Error Handling

MAPI defines several macros for making it easier to work with SCODE values on 16-bit platforms and HRESULT values on both platforms. Some of the macros and functions below provide conversion between the two datatypes and can be used in client and service provider code that:

- Runs only on 16-bit platforms.
- Runs on both 16-bit and 32-bit platforms.
- Is 16-bit code being ported to 32-bit platforms.

For 32-bit environments, these datatype conversion macros are meaningless. They exist to provide compatibility and to ease the porting effort.

There are two sets of macros that test for failure or success: HR_SUCCEEDED and SUCCEEDED and HR_FAILED and FAILED.

On 32-bit platforms, because SCODE and HRESULT values are the same, SUCCEEDED is the same as HR_SUCCEEDED and FAILED is the same as HR_FAILED. Developers that want to write cross-platform clients or service providers are the only ones that need to distinguish between an HRESULT and an SCODE.

Do not fall into the trap of confusing the two error types as is demonstrated in the following code sample:

```
HRESULT hr = S_OK;
return hr;
```

Instead, use the **ResultFromScode** macro to set the HRESULT variable to the corresponding HRESULT value for S_OK.

Warning The macro SCODE_FACILITY retrieves twelve bits from the SCODE in its Win16 implementation when it should only retrieve four. This can cause false negative results to occur when comparing the facility retrieved from a particular SCODE against predefined facility values. Programmers writing cross-platform code need to be aware of this problem.

Commonly used macros are briefly described in the following table.

Macro	Description
GetScode	Returns an SCODE given an HRESULT.
ResultFromScode	Returns an HRESULT given an SCODE.
MAKE_SCODE	Constructs an SCODE from its components.
HR_SUCCEEDED	Tests an HRESULT for a success or warning condition.
HR_FAILED	Tests an HRESULT for an error condition.
HRESULT_CODE	Extracts the error code part of the HRESULT.
HRESULT_FACILITY	Extracts the facility from the HRESULT.
HRESULT_SEVERITY	Extracts the severity bit from the SEVERITY.
SCODE_CODE	Extracts the error code part of the SCODE.
SCODE_FACILITY	Extracts the facility from the SCODE.
SCODE_SEVERITY	Extracts the severity field from the SCODE.
SUCCEEDED	Tests an SCODE for a success or warning condition.
FAILED	Tests an SCODE for an error condition.

Note Calling **ResultFromScode** for S_OK verification carries a performance penalty. Your client or service provider should not routinely use **ResultFromScode** for successful results. **GetScode** and **ResultFromScode** are externally defined and not MAPI support functions. They are not in the table of support functions passed to service providers, but in a separate DLL that can be linked with providers.

How Return Values are Documented

The reference entries in the *MAPI Programmer's Reference* document only those return values that require some handling by client applications or service providers. Return values that indicate common error conditions and can be deduced by checking for failure are not included in the documentation. For example, many interface methods can return MAPI_E_INVALID_PARAM if a caller specifies the wrong value for an input parameter. This value is typically not listed in the set of expected return values because there is no need to look specifically for MAPI_E_INVALID_PARAMETER and no need to process it differently from any other error. On the other hand, some service providers do not support event notification and will return MAPI_E_NO_SUPPORT to the **Advise** method made by clients through **IMAPISession**. Because clients need to explicitly check for this value and provide code for handling the condition that it represents should it occur, MAPI_E_NO_SUPPORT is included in the list of return values for **IMAPISession::Advise**.

The following table describes error values that are commonly returned from methods and functions and require explicit handling on the part of a client or service provider. These values fall into four categories: values that indicate invalid input data, values that indicate resource problems, values that indicate character set incompatibility, and values that indicate failure from an unknown origin.

Return value	**Description**
MAPI_E_INVALID_PARAMETER	One or more of the parameters passed into the method or functions were not valid.
MAPI_E_UNKNOWN_FLAGS	One or more values for a flags parameter were not valid.
MAPI_E_DISK_ERROR	There was a problem writing to or reading from disk.
MAPI_E_NOT_ENOUGH_DISK	Not enough disk space was available to complete the operation.
MAPI_E_NOT_ENOUGH_MEMORY	Not enough memory was available to complete the operation.
MAPI_E_NOT_ENOUGH_RESOURCES	Not enough system resources were available to complete the operation.
MAPI_E_BAD_CHARWIDTH	An incompatibility exists in the character sets supported by the caller and the implementation.
MAPI_E_CALL_FAILED	An error of unexpected or unknown origin occurred.

Errors regarding invalid data passed in by a caller can be determined through either the parameter validation API functions provided by MAPI or a set of macros. For more information on how to use the MAPI parameter validation model in your programs, see Validating Parameters to Interface Methods.

Character set incompatibility arises when either of the following situations occurs:

- A client or service provider sets the MAPI_UNICODE flag on a method or function call and the implementation does not support Unicode. Setting MAPI_UNICODE indicates that character strings passed in as input are Unicode strings and that character strings passed back as output are expected to be Unicode strings.

- A client or service provider does not set the MAPI_UNICODE flag on a method or function call and the implementation only supports Unicode.

Strategies for Error Handling

Because interface methods are virtual, it is not possible to know, as a caller, the full set of values that can be returned from any one call. One implementation of a method might return five values; another might return eight. The reference entries in the *MAPI Programmer's Reference* list a few values that can be returned for each method; these are the values that your client or service provider can check for and handle because they have special meanings. Other values can be returned, but since they are not meaningful, special code to handle those is not necessary. A simple check for success or failure is adequate.

A few of the interface methods return warnings. If a method that your client or service provider calls can return a warning, use the **HR_FAILED** macro to test the return value rather than a check for zero or nonzero. Warnings, although nonzero, differ from error codes in that they do not have the high bit set. If your client or service provider does not use the macro, it is likely that a warning might be mistaken for a failure.

Although most interface methods and functions return HRESULT values, some functions, primarily those defined for Simple MAPI, return unsigned long values. Also, some methods used in the MAPI environment come from OLE and return OLE error values rather than MAPI error values. Keep in mind the following guidelines when making calls:

- Never rely on or use the return values from **IUnknown::AddRef** or **IUnknown::Release**. These return values are for diagnostic purposes only.
- **IUnknown::QueryInterface** always returns generic OLE errors where the facility is FACILITY_NULL or FACILITY_RPC, rather than MAPI errors.
- All other interface methods return MAPI interface errors with a facility of FACILITY_ITF, or FACILITY_RPC or FACILITY_NULL errors.
- Simple MAPI calls return Simple MAPI errors rather than SCODE or HRESULT values.
- CMC calls have their own defined set of error codes. CMC codes are unsigned long values.
- No MAPI call ever returns a Simple MAPI error value.

If your service provider works with clients that use both the Simple MAPI and MAPI client interfaces, be aware that although the underlying values for some of the errors are different, the constants defined for them are quite similar. For example, when a call is made to an unsupported feature in Simple MAPI, the error MAPI_E_NOT_SUPPORTED is returned. When the same type of call is made to a MAPI method, one of four possible errors can be returned: MAPI_E_NO_SUPPORT, MAPI_E_INTERFACE_NOT_SUPPORTED, MAPI_E_INVALID_PARAMETER, or MAPI_E_VERSION. For this reason, avoid including the Simple MAPI header file, MAPI.H, in your client or service provider. The table below points out these similarities and differences.

Simple MAPI return value	MAPI return value
MAPI_E_NOT_SUPPORTED	MAPI_E_NO_SUPPORT MAPI_E_INTERFACE_NOT_SUPPORTED MAPI_E_INVALID_PARAMETER MAPI_E_VERSION
MAPI_E_DISK_FULL	MAPI_E_NOT_ENOUGH_DISK
MAPI_E_NETWORK_FAILURE	MAPI_E_NETWORK_ERROR
MAPI_E_USER_ABORT	MAPI_E_USER_CANCEL
MAPI_E_ACCESS_DENIED	MAPI_E_NO_ACCESS
MAPI_E_AMBIGUOUS_RECIPIENT	MAPI_E_AMBIGUOUS_RECIP

Using Extended Errors

Implementors of interface methods can choose to simply return success (S_OK) and failure (MAPI_E_CALL_FAILED) or differentiate between error conditions, returning as many error values as make sense for the situation. Most situations can use one of the error values defined by MAPI in the MAPICODE.H header file. However, for situations that are not covered by a predefined code, the value MAPI_E_EXTENDED_ERROR can be used. MAPI_E_EXTENDED_ERROR indicates to the caller that more information about the error is available. The caller retrieves the additional information by calling the **GetLastError** method on the same object that returned MAPI_E_EXTENDED_ERROR.

GetLastError can be called to retrieve information about any error code, not only MAPI_E_EXTENDED_ERROR. Many MAPI objects implement interfaces that include the **GetLastError** method. **GetLastError** returns a single **MAPIERROR** structure that, in theory, includes a concatenation of all of the errors generated by the previous method call. As a caller, it is wise not to depend on having this extra error information available because object implementors are not required to provide it. However, it is strongly recommended that whenever implementors return MAPI_E_EXTENDED_ERROR, they make it possible for callers to retrieve a **MAPIERROR** structure with useful information about the error.

Because **GetLastError** is also an API function that is part of the Win32 SDK, it can be easy to forget that in MAPI, **GetLastError** is an interface method and can only be called on MAPI objects. Another easy mistake to make is calling **GetLastError** on the wrong object. **GetLastError** must be called on the object that generated the error. For example, if your client makes a session call, and MAPI forwards the call to a service provider to do the work, your client should not call **GetLastError** on the service provider object. **IMAPISession::GetLastError** is the correct call; **GetLastError** should be invoked on the session object.

Deferring Errors

Some interface methods accept the MAPI_DEFERRED_ERRORS flag as an input parameter. When this flag is set, the method does not have to return immediately with a value; it can let the caller know the result of the call at some later time.

Deferring errors helps service providers in their implementation of complex methods, making processing faster. Rather than handling many requests and returning a value for each, deferring errors allows the calls to be bundled within the service provider. Processing many requests at once cuts down on network traffic, thereby improving performance. Deferring errors is especially useful in calls to delete or copy properties, which can be very time-consuming operations.

When a client makes a call without setting the MAPI_DEFERRED_ERRORS flag that can only be handled in a deferred manner, service providers can either defer the errors regardless or return MAPI_E_TOO_COMPLEX. Most clients should defer errors as a preferable strategy to failing the call.

Setting the MAPI_DEFERRED_ERRORS flag changes a client's error handling implementation in that the returned information can be delivered at any time rather than at a planned time. It is possible for an error to be returned when it is too late to do anything about it or after data about the original request is no longer available. For example, if a client calls **IMsgStore::OpenEntry** to open a deleted folder with MAPI_DEFERRED_ERRORS set, the client will not know of the problem until an **IMAPIProp::GetProps** call is made to retrieve one of the folder's properties. **GetProps** will then return MAPI_E_NOT_FOUND.

About Validating Data

Implementors and users of interface methods need to validate all types of data from simple integer values to object pointers. Depending on the source of the data and its type, MAPI provides guidelines for validation. Standard guidelines work to ensure consistent and accurate validation, an important feature that helps to promote interoperability between MAPI components.

Validating Parameters to Interface Methods

Interface method implementors can perform two types of validation:

- Debug validation whereby the implementor displays the values of the input parameters and provides verbose information should there be errors, optionally calling a macro provided by MAPI.
- Full validation whereby the implementor calls a macro provided by MAPI to check that all necessary parameters are present and set to valid values.

The type of validation performed depends on the caller. When the caller is MAPI or a service provider, debug validation is adequate because MAPI and service providers are expected to pass parameters correctly. When the caller is a client, however, full validation is recommended. Service providers cannot and should not count on clients to always pass appropriate parameters.

MAPI performs debug validation in those interface implementations that are called internally or by service providers and full validation in the implementations that are called by clients. All functions and the utility interfaces, **ITableData** and **IPropData**, perform debug validation.

For every interface method, there are three macros that clients and service providers can call in their implementations. Clients and providers can use one macro for debug validation and the other two for full validation. One of the macros for full validation is called for methods that return HRESULT values and the other is called for methods that return unsigned long values.

These macros conform to a three part naming convention with each part separated by an underscore. The prefix identifies the macro being called, as described in the following table.

Macro name prefix	Description
Validate	Identifies a macro to be used for fully validating methods that return HRESULT values.
UlValidate	Identifies a macro to be used for fully validating methods that return unsigned long values.
CheckParameters	Identifies a macro to be used for performing debug validation.

The other two parts identify the interface and method being validated. For example, to fully validate **IMAPIProp::GetProps**, a method that returns an HRESULT value, service providers call **Validate_IMAPIProp_GetProps**. To perform debug validation for this method, service providers call **CheckParameters_IMAPIProp_GetProps**.

The parameters to these macros varies slightly depending on whether the call is being made in C or C++. As with calls made to interface methods, the first parameter in C must be a pointer to the object implementing the method. In C++, the *this* pointer should be the first parameter. The rest of the parameters are the same in either language; they are the same parameters that are passed to the method being validated. Therefore, the number and type of parameters passed to a validation macro varies depending on the method being validated. All parameters passed to the method must be passed to a validation macro.

The three types of validation macros are platform independent; they are guaranteed to work on both RISC and Intel platforms. Their predecessors, the **ValidateParameters**, **UlValidateParameters**, and **CheckParameters** functions, perform the same functionality as the macros. However, these functions are available only for Intel platforms. Clients and service providers operating on RISC platforms must use the validation macros.

The **ValidateParameters** and **UlValidateParameters** macros and functions perform the following tasks:

- Examine the validity of read and write pointers
- Examine the validity of structures
- Examine the validity of reserved flags
- Examine the incoming size of parameters if there is a size limit

The **CheckParameters** macro and function are intended to be used by service providers in method implementations that are called by MAPI. They do not perform any real checking of parameters; they supply assert statements. Because the parameters passed by MAPI should be assumed to be correct, service providers do not need to perform full validation in these methods.

The following code samples illustrate how to call the **ValidateParameters** macro in C and C++. The method being validated is **IMAPITable::QueryRows**, a method implemented by service providers that returns an HRESULT value and requires three parameters: the number of rows that are requested, a bitmask of flags, and the address of a pointer to an **SRowSet** structure.

For C, the validation occurs as follows:

```
STDMETHODIMP Table_QueryRows(LPTABLE lpTable, LONG lcRows,
                             ULONG ulFlags, LPSRowSet FAR * lplprows)
{
    Validate_IMAPITable_QueryRows(lpTable, lcRows, ulFlags, lplprows);

    // rest of method implementation
}
```

For C++, this example would be written as follows:

```
STDMETHODIMP CTable::QueryRows(LONG lcRows, ULONG ulFlags,
                               LPSRowSet FAR * lplprows)
{
    Validate_IMAPITable_QueryRows(this, lcRows, ulFlags, lplprows);

    // rest of method implementation
}
```

Validating Object Pointers

The need to validate object pointers arises when calls are made to retrieve MAPI objects or when private method calls are made within your client or service provider. It is always a good idea to validate these pointers. However, unlike parameter validation, MAPI does not provide a single API function to use. Instead there is a recommended series of steps to ensure that object pointers given to your client or service provider are valid. Not all steps are appropriate for all situations; the vtable verification step, for example, is unnecessary in C++. These steps are as follows:

1. Check that the object pointer points to the correct amount of writeable memory.
2. Check that the vtable of the object contains the expected number of readable entries.
3. Check that one or more methods in the vtable have the expected address.
4. Check that the object's reference count is nonzero.

MAPI provides two API functions for validating pointer memory: **IsBadWritePtr** and **IsBadReadPtr**. The following C code sample illustrates how to validate an object pointer using all of the steps outlined above and these two API functions.

```
if (IsBadWritePtr(lpMyObject, sizeof(MYOBJECT)))
    return failure

if (IsBadReadPtr(lpMyObject->lpVtbl, size(MYOBJECT_Vtbl)))
    return failure

if (lpMyObject->lpVtbl->SetProps != MYOBJECT_SetProps)
    return failure

if (lpMyObject-> cRef == 0)
    return failure
```

Validating Data Structures

Standard MAPI data structures are often passed between methods. MAPI provides several API functions to check the validity of these structures. The following table describes the common functions used by clients and service providers to validate MAPI data structures.

Validation function	Purpose
FBadColumnSet	Validates an array of property tags.
FBadEntryList	Validates a list of entry identifiers.
FBadProp	Validates a single property value.
FBadPropTag	Validates a single property tag.
FBadRestriction	Validates a restriction.
FBadRglpNameID	Validates an array of name identifiers.
FBadRglpszW	Validates an array of Unicode strings.
FBadRow	Validates a single row in a table.
FBadRowSet	Validates a set of rows in a table.
FBadSortOrderSet	Validates a sort order for a table.

About Notification

Event notification is the communication of information between two MAPI objects. Through one of the objects, a client or service provider registers an interest in learning of a change or error, called an event, that may take place in the other object. After the event occurs, the first object is informed, or notified. The object receiving the notification is called the advise sink; the object responsible for the notification is called the advise source.

There are three types of advise sink objects; all types are standard MAPI objects:

- Advise sink objects
- Form advise sink objects
- View advise sink objects

Advise sink objects are the most common type. Advise sinks are typically implemented by client applications to receive address book and message store notifications and support the **IMAPIAdviseSink** interface. **IMAPIAdviseSink** contains a single method, **OnNotify**. Form and view advise sinks are less common, they are implemented to receive notifications about changes to custom forms. Form advise sinks support the **IMAPIFormAdviseSink** interface and view advise sinks support the **IMAPIViewAdviseSink** interface. Because most clients implement standard advise sink objects, assume that discussions of notifications relate to address book and message store notifications rather than forms notifications. For more information about forms notifications, see About Forms Notifications and Writing Form Server Code.

Advise source objects are implemented by service providers and by MAPI. Not all service providers support event notification; it is optional, but strongly recommended. Message store and address book providers usually support object notifications on several of their objects and table notifications on their contents and hierarchy tables. Transport providers do not support notifications directly; they rely on alternative methods of communication with clients.

Unlike advise sinks, advise source objects are not a unique type of MAPI object. Many MAPI objects, such as message stores and tables, can take on the role of advise source. An advise source is any MAPI object that:

- Implements an **Advise** method to receive notification registrations.
- Implements an **Unadvise** method to receive notification cancellations.
- Generates notifications of the appropriate type to the appropriate advise sink objects that have registered by calling their **IMAPIAdviseSink::OnNotify** methods.

Clients implementing advise sink objects call **Advise** when they want to register for a notification, in most cases passing in the entry identifier of the object with which registration should occur, and **Unadvise** when they want to cancel the registration. Clients pass a parameter to Advise that indicates which of the several types of events they want to monitor. **Advise** returns a nonzero number that represents a successful connection between the advise sink and advise source.

Before calling **Advise**, clients can determine if a message store provider supports notification by checking that the STORE_NOTIFY_OK flag is set in the message store's PR_STORE_SUPPORT_MASK property. There is no way for clients to determine ahead of time whether or not an address book provider supports notifications. Clients must attempt to register and if the attempt fails, they can assume notifications are unsupported.

When an event for which a client has registered occurs, the advise source notifies the advise sink by calling its **IMAPIAdviseSink::OnNotify** method with a notification data structure that contains information about the event. An advise sink's implementation of **OnNotify** can perform tasks in response to the notification, such as updating data in memory or refreshing a screen display.

Service providers can implement support for notifications manually or take advantage of the help provided in three **IMAPISupport** methods: **Subscribe**, **Unsubscribe**, and **Notify**. The **Subscribe** and

Unsubscribe methods handle notification registration and de-registration for providers; the **Notify** method handles sending notifications when appropriate.

To use the support object methods for notification registration, service providers call **IMAPISupport::Subscribe** in their **Advise** methods and pass to **Subscribe** the advise sink pointer that clients pass to **Advise**. If an entry identifier is passed as an input parameter to specify an advise source, service providers convert it to a binary key. **Subscribe** creates a unique connection number and it is this number that service providers return to clients. Service providers can release the client's advise sink object pointer at any time after the **Advise** call has completed.

When clients call **Unadvise** to cancel a registration, service providers either decrement the reference count on the client's advise sink pointer or call **Unsubscribe** to do the same.

When it is time to generate a notification, service providers perform any internal processing that relates to the notification and initializes a **NOTIFICATION** structure by setting all of its unused members to zero. This technique for initializing the NOTIFICATION structure can help clients create smaller, faster, and less error-prone **OnNotify** implementations.

The following diagram shows the communication between advise sink objects, advise source objects, and MAPI. MAPI is involved only when the advise source calls the **IMAPISupport** methods for notification support.

{ewc msdncd, EWGraphic, groupx834 3 /a "MAPI_51.WMF"}

About Notification Events

When clients register for event notification, they must specify one or more events of interest. The events that they can specify depend on the set of events that the intended advise source supports. There are nine types of notifications, each represented by a constant. Clients can register for eight of the nine types; the only exception is the status object notification. This is an internal MAPI notification; clients cannot register for it and service providers cannot generate it. The following table describes the types of events and the advise source objects that can support them. The event constant is included with the event type.

Event type	Description	Advise source objects
Critical error (*fnevCriticalError*)	A global error or event has occurred, such as a session shut down in progress.	Session, all types of message store and address book objects, table, status
Object modified (*fnevObjectModified*)	A MAPI object has changed.	Folders, messages, all types of address book objects
Object created *(fnevObjectCreated)*	A MAPI object has been created.	Folders, messages, all types of address book objects
Object moved *(fnevObjectMoved)*	A MAPI object has been moved.	Folders, messages, all types of address book objects
Object deleted *(fnevObjectDeleted)*	A MAPI object has been deleted.	Folders, messages, all types of address book objects
Object copied *(fnevObjectCopied)*	A MAPI object has been copied.	Folders, messages, all types of address book objects
Extended event *(fnevExtended)*	An internal event defined by a particular service provider has occurred.	Any advise source object
Search complete *(fnevSearchComplete)*	A search operation has finished and the results of the search are available.	Folders
Table modified *(fnevTableModified)*	Information in a MAPI table object has changed.	Tables
New mail (*fnevNewMail*)	A message has been delivered and is waiting4 to be processed.	Message store, folders

The extended event is defined by a service provider to represent an event that cannot be covered by any of the other pre-defined events. Only clients that know before they register that a service provider supports an extended event can register for that event. It is not possible for clients to determine without advanced knowledge if a service provider supports an extended event and, if it does, how to handle such an event when it is received.

About Forms Notifications

Registering for and handling notifications from form objects is a different process than for other MAPI objects. Advise sinks for form notifications implement either the **IMAPIViewAdviseSink** or **IMAPIFormAdviseSink** interface rather than **IMAPIAdviseSink**. **IMAPIViewAdviseSink** and **IMAPIFormAdviseSink** each have multiple methods, one for each of the possible events that the corresponding advise source is capable of generating. For example, **IMAPIFormAdviseSink** has two methods: **OnChange** to handle a change to the form viewer's status and **OnActivateNext** to display a new message with the correct form.

The event handling strategy for forms is similar to the event handling strategy implemented in OLE. Clients do not register for specific event types as they do for most MAPI objects. The assumption is that registering for notification enables them to receive any type of event that can be generated by the particular advise source. Because **IMAPIAdviseSink::OnNotify** must be written so as to handle all registered events, implementing it can be complex for clients that register for many different events. Because the methods in the form advise sink objects target a single event, implementing these methods is simpler.

About Threading in MAPI

A thread is the basic entity to which a 32-bit operating system allocates CPU time. A thread has its own registers, stack, priority, and storage, but shares an address space and process resources such as access tokens. Threads also share memory, with one thread reading what another thread has written.

MAPI clients and service providers use the following generic threading models.

Threading model	Description
Single threading model	All objects are used on the single thread.
Apartment threading model	An object can be used only on the thread that created it.
Free threading, or thread-party, model	An object can be used on any thread.

MAPI and MAPI service providers use the free threading model, supporting thread-safe objects that can be used on any thread at any time. The current version of OLE uses the apartment threading model. The apartment threading model supports objects that must be explicitly transferred when a thread other than the one that created the object needs to use that object.

The mechanism that OLE uses to transfer objects from one thread to another is known as marshalling. Marshalling involves a stub object and a proxy object. These special objects package the parameters of the interface in the object to be marshalled, transfer these parameters to the other thread, and unpackage them upon arrival. Conflict between the two multithreaded models arises when a free-threading MAPI object is sent to another process using OLE "lightweight" Remote Procedure Call, or LRPC. LRPC changes the object's semantics from free threading to apartment threading by interposing stub and proxy interfaces with apartment threading behavior between the object and its caller. Awareness of the situations in MAPI that lead to this conflict can help clients and service providers prevent problems from occurring.

A MAPI object can be accessed in the following ways:

- Through direct calls to its methods using an interface pointer returned by a service provider or MAPI linked to the client's process, such as the session object returned from **MAPILogonEx**.
- Through indirect calls to its methods using an interface pointer returned by any service provider, such as the folder object copied from another folder in **IMAPIFolder::CopyFolder**.
- Through a callback function, such as the **IMAPIAdviseSink::OnNotify** method passed to a service provider or to MAPI in an **Advise** call or the methods that can show progress on a lengthy operation.

Using Thread-Safe Objects

Client applications and service providers can assume that objects used directly or as callbacks are always thread-safe except in the following cases:

- A transport provider's status object obtained through a client call to **IMAPISession::OpenEntry** with an entry identifier from the provider's status table row.
- All MAPI form objects obtained through a client call to **MAPIOpenFormManager**. Form objects obey apartment model rules and clients must use them and all objects contained by them only on the thread that created them.

When a client accesses a transport provider's row in the status table that includes the entry identifier of the associated status object, the client can call **OpenEntry** with that entry identifier to open the status object. This status object is not thread-safe because transport providers run in the context of the MAPI spooler and do not maintain a separate context for their status object. The status object obeys apartment model rules and clients must use it only on the thread that created it.

A client must also invoke **MAPIInitialize** on every thread before using any MAPI objects and **MAPIUninitialize** when that use is complete. These calls should be made even if the objects to be used are passed to the thread from an external source. **MAPIInitialize** and **MAPIUninitialize** can be called from anywhere except from within a Win32 **DLLEntryPoint** function, a function that is invoked by the system when processes and threads are initialized and terminated, or upon calls to the **LoadLibrary** and **FreeLibrary** functions. Clients should not need to use **DLLEntryPoint** functions because the MAPI service provider interface has its own initialization and deinitialization entry points.

Indirect use objects should never be assumed to be thread-safe. Indirect use objects are returned by methods that require destination interface pointers as input parameters. Examples of such methods are **IMAPIProp::CopyTo** and **CopyProps**, **IMAPIFolder::CopyFolder** and **CopyMessage**, and **IMsgServiceAdmin::CopyMsgService**. If a service provider wants to call such an object from a thread other than the one on which it was passed, the provider is responsible for explicitly marshalling the object.

Implementing Thread-Safe Objects

With objects that are returned from interface method calls directly, it is the provider's responsibility to insure thread-safety. With callback objects, it is the client's responsibility. A service provider implements a thread-safe object by serializing access to shared data within the object, insuring that one thread does not inadvertently replace the work of another thread. A service provider can implement serialized access to data in three ways:

1. Provide every object with its own critical section, calling the Win32 API function **EnterCriticalSection** at the beginning of every method and **LeaveCriticalSection** at the end. Most of the samples in the MAPI SDK use this method. This option has two drawbacks: a high overhead and the possibility of deadlock. Deadlock can occur if, for example, the thread-safe object calls the MAPI support object or an object returned indirectly through one of the copy methods.
2. Use a single critical section for all objects, calling **EnterCriticalSection** when the provider is loaded and **LeaveCriticalSection** when the provider is unloaded. This option provides the most simplicity, but suffers in performance when used with multithreaded clients.
3. Create a small number of critical sections to be associated with crucial shared data structures. The data is isolated from the object, placed, for example, in memory or in a parent object. Access to the data is handled through an internal interface. This option offers the best balance between performance and simplicity.

A client can implement a thread-safe notification callback by calling the MAPI utility **HrThisThreadAdviseSink**. **HrThisThreadAdviseSink** transforms a non-thread-safe advise sink into a thread-safe one. For progress callbacks, there is no such utility at this time. A client can choose to use the MAPI thread-safe progress object or create one manually.

A thread-safe object might or might not also be thread-aware. A thread-aware object maintains a separate context for every thread that is using it. Service providers are not required to support thread-awareness in their thread-safe objects, although supporting thread-awareness can be useful in some situations. Two MAPI tables always provide their own context by definition. One table used on different threads does not and should not provide unique context.

A client can choose between receiving notifications on the same thread that was used for the **MAPIInitialize** call, on the same thread that was used for the **Advise** call, or on a separate thread owned by MAPI. To insure that notifications arrive on the same thread that was used to call **MAPIInitialize**, a client calls **MAPIInitialize** and passes zero in the *ulFlags* member of the **MAPIINIT_0** structure. Notifications are then delivered during the main message loop.

To receive notifications on the MAPI-owned thread, a client calls **MAPIInitialize** with the **ulFlags** member of the **MAPIINIT_0** structure set to MAPI_MULTITHREAD_NOTIFICATIONS. The **Advise** call is made with the client's advise sink object rather than a wrapped version.

To insure that notifications arrive on the same thread that was used to call **Advise**, a client calls **HrThisThreadAdviseSink** and passes the newly created wrapped advise sink to **Advise** rather than the original advise sink. **MAPIInitialize** can be called with either flag value.

About Profile Administration

Profile administration is an important part of MAPI because without valid profiles, logon cannot occur and a session cannot be established. Profile administration provides a means for adding, deleting, or changing information about the message services and service providers installed on the workstation, ensuring the validity of each profile using those services and providers.

Both clients and providers perform profile administration using interfaces implemented by MAPI. Some clients choose not to perform profile administration, relying instead on the applications that MAPI supplies, the Control Panel applet and the Profile Wizard.

Clients that perform profile administration can use the MAPI interfaces to:

- Create new profiles.
- Delete or copy existing profiles.
- Add message services to profiles.
- Access the message service and provider tables.
- Delete message services from profiles.
- Copy message services.
- Configure message services.
- Make modifications to profile sections belonging to service providers or message services, such as add a service provider to a message service.

One of the major restrictions to clients' abilities to perform profile administration is that MAPI does not allow them to open and modify profile sections that belong to service providers and message services. This is because service providers are encouraged to store private information such as credentials in their profile sections.

Service providers have a more limited use of the MAPI interfaces for profile administration. They make changes to profile sections that they own or that describe the message service to which they belong. They also provide one or more property sheets so that clients can view their configuration data and optionally make changes.

About Interfaces for Profile Administration

To support profile administration, MAPI provides the following interfaces:

IProfAdmin
IMsgServiceAdmin
IProviderAdmin)
IProfSect

The first two interfaces, **IProfAdmin** and **IMsgServiceAdmin**, are only used by clients. **IProfAdmin** allows clients to create, delete, copy, and rename profiles as well as set passwords, establish a default profile, and view the profile table. To retrieve an **IProfAdmin** pointer, clients call **MAPIAdminProfiles**.

IMsgServiceAdmin allows clients to make message service modifications within a particular profile. To access an **IMsgServiceAdmin** pointer, clients call **IMAPISession::AdminServices** or **IProfAdmin::AdminServices** and specify the name of the target profile. Messaging clients typically use the session object for access while configuration clients use the profile administration object.

With **IProfAdmin**, all tasks are performed by MAPI; message services are not involved. Conversely, most **IMsgServiceAdmin** methods require that MAPI start the message service to perform the requested task by calling its entry point function. To enhance performance when either interface would work, use **IProfAdmin** rather than **IMsgServiceAdmin**.

Both clients and service providers use the **IProviderAdmin** and **IProfSect** interfaces. **IProviderAdmin** enables changes to be made to the members of a particular message service in a profile and grants access to the provider table. To retrieve an **IProviderAdmin** pointer, clients call **IMsgServiceAdmin::AdminProviders** and pass the **MAPIUID** for the target message service. Service providers are given **IProviderAdmin** pointers by MAPI as an input parameter to their message service entry point functions.

IProfSect is a derivative of **IMAPIProp** that has no additional methods of its own. Its purpose is solely to manipulate the properties of a profile section. To retrieve an **IProfSect** pointer, clients call **IMAPISession::OpenProfileSection** or **IProviderAdmin::OpenProfileSection** and service providers call **IMAPISupport::OpenProfileSection** or **IProviderAdmin::OpenProfileSection**.

Note Message store providers can modify their profile sections by calling **IMAPISupport::ModifyProfile**.

The following diagram shows all of the different calls that can be made to provide access to a profile containing message service and service provider information.

{ewc msdncd, EWGraphic, groupx834 4 /a "MAPI_60.WMF"}

About Profile Administration Objects

Profile administration objects, or objects that implement the **IProfAdmin** interface, enable clients to create, copy, and delete profiles, and to change profile passwords.

To create a new profile, clients call **IProfAdmin::CreateProfile**. **CreateProfile** calls the entry point function for each message service to be added to the profile with MSG_SERVICE_CREATE set as the *ulContext* parameter. Clients pass one or more option flags that provide MAPI with information. For example, clients set the MAPI_DEFAULT_SERVICE flag if the new profile should contain the message services listed in the [Default Services] section of the MAPISVC.INF file. Clients set MAPI_DIALOG to enable interactive configuration of each message service using the service's property sheets. If a message service cannot display a user interface, it remains unconfigured.

To change a profile's password, clients call **IProfAdmin::ChangeProfilePassword**. **ChangeProfilePassword** enables clients to change profile passwords on 16-bit platforms only; it is unsupported on 32-bit platforms.

To delete a profile, clients call **IProfAdmin::DeleteProfile**. **DeleteProfile** calls the entry point function of every message service in the profile with the *ulContext* parameter set to MSG_SERVICE_DELETE. The calls to the entry point functions occur before the services are physically removed from the profile. If the profile to be deleted is currently being used, **DeleteProfile** will wait until a later safer time to delete it. The profile does not actually disappear until every client with an active session has disconnected.

To mark a profile as the default, clients call **IProfAdmin::SetDefaultProfile**. The default profile is the one that is used at logon time when a client calls the API function **MAPILogonEx** and sets the MAPI_USE_DEFAULT flag. To set a new default profile, **SetDefaultProfile** sets the PR_DEFAULT_PROFILE property for the new default profile and removes the setting for the previous default profile.

Note A word of caution about modifying profiles. There are no safeguards to protect a client from adversely modifying a profile that is in use. MAPI will prevent clients from deleting a profile in use, but will not prevent them from deleting every message service in it, causing every message service provider to stop and unpredictable results to occur.

About Message Service Administration Objects

Message service administration objects, or objects that implement the **IMsgServiceAdmin** interface, enable clients to:

- Create and reconfigure message services.
- Copy and rename message services.
- Display a message service's property sheet.
- Access the message service and provider tables.
- Access a provider administration object.
- Establish transport provider order.
- Establish a primary identity for the session.

Because clients are not allowed to directly access the profile sections that belong to service providers and message services, all message service administration calls cause MAPI to call the message service's configuration entry point function to perform the real work. Therefore, whenever possible, clients should call **IProfAdmin** methods rather than **IMsgServiceAdmin** methods because profile administration tasks can be completed without starting the message service.

To access a provider administration object, clients call **IMsgServiceAdmin::AdminProviders**. Provider administration objects allow clients to add or delete instances of service providers from a message service. Some message services do not allow this type of dynamic modification; whether or not it is supported is up to the message service.

To access the message service table, clients call **IMsgServiceAdmin::GetMsgServiceTable**. The message service table provides a view of properties for all of the message services in the current profile. Three of the most commonly used properties in the table are:

PR_SERVICE_UID
PR_SERVICE_NAME
PR_RESOURCE_FLAGS

PR_SERVICE_UID is the unique identifier for the message service; many profile administration methods require it as a parameter. PR_SERVICE_NAME can be used to programmatically identify the type of message service. It comes from the [Services] section in MAPISVC.INF and should never be changed or localized. PR_RESOURCE_FLAGS is a bitmask that describes the message service's capabilities.

Once a client calls **GetMsgServiceTable** to access a view of the message service table, any changes that occur to message services within the profile will not be reflected in the view.

To access another table, the provider table, clients call the **IMsgServiceAdmin::GetProviderTable** method. The provider table lists all of the service providers for a message service and includes the following properties in its column display:

PR_PROVIDER_DISPLAY
PR_PROVIDER_ORDINAL
PR_RESOURCE_TYPE
PR_RESOURCE_FLAGS

To add a new message service to a profile, clients call **IMsgServiceAdmin::CreateMsgService**. MAPI first copies all of the relevant information in the MAPISVC.INF file and then calls the message service's entry point function with the *ulContext* parameter set to MSG_SERVICE_CREATE. **CreateMsgService** creates a provider section in the profile for every provider section in MAPISVC.INF. Once the message service has been added, MAPI sets the service's PR_SERVICE_UID property. Clients must call **IMsgServiceAdmin::GetMsgServiceTable** to access it.

To configure a newly added message service, clients call **IMsgServiceAdmin::ConfigureMsgService**. Like many of the method calls in MAPI, **ConfigureMsgService** accepts a flag that specifies whether a user interface can be displayed. Clients can ask that property sheets never be displayed, that property sheets appear only if the service requires additional configuration, or have them appear regardless of whether additional configuration is required. **ConfigureMsgService** configures all of the service providers in a message service. To configure a single service provider, clients should call the provider's **IMAPIStatus::SettingsDialog** method.

To copy a message service to the same profile or a different profile, clients call **IMsgServiceAdmin::CopyMsgService**. When a message service is copied, the new instance of the service is configured in exactly the same way as the original. Sometimes **CopyMsgService** returns the error MAPI_E_ACCESS_DENIED. The most common cause of this error return is a message service that does not allow itself to be duplicated.

To delete a message service from a profile, clients locate its unique identifier in the message service table and call **IMsgServiceAdmin::DeleteMsgService**. **DeleteMsgService** calls the message service's entry point function with the *ulContext* parameter set to MSG_SERVICE_DELETE. Message services perform any clean up tasks at this time before they are removed from the profile.

To set a message service's primary identity, clients call **IMsgServiceAdmin::SetPrimaryIdentity**. Usually the entry identifier for an entry in an address book container serves as a message service's primary identity. Service providers that can supply an identity set the following status table properties:

PR_RESOURCE_FLAGS (to contain the STATUS_PRIMARY_IDENTITY flag)
PR_IDENTITY_DISPLAY
PR_IDENTITY_SEARCH_KEY
PR_IDENTITY_ENTRYID

Service providers that cannot supply an identity set the STATUS_NO_PRIMARY_IDENTITY flag in their PR_RESOURCE_FLAGS property and do not set the other three PR_IDENTITY properties.

To change the order in which transports are called to deliver messages, clients use the information in the provider table and the **IMsgServiceAdmin::MsgServiceTransportOrder** method. Transport order is used by MAPI to resolve conflicts when multiple transport providers register to handle addresses of the same type. Whenever clients make a change to the order, they pass in a list of all of the transport providers in the profile in the desired order. It is not possible to switch the position of a few transports by passing in only those few. A call to **MsgServiceTransportOrder** overrides any preferences a transport provider specifies, such as the existence of the STATUS_XP_PREFER_LAST flag in its PR_RESOURCE_FLAGS property.

About Profile Section Objects

Profile section objects, or objects that implement the **IProfSect** interface, are used by client applications and service providers to access and modify the properties of a service provider or message service profile section.

Every profile has a name, stored in its PR_PROFILE_NAME property, and a binary key, stored in its PR_SEARCH_KEY property. The profile name is a character string specified either by the client or by a user. The search key is set equal to a value defined in MAPIGUID.H as MUID_PROFILE_INSTANCE. Whereas it is possible for a profile's name to be ambiguous, such as when a profile is deleted and a new one is created with the same name, a profile's search key is guaranteed to always be unique among all profiles ever created. Clients and service providers using a profile section's search key should treat it as binary data rather than data of any particular type.

There are two major differences between profile section objects and other objects that inherit from **IMAPIProp**:

- Profile sections do not support transactions.
- Profile sections do not support named properties.

Because profile sections do not support transactions, any changes made with calls to **IMAPIProp::CopyProps**, **CopyTo**, or **SetProps** immediately take effect. Clients and service providers can call **IMAPIProp::SaveChanges** and it will succeed, but it has no affect on the profile section data. Service providers wanting to implement an undo feature in their property sheets should implement the property sheets with copies of their profile sections rather than the real sections. With copies, users and clients can make property changes and later cancel those changes, leaving the original profile sections untouched.

Because profile sections do not support the use of named properties, when clients or service providers call **IMAPIProp::GetIDsFromNames** or **GetNamesFromIDs**, these methods return MAPI_E_NO_SUPPORT.

Message store providers have some special rules in regards to their profile sections. They must maintain the PR_DISPLAY_NAME property for their message store both in the message store and in the profile section, keeping these two locations in sync. When the message store is created, its PR_DISPLAY_NAME property is set based on the value in the profile section. When the message store provider logs on or the message store is assigned a new display name, the profile section's PR_DISPLAY_NAME property is set based on the new message store property.

Message store provider profile sections must also include the message store's PR_RECORD_KEY and PR_ENTRYID properties, entered at the same time.

About Provider Administration Objects

Provider administration objects, or objects that implement the **IProviderAdmin** interface, are used by clients and service providers for managing service providers within a message service. Clients access provider administration objects by calling:

1. **IMAPISession::AdminServices** to access a message service administration object.
2. **IMsgServicesAdmin::AdminProviders** to access the provider administration object.

Service providers are given pointers to provider administration objects by MAPI when their message service entry point functions are called.

To access the provider table, clients and service providers call **IProviderAdmin::GetProviderTable**. The provider table lists information about all of the service providers currently installed in the message service. Clients and service providers can use the provider table to access the name of the provider DLL file, for example, or the **MAPIUID**, display name, and type of the provider as well as information about the message service.

To access a service provider's profile section, clients and service providers call **IProviderAdmin::OpenProfileSection**. **OpenProfileSection** uses the provider's **MAPIUID** to locate the correct profile section; callers can access this **MAPIUID** through the provider table. **OpenProfileSection** returns an **IProfSect** interface pointer.

IProviderAdmin::CreateProvider and **IProviderAdmin::DeleteProvider** are used to dynamically add or remove a service provider from a message service in the active profile. Many message services do not support this functionality, allowing configuration of this kind to occur only when the profile is not in use.

Working with Character Sets

MAPI-compliant client applications and service providers can use ANSI characters (single byte) or Unicode characters (double byte). OEM character sets are not supported; an OEM string passed to a MAPI method or function will cause that method or function to fail. Client applications that work with filenames in the OEM character set must be careful to convert them to ANSI before passing them to a MAPI method or function.

Supporting the Unicode character set is optional, both for clients and service providers. All service providers should write their code so that they can compile the same regardless of whether or not they support Unicode. Clients compile conditionally, depending on their level of support, but service providers do not. They should not have to be recompiled when the character set changes. Nothing in service provider code should be conditional.

When clients or service providers that support Unicode make a method call that includes character strings as input or output parameters, they set MAPI_UNICODE flag. Setting this flag indicates to the implementation that all incoming strings are Unicode strings. On output, setting this flag requests that all strings passed back from the implementation should be to be Unicode strings if possible. Method implementors that support Unicode will comply with the request; method implementors that do not provide Unicode support will not comply. String properties that are not in Unicode format are of type PT_STRING8.

MAPI defines the **fMapiUnicode** constant in the header file MAPIDEFS.H to represent the default character set. If a client or service provider supports Unicode, **fMapiUnicode** is set to MAPI_UNICODE. Clients and service providers that do not support Unicode set **fMapiUnicode** is set to zero.

Service providers that do not support Unicode should:

- Refuse to perform conversions between character sets.
- Never pass the MAPI_UNICODE flag in method calls.
- Return MAPI_E_BAD_CHARWIDTH when the MAPI_UNICODE flag is passed in.
- Declare ANSI string properties explicitly.

Service providers can also return MAPI_E_BAD_CHARWIDTH when they only support Unicode and clients do not pass the MAPI_UNICODE flag.

Note The current version of MAPI does not support Unicode. That is, clients and service providers can expect strings returned from any of the methods in interfaces implemented by MAPI to be ANSI character strings.

About Supporting Formatted Text

The text of a message can be stored and transmitted using a plain text or formatted text. Formatted text enhances the message text by altering its appearance with, for example, one or more fonts, font sizes, or text colors. It is recommended that all clients and whenever possible, all message store providers, support the formatted text. Supporting formatted text in messages adds value by improving message readability and making message handling easier and more efficient.

Formatted text can be implemented in a variety of ways. The most common way is with the Rich Text Format, or RTF. MAPI defines two transmittable properties for holding message text information: PR_BODY for plain text and PR_RTF_COMPRESSED for RTF text that has been compressed. Because the formatted version of a message text can be twice as large as the version without the formatting, the formatted RTF text is compressed before it is transferred with the message and stored in the PR_RTF_COMPRESSED property. When it is time to display the message on the screen, it is uncompressed using a utility function provided by MAPI.

MAPI defines these two message text properties and mechanisms for conversion between them so that RTF-aware clients can interoperate with clients and messaging systems that do not support formatted text.

Synchronizing Text and Formatting

The main challenge in sending Rich Text Format (RTF) messages is keeping the text synchronized with the formatting. To insure that when messages arrive at their destination, they are as their originators intended and that the text and formatting are synchronized, MAPI provides the **RTFSync** function. **RTFSync** is typically called by RTF-aware clients before displaying incoming messages and by the MAPI spooler when it downloads messages to a transport provider. Callers specify the area of possible discrepancy by passing one or two flags to **RTFSync**:

- RTF_SYNC_BODY_CHANGED to indicate a modification in message text.
- RTF_SYNC_RTF_CHANGED to indicate a modification in message formatting.

The synchronization process that occurs in **RTFSync** is a sophisticated cyclic redundancy check (CRC) of the message text that ignores some characters and converts others. Characters that most likely were added by transport providers are ignored. MAPI defines several auxilliary properties for working with RTF as described in the following table.

RTF property	**Description**
PR_RTF_SYNC_BODY_TAG	Indicates the beginning of the real message text.
PR_RTF_SYNC_BODY_CRC	Contains the result of the cyclic redundancy check of the message text.
PR_RTF_SYNC_BODY_COUNT	Contains the number of characters in PR_RTF_SYNC_BODY_CRC.
PR_RTF_IN_SYNC	Set to TRUE when the message text and formatting have been synchronized.
PR_RTF_SYNC_PREFIX_COUNT	Contains the number of non-whitespace characters that preceed the message text.
PR_RTF_SYNC_TRAILING_COUNT	Contains the number of non-whitespace characters that trail the message text.

Supporting Formatted Text in Outgoing Messages: Client Responsibilities

Clients set either the PR_BODY property, the PR_RTF_COMPRESSED property, or both properties for an outgoing message. Clients that support only plain text set only the PR_BODY property. Rich Text Format (RTF)-aware clients might set both properties or only PR_RTF_COMPRESSED, depending on the message store provider being used.

It is important for a client to check its message store's PR_STORE_SUPPORT_MASK property to determine whether the store supports rich text. If the message store is not RTF-aware, an RTF-aware client sets both the PR_BODY and PR_RTF_COMPRESSED properties for each outgoing message.

If the message store is RTF-aware, only the PR_RTF_COMPRESSED property needs to be set.

RTF-aware clients should perform the following steps to set PR_RTF_COMPRESSED and ensure that the synchronization process occurs as necessary:

1. Call the **IMAPIProp::OpenProperty** method to open the PR_RTF_COMPRESSED property, setting both the MAPI_CREATE and MAPI_MODIFY flags. MAPI_CREATE insures that any new data replaces any old data and MAPI_MODIFY enables your client to make those replacements.
2. Call the **WrapCompressedRTFStream** function, passing STORE_UNCOMPRESSED_RTF if the message store sets the STORE_UNCOMPRESSED_RTF bit in its PR_STORE_SUPPORT_MASK property, to get an uncompressed version of the PR_RTF_COMPRESSED stream returned from **OpenProperty**.
3. Write the message text data to the uncompressed stream returned from **WrapCompressedRTFStream**.
4. Commit and release both the uncompressed and compressed streams.

At this point, if the message store provider supports RTF, your client has done all that is required. Your client can depend on the message store provider to handle the synchronization process and the creation of the PR_BODY property, if necessary. However, if the message store provider does not support RTF, your client must call the **RTFSync** function to synchronize the text with the formatting, setting the RTF_SYNC_RTF_CHANGED flag.

Supporting Formatted Text in Incoming Messages: Client Responsibilities

As messages are transferred between messaging systems, the MAPI spooler makes sure that the rich text formatting remains synchronized with the message text. The MAPI spooler calls the **RTFSync** function from within a wrapped version of the message that it passes to the transport provider. The transport provider saves the changes made to the message by calling the **IMAPIProp::SaveChanges** method and then routes it to the new recipient.

When the recipient's RTF-aware client application opens the message to display the text, it must synchronize the text with the formatting and open either PR_RTF_COMPRESSED or PR_BODY, depending on which property is available. RTF-aware clients should follow these steps for opening a message:

1. Call **RTFSync** to synchronize the message text with the formatting if the message store is not RTF-aware. The RTF_SYNC_BODY_CHANGED flag should be passed in the *ulFlags* parameter if the PR_RTF_IN_SYNC property is missing or set to FALSE. Clients working with RTF-aware message stores need not make the **RTFSync** call because the message store takes care of it.
2. Call **IMAPIProp::SaveChanges** if the message text has been updated.
3. Call **IMAPIProp::OpenProperty** to open the PR_RTF_COMPRESSED property. If PR_RTF_COMPRESSED is not available, your RTF-aware client should open the PR_BODY property instead to display the message content.
4. Call the **WrapCompressedRTFStream** function to create an uncompressed version of the compressed RTF data, if available.
5. Display the uncompressed RTF data or the plain text data to the user.

RTFSync returns a boolean value that indicates whether or not the message has been updated. If this value returns TRUE, call **SaveChanges** at some point to make the update permanent. The call does not have to be made immediately after **RTFSync** returns.

Note Because so many components handle the rich text before your client receives it, there is the possibility of corruption. This corruption could come from the message store provider, a third party application, a gateway, or a transmission error.

Supporting Formatted Text: Message Store Responsibilities

Message store providers use the PR_STORE_SUPPORT_MASK property to publish whether or not they can handle rich text and, if they are RTF-aware, whether they store rich text in a compressed or uncompressed format. Message store providers indicate that they are RTF-aware by setting the STORE_RTF_OK bit and that they store the rich text is an uncompressed form by setting the STORE_UNCOMPRESSED_RTF bit.

While it is important for an RTF-aware client to check for the STORE_RTF_OK bit to determine whether or not a message store supports rich text, it is not necessary for a client to be concerned with the format of an RTF-aware store's rich text.

All message stores must support for non-RTF-aware clients. A non-RTF-aware message store must delete the PR_RTF_IN_SYNC property during a call to the message's **IMAPIProp::SaveChanges** method if a client has changed PR_BODY without updating either PR_RTF_IN_SYNC or PR_RTF_COMPRESSED. Deleting PR_RTF_IN_SYNC causes the PR_RTF_COMPRESSED property to be recomputed from the PR_BODY property the next time an RTF-aware client calls **RTFSync**.

Most RTF-aware message stores are not given the message text by clients; it must be computed on request. Because this computation is time consuming and expensive, clients should use PR_RTF_COMPRESSED whenever possible. To compute the PR_BODY property, the message store provider must uncompress the contents of the PR_RTF_COMPRESSED property and remove the rich text formatting. Clients that do not support the PR_RTF_COMPRESSED property require this computation to take place for every message.

When copying messages, message store providers that do not use the **IMAPISupport::DoCopyProps** or **IMAPISupport::DoCopyTo** methods run the risk of creating a message with no content if their implementation excludes the PR_BODY property and relies on PR_RTF_COMPRESSED. It is possible for the data in the PR_RTF_COMPRESSED property to be corrupt. Before excluding either of these message content properties in the copy operation, check for corruption as follows:

1. If the value of PR_RTF_COMPRESSED is not larger than the compressed RTF, the property is corrupt.
2. If the magic value in the RTF header is not *dwMagicCompressedRTF* or *dwMagicUncompressedRTF*, the property is corrupt.

Message store providers using the support methods need not be concerned with implementing a check for PR_RTF_COMPRESSED corruption; MAPI ensures that the appropriate properties exist and are valid.

There are three different levels of RTF support that message store providers can implement; MAPI recommends that RTF-aware message store providers implement their support at the middle or highest level. All RTF-aware message store providers take care of generating PR_BODY from the data included in PR_RTF_COMPRESSED on outgoing messages and make a call to **RTFSync** to synchronize text and formatting on incoming messages.

The differences between these three levels are described in the following table.

Level of support	**Description**
Low	Message store provider calls **RTFSync** whenever changes are saved to a message and extracts the data for the PR_BODY property from PR_RTF_COMPRESSED rather than requiring clients to set it. Both PR_BODY and PR_RTF_COMPRESSED are stored.
Middle	Message store provider stores only the

	PR_RTF_COMPRESSED property, computing PR_BODY when necessary.
High	Message store provider stores neither PR_BODY or the auxilliary RTF properties. **RTFSync** is called when the message text has changed and the formatting remains unchanged or when a new message is downloaded by a transport provider.

Supporting Formatted Text: Rendering Attachments

A client that cares about where in a message its attachments are rendered sets the PR_RENDERING_POSITION property for these attachments during message composition. A client that does not care about rendering placement leaves this property unset.

When a client opens a message with attachments, it attempts to retrieve each attachment's PR_RENDERING_POSITION property to determine where in the message text the attachment should be rendered. A client can use one of the following methods to retrieve PR_RENDERING_POSITION:

- **IMAPIProp::GetProps** on the open attachment to retrieve the PR_RENDERING_POSITION property.
- **IMessage::GetAttachmentTable** on the open message to retrieve its attachment table. PR_RENDERING_POSITION is a required column in all attachment tables. This is the preferable method because it results in better performance.

RTF-aware messages stores can choose whether to return an accurate or approximate value for PR_RENDERING_POSITION. Because message stores recalculate an attachment's PR_RENDERING_POSITION value when asked for the message's PR_BODY property, some RTF-aware message stores only guarantee the accuracy of rendering positions when a client asks first for PR_BODY. RTF-aware message stores are allowed to provide clients with approximate rendering position values to enhance performance. Providing an approximate rather than an accurate rendering position saves time and is sufficient for most situations.

RTF-aware message stores should base their approximation on the value specified by the client responsible for creating the attachment. Although all clients should set PR_RENDERING_POSITION, message store providers should be prepared to deal with the possibility of its absence. When the client does not set PR_RENDERING_POSITION, a message store can set it to -1 to indicate that the rendering position is not within the message text. Attachments with a rendering position of -1 can be displayed at any place within the message depending on the client. Many clients position these types of attachments at the top of the message.

The degree of accuracy for a PR_RENDERING_POSITION property depends on whether or not a message store saves both a message's PR_BODY and PR_RTF_COMPRESSED properties or only PR_RTF_COMPRESSED. If the client generates PR_BODY and the message store saves it along with the rich text, the rendering positions will be accurate. However, if the message store must generate its own version of PR_BODY because it only saves PR_RTF_COMPRESSED, it is probable that the rendering positions will be somewhat inaccurate. This is because of the differences in the way that clients and message store providers generate the PR_BODY property.

To calculate an accurate PR_RENDERING_POSITION value, an RTF-aware store uses a tag embedded in the rich text. The utility function **RTFSync** can be called to perform this calculation and update an attachment's rendering position. Depending on the amount of state information available, the message store can pass either RTF_SYNC_BODY_CHANGED, RTF_SYNC_RTF_CHANGED, or both values to **RTFSync**.

Supporting Formatted Text: Gateway Responsibilities

For outgoing messages, gateways should follow these steps for the correct handling of rich text:

1. Retrieve only a message's PR_RTF_COMPRESSED property from the message store. The main advantage in retrieving only the PR_RTF_COMPRESSED property is that the message text does not need to be sent between machines if the gateway and the message store exist on different machines.
2. Generate the message text from the rich text either by calling the RTF library function **HrTextFromCompressedRTFStream** or, if the message is stored locally, **RTFSync**. The RTF_SYNC_RTF_CHANGED flag should be set in the call to **RTFSync**.
3. Make any irreversible modifications to the message text, such as dropping unsupported characters.
4. Ensure that both PR_RTF_IN_SYNC and all of the RTF auxilliary properties are either set or absent.
5. If any modifications were made, call **RTFSync** with both the RTF_SYNC_RTF_CHANGED and RTF_SYNC_BODY_CHANGED flags set. **RTFSync** will recalculate the RTF auxilliary properties from the modified text.
6. Make any reversable modifications to the message text, such as inserting attachment placeholders and performing nondestructive code page conversions.
7. Send the message.

For incoming messages, gateways should perform the following steps for handling rich text:

1. Reverse any message text modifications that were made directly before the message was sent.
2. Call **RTFSync** if the message contains both the PR_RTF_COMPRESSED and PR_BODY properties.
3. Update the message in the message store with the PR_RTF_COMPRESSED property if the message contains it; update with the PR_BODY property only if PR_RTF_COMPRESSED is absent.
4. Discard PR_BODY if the message contains both this property and PR_RTF_COMPRESSED.

Gateways call **RTFSync** to avoid transmitting both the message text and rich text if the message store is on a different machine. If the gateway is local, it can set both properties and allow the message store to perform the synchronization.

About Sending Across Messaging Domains

A messaging domain represents one or more messaging systems that share a common address format. Communication across multiple messaging domains involves translating a message sent in the format of the original messaging domain into the format of the destination messaging domain. Because not all address formats are compatible, a gateway is needed to translate the addressing information from the source format into the destination format. To ensure validity across messaging domains, client applications store important addressing information in MAPI properties. In addition, gateways perform the translation, examining the properties known to need translation and changing them to a format that the destination messaging domain can use.

Previously, MAPI allowed this addressing information to be associated with only the users who comprise a message's current recipient list. The properties describing each member of the recipient list underwent the required translation by the gateway to ensure validity across messaging domains. However, some applications require that their messages include addressing information about users that perhaps were recipients in the past, will be recipients in the future, or will never be recipients. For example, routing applications, which send messages in a specified order to a group of users, embed addressing information about these users in the messages. The embedded information typically includes the address and address type of the future recipients, and perhaps also their roles and positions in the routing order, their names, and one or more binary identifiers per recipient.

To enable messages to include information about these non-recipient users, MAPI now includes a strategy for ensuring that this non-recipient information is also translated correctly across messaging domains. This strategy is based on the concept of gateway-mappable properties.

About Gateway Mappable Properties

Gateway-mappable properties are properties that may require translation when sent from one messaging domain to another. MAPI's gateway-mappable properties allow messages to include information that requires a gateway to ensure the destination messaging system uses it properly. Although gateway developers are not required to provide this translation capability, they should consider gateway-mappable properties as an opportunity to handle message content even better.

MAPI specifies five types of gateway-mappable properties:

- Display name
- E-mail address
- E-mail type
- Entry identifier
- Search key

This is the set of addressing properties that are associated with recipients, senders, report recipients, and delegated senders and recipients. To help your client define these properties so that a gateway handles them specially, MAPI specifies a naming convention using named properties and property sets. Five property sets exist to hold named properties, the addressing properties that require mapping. There is one property set for each type of mappable property. The property sets that will hold these named addressing properties are as follows:

Property set	Description
PS_ROUTING_DISPLAY_NAME	Contains string properties used as display names.
PS_ROUTING_EMAIL_ADDRESSES	Contains string properties used as e-mail addresses.
PS_ROUTING_ADDRTYPE	Contains string properties used as e-mail address types.
PS_ROUTING_ENTRYID	Contains binary properties used as long-term entry identifiers.
PS_ROUTING_SEARCH_KEY	Contains binary properties used as search keys.

About Client Naming Responsibilities

Clients must follow a naming convention for their properties that need to be translated by a gateway. All properties to be translated should be created as named properties in one of the five property sets designated to hold mappable properties: PS_ROUTING_EMAIL_ADDRESSES, PS_ROUTING_ADDRTYPE, PS_ROUTING_DISPLAY_NAME, PS_ROUTING_ENTRYID, or PS_ROUTING_SEARCH_KEY. Addressing properties that relate to the same user must be given the same name. Gateways rely on this naming convention, which enables them to match an address with its correct address type. For address parsing, the mapping between address and address type must be accurate.

MAPI named properties are represented with the **MAPINAMEID** data structure, which specifies that the property name can be either a Unicode string or a 32-bit integer. Integer naming is recommended for groups of addresses because it is a straightforward way to differentiate between sets of mappable properties, and it can easily serve as an index to the user. The alternative to using integers is to assign one string as the name for all five of a user's mappable properties. If only one user requires mapping, assigning a string is acceptable. However, when there are multiple users, it is better to use integer naming. The name, whether it be numeric or string-based, should be stored in either a message class-specific property or in a property belonging to a property set that is defined by the client application.

Note Avoid translating integer names to strings, such as "route_recipient_1" and "route_recipient_2." This effort is unnecessary.

To illustrate how this naming convention works, consider a routing application that sends a message to each member of a four-person team. When one member receives the message, he or she must respond to it before it can be sent along with the compiled responses to the next member. The original message contains a recipient list with one entry: the first member of the team. Embedded within the message are the gateway-mappable properties for the other three team members. Each member has five core user properties residing in the gateway-mappable property sets, assigned a unique number as a name.

The following table illustrates the settings for each of set of gateway-mappable properties stored for the three remaining team members to whom the message is routed when the first team member is finished with it.

Property Set	**Second Team Member**	**Third Team Member**	**Fourth Team Member**
PS_ROUTING_EMAIL_ADDRESSES	Address = 0	Address = 1	Address = 2
PS_ROUTING_ADDRTYPE	Address type = 0	Address type = 1	Address type = 2
PS_ROUTING_DISPLAY_NAME	Display name = 0	Display name = 1	Display name = 2
PS_ROUTING_ENTRYID	Entry identifier = 0	Entry identifier = 1	Entry identifier = 2
PS_ROUTING_SEARCH_KEY	Search key = 0	Search key = 1	Search key = 2

Clients that use mappable search keys as references to other message properties must recognize that the other message properties will not be translated unless they are placed in one of these mappable property sets. When a message with unmapped references to mapped search keys is sent to a destination in another messaging domain, the references are invalid. To allow these other properties to remain synchronized with the search keys, you can assign them string names in the PS_ROUTING_SEARCH_KEY property set that do not interfere with the names given to any of the core mappable properties. For example, suppose a client needs to transmit a property that contains the search key for the last person in a long routing list. The client can create a named property in the PS_ROUTING_SEARCH_KEY property set called "last_search_key." Because it is stored in a mappable property set, the "last_search_key" property is translated along with the rest of the gateway-mappable properties.

About Gateway Mapping Responsibilities

When a MAPI-aware gateway receives a message containing named properties in one of the special property sets designated to contain gateway-mappable properties, the gateway should map all of the properties to the protocol of the destination messaging system. Although MAPI recommends that gateways handle all named properties in the special property sets, gateways are expected to handle only two: e-mail address and address type. Because the e-mail address and address type properties directly affect message transmission, it is critical that gateways support the mapping of these two properties. Because search keys consist of a user's address type and address, they should also be translated if at all possible.

Entry identifiers are not expected to be handled frequently. To enable mapping of an entry identifier that originates in one messaging system to an entry identifier that is usable by another messaging system, the gateway must be able to use the format of both systems. Because most gateways are not aware of entry identifier formats, the translation of entry identifiers is rare.

Another mappable property that is not expected to be translated frequently is the display name. Gateways should store display names and transmit them, but not necessarily translate them.

About the Idle Utility

MAPI provides several functions that are collectively known as the idle utility. These functions allow clients, address book providers, and message store providers to perform various tasks during slow times in the session or in response to a slow time. For example, clients and service providers can defer slow operations or close files that have remained unused for a lengthy period. Transport providers typically do not use the idle utility because the **IXPLogon::Idle** method takes its place.

To use the idle utility, clients and service providers create a callback function that contains the tasks that should occur when the MAPI subsystem is idle. When MAPI detects idle time, it invokes this callback function. The callback function follows the **FNIDLE** prototype, defined as follows:

```
BOOL (STDAPICALLTYPE FNIDLE) (LPVOID lpvContext)
```

The functions that make up the idle utility are:

ChangeIdleRoutine
DeregisterIdleRougine
EnableIdleRoutine
FtgRegisterIdleRoutine
MAPIDeInitIdle
MAPIInitIdle

To register a callback function, clients and service providers call the **FtgRegisterIdleRoutine** function. The input parameters include an optional priority, a block of memory that is passed to your callback function as input, an amount of time to be used as in any way appropriate, and a set of option flags.

Clients and service providers can specify a priority in the *priIdle* parameter that controls how the idle function runs or specify zero if priority is not an issue. Because negative numbers represent higher priorities than positive numbers or zero, compression and search operations should be assigned negative numbers. Tasks that occur once should be assigned positive numbers.

To deregister an active callback function, clients and service providers call the **DeregisterIdleRoutine** function. Because **DeregisterIdleRoutine** operates asynchronously, it is possible for the callback function to be invoked at any time during the deregister call and possibly even after **DeregisterIdleRoutine** has returned.

To modify some or all of the characteristics of a callback function, clients and service providers call the **ChangeIdleRoutine** function. **ChangeIdleRoutine** any of the options set with the flags parameter *ircIdle* to be changed, including the function itself, its priority, time setting, and input parameter.

MAPI's definition of idle is the same as the operating system, when the operating system has a definition. On Win16, MAPI's installs a journalling hook that updates a variable with the time of the last user action. Idle time calculations are based on this variable. On Win32, MAPI creates a thread with idle-class priority to schedule idle tasks. This thread keeps track of the time and posts a message to the thread that is to execute the idle task when the time for its execution arrives. Win32 schedules threads, not processes. If tasks that have a priority higher than the idle priority are occurring on the workstation, the idle task should not get scheduled for execution until the tasks have completed.

All idle tasks run on the thread that called **MAPIInitIdle**. MAPI has a separate thread for scheduling, but when an idle task becomes eligible, it posts a message back over to the initialization thread and the idle task is executed there. The implications for different types of clients are as follows:

Threading model	Implication
Single-threaded	No problem. Idle functions execute on your client's main thread and are serialized through the message loop.

Free-threaded	Idle functions must be thread-safe, but your client already has the necessary infrastructure. Your client might not need MAPI's idle utility at all.
Apartment-threaded	Idle function has to execute on the same thread that registered it if it wants to use MAPI, OLE, or any other COM interfaces. The most straightforward way is to register an idle function with MAPI that posts a message to the right thread and dispatch the "real" idle function directly from that thread's message loop.

Developing a Client Application

MAPI client applications are applications that are written with the object oriented MAPI client interface. MAPI client applications interact with a messaging system through service providers and through the MAPI subsystem. This interaction can occur in many different ways; there is an enormous variety of client applications. The most common client application either integrates messaging into its feature set or performs messaging as its primary feature. Other features that MAPI clients might provide include profile administration or address book and message store management.

Regardless of the type of MAPI client you are developing, you will:

- Initialize the MAPI libraries.
- Start up a session.
- Communicate with one or more service providers either directly or through the MAPI subsystem.
- End a session.

Use the following topics to help you implement these basic tasks and the specific features that will make your MAPI client application unique.

Types of Client Applications

There are primarily two types of messaging clients: those that handle interpersonal messages (IPM) and those that handle interprocess communication, or IPC, messages. Within those types, messaging client applications can be categorized as follows:

- Person-to-person
- Person-to-machine
- Machine-to-person
- Machine-to-machine
- Mix of persons and machines

Person-to-person applications involve a person initiating the exchange of messages and another person responding. This category of applications includes traditional e-mail applications as well as more structured exchanges such as document routing or expense approval.

Person-to-machine applications involve a person initiating the exchange of messages and a machine responding. This category includes applications that use e-mail to, for example, submit a database query or subscribe to a mailing list.

Machine-to-person applications involve a machine initiating the exchange of messages and a person responding. This category includes applications that distribute documents such as news feeds and opinion surveys.

Machine-to-machine applications involve a machine initiating the exchange of messages and a machine responding. This category includes applications such as link heartbeat monitoring and directory and database replication.

The final category, a mix of persons and machines, involves a more complex scenario. This category includes applications that do not necessarily transmit messages between senders and recipients. Instead they might post them directly into a public folder or to a bulletin board forum supported by a message store. The messages can then be consumed on demand by other readers, an administrator, or a software agent.

If you are writing a person-to-person application, machine-to-person application, or an application that posts messages to public forums, set up your application to send and receive IPM messages. If you are writing a person-to-machine or machine-to-machine application, it should be set up to send and receive IPC messages. Any application that requires the interaction of a human user must support IPM messages. Applications that involve both people and machines in a variety of scenarios often must support both IPM and IPC messages. The only real difference between the two classes is that IPM messages in a message store are visible to messaging clients, while IPC messages usually are not.

Rather than limiting your messages to the capabilities provided by MAPI's superclasses, IPM and IPC, you can customize and enhance these classes by creating new IPM or IPC subclasses. Creating message subclasses involves inventing new message classes that inherit from the superclasses. For example, if your person-to-person application specializes in sending documents, you can subclass the IPM superclass by defining an IPM.Document.MyDoc class and create properties that describe your documents. In addition to supporting these custom properties, your IPM.Document.MyDoc messages will inherit the properties supported by all IPM messages.

Linking to the MAPI DLLs

MAPI clients can link to the MAPI DLLs implicitly or explicitly through the Windows functions **LoadLibrary** and **GetProcAddress**.

MAPI provides type definition statements in the MAPIX.H header file for each of the following functions:

- **MAPILogonEx**
- **MAPIUninitialize**
- **MAPIInitialize**
- **MAPIAllocateBuffer**
- **MAPIAllocateMore**
- **MAPIFreeBuffer**
- **MAPIAdminProfiles**

Use these type definitions to correctly call the corresponding entry points if your client links explicitly to the MAPI DLLs.

Service providers can contain non-MAPI functionality, features that are completely unrelated to MAPI, in their DLL. If your client needs to use these features, it is necessary to call **LoadLibrary** before using the DLL and **FreeLibrary** to remove the DLL from memory after its use. Because MAPI is unaware of non-MAPI uses of a service provider DLL, there is no guarantee that the DLL will be available when needed. MAPI releases a service provider DLL when there are no longer any clients with active sessions that require its services.

Session Handling

Before your client can communicate with service providers and an underlying messaging system, a session must be established. A MAPI session is a link from a client to other MAPI components. As the result of successfully starting a session, MAPI returns to clients a pointer to a session object, an object that implements the **IMAPISession** interface.

Your client can use the methods of the **IMAPISession** interface to open and compare address book and message store objects, to access several tables, and to perform profile and message service administration.

Your client starts a session by initializing the MAPI libraries and, if required, the OLE libraries, and by selecting a valid profile. MAPI verifies the configuration of each of the service providers in the message services included in the profile, prompting the user for additional information if necessary and your client allows it. When session start up is complete, the configured service providers are ready to service your client.

When it is time to end a session, your client uninitializes the initialized libraries and makes a call to **IMAPISession::Logoff**. This call causes MAPI to shut down each of the active service providers.

Starting a MAPI Client Session

Although there is a significant amount of work performed during session start up, the tasks required of your client are minimal. Much of this work is done in MAPI's processing of the **MAPIInitialize** and **MAPILogonEx** calls. Both of these functions accept flags as input parameters for controlling aspects of the session such as notification handling and the user interface. It is important to understand the consequences of setting each of these flags when calling **MAPIInitialize** to initialize the MAPI libraries and **MAPILogonEx** to log on to the MAPI subsystem.

To start a MAPI session

1. Call **MAPIInitialize** to initialize the standard set of MAPI libraries.
2. If your client needs to use the OLE libraries, call the OLE function **OleInitialize**.
3. If your client needs to use the MAPI utility library, call **ScInitMapiUtil**.
4. Call **MAPILogonEx** to log on to the MAPI subsystem.

Initializing the MAPI Libraries

All client applications that use the MAPI libraries must call the **MAPIInitialize** function. **MAPIInitialize** initializes global data for the session and prepares the MAPI libraries to accept calls. There are two flags that are important to set in some situations:

MAPI_NT_SERVICE
MAPI_MULTITHREAD_NOTIFICATIONS

The MAPI_NT_SERVICE flag must be set by clients that are implemented as Windows NT services. If your client is a Windows NT service that does not pass the MAPI_NT_SERVICE flag in its call to **MAPIInitialize**, MAPI will not recognize it as a service.

The MAPI_MULTITHREAD_NOTIFICATIONS flag relates to how MAPI manages notifications for service providers. MAPI supports event notification by creating a hidden window that receives window messages when changes occur in an advise source. Concurrently, MAPI places a copy of the parameters to be included in the notifications in shared memory. The window messages are processed at some point, causing the appropriate **IMAPIAdviseSink::OnNotify** methods to be called.

Because it must process messages, MAPI expects the thread that is used to create the notification window to:

- Have a message loop.
- Remain unblocked throughout the life of the session.
- Have a longer lifetime than any other thread created by the client.

MAPI either creates the notification window on the thread that was used for your client's first **MAPIInitialize** call or on a separate thread, dedicated to handling notifications. Your client can choose which thread is used by setting a flag in the first **MAPIInitialize** call. The danger in allowing one of your client's threads to handle the notifications is that if the thread disappears, the notification window is destroyed and notifications can no longer be sent to any of the other threads in your client. Also, special processing might be needed to control the dispatching of the notification messages that are posted to the hidden window's message queue.

If your client uses a separate window to handle notifications, be assured that notifications will appear at the appropriate time on an appropriate thread. Your client will not need any special code to check for and process the Windows messages that are posted to the notification window.

MAPI recommends that the following types of client applications use a separate thread to create the hidden window for notification support:

- All 32-bit multithreaded clients
- Single-threaded Windows NT services and Win 32 console applications
- Single-threaded clients that do not need to use their main thread for notification

To use the separate thread approach, call **MAPIInitialize** on every thread in your client, setting the MAPI_MULTITHREAD_NOTIFICATIONS flag. Calls made from 16-bit clients operate as if the MAPI_MULTITHREAD_NOTIFICATIONS flag is not set.

Note Only a client's first call to **MAPIInitialize** causes a hidden window to be created to support notifications. Subsequent calls only cause a reference count to be incremented.

Initializing the OLE Libraries

If your client application also uses OLE, an additional call to the OLE function **OleInitialize** must be made. If your client is written for a 32-bit platform, such as Windows NT or Windows 95, this call must be made on every thread in your application. **OleInitialize** initializes global data for the session and prepares the OLE libraries to accept calls. For information about calling **OleInitialize**, see the *OLE Programmer's Reference*.

Initializing the MAPI Utilities

If your client needs to use only the MAPI utilities, such as the **ITableData** and **IPropData : IMAPIProp** interfaces, it does not need to call **MAPIInitialize** for initialization. Instead, call the API function **ScInitMapiUtil**. **ScInitMapiUtil** allows client applications to use utility functions and methods that require MAPI allocators, but that do not ask for them explicitly.

At shutdown time, your client must make a call to **DeinitMapiUtil** to free resources connected to the utilities. Do not call **MAPIUninitialize**.

Be aware that the **ITableData** interface does not support table notifications for clients that have called **ScInitMapiUtil** rather than **MAPIInitialize**.

Logging On a MAPI Client

Client applications log on to the MAPI subsystem by calling the **MAPILogonEx** function with the appropriate flags set. **MAPILogonEx** starts up and initializes each provider in the profile. If the message service that your client is using has multiple service providers, the providers in the address book are started first.

The logon process validates a client's choice of profile, the password for the profile, service provider configuration, and establishes either a shared, individual, or non-messaging session with the MAPI subsystem. Clients that supply partial information to **MAPILogonEx** must prompt the user for the additional data by allowing a dialog box to be displayed. Clients that do not need user input can suppress the dialog box display.

To specify a choice of profile

- Pass in a character string that represents the name of the profile in the *lpszProfileName* parameter.

  - Or -

- Allow the user to enter the profile by passing NULL in the *lpszProfileName* parameter and setting the MAPI_LOGON_UI flag.

  - Or -

- Select the default profile by passing NULL in the *lpszProfileName* parameter and setting the MAPI_EXPLICIT_PROFILE flag.

To specify a password

- Pass in a character string that represents the password in the *lpszPassword* parameter.

  - Or -

- Allow the user to enter the password by passing NULL in the *lpszProfileName* parameter and setting the MAPI_PASSWORD_UI flag.

The flags that **MAPILogonEx** uses to enable a user interface are mutually exclusive; only one can be set. Leaving these flags unset disables the user interface completely, causing **MAPILogonEx** to fail if necessary information is missing. That is, if your client disables the user interface, passes NULL for the *lpszProfileName* parameter, and does not set the MAPI_EXPLICIT_PROFILE flag, **MAPILogonEx** will fail because it cannot retrieve a profile name.

There are three types of sessions that a client can choose: an individual messaging session, a shared messaging session, and a non-messaging session. Individual messaging sessions are private connections between your client and the MAPI subsystem and can be established by setting the MAPI_NEW_SESSION flag in the call to **MAPILogonEx**. Shared messaging sessions are connections that multiple messaging clients can use. Shared sessions are typically established for clients that are members of a workgroup and use the same profile. To establish a new session as a shared session, your client sets the MAPI_ALLOW_OTHERS flag.

To use an existing shared session

- Do not set the MAPI_NEW_SESSION flag.
- Do not set the MAPI_ALLOW_OTHERS flag.
- Pass NULL for the *lpszProfileName* parameter.
- Pass NULL for the *lpszPassword* parameter.

Non-messaging sessions allow clients to access the MAPI subsystem, but do not allow messages to be sent or received. Configuration or administration applications are examples of clients that might need to establish non-messaging sessions. To request a non-messaging session, your client sets the MAPI_NO_MAIL flag. Setting this flag logs your client on without informing the MAPI spooler.

Messaging and non-messaging clients can share sessions. If your client logs on with the MAPI_NO_MAIL flag set and establishes a shared session, a second client logging onto this session that doesn't set the MAPI_NO_MAIL flag has no affect on the operation of your client. That is, as a messaging client, the second client will be able to send and receive messages while your client will not.

Clients should only set the MAPI_NO_MAIL flag under the following conditions:

- When they do not send or receive messages during the session, such as with clients performing profile configuration or message store access.
- When they have complete control over the contents of the profile, and messages are sent and received using only the Microsoft Exchange default message store and transport providers.

MAPILogonEx defines a few other flags that your client can set:

- MAPI_FORCE_DOWNLOAD indicates that incoming messages should be downloaded before **MAPILogonEx** returns. Not setting this flag causes messages to be downloaded in the background at a later time.
- MAPI_SERVICE_UI_ALWAYS requests that every message service in the profile display a configuration dialog box.

With every successful logon, **MAPILogonEx** returns a pointer to a MAPI session. Your client can use this pointer to call the methods of the **IMAPISession** interface. Session pointers, regardless of the type of session, are unique to the clients that receive them and are not valid across tasks.

Ending a MAPI Client Session

When your client needs to terminate its session with the MAPI subsystem, it must cancel the registrations for all of its notifications, release all of its open objects, call **IMAPISession::Logoff**, and release its session pointer. The call to **Logoff** is optional; it gracefully shuts down all of the service providers.

Clients can end their sessions in response to a user's request, either immediately or after all outbound messages have been processed, and when a critical error occurs. Some clients need to stay logged on so that pending outbound messages can reach the transport provider and the destination messaging system. If this type of client sends a message and immediately logs off, the message remains in the outgoing queue until a user logs back on and stays logged on long enough for the message to be transmitted.

To end a session

1. Cancel all registrations for notification by calling the **Unadvise** method of every advise source.
2. If your client has active references to any of the following objects, release them by calling their **IUnknown::Release** methods:

   Advise sink

   Status table

   Outbox folder

   Message store

   Address book
3. Call **MAPIFreeBuffer** to free the memory for any cached entry identifiers, such as PR_IPM_SUBTREE_ENTRYID.
4. Call **IMAPISession::Logoff**, setting the MAPI_LOGOFF_UI flag if your client allows a user interface and the MAPI_LOGOFF_SHARED flag if your client owns the current shared session. **Logoff** notifies all other clients that are using the current shared session that they should log off by sending an error notification.
5. Release the session pointer by calling the session's **IUnknown::Release** method.
6. If your client called **OleInitialize** during session startup to initialize the OLE libraries, uninitialize them now by calling **OleUninitialize**. Only clients that have called **OleInitialize** must call **OleUninitialize**.
7. Uninitialize the MAPI libraries by calling **MAPIUninitialize**. If your client called **OleInitialize** at some point, make sure that a call to **OleUninitialize** occurs before this call to **MAPIUninitialize**. The timing is crucial. If the call to **OleUninitialize** follows the call to **MAPIUninitialize**, your client might terminate ungracefully.
8. If your client called **ScInitMapiUtil** during session startup to initialize the MAPI utility library, uninitialize it now by calling **DeinitMapiUtil**. Only clients that have called **ScInitMapiUtil** must call **DeinitMapiUtil**.

Note All open objects must be released before the call to **IMAPISession::Logoff**. Objects that remain open after **Logoff** is called become invalid, they cannot accept any calls and might never be freed. If a call is made to one of these objects, your client should expect the call to fail.

When a MAPI session ends, all of the service provider DLLs are unloaded. They are not deleted. MAPI has no mechanism for deleting DLLs during the session shutdown process. A service provider's DLL can only be deleted when a configuration client such as the Control Panel calls its message service entry point function with the MSG_SERVICE_INSTALL event.

The MAPI spooler remains running as long as there is one client with an active session on the system. When the last client ends its session, the MAPI spooler is automatically shut down.

Opening and Comparing Objects

Your client can use the **IMAPISession : IUnknown** interface for opening objects of several different types. **IMAPISession** has the following methods for opening objects:

- **OpenProfileSection** to open a profile section.
- **OpenMsgStore** to open a message store.
- **OpenAddressBook** to open the MAPI integrated address book.
- **AdminServices** to open a message service administration object.
- **OpenEntry** to open any object that has an assigned entry identifier.

Some clients implement a feature enabling them to reconfigure the message services and service providers in a profile. To support profile administration, your client can call the **IMAPISession::AdminServices** method. **AdminServices** returns a pointer to a message service administration object, or object that supports the **IMsgServiceAdmin : IUnknown** interface. With an **IMsgServiceAdmin** pointer, your client can create, delete, copy, and configure a message service, specify the sequence that MAPI uses for calling transport providers to deliver messages, access the message service table, and access the profile.

With the **OpenEntry** method and a valid entry identifier, your client can open any address book or message store provider object. There are many **OpenEntry** methods in MAPI, implemented by the following objects:

Address book provider's logon object	Message store provider's logon object
MAPI address book	Message store
Address book container	Folder
Session	Support object

Some **OpenEntry** methods require an entry identifier of the object to be opened, as does **IMAPISession::OpenEntry**; other methods allow NULL to be specified. A NULL entry identifier is interpreted differently depending on the object. For example, when your client calls **IAddrBook::OpenEntry** with a NULL entry identifier, MAPI opens the root container of the address book. The message store's **OpenEntry** method behaves similarly; it opens the root folder of the message store. **IMAPIContainer::OpenEntry**, implemented by both folders and address book containers, might return MAPI_E_INVALID_PARAMETER or the root container, depending on the implementor.

In addition to disallowing a NULL value for the entry identifier, the session's **OpenEntry** method differs from other **OpenEntry** methods because its job is not to open objects. Instead, it examines the entry identifier and forwards the call to another **OpenEntry** method implemented by the appropriate service provider. For example, if your client calls **IMAPISession::OpenEntry** with the entry identifier of a message, MAPI calls the **IMSLogon::OpenEntry** method of the message store responsible for the message.

The **IMAPISession::CompareEntryIDs** method compares two entry identifiers. If the **MAPIUID** structures contained within the entry identifiers belong to the same service provider, MAPI forwards the call to that provider. **CompareEntryIDs** returns an error value when the two entry identifiers do not match. Although this method can compare entry identifiers that belong to either higher level objects such as message stores and lower level objects such as messaging users, use **CompareEntryIDs** for the higher level objects. To compare lower level objects, use the objects' search keys (PR_SEARCH_KEY) or record keys (PR_RECORD_KEY). These values can be compared directly, without calling a method.

Choose which **OpenEntry** and **CompareEntryID** method to use according to the amount of information that your client has about the object or objects to be opened or compared. Use the following guidelines when deciding which interface method to call:

- If your client has no information about the target objects, call **IMAPISession::OpenEntry** or **IMAPISession::CompareEntryIDs**. This approach enables access to any object, but is the slowest of the three.
- If your client knows that the target objects are address book entries rather than, for example, folders, call the **IAddrBook::OpenEntry** or **IAddrBook::CompareEntryIDs** method. **IAddrBook::OpenEntry** opens the root container of the address book when NULL is specified as the target object. This approach enables access to any address book object and is faster than using **IMAPISession**, but slower than using **IMAPIContainer**.
- If the entry identifier being used is a short-term entry identifier or if your client knows that the target objects belong to a particular address book container or folder, call the **IMAPIContainer::OpenEntry** method. This approach yields the fastest performance, but it enables access only to objects in a specific container or folder.

About Provider and Session Identity

Service providers, typically address book providers, have the option of supplying an identity that can be used to represent the session in a variety of situations. Three properties describe a provider's identity:

PR_IDENTITY_ENTRYID
PR_IDENTITY_DISPLAY
PR_IDENTITY_SEARCH_KEY

These properties are set to the entry identifier, display name, and search key of the corresponding identity object, which is typically a messaging user. Providers that supply an identity also set the STATUS_PRIMARY_IDENTITY flag in their PR_RESOURCE_FLAGS property.

Depending on your client's needs, it might need to use a particular provider's identity or the primary identity for the session. For example, your client can use the primary identity to include it on printouts or use a provider's identity for display purposes or to retrieve properties, such as PR_RESOURCE_PATH. PR_RESOURCE_PATH, if set, contains the path to miscellaneous files used or created by a provider. The PR_RESOURCE_PATH property for the provider supplying the primary identity when your client is interested in locating files that pertain to the user of the session rather than a specific provider.

To retrieve the identity of a specific provider

1. Call **IMAPISession::GetStatusTable** to access the status table.
2. Build a restriction using an **SPropertyRestriction** structure to match the PR_PROVIDER_NAME column with the name of the specified provider.
3. Call **IMAPITable::FindRow** to locate the provider's row. The provider's identity will be stored in the PR_IDENTITY_ENTRYID column, if it exists.

To retrieve the primary identity for a session

- Call **IMAPISession::QueryIdentity**.

QueryIdentity bases session identity on the existence of the STATUS_PRIMARY_IDENTITY value in the PR_RESOURCE_FLAGS column for one of the rows in the status table. If none of the status rows have this value set, **QueryIdentity** assigns identity to the first service provider that sets the three PR_IDENTITY properties. If no service provider supplies an identity, **QueryIdentity** returns MAPI_W_NO_SERVICE. When this happens, your client should create a character string to represent a generic user that can serve as the primary identity.

Note Do not call **QueryIdentity** during session start up; it will unnecessarily increase the amount of time it takes to start your client's session. This is because to access the status table and query for the PR_IDENTITY properties, the MAPI spooler must be available. The extra time it takes to start up the MAPI spooler will add to your client's start up time.

To explicitly set the primary identity for a session

- Call **IMsgServiceAdmin::SetPrimaryIdentity**. Pass the MAPIUID for the target service provider.

Using the Status Table and Status Objects

Your client calls **IMAPISession::GetStatusTable** to access the status table implemented by MAPI. The status table provides information about the MAPI subsystem, MAPI spooler, address book, or a particular service provider.

Each row in the status table represents a status object implemented by MAPI or a service provider. Your client can use a status object to display a provider's configuration property sheet, to change a provider password, to upload or download messages, and to communicate with a particular transport provider.

To access the status object of a particular service provider

1. Call **IMAPISession::GetStatusTable** to retrieve the status table.
2. Call the status table's **IMAPITable::SetColumns** method to limit the column set to PR_ENTRYID, PR_RESOURCE_TYPE, and PR_DISPLAY_NAME.
3. Build a property restriction using an **SPropertyRestriction** structure to match PR_DISPLAY_NAME with the display name of the target provider.
4. Call **HrQueryAllRows**, passing in the **SPropertyRestriction** structure, to retrieve the row that represents the status of the provider.
5. Pass the PR_ENTRYID column to **IMAPISession::OpenEntry** to open the provider's status object.

To display a property sheet, call the status object's **IMAPIStatus::SettingsDialog** method for the target provider. **SettingsDialog** displays a property sheet for viewing and in some cases, changing the configuration properties of a provider.

To communicate with a transport provider, call its status object's **IMAPIStatus::ValidateState** method. **ValidateState** can reconfigure a transport provider, prevent the provider from displaying a user interface, and control a session that involves downloading message headers from a remote server, depending on the flags that your client passes in. For example, to cancel the downloading of remote headers, pass the ABORT_XP_HEADER_OPERATION to **ValidateState**. To connect or disconnect from the remote server, pass FORCE_XP_CONNECT or FORCE_XP_DISCONNECT. To reconfigure the provider, pass CONFIG_CHANGED.

Clients that implement sending or receiving of messages on demand call either a transport provider's or the MAPI spooler's **IMAPIStatus::FlushQueues** method. Your client can pass three flags into the method: FLUSH_UPLOAD, FLUSH_DOWNLOAD, and FLUSH_FORCE. FLUSH_UPLOAD instructs the provider or the spooler to send any messages waiting in the output queue while FLUSH_DOWNLOAD instructs the provider or the spooler to receive any incoming messages. FLUSH_FORCE can be set with either of the other flags to cause the status object to perform the flush regardless of the timing.

Do not expect to be able to call **SettingsDialog** or **ChangePassword** on any of the MAPI subsystem, MAPI spooler, or address book status objects. Both the subsystem and address book status objects only support **ValidateState**; the MAPI spooler status object supports **FlushQueues** in addition to **ValidateState**.

For more information about the status table and status objects, see About Status Tables and Status Objects.

Requesting Read and Delivery Status Reports

Your client can register to be informed when a message that it has sent has or has not been read or delivered. MAPI is designed to automatically send nondelivery, or NDR, reports to senders when messages cannot be successfully delivered. Reports of successful delivery and read status reports are only generated on demand.

To request a delivery report, set the PR_ORIGINATOR_DELIVERY_REPORT_REQUESTED property to TRUE.

Read status reports are generated as a pair. That is, clients cannot request only read reports or only non-read reports. To request a read status report, set the PR_READ_RECEIPT_REQUESTED property to TRUE.

To suppress the delivery of nondelivery reports, set PR_ORIGINATOR_NON_DELIVERY_REPORT_REQUESTED to FALSE.

For more information about message reports, see About Report Messages.

Handling Outgoing Messages

To prepare a message for transmission, your client must open the default message store and the Outbox folder, create the message, set several message and recipient properties, perform name resolution, and finally save the message. When your client has completed these steps, the message is ready to be sent.

Assuming that your client has opened the default message store, perform the following procedure to create a transmittable message.

To create an outgoing message

1. Call the message store's **IMAPIProp::GetProps** method to retrieve the PR_IPM_OUTBOX_ENTRYID property, which contains the entry identifier for the Outbox.
2. Call **IMsgStore::OpenEntry**, passing PR_IPM_OUTBOX_ENTRYID as the entry identifier, to open the Outbox.
3. Call the Outbox folder's **IMAPIFolder::CreateMessage** method to create the new message.
4. Add one or more recipients to the message.
5. Optionally, add a subject.
6. Add the message text.
7. Optionally, add one or more attachments.
8. Add any other necessary properties, such as PR_CONVERSATION_INDEX.
9. Call the message's **IMAPIProp::SaveChanges** method to save it or **IMessage::SubmitMessage** to save the message before it is sent.

Opening the Default Message Store

In any particular session, one message store acts as the default message store. A default message store supports the PR_DEFAULT_STORE property and sets it to TRUE. A default message store also has a collection of special folders that MAPI automatically creates when the message store is opened. The special folders include:

- IPM subtree of special folders (Inbox, Outbox, Deleted Items, and Sent Items)
- Root folder for search results
- Root folder for common views
- Root folder for personal views

The names that MAPI assigns to the IPM subtree folders are default names; your client can keep these names or change them as necessary. MAPI assigns default names and associations for these folders to keep messages from inadvertently disappearing if a client neglects to establish receive folders for messages.

To open the default message store

1. Call the **IMAPISession::GetMsgStoresTable** method to access the message store table.
2. Build a restriction using the **SAndRestriction** structure to combine:
   - A restriction using the **SExistRestriction** structure that tests for the existence of the PR_DEFAULT_STORE property.
   - A restriction using the **SPropertyRestriction** structure that checks for the TRUE value in the PR_DEFAULT_STORE property.
3. Call **HrQueryAllRows** to retrieve the row that represents the default message store.
4. Pass the entry identifier included in the row's column set to **IMAPISession::OpenMsgStore** to open the default message store.

Opening the Outbox Folder

The Outbox folder is the folder that contains all outgoing messages. For clients that handle IPM messages, this is the Outbox folder in the IPM subtree. For clients that handle IPC messages, this is the receive folder.

To open the Outbox

1. For IPM classes:
   a. Call the message store's **IMAPIProp::GetProps** method to retrieve the PR_IPM_OUTBOX_ENTRYID property.
   b. Call **IMsgStore::OpenEntry** with the entry identifier in PR_IPM_OUTBOX_ENTRYID to open the Outbox.
2. For IPC classes, use the receive folder for creating outgoing messages.

Creating an Outgoing Message

To create an outgoing message, call the Outbox folder method, **IMAPIFolder::CreateMessage**. **IMAPIFolder::CreateMessage** assigns an entry identifier for the newly created message that uniquely identifies it in the message store. The entry identifier is made up of a part that represents the message store provider and a part that represents the individual message. A message's entry identifier is unique within its message store and all the message stores that are open concurrently.

Your client, service providers, and possibly other clients will use this entry identifier, stored in the message's PR_ENTRYID property, to open the message once it has been saved. Some message store providers allow the entry identifier to be available immediately after **CreateMessage** returns; others delay its availability until the message has been saved.

Outgoing messages, after they have been sent, can be deleted or saved in a folder, depending on a property that your client sets. If your client sets PR_DELETE_AFTER_SUBMIT, each outgoing message is removed after it is sent. If your client sets PR_SENTMAIL_ENTRYID, each outgoing message is placed in the folder represented by the entry identifier in this property. Outgoing messages are saved when your client sends them in a call to **IMessage::SubmitMessage**. Messages not being sent must be saved with an explicit call to their **IMAPIProp::SaveChanges** method.

Adding a Message Recipient

All outgoing messages require at least one recipient, a collection of properties that represents a particular destination for the message. A recipient can represent a human user, a machine, or a folder.

Recipients are considered subobjects of the message with which they are associated because they can only be opened when the message is open.

By the time the message is sent, its recipients always have the following properties:

PR_ADDRTYPE
PR_DISPLAY_NAME
PR_EMAIL_ADDRESS
PR_ENTRYID
PR_SEARCH_KEY

These properties are used to access the recipient, send messages to it, and to compare it to others. Not all of these properties need to be available right away. Your client can add a recipient without knowing its entry identifier. Recipients undergo a process called name resolution, whereby an address book provider associates display names with entry identifiers. Clients typically initiate name resolution for a message by calling **IAddrBook::ResolveName** before they call **IMessage::SubmitMessage** to send it.

Another property, PR_RECIPIENT_TYPE, is assigned to each recipient by the message store provider. It indicates whether a recipient appears as a primary, carbon copy, or blind carbon copy recipient on a message. It also indicates for resent messages whether or not a recipient successfully received the message with the original transmission.

Recipients undergo a process called name resolution, whereby an address book provider associates an entry identifier with the display name.

Rather than calling an **IMAPIProp** method to manipulate these properties and others, clients call one message method, **GetRecipientTable**, to access existing recipient properties; a table method, **QueryRows**, to retrieve the rows in the table; and another message method, **ModifyRecipients**, to add new properties or make changes. For more information about recipient properties, see About Base Address Properties.

Recipients can be represented by permanent entries in an address book container or by independent entries known as one-offs. One-offs are recipients that are not part of the address book. As with other types of recipients, the address for one-offs follows a specific format. Their entry identifiers also conform to a specific format. For more information about one-off addresses and entry identifiers, see About One-Off Addresses and About One-Off Entry Identifiers.

Some clients limit their users to selecting recipients only from the address book. Other clients allow a mixture of address book recipients and one-off recipients. Still other clients do not use the address book at all.

To create a recipient list from the address book

1. Allocate an **ADRPARM** structure and a pointer to an **ADRLIST** structure.
2. Zero the memory in the **ADRPARM** structure and set the **ADRLIST** pointer to NULL.
3. Call **IAddrBook::Address** to display the address dialog box and populate the **ADRPARM** structure.
4. Call **IMessage::ModifyRecipients**, passing the **ADRLIST** pointer. This structure will contain the properties of each of the recipients selected by the user.

To add one or more recipients to a recipient list

1. Allocate an **ADRLIST** structure that contains one **ADRENTRY** structure for each of the recipients to be added. Make each **ADRENTRY** structure large enough to hold at least the following properties:

   PR_ADDRTYPE

PR_DISPLAY_NAME
PR_EMAIL_ADDRESS
PR_ENTRYID
PR_RECIPIENT_TYPE
PR_SEARCH_KEY

2. For each recipient and each property, place the property tag and value in the appropriate **ADRENTRY** structure. For example, the following code sample sets two properties for a recipient: its entry identifier and display name.

```
LPADRLIST lpAdrList = NULL;

hr = MAPIAllocateBuffer(numProps * sizeof(SPropValue),
                (LPVOID FAR *(&lpAdrList->aEntries[0].rgPropVals);

lpAdrList->aEntries[0].rgPropVals[0].ulPropTag = PR_ENTRYID;
lpAdrList->aEntries[0].rgPropVals[0].Value.bin      = RecipEID;

lpAdrList->aEntries[0].rgPropVals[1].ulPropTag = PR_DISPLAY_NAME;
lpAdrList->aEntries[0].rgPropVals[1].Value.LPSZ     = szDisplay;

// other properties

lpAdrList->aEntries[0].cValues=numProps;
```

3. Call **IMessage::ModifyRecipients** with the MODRECIP_ADD flag set.

For more information about adding recipients, see About Message Recipients.

Adding a Message Subject

Most message forms include a subject line where users can summarize the intent of a message. MAPI defines two properties that clients can set to describe their message subjects: PR_SUBJECT and PR_SUBJECT_PREFIX. PR_SUBJECT contains the text of the message subject. PR_SUBJECT_PREFIX contains the characters that make up the subject's prefix, if there are any. Both of these properties are optional; your client can choose to set one, both, or none.

For new outgoing messages, set PR_SUBJECT to a character string 128 bytes or less using the message's **IMAPIProp::SetProps** method. The 128 byte limit is not a limit imposed by MAPI; it is a limit imposed by some message store providers. To insure interoperability with providers that do impose it, limit your client's subjects to 128 bytes.

The PR_SUBJECT_PREFIX property is a character string that holds the prefix characters of a subject line. Therefore, for new messages, clients either do not set PR_SUBJECT_PREFIX or set it to a blank string.

When messages are saved in a folder, the message store provider calculates another subject property, PR_NORMALIZED_SUBJECT. PR_NORMALIZED_SUBJECT contains the message's subject with all prefix characters stripped out. For more information about the relationship between these three subject properties, see About Message Subject Properties.

Adding Message Text

Although some messages are made up of nothing more than a recipient list and a subject line, the content of most messages, specifically IPM.Note messages, includes text, either plain or formatted. Message text is stored in two properties: PR_BODY and PR_RTF_COMPRESSED. If your client is plain text-based, it will set PR_BODY. If your client supports formatted text in the Rich Text Format (RTF), it will set either PR_RTF_COMPRESSED only or both PR_RTF_COMPRESSED and PR_BODY, depending on the message store provider being used. When an RTF-aware client is using an RTF-aware message store, it sets PR_RTF_COMPRESSED only. When an RTF-aware client is using a non-RTF-aware message store, it set both properties.

Note Clients can use other types of formatted text in addition to RTF, but MAPI has not defined properties for managing types other than RTF.

To add formatted message text

1. Determine whether or not the message store that your client is using is RTF-aware. Perform the following tasks to find out if and how a message store supports RTF:
   a. Call the message store's **IMAPIProp::GetProps** method to retrieve the PR_STORE_SUPPORT_MASK property.
   b. Check for the STORE_RTF_OK value. If STORE_RTF_OK is set, the message store provider supports RTF text. If it is not set, the message store provider supports plain text only.
2. Set the appropriate message content property or properties. If your client is using an RTF-aware message store, set only the PR_RTF_COMPRESSED property. If your client is using a non-RTF-aware message store, set both PR_BODY and PR_RTF_COMPRESSED.

   Your client can call **RTFSync** to generate the PR_BODY property from PR_RTF_COMPRESSED.
3. To set the PR_RTF_COMPRESSED property:
   a. Call the message's **IMAPIProp::OpenProperty** method to open the PR_RTF_COMPRESSED property, specifying IID_IStream as the interface identifier and setting the MAPI_CREATE flag.
   b. Call the **WrapCompressedRTFStream** function, passing the STORE_UNCOMPRESSED_RTF flag if the STORE_UNCOMPRESSED_RTF value is set in the message store's PR_STORE_SUPPORT_MASK property.
   c. Release the original stream by calling its **IUnknown::Release** method.
   d. Call either **IStream::Write** or **IStream::CopyTo** to write the message text to the stream returned from **WrapCompressedRTFStream**.
   e. Call the **Commit** and **Release** methods on the stream returned from the **OpenProperty** method.

At this point, if the message store provider supports RTF, your client has done all that is required. Your client can depend on the message store provider to handle synchronize the message content and formatting and to create the PR_BODY property if necessary. RTF-aware message stores call **RTFSync** to handle the synchronization. If the RTF_SYNC_BODY_CHANGED flag is set to TRUE, the provider will recompute the PR_BODY property.

If your client's message store provider does not support RTF, your client must also add non-RTF message content by setting the PR_BODY_property.

To add non-formatted message text

1. Call the **IMAPIProp::OpenProperty** method to open the PR_BODY property with the **IStream** interface.
2. Call **IStream::Write** to write the message text data to the stream returned from **OpenProperty**.
3. Call the **RTFSync** function to synchronize the text with the formatting. Because this is a new message, set both the RTF_SYNC_RTF_CHANGED and RTF_SYNC_BODY_CHANGED flags to indicate that both the RTF and plain text version of the message text has changed. **RTFSync** will set several related properties that the message store provider requires, such as PR_RTF_IN_SYNC,

and write them to the message.

4. Call **IStream::Commit** and **IUnknown::Release** on the stream to commit the changes and free its memory.

Adding Rendering Information to Formatted Text

To indicate where in formatted text an attachment is to be rendered, your client must insert a sequence of placeholder characters in the message's PR_RTF_COMPRESSED property. The placeholder sequence is made up of the following characters:

```
\objattph
```

To add rendering information to formatted message text

- When writing the stream of text to the message's PR_RTF_COMPRESSED property, insert the placeholder sequence and a space character at the position where the attachment should be rendered.
- Set the PR_RENDERING_POSITION property of each attachment to a numeric value. The lowest value should be assigned to the PR_RENDERING_POSITION property of the first attachment to appear in the formatted text; the highest value to the last attachment.

Adding a Message Attachment

A message attachment is some additional data, such as a file, another message, or an OLE object, that your client can send or save along with the message. Your client can support all of the available types, some of them, or none of them. Attachments are considered subobjects of messages because they cannot be accessed independently. The message to which an attachment is attached must be open for the attachment to be usable.

To add an attachment to a message

1. Call the message's **IMessage::CreateAttach** method and pass NULL as the interface identifier. **CreateAttach** returns a number that uniquely identifies the new attachment within the message. The attachment number is stored in the PR_ATTACH_NUM property and is valid only as long as the message containing the attachment is open.
2. Call **IMAPIProp::SetProps** to set PR_ATTACH_METHOD to indicate how to access the attachment. PR_ATTACH_METHOD is required. Set it to:
   - ATTACH_BY_VALUE if the attachment is binary data.
   - ATTACH_BY_REFERENCE, ATTACH_BY_REF_RESOLVE, or ATTACH_BY_REF_ONLY if the attachment is a file.
   - ATTACH_EMBEDDED_MSG if the attachment is a message.
   - ATTACH_OLE if the attachment is an OLE object.
3. Set the appropriate attachment data property:
   - PR_ATTACH_DATA_BIN for binary data and OLE 1 objects.
   - PR_ATTACH_PATHNAME for files.
   - PR_ATTACH_DATA_OBJ for messages and OLE 2 objects.
4. Set PR_ATTACH_RENDERING to hold the graphic representation of the attachment for file or binary attachments. Do not set it for OLE objects, which store the rendering information internally, or for attached messages.
5. Set PR_RENDERING_POSITION to indicate where the attachment should be displayed. PR_RENDERING_POSITION applies only to clients that set the PR_BODY property. If your client only supports PR_RTF_COMPRESSED, place the following placeholder information in the compressed stream:

   ```
   \objattph
   ```

   To set PR_RENDERING_POSITION, assign either a number that represents an ordinal offset in characters, with the first character of PR_BODY being 0, if your client needs to know where in the message the attachment is rendered, or 0xFFFFFFFF, if your client does not render attachments within PR_BODY.
6. Set PR_ATTACH_FILENAME to indicate the short name of the file for a file attachment and PR_ATTACH_LONG_FILENAME to indicate the name of the file as supported on a platform that handles the long filename format. Both properties are optional. However, if your client sets PR_ATTACH_LONG_FILENAME, it should also set the short version, PR_ATTACH_FILENAME.
7. Set PR_DISPLAY_NAME to indicate the name for the attachment that can appear in a dialog box. PR_DISPLAY_NAME is optional.

To set PR_ATTACH_DATA_BIN

1. Call **IMAPIProp::OpenProperty** to open the property with the **IStream** interface.
2. If a file contains the data and it is open or if your client needs explicit control over buffer size, call **IStream::Write** in a loop to place the data in the stream.
3. Another option is to call **OpenStreamOnFile** to create a stream to access the data file and then call this stream's **IStream::CopyTo** method to copy the data to the stream returned by **OpenProperty**.
4. Call the new stream's **IStream::Commit** method.

To set PR_ATTACH_DATA_OBJ

1. Call **IMAPIProp::OpenProperty** to open the property with the **IStreamDocfile** interface to create a stream that works with structured storage. If this call succeeds, create the stream with the same steps outlined for setting PR_ATTACH_DATA_BIN.
2. If **OpenProperty** fails:
   a. Call **OpenProperty** again asking for **IStorage**.
   b. Call **StgOpenStorage** to open the OLE object and return a storage object.
   c. Call the returned storage object's **IStorage::CopyTo** method to copy to the storage object returned from **OpenProperty**.
   d. Call the new storage object's **IStorage::Commit** method.

To set PR_ATTACH_PATHNAME

1. Allocate an **SPropValue** structure, setting the **ulPropTag** member to PR_ATTACH_PATHNAME and the **Value.LPSZ** member to the character string that represents the filename.
2. Call the attachment's **IMAPIProp::SetProps** method.

Note If your platform supports long filenames, your client needs to set both PR_ATTACH_PATHNAME and PR_ATTACH_LONG_PATHNAME. It might be necessary to make an operating system call to retrieve the short filename.

For more information, see About Message Attachments.

Saving an Outgoing Message

When your client is ready to make an outgoing message a permanent part of a folder, call **IMAPIProp::SaveChanges** on the message and all of its attachments. It is a good idea to specify the KEEP_OPEN_READWRITE flag, which allows the message to be modified at a later time. Other flags your client can set include FORCE_SAVE, which indicates that the message or attachment should be closed after changes are committed, KEEP_OPEN_READONLY, which indicates that no further changes will be made, and the flag to allow the message store provider to batch client requests, MAPI_DEFERRED_ERRORS.

It is essential that your client calls **SaveChanges** for every attachment in the message before calling **SaveChanges** for the message. If your client fails to save an attachment, the attachment will not be included with the message when it is sent and it will not appear in the message's attachment table. If your client fails to save the message after saving all of the attachments, both the message and the attachments will be lost.

When **SaveChanges** is called, the message store provider updates the following properties.

- PR_DISPLAY_TO lists all primary recipients.
- PR_DISPLAY_CC lists all carbon copy recipients.
- PR_DISPLAY_BCC lists all blind carbon copy recipients.
- PR_MESSAGE_FLAGS sets MSGFLAG_HASATTACH if one or more attachments have been saved and clears MSGFLAG_UNMODIFIED to show the message has changed.
- PR_MESSAGE_SIZE contains the most current size of the message.

If **SaveChanges** returns MAPI_E_CORRUPT_DATA, assume that the data being saved is now lost. Message store providers that use a client-server model for their implementation might return this value when a network connection is lost or the server is not running. Before returning an error to the user, try to write and save the data a second time by making a call to **SetProps** followed by another call to **SaveChanges**. If the data is cached locally, this should not be a problem. However, if there is no local cache or the second **SaveChanges** call fails, display an error to alert the user to the problem.

Sending an Outgoing Message

Your client calls **IMessage::SubmitMessage** when it is ready to send a message. **SubmitMessage** places the message in the outgoing queue and sets the MSGFLAG_SUBMIT flag in the message's PR_MESSAGE_FLAGS property. The message store provider, if tightly coupled to a transport provider, gives the message directly to the transport which delivers it to the messaging system. If not tightly coupled, the message store provider informs the MAPI spooler that the outgoing queue has changed and the MAPI spooler transfers the message to an appropriate transport provider.

If your client allows its users to cancel a send operation, call **IMsgStore::AbortSubmit** to implement this feature. **AbortSubmit** removes the message from the output queue. Users can be allowed to stop a send from happening until the message is given to the underlying messaging system.

If **SubmitMessage** returns MAPI_E_CORRUPT_DATA, assume that the data being sent is now lost. Before attempting to send a second time, re-write the message by calling **SetProps** and **SaveChanges**. Display an error to the user if these **IMAPIProp** calls fail or if **SubmitMessage** fails a second time.

After a successful call to **SubmitMessage**, free any memory that was allocated for the recipient list and release the message and its attachments. Once a message has been sent, MAPI does not permit any further operations on the pointers for these objects. The one exception is calling **IUnknown::Release**. No other calls are allowed because many message store providers invalidate entry identifiers for messages that have been sent.

Posting a Message

Posting a message is similar to sending a message. The main difference is the destination. Rather than being directed to one or more recipients across one or more messaging systems, a posted message remains in a folder in your client's message store.

To post a message

1. Call **IMAPIFolder::CreateMessage** to create the message.
2. Call the message's **IMAPIProp::SetProps** method to set:
   - The MSGFLAG_UNSENT flag in the PR_MESSAGE_FLAGS property.
   - The PR_SENDER properties.
   - The PR_SENT_REPRESENTING properties.
   - The PR_RECEIPT_TIME property.
3. Call the message's **IMAPIProp::SaveChanges** method to save the message.

Notice that your client does not create a recipient list. Instead, it must set several properties that are normally set by a transport provider for a sent message.

Resending an Undelivered Message

When your client application receives a nondelivery report, it should contain a copy of the undelivered message as an attachment. Your client can choose whether or not to allow a user to attempt a resend. Most clients use a form for resending that is provided through MAPI. The form can be a custom version registered by your client to use with resent messages of specific classes or a default form provided by MAPI. Your client can also implement the resending of a message manually.

To resend an undeliverable IPM message

1. Create a new outgoing message in the Outbox folder and copy the properties from the undelivered message to the new message. Do not copy PR_MESSAGE_CLASS, PR_MESSAGE_FLAGS, or PR_ORIGINAL_ENTRYID.
2. Set the new message's PR_MESSAGE_CLASS property to the message class of the original message.
3. Set the MSGSTATUS_RESEND flag in the new message's PR_MESSAGE_FLAGS property.
4. Copy the recipient list and the corresponding PR_RECIPIENT_TYPE properties from the undelivered message to the new message.
5. Set the MAPI_SUBMITTED flag in the PR_RECIPIENT_TYPE property for all recipients.
6. Make a copy of all recipients that failed to receive the message and set each of their PR_RECIPIENT_TYPE properties to MAPI_P1.
7. Call **IMessage::SubmitMessage** to save and send the new message.

Handling Replies and Forwarded Messages

Before an outgoing reply or forwarded message can be created, your client must open its default message store, the folder that is designated to hold outgoing messages, and the folder that is holding the original message. Typically the outgoing message folder is the Outbox and the folder holding the original message is the Inbox. See Opening the Outbox Folder for information about how to open this folder. Also, the entry identifier for the original message must be accessible.

A reply or forwarded message is based in part on the properties of the original message, the message that is being replied to or forwarded. Some of the properties are copied exactly and some are explicitly excluded.

The properties that are copied, excluded, and set vary somewhat depending on whether your client is replying to or forwarding a message and the properties that are available from the original message. To copy message properties, your client can call either **IMAPIProp::CopyTo** or **IMAPIProp::CopyProps** on the original message. With **IMAPIProp::CopyTo**, your client specifies the particular properties to be excluded; with **IMAPIProp::CopyProps**, it specifies the properties to be included.

To create an outgoing reply or forwarded message

1. Call the Outbox's **IMAPIFolder::CreateMessage** method to create the new reply or forwarded message.
2. Call the original message's **IMAPIProp::CopyTo** method to transfer properties to the new message and make modifications as necessary.
3. Call the new message's **IMAPIProp::SetProps** method to set properties that are unique to reply or forwarded messages.
4. Call the new message's **IMAPIProp::SaveChanges** method to save the message or **IMessage::SubmitMessage** to save and send it.

Setting Properties on a Reply or Forwarded Message

Reply and forwarded messages require many properties, some that are copied from the original message and some that are new.

To set properties on a reply or forwarded message

1. Copy PR_BODY or PR_RTF_COMPRESSED, depending on whether or not your client supports formatted text, and add some separator text.
2. Copy PR_CONVERSATION_TOPIC.
3. Update PR_CONVERSATION_INDEX by calling **ScCreateConversationIndex** and passing in the value of the original message's PR_CONVERSATION_INDEX property.
4. Set the PR_SENT_REPRESENTING properties to the corresponding values in the PR_RCVD_REPRESENTING properties. Do not copy this group of properties from the original message.
5. Copy PR_MESSAGE_ATTACHMENTS for forwarded messages only.
6. Copy PR_MESSAGE_RECIPIENTS only for replies that go to all recipients in the recipient list.
7. If your client uses a subject prefix, copy PR_NORMALIZED_SUBJECT from the original message and concatenate the prefix onto the beginning of the string. Set the PR_SUBJECT property to this new string that includes the prefix and the original subject.
8. If your client uses a non-standard prefix, such as a string that is longer than three characters, set PR_SUBJECT_PREFIX to its value. Otherwise, do not set PR_SUBJECT_PREFIX. Typically clients use RE: or FW: as prefixes for their reply and forward messages.
9. Set each of the entries in PR_REPLY_RECIPIENT_ENTRIES and PR_REPLY_RECIPIENT_NAMES to the entry identifier and display name of a primary recipient for replies only. Do not copy these properties from the original message. PR_REPLY_RECIPIENT_ENTRIES and PR_REPLY_RECIPIENT_NAMES must be kept synchronized, meaning that they must contain the same number of entries and entries at the same position in each of the properties must represent the same recipient.
10. Do not copy the following properties from the original message to the new reply or forwarded message:

PR_CLIENT_SUBMIT_TIME	PR_MESSAGE_DELIVERY_TIME
PR_MESSAGE_DOWNLOAD_TIME	PR_MESSAGE_FLAGS
PR_ORIGINATOR_DELIVERY_REPORT_REQUESTED	PR_RCVD_REPRESENTING properties
PR_READ_RECEIPT_ENTRYID	PR_READ_RECEIPT_REQUESTED
PR_RECEIVED_BY properties	PR_REPORT_ENTRYID
PR_SENDER properties	PR_SENTMAIL_ENTRYID
PR_SUBJECT_PREFIX	

Selecting a Recipient for a Reply or Forwarded Message

Your client can enable users to reply only to the sender of the original message or to all of the primary and carbon copy recipients in addition to the sender. When a reply is sent only to the sender, its recipient list contains only that single entry, as represented by the original message's PR_SEND_REPRESENTING properties. When a reply is sent to all recipients, its recipient list is based on the original message's recipient list, with blind carbon copy recipients and PR_RECEIVED_BY_SEARCH_KEY, the recipient that represents your client's user, removed.

To create a recipient list when replying to all recipients

1. Call the original message's **IMessage::GetRecipientTable** method to access its recipient table.
2. Call **HrQueryAllRows** to retrieve all of the rows in the table. Determine if each row represents a primary or carbon copy recipient and should remain in the list or if it represents a blind carbon copy recipient or your client's user and should be removed from the list.
3. Differentiate between recipient types by looking at the PR_RECIPIENT_TYPE column. This column will be set to MAPI_TO for primary recipients, MAPI_CC for carbon copy recipients, and MAPI_BCC for blind carbon copy recipients.
4. Compare the PR_SEARCH_KEY column with the PR_RECEIVED_BY_SEARCH_KEY property of the original message to determine if the row represents the user.
5. Remove unwanted rows from the recipient list by calling **MAPIFreeBuffer** to free the memory associated with the corresponding entries in the recipient table's **SRowSet** structure. Set all of the values in the property value array to zero, all of the **cValues** members to zero, and all of the **lpProps** members in each **SRow** structure in the **SRowSet** to NULL.
6. Add the sender to the recipient list, as represented by the original message's PR_SENT_REPRESENTING_NAME and PR_SENT_REPRESENTING_ENTRYID properties. Check that the sender is not duplicated in the list.
7. Call the reply message's **IMessage::ModifyRecipients** method, setting the *ulFlags* parameter to zero, to create a new recipient list for the reply or forwarded message based on the list from the original message.

Note Before calling **IMessage::ModifyRecips** to store changes in the recipient list, your client can allow users to make additional modifications. Allowing users to make changes in a recipient list is an optional client feature.

Copying an Attachment on a Forwarded Message

When your client application is forwarding a message that includes one or more attachments, it should include all of the properties associated with those attachments. Attachments are never included with a reply.

There are a few ways to copy message attachments from the message being forwarded. The first way is to call the original message's **IMAPIProp::CopyProps** method to copy the PR_MESSAGE_ATTACHMENTS property, causing all of the attachment information to be copied.

The second way is to call the original message's **IMAPIProp::CopyTo** method. Make one call to copy all of the message properties, including the attachments. Do not make a separate **CopyTo** call to copy only the attachments.

The third way is to follow the following procedure for each attachment to be copied:

1. Call the new forwarded message's **IMessage::CreateAttach** method to create a new attachment.
2. Call the original message's **IMessage::OpenAttach** method to open the attachment to be copied.
3. Call the original message's **IMAPIProp::CopyTo** method to copy all of the attachment properties from the old attachment to the new one.

Format the message content by adding indentation and a header paragraph that includes the original sender, date of transmission, subject, and recipient list. Do not include Internet-style prefix characters with the content.

Handling Incoming Messages

Clients that have registered for new mail notifications learn of new incoming messages when the message store provider calls their **IMAPIAdviseSink::OnNotify** method. Clients register for new mail notifications by calling the message store's **IMsgStore::Advise** method with a NULL entry identifier. Clients that do not register for new mail notifications must open their receive folders periodically to check for the arrival of new messages.

Message store providers pass a NEWMAIL_NOTIFICATION structure in their **IMAPIAdviseSink::OnNotify** call that contains the message class of the new incoming message. Your client might want to handle only those messages with classes that it supports and ignore any incoming messages with foreign message classes.

To handle an incoming message

- If your client only supports specific message classes, determine if the message class of the incoming message is a class that your client supports.
- As an option, open the receive folder that contains the message and display its contents table (optional if your client already has the entry identifier of the message).
- If your client is interactive, determine the form to be used to display the message to the user.
- Process the message and its attachments.

Selecting a Receive Folder

A receive folder is where incoming messages of a particular class are placed. For IPM and related report messages, MAPI assigns the Inbox as the default receive folder. For IPC and related report messages, MAPI assigns the root folder of the message store as the default receive folder. Your client can change these assignments or make additional assignments for other message classes. Making explicit receive folder assignments for your client's supported message classes is optional.

When an incoming message class does not have an assigned receive folder, the message store provider automatically uses the receive folder for the class that matches the longest possible prefix of the incoming class. For example, if your client receives a message of class IPM.Note.MyDocument and the only receive folder that has been established is the Inbox for IPM messages, this message will be placed in the Inbox because IPM.Note.MyDocument derives from the base class IPM.

When your client is assigning a receive folder for IPC messages, never use a folder from the IPM subtree. These folders should be reserved for IPM messages only. Use instead a folder that is contained within the message store's root folder.

To create a receive folder for an IPM message class

1. Call the message store's **IMAPIProp::GetProps** method to retrieve the PR_IPM_SUBTREE_ENTRYID property.
2. Call **IMsgStore::OpenEntry** with PR_IPM_SUBTREE_ENTRYID as the entry identifier to open the root folder of the IPM subtree in the message store.
3. Call **IMAPIFolder::CreateFolder** to create the receive folder.
4. Call **IMsgStore::SetReceiveFolder** to map the new folder to your client's IPM message class.

To create a receive folder for an IPC message class

1. Call **IMsgStore::OpenEntry** with a null entry identifier to open the root folder of the message store.
2. Call **IMAPIFolder::CreateFolder** to create the receive folder.
3. Call **IMsgStore::SetReceiveFolder** to map the new folder to your client's IPC message class.

Assign the receive folder that your client uses for messages for related report messages. For example, if your client receives IPM.Note messages, set up one receive folder for future IPM.Note messages and the same receive folder for future Report.IPM.Note messages.

Note Your client can use the **IMsgStore::SetReceiveFolder** method for more than establishing a receive folder association with messages of a particular class. The character string passed to **SetReceiveFolder** does not necessarily need to represent a message class. For example, you can map a folder to a character string, making it easier to find the folder when necessary. The most obvious alternative for associating folders to strings is to use message store named properties, an approach that is not recommended because MAPI does not define a range of message store properties for use by clients and message store providers do not always support named properties.

Opening a Receive Folder

To access an incoming message, open the receive folder that is registered for the message class of the message. Receive folders for IPM message classes will typically be the Inbox; for IPC messages, they will typically be hidden folders. The message store method **GetReceiveFolder** can be used to retrieve the entry identifier for any receive folder for any message class. The **OpenEntry** method for the message store can be used to open the receive folder.

To open a receive folder

1. Call **IMsgStore::GetReceiveFolder**, specifying a character string that represents the message class of the incoming message. If there is no registered receive folder for the message class, **GetReceiveFolder** chooses the receive folder whose associated message class matches the longest possible prefix of the message class passed in.
2. Call **IMsgStore::OpenEntry** with the entry identifier returned by **GetReceiveFolder** to open the receive folder.

It is possible although undesirable to receive messages with no class, but your client should never receive a message with a NULL class. When clients fail to set the PR_MESSAGE_CLASS property on their outgoing messages, message store providers set it to IPM. On incoming messages, transport providers set it either to IPM or IPM.Note.

Determining Message Ownership

Your client might need to locate all messages that it has created and saved during composition or has sent to one or more recipients. There are a few ways to evaluate message ownership. For example, it is usually safe to assume that an incoming message was sent by your client if:

- Your client keeps outgoing messages in the Sent Items folder.
- The Sent Items folder is the same as the parent folder of the incoming message.

To compare the Sent Items folder with the incoming message's parent folder, call **IMsgStore::CompareEntryIDs** and specify PR_IPM_SENTMAIL_ENTRYID and PR_PARENT_ENTRYID as the entry identifiers to compare. A potential drawback to this assumption is that it is possible for messages to have moved from the Sent Items folder to another folder.

Another way to evaluate message ownership is to assume that a message was never handled by a transport provider if all of the properties that are rarely set by anyone except transport providers are missing on the incoming message. These properties include:

PR_RECEIVED_BY properties
PR_MESSAGE_DOWNLOAD_TIME
PR_TRANSPORT_MESSAGE_HEADERS
PR_MESSAGE_TO_ME
PR_MESSAGE_CC_ME
PR_MESSAGE_RECIP_ME

Transports are required to set the PR_SENDER and PR_SENT_REPRESENTING groups of properties, but sometimes also clients set these.

Your client can find all received messages that it has sent as well as messages that it has posted and, depending on your client, messages that it has saved during composition by performing the following tasks:

1. Call **IMAPISession::GetStatusTable** to retrieve the status table.
2. Retrieve all of the rows in the table, saving only the distinct values for the PR_IDENTITY_SEARCH_KEY column.
3. Compare the value of the message's PR_SENT_REPRESENTING_SEARCH_KEY with the value of each of the PR_IDENTITY_SEARCH_KEY values. If any of these match, your client can assume that it sent the message.

If the PR_SENT_REPRESENTING properties are missing, your client can assume that it either sent the message or saved it during composition and has not yet sent it. To determine if a message has been sent, check if the MSGFLAG_UNSENT flag is set in its PR_MESSAGE_FLAGS property. If it is set, the message has not been sent.

Viewing a Folder Contents Table

To make summary information about new messages available, retrieve the contents table of the folder containing the messages and display the relevant columns from each row.

To display the contents of a folder that contains new incoming messages

1. Call **IMsgStore::GetReceiveFolder** to retrieve the entry identifier of the receive folder for the message class of the incoming message.
2. Call **IMsgStore::OpenEntry**, passing the entry identifier returned from **GetReceiveFolder**, to open the receive folder.
3. Call the folder's **IMAPIContainer::GetContentsTable** method to open its contents table.
4. Optionally, limit your client's view of the contents table to rows representing the incoming message class by:
   a. Creating a property restriction in an **SPropertyRestriction** structure that matches the PR_MESSAGE_CLASS property with the message class of the incoming message.
   b. Creating a bitmask restriction in an **SBitmaskRestriction** structure that uses PR_MESSAGE_FLAGS as the property tag and the MSGFLAG_UNREAD value as the mask.
   c. Creating a restriction in an **SAndRestriction** structure that joins the property and bitmask restrictions.
   d. Passing the joined restriction structures to the contents table's **IMAPITable::Restrict** method.
5. Call **IMAPITable::QueryRows** to retrieve all of the rows from the contents table for processing.
6. Rely on new mail notifications to advise you of subsequent incoming messages.

Opening an Incoming Message

When your client application is informed that a message has been placed in a receive folder, open the message by passing its entry identifier to one of the following methods, listed in order from fastest to slowest in response time:

- The receive folder's **IMAPIFolder::OpenEntry** method
- The message store's **IMsgStore::OpenEntry** method
- **IMAPISession::OpenEntry**

If your client has registered for message store notifications, it will learn of new incoming messages to be processed through the new mail notification that it will receive. Included in the NEWMAIL_NOTIFICATION structure that is sent in the **IMAPIAdviseSink::OnNotify** call is the entry identifier and message class of the incoming message, the entry identifier of its folder, and its PR_MESSAGE_FLAGS property. If your client has not registered to receive this type of notification, it will have to periodically scan each receive folder to learn of incoming messages.

Interactive clients display incoming messages to their users with a form that is appropriate for the message class of the message. The choice of forms is based on the message class of the incoming message. Messages that belong to the IPM class use a default form implemented by MAPI. Messages that belong to custom classes defined by clients can use either client-defined specialized forms or MAPI's default form.

Opening Message Text

The text of a message is stored either in its PR_BODY property or PR_RTF_COMPRESSED property. If your interactive client supports Rich Text Format (RTF), open PR_RTF_COMPRESSED. If your client does not support RTF, open PR_BODY. Because the contents of a message can be large regardless of whether or not it is formatted, use **IMAPIProp::OpenProperty** to open these properties and specify IID_IStream as the interface identifier, if necessary.

To display formatted message text

1. If your client is using a non-RTF aware message store, as indicated by the absence of the STORE_RTF_OK flag in the store's PR_STORE_SUPPORT_MASK property:
   a. Call the message's **IMAPIProp::GetProps** method to retrieve the PR_RTF_IN_SYNC property.
   b. Call **RTFSync** to synchronize the message's PR_BODY property with the PR_RTF_COMPRESSED property. Pass the RTF_SYNC_BODY_CHANGED flag if either the call to retrieve PR_RTF_IN_SYNC failed because the property does not exist or it is set to FALSE.
   c. If **RTFSync** returned TRUE, indicating that changes were made, call the message's **IMAPIProp::SaveChanges** method to permanently store them.
2. Regardless of whether or not your client is using either an RTF-aware message store:
   a. Call **IMAPIProp::OpenProperty** to open the PR_RTF_COMPRESSED property.
   b. If PR_RTF_COMPRESSED is not available, call **OpenProperty** to open the PR_BODY property.
   c. Call the **WrapCompressedRTFStream** function to create an uncompressed version of the compressed RTF data, if available.
   d. Copy the plain text from the stream to the appropriate place in the message form.

To display plain message text

1. Call the message's **IMAPIProp::GetProps** method to retrieve the PR_BODY property.
2. If **GetProps** returns either PT_ERROR for the property type in the property value structure or MAPI_E_NOT_ENOUGH_MEMORY, call the message's **IMAPIProp::OpenProperty** method. Pass PR_BODY as the property tag and IID_IStream as the interface identifier.
3. Copy the plain text from the stream to the appropriate place in the message form.

Rendering Attachments in Plain Text

To render an attachment in a message with plain text, retrieve the attachment's PR_RENDERING_POSITION property and apply it to the data in the PR_ATTACH_RENDERING property. There are two ways to retrieve PR_RENDERING_POSITION:

- Open the attachment by calling the message's **IMessage::OpenAttach** method and then ask for the PR_RENDERING_POSITION property by calling the attachment's **IMAPIProp::GetProps** method.
- Call the message's **IMessage::GetAttachmentTable** method to access its attachment table and retrieve the column that holds the PR_RENDERING_POSITION property. This way is always preferable.

Keep in mind that many RTF-aware message stores do not calculate PR_RENDERING_POSITION until a client requests the PR_BODY property of a message. Until that time, PR_RENDERING_POSITION usually represents an approximate value. Message store providers are allowed to supply clients with an approximate value to enhance performance.

The rendering for a file or binary attachment is stored in its PR_ATTACH_RENDERING property. Your client has the choice of retrieving PR_ATTACH_RENDERING in the same ways as it retrieved PR_RENDERING_POSITION: directly from the attachment or from the attachment table. For PR_ATTACH_RENDERING, the first strategy, although more time-consuming, is the safer of the two. Because some message store providers truncate their table columns to 255 bytes, or in a few cases, 510 bytes, it is difficult to be sure that the PR_ATTACH_RENDERING column contains the complete rendering. When retrieving the property directly from the attachment, it will always be complete.

Neither OLE or message attachments set PR_ATTACH_RENDERING. Instead, rendering information for OLE 1 attachments is stored in the content stream. For OLE 2 attachments, it is stored in a special child stream of the storage object. Rendering information for message attachments is available through the form manager.

To retrieve the rendering for a message attachment, your client must

1. Use the message class of the message to access the form manager.
2. Access the form manager's PR_MINI_ICON property.

Rendering Attachments in Formatted Text

RTF-aware clients can retrieve rendering position information from RTF message text by looking for the following escape sequence in the message's PR_RTF_COMPRESSED property:

```
\objattph
```

To display attachments using the PR_RTF_COMPRESSED property of a message

1. Call **IMessage::GetAttachmentTable** to access the attachment table.
2. Build a property restriction that limits the table to rows that have PR_RENDERING_POSITION not equal to -1.
3. Call **IMAPITable::Restrict** to register the restriction.
4. Call **IMAPITable::SortTable** to sort the attachments.
5. Call **IMAPITable::QueryRows** to retrieve the appropriate rows.
6. Call the message's **IMAPIProp::OpenProperty** method to retrieve PR_RTF_COMPRESSED with the **IStream** interface.
7. Scan the stream, looking for the rendering placeholder, `\objattph`. The character following this placeholder is the place for the next attachment in the sorted list.

Opening an Attachment

Before your client can open an attachment, its attachment number must be available. The attachment number is stored in the PR_ATTACH_NUM property. An attachment's PR_ATTACH_NUM property is similar to a message's short-term entry identifier in that it is used to open the object. Your client can access an attachment number through a message's attachment table. The attachment number is a required column in this table.

To open every attachment in a message

1. Call the message's **IMessage::GetAttachmentTable** method.
2. Call **HrQueryAllRows** to retrieve all of the rows in the table.
3. For each row:
   a. Open the attachment by passing the attachment number represented in the PR_ATTACH_NUM column in a call to the message's **IMessage::OpenAttach** method.
   b. Open the attachment's data by first examining the attachment's PR_ATTACH_METHOD property.
   c. If PR_ATTACH_METHOD is set to ATTACH_BY_REF_ONLY, call **IMAPIProp::GetProps** to retrieve the PR_ATTACH_PATHNAME property.
   d. If PR_ATTACH_METHOD is set to ATTACH_BY_VALUE, call **IMAPIProp::OpenProperty** to open the PR_ATTACH_DATA_BIN property with the **IStream** interface.
   e. If PR_ATTACH_METHOD is set to PR_ATTACH_OLE and the attachment is an OLE 2 object:
      1. Call **IMAPIProp::OpenProperty** to open the PR_ATTACH_DATA_OBJ property with the **IStreamDocfile** interface . Attempt to use this interface. **IStreamDocfile** is the best choice for accessing OLE 2 attachments because it involves the least amount of overhead.
      2. If the **OpenProperty** call fails, call it again to retrieve the PR_ATTACH_DATA_BIN property with the **IStreamDocfile** interface.
      3. If this second **OpenProperty** call fails, try again to call **OpenProperty** to retrieve PR_ATTACH_DATA_OB\J. However, rather than specifying **IStreamDocfile**, specify the **IStorage** interface.

Your client can request that an attachment is opened in read/write or read-only mode. Read-only is the default mode, and many message store providers open all attachments in this mode regardless of what clients request. Pass the MAPI_BEST_ACCESS flag to request that the message store provider grant the highest level of access it can and then retrieve the open attachment's PR_ACCESS_LEVEL property to determine the level of access that was actually granted.

Handling Notifications

Notifications enable one object to inform another object that it has undergone a change. The changed object, referred to as the advise source, might be the session object, under MAPI's control, or an object created by a service provider, such as a message. The informed object, referred to as the advise sink, usually contains either an implementation of the **IMAPIAdviseSink** interface or the **IMAPIViewAdviseSink** interface and is within a client application. Objects that can act as advise sources implement an **Advise** method, which is called by clients to register for notifications. One of the parameters to **Advise** is a pointer to an implementation of **IMAPIAdviseSink** or **IMAPIViewAdviseSink**. The advise source caches this pointer so that it can call **IMAPIAdviseSink::OnNotify** or one of the methods in **IMAPIViewAdviseSink** when a change occurs for which the advise sink has registered.

Because receiving notifications enables users to view the most up-to-date information, it is recommended that all clients register for and handle notifications. However, it is optional.

Timing a Notification

Because event notification is an asynchronous process, your client can be notified at any time, not necessarily immediately after the event has occurred. The timing of calls to your client's **IMAPIAdviseSink::OnNotify** method varies depending on the service provider implementing the advise source. Service providers can notify your client either:

- Simultaneously with the event.
- Directly after the event.
- At some later point following the event, possibly after an **Unadvise** call.

Most service providers call **OnNotify** after the MAPI method responsible for the event has returned to its caller. For example, notifications on messages are sent either when changes are saved through a call to **IMAPIProp::SaveChanges** or when the message object is released with a call to **IUnknown::Release**. Until the notification is sent, no changes are visible in the message store.

Clients can receive notifications from an advise source after they have called **Unadvise** to remove their registration. Be sure to release your client's advise sink only after its reference count has fallen to zero, not following a successful **Unadvise** call. Do not assume that because your client called **Unadvise** that the advise sink is no longer necessary.

Ensuring a Thread Safe Notification

If your client runs on a multithreaded platform such as Windows NT, it may need insurance that calls to its **IMAPIAdviseSink::OnNotify** methods occur on a particular thread. Because calls to **OnNotify** can typically occur on any thread, it is possible for your client to receive notifications on unexpected and unwanted threads, leading to errors that are difficult to debug.

To guarantee that calls to **OnNotify** for a particular notification are made on the same thread that was used for its **Advise** call, call **HrThisThreadAdviseSink** before calling **Advise**. **HrThisThreadAdviseSink** creates a new advise sink object from your client's advise sink object. Pass this new object in the call to **Advise**. All clients with advise sink objects that might not work outside of the context of a particular thread should always register advise sink objects created with **HrThisThreadAdviseSink**.

Clients that run on 16-bit platforms can call **HrThisThreadAdviseSink** if desired without any negative side effects; the only processing that occurs is an increment of the reference count on the original advise sink.

Handling an Object Notification

The vast majority of object notifications are generated by the message store provider. Because message store providers can be flexible in how they implement notifications, there is variation from provider to provider. Client applications must be aware that they will not always receive the same type of notification in response to a particular event from all providers.

At one end of the spectrum are the message store providers that support "rich" notifications; these providers generate the most descriptive notification possible and send that notification for all registered advise sources. At the other end are the message store providers that support limited notifications; these providers send the most general notifications for a restricted number of advise sources. For example, a message being moved in one provider will cause the *fnevObjectMoved* notification to be sent for the message store, the original parent of the moved message, the new parent of the moved message, and the message itself while in another provider the same event will cause the *fnevObjectChanged* notification to be sent only for the message store or the message. Some message store providers do not allow clients to register with message stores at all.

When your client receives an object modified notification, bear in mind that the property tag array portion of the OBJECT_NOTIFICATION may or may not be NULL. Message store providers are not required to insert property information in this array and most do not. Implement your **OnNotify** method to handle the case where the *lpPropTagArray* pointer is NULL.

For most, if not all object notifications, your client must update the view of the affected folder or folders.

Registering for a Notification

Your client can register for address book or message store notifications as part of its initialization process. MAPI supports notification on the address book regardless of whether any of the address book providers support it. Support for notification on message stores depends on the particular message store provider. To determine whether a particular message store provider supports notifications, your client must check its PR_STORE_SUPPORT_MASK property. If the message store that your client is using supports notifications, the STORE_NOTIFY_OK flag will be set. If it doesn't provide notification support, this flag will not be unset.

Register for notifications by calling an **Advise** method. Many objects implement **Advise** and clients can register with those objects in a variety of ways. Clients can register through the session, the address book, or with a specific service provider object, such as a folder.

To register through the session, call the **IMAPISession::Advise** method. **IMAPISession::Advise** enables clients to register for critical error notifications on the overall session or for various notifications on individual objects. Sessions send critical error notifications to clients logged onto shared sessions when another client using the shared session calls **MAPILogoff**. To register for session notifications, pass NULL for the entry identifier parameter. To register for notifications on a particular object, specify the entry identifier of the object and one or more event types that the object supports. **IMAPISession::Advise** forwards the call to the appropriate service provider, as determined by the MAPIUID portion of the entry identifier. Use **IMAPISession::Advise** rather than a service provider's **Advise** method to register for object notifications as a short cut.

To register through the address book, call **IAddrBook::Advise**. Registering with the address book is similar to registering with the session. To register for critical error notification from the address book, pass NULL for the entry identifier. To register for notifications on a particular address book object, specify the appropriate entry identifier and event or events of interest. Your client should be aware that many address book providers do not support notifications on individual objects. Rather, they support table notifications on their contents and hierarchy tables. Register for table notifications by calling the particular table's **IMAPITable::Advise** method.

To register for notifications

1. Create a MAPI advise sink object and increment its reference count.
2. If appropriate, call **HrThisThreadAdviseSink** to create an advise sink object that wraps your client's original advise sink.
3. Release the original advise sink.
4. Call the appropriate **Advise** method to complete the registration.
5. Cache the connection number returned from **Advise**.
6. If using a wrapped advise sink, release it. Once the wrapped advise sink is registered, your client no longer needs it.

It is good practice to release the advise sink your client implements or creates with **HrAllocAdviseSink** immediately after a successful return from an **Advise** call. This is because it is possible for service providers to release your advise sink after the **Advise** call, but before an **Unadvise** call is made. Once your client has given the advise source a pointer to its advise sink and the reference count has been incremented on the advise sink, it is wise to release it unless your client has a long term use for it.

Note   All connection numbers that represent valid advisory registrations will not be released until the **Unadvise** call is made.

Registering for a Message Store Notification

To register for message store notifications, call either the **IMAPISession::Advise** or **IMsgStore::Advise** method and specify the entry identifier of the message store, folder, or message to be the advise source. As do address book providers, message store providers support both object and table notifications. Whether your client registers with particular message store objects, with the folder hierarchy and contents tables that describe these objects, or with both objects and tables depends on the notifications your client expects to see and the message store provider's implementation. Some message store providers generate multiple notifications for events; other message store providers do not. For example, suppose your client copies a message to a folder with which it has registered to receive both object copied and object moved notifications. Whether or not your client actually receives the object copied notification depends on:

- The method that your client called to perform the copy. This could be **IMAPIFolder::CopyMessages**, **IMAPIProp::CopyTo**, or **IMAPIProp::CopyProps**.
- How the message store provider implements the copy method.
- Whether or not the message store provider supports object copied notifications on folders.

Because there are no strict guidelines that describe how to implement event notification for message store providers, clients cannot expect consistent behavior. MAPI does make recommendations as to how message store providers implement event notification and the following table outlines these recommendations. Read the table as follows: after your client performs the operation in the first column, it should expect to receive a notification of the type listed in the second column if it has registered for that type with the object listed in the third column. For example, after your client has created a folder, it will receive an *fnevObjectCreated* notification only if it has registered for *fnevObjectCreated* notifications with the message store.

Operation	Event type	Advise source
Create a folder	*fnevObjectCreated*	Message store
Delete a folder	*fnevObjectDeleted*	Message store Deleted folder
Move a folder from one folder to another	*fnevObjectMoved*	Message store Moved folder
Copy a folder from one folder to another	*fnevObjectCopied*	Message store and copied folder (no *fnevObjectCreated* notification sent for the new copy of the folder)
Change in a computed folder property (PR_SUBFOLDERS, PR_CONTENT_UNREAD, PR_CONTENT_COUNT)	*fnevObjectChanged*	Message store Changed folder (No notification to parent folder)
Create a message	*fnevObjectChanged*	Message store
Delete a message, causing a change in the parent folder's PR_CONTENT_COUNT	*fnevObjectDeleted*	Message store Deleted message

Move a message from one folder to another	*fnevObjectMoved*	Message store Moved message
Copy a message from one folder to another	*fnevObjectCopied*	Message store Copied message (No *fnevObjectCreated* notification for new copy of the message)
Save a message, causing a change in the parent folder's PR_CONTENT_COUNT property	*fnevObjectCreated*	Message store on first save only
Save a message	*fnevObjectModified*	Message store on saves after the first save Changed message (No notification to parent folder)
Complete a search operation	*fnevSearchComplete*	Message store Search folder
New message	*fnevNewMail*	Message store

Registering for an Address Book Notification

Address book notifications enable your client to learn of events that occur to any recipient or to a particular recipient in the address book. Your client can register for these notifications either through the MAPI address book by calling **IAddrBook::Advise** or through an address book container's hierarchy or contents table by calling **IMAPITable::Advise**.

Specify the entry identifier of an address book container, distribution list, or messaging user if registering for notifications for a particular recipient and NULL if registering for notifications on the entire address book. The entry identifier must represent an object in the address book. **IAddrBook::Advise** examines this entry identifier to determine which address book provider is responsible for the corresponding object and forwards the call to the appropriate address book provider's **IABLogon::Advise** method.

The event type that your client specifies in the *ulEventMask* parameter represents the kind of change that the client is interested in and that will cause the address book provider to call the client's **IMAPIAdviseSink::OnNotify** method. For example, if a client needs to be notified when a recipient is added to the container designated as the Personal Address Book, it can register for table modified events on the container's contents table. Clients can register for the following types of events:

- Critical error
- Any of the object events (created, modified, deleted, moved, or copied)
- Table modified

Typically clients register only for notifications on address book container contents and hierarchy tables. They do not register directly with address book objects such as messaging user objects or distribution lists. This is because:

- Many address book providers do not support notifications on their messaging user and distribution list objects.
- Table notifications are sufficient for tracking changes and reporting them to users.

Registering for a Table Notification

As an alternative to registering directly with an advise source object, such as a folder or a messaging user, your client can register for notifications on a contents or hierarchy table. Tracking changes to address book entries, folders, and messages through a contents or hierarchy table can be simpler and more straightforward than through individual objects. For example, your client can call **IMAPITable::Advise** on a folder's hierarchy table to register to receive notifications whenever a change occurs to any of the folders corresponding to the rows in the table. If your client supports the viewing of remote messages, it can register with the status table to observe activity by transport providers and the MAPI spooler.

However, it is not always preferable to use table notifications instead of object notifications. Monitoring changes in the number of messages in a folder is an example of when your client might need to register for object notifications on a folder rather than on a table implemented by the folder.

Canceling a Notification

To cancel a notification for a particular advise source, client applications call its **Unadvise** method. Calling **Unadvise** is important because it causes the service provider to release its reference to an advise sink that no longer should receive notification.

Clients must make one call to an advise source's **Unadvise** method for every successful **Advise** call that is made to prevent service providers from maintaining invalid references to their advise sink objects possibly after the corresponding advise source objects have been destroyed. As long as a service provider maintains a reference to an advise sink, the advise sink can continue to receive **IMAPIAdviseSink::OnNotify** calls.

Because service provider implementations differ, clients that fail to call **Unadvise** to cancel notification registration cannot assume anything about the timing of a service provider's release of the advise sink reference. Some service providers release their references to advise sinks when they release their advise sources. Some service providers do not. As long as a service provider maintains a reference to an advise sink, that advise sink can continue to receive notifications. In fact, because of the asynchronous nature of event notification, clients can be notified even after a successful **Unadvise** call. You must write the notification handler of your client so that it can receive notifications at any time.

Implementing an Advise Sink Object

Your client can either implement one or more of its own advise sink objects or use a utility function, **HrAllocAdviseSink**. **HrAllocAdviseSink** creates an advise sink object with an implementation of **OnNotify** that invokes a callback function provided by the client.

There are advantages and disadvantages to using **HrAllocAdviseSink**. It can save work, but it provides no control over reference counting the resulting advise sink object. Therefore, clients that need to carefully control their advise sink's release or that have interdependencies between their advise sink and another client object should construct their own **IMAPIAdviseSink** implementation and avoid using **HrAllocAdviseSink** altogether.

Clients implementing their own advise sink should make it an independent object not related to or dependent upon any other objects so as to eliminate potential complications in reference counting and object release. However, if your client must implement its advise sink as part of another object or include a back pointer to another object as a data member, it is recommended that two separate reference counts be maintained: one for the object referenced by the advise sink and one for the advise sink.

When the object's reference count falls to zero, all of its methods can fail and its vtable can be destroyed, but the memory for the object must remain intact until after the advise sink's reference count also falls to zero. This means that in addition to decrementing its reference count, the advise sink's **Release** implementation must also finish destroying the object when the count reaches zero. When two separate reference counts are not maintained, it would be easy to inadvertently destroy the advise sink as part of the encompassing object's **Release** process.

Clients must be prepared to receive a notification even after the completion of an **Unadvise** call and free their advise sink objects only when their reference counts reach zero. This is due to the asynchronous nature of event notification.

Clients using **HrAllocAdviseSink** to implement an advise sink must be equally careful not to include their callback function as a method in another advise sink object. It is tempting to do this with C++ clients and pass the *this* pointer as a parameter. This is a dangerous strategy because clients typically free an object when its reference count reaches zero. Freeing the memory for the advise sink object would render the *this* pointer invalid.

Note Notification handlers such as implementations of **IMAPIAdviseSink::OnNotify** must be reentrant if they explicitly yield control of the computer.

Depending on the type of event and the advise source responsible, an **OnNotify** implementation can handle events in various ways. The following table offers suggestions for some of the standard events.

Type of event	Handling in OnNotify
Object moved	If the moved object's original parent is related to the new parent, update the view beginning with the folder or address book container highest in the hierarchy. If the two parent containers are unrelated, update both of their views.
New message	Change the user interface to inform the user of the arrival of one or more new messages. Place the receive folder in the current view.

Error	For all objects except the session, log the error if necessary and return. For the session object, log off if possible.
Search complete	No processing necessary.

Forcing a Notification

When service providers use the MAPI notification utility, MAPI delivers notifications using a hidden window and its corresponding window procedure. For each process to receive a notification, MAPI posts a special message to the hidden window. This message is named with the constant **szMAPINotificationMsg** which is defined in MAPIDEFS.H.

Your client receives these notifications when the hidden window's window procedure processes the **szMAPINotificationMsg** message. To guarantee that notifications are delivered to your client, it is necessary to wait for and dispatch this **szMAPINotificationMsg** message. Implementing the logic to achieve this can be done fairly simply, but MAPI now provides an entry point into the MAPI(32) DLL called **HrDispatchNotifications** to make processing even simpler. Your client can use **HrDispatchNotifications** as follows or implement a loop that accomplishes the same purpose as this entry point.

```
HRESULT hr = HrDispatchNotifications(0);
```

The following code sample shows how to force notifications to occur without including logic for processing errors.

```
void ForceNotifications()
{
    UINT wmNotify;
    MSG  msg;

    wmNotify = RegisterWindowMessage(szMAPINotificationMsg);
    while (1)
    {
        if (PeekMessage(&msg, NULL, wmNotify, wmNotify, 0))
        {
            DispatchMessage (&msg);
            break;
        }
    }
    // at this point, calls to OnNotify have occurred
}
```

The next code sample extends the previous sample by adding error handling and a processing timer. It returns TRUE if notifications were processed.

```
void ForceNotifications()
{
    UINT wmNotify;
    MSG  msg;
    DWORD dwStart;
    DWORD dwNow;

    wmNotify = RegisterWindowMessage(szMAPINotificationMsg);
    dwStart = GetCurrentTime();
    dwNow = dwStart;

    do
    {
        if (PeekMessage(&msg, NULL, wmNotify, wmNotify, 0))
        {
            DispatchMessage (&msg);
```

```
                return TRUE;
            }
            if (PeekMessage(&msg, NULL, 0, 0, PM_NOREMOVE))
            {
                if (msg.message == WM_QUIT || msg.message == WM_CLOSE)
                    return FALSE;
                if (msg.message == WM_PAINT)
                {
                    GetMessage(&msg, NULL, 0, 0);
                    TranslateMessage(&msg);
                    DispatchMessage(&msg);
                }
            }
            dwNow = GetCurrentTime();
        }  while (dwNow - swStart < dwTimeout);

        return FALSE;        // timed out

}
```

Because the **HrDispatchNotifications** entry point was added as of MAPI version 1.0a, your client might want to implement logic that binds dynamically to the entry point if it is available and uses the following code if it is not available.

```
#include <mapiwin.h>
HRESULT (STDAPICALLTYPE FAR *pDispatchFn)(ULONG ulFlags);
HINSTANCE hInst;
HRESULT hr;

hInst = GetModuleHandle("MAPI"szMAPIDLLSuffix".DLL");
pDispatchFn = GetProcAddress(hInst, "HrDispatchNotifications");
if (pDispatchFn)
    hr = (*pDispatchFn)(0);
```

Handling the Address Book

Your client's handling of the MAPI address book includes the following tasks:

- Placing entries from one or more containers into the recipient lists of messages to be transmitted
- Adding entries into containers
- Modifying the properties of recipients
- Displaying common dialog boxes to allow users to browse address information and make changes.

Opening the Address Book

After your client application calls **MAPILogonEx** to start up each of the address book providers in the profile, call **IMAPISession::OpenAddressBook** to retrieve a pointer to MAPI's **IAddrBook** interface. Clients use the **IAddrBook** methods to work with the entries in the containers of each of the address book providers.

The MAPI address book is available regardless of whether all of the address book providers in the profile are running, some of them, or none of them. Therefore, your client must always release its **IAddrBook** pointer when the session ends.

To open the top-level container of the address book, clients call **IAddrBook::OpenEntry** with a NULL entry identifier. To open any container, clients specify the entry identifier of the container in the **OpenEntry** call.

OpenAddressBook might return a warning, MAPI_W_ERRORS_RETURNED, to indicate that there were problems with one or more of the address book providers. Interactive clients should call **IMAPISession::GetLastError** to retrieve additional error information and display the returned information the first time that they call **OpenAddressBook** and it returns a warning.

Non-interactive clients should ignore the warning and proceed as if the method succeeded. The returned **IAddrBook** interface is valid; release it at the end of the session.

Opening an Address Book Container

After the integrated address book provided by MAPI is open, your client must open one or more address book containers to access the recipients within them. Address book containers can be modifiable, as are all Personal Address Book containers, or read-only used solely for browsing. The procedures for opening these two types of containers are different.

To open the Personal Address Book container

1. Call **IAddrBook::GetPAB** to retrieve the personal address book's entry identifier.
2. Call **IAddrBook::OpenEntry** with the personal address book's entry identifier.

To open a container that is not the Personal Address Book

1. Call the **IAddrBook::OpenEntry** method with a NULL entry identifier to open the address book's root container.
2. Call the **IMAPIContainer::GetHierarchyTable** method to retrieve the root container's hierarchy table, a list of all of the top-level containers in the address book.
3. If the container is of a specific type:
   a. Create an **SPropertyRestriction** structure with PR_DISPLAY_TYPE for the property tag, the container's type for the property value, and RELOP_EQ for the relation. PR_DISPLAY_TYPE can be set to many values, among them being:
      - DT_GLOBAL to limit the hierarchy table to containers that belong in the global address list.
      - DT_LOCAL to limit the table to containers belonging to a local address book.
      - DT_MODIFIABLE to limit the table to containers that can be modified.

   b. Create an **SPropTagArray** structure that includes PR_ENTRYID, PR_DISPLAY_TYPE, and any other columns of interest.
   c. Call **HrQueryAllRows**, passing your property restriction and property tag array. **HrQueryAllRows** will return zero or more rows, one row for every container that belongs to the specified type. Your client must be prepared to handle the return of any number of rows.
   d. Call **IAddrBook::OpenEntry** with the entry identifier from the PR_ENTRYID column of the row that represents the container of interest.
4. If the container belongs to a specific address book provider:
   a. Create an **SPropertyRestriction** structure with PR_AB_PROVIDERS for the property tag, a provider-specific value for the property value, and RELOP_EQ for the relation. Typically the provider-specific value is a globally unique identifier or GUID. You will find this value published in one of the address book provider's header files.
   b. Create an **SPropTagArray** structure that includes PR_ENTRYID, PR_AB_PROVIDER, and any other columns of interest.
   c. Call **HrQueryAllRows**, passing your property restriction and property tag array. **HrQueryAllRows** will return zero rows if the specified address book provider is not in the profile. It can return one or more rows for the provider's top-level containers, depending on how the provider is organized.
   d. Call **IAddrBook::OpenEntry** with the entry identifier from the PR_ENTRYID column of the row that represents the container of interest. If the container that your client is interested in is not a top-level container, find the top-level container and traverse the hierarchy.

Setting Address Book Options

Your client can set three properties that describe options for using the address book:

- PR_AB_SEARCH_PATH
- PR_AB_DEFAULT_DIR
- PR_AB_DEFAULT_PAB

The PR_AB_SEARCH_PATH property is used by **IAddrBook::ResolveName** to determine the containers to be involved in name resolution and the order that they should be involved. For each container in PR_AB_SEARCH_PATH, **IAddrBook::ResolveName** calls its **IABContainer::ResolveNames** method. Containers at the beginning of PR_AB_SEARCH_PATH are searched before containers at the end of PR_AB_SEARCH_PATH.

The search order in PR_AB_SEARCH_PATH is specified using an **SRowSet** data structure, the same structure that is used to represent rows in a table. Your client can view the current search path by calling the **IAddrBook::GetSearchPath** method and change it by calling the **IAddrBook::SetSearchPath** method.

The PR_AB_DEFAULT_DIR property is entry identifier of the address book container tthat o be displayed initially when the address book is displayed. The default directory setting remains in effect until your client changes it by calling the **IAddrBook::SetDefaultDir** method. Your client can view the default directory by calling the **IAddrBook::GetDefaultDir** method.

The PR_AB_DEFAULT_PAB property is the entry identifier of the modifiable address book container that is to be used as the session's Personal Address Book Like PR_AB_DEFAULT_DIR, the value of PR_AB_DEFAULT_PAB remains in effect until it is changed. Logging off and logging back on do not affect their settings. Your client can access PR_AB_DEFAULT_PAB by calling **IAddrBook::GetPAB** and change it by calling **IAddrBook::SetPAB**.

These three properties are special because your client cannot work with them using the standard **IMAPIProp** methods. Instead, your client must use **IAddrBook** methods. Because none of these properties can be changed with **IMAPIProp::SetProps**, there is no need to call **IMAPIProp::SaveChanges** to make changes permanent. Modifications made through the **IAddrBook** methods take effect immediately.

Creating an Address Book Entry

Your client creates new entries for the address book when it is preparing to send an outgoing message or when it is modifying a particular container. Depending on the address book provider, your client can create new entries of one or more type. To create a new entry, call one of the following methods:

IAddrBook::CreateOneOff
IAddrBook::NewEntry
IABContainer::CreateEntry

The **IAddrBook::CreateOneOff** method creates recipients to be used in the recipient lists of outgoing messages. When clients call **CreateOneOff,** they specify requires that clients specify the name of the new recipient and the new recipient's e-mail address and address type. The resulting entry identifier adheres to the special format for one-off entry identifiers. See About One-Off Entry Identifiers for information about this format.

Creating an Entry with IAddrBook::NewEntry

The **IAddrBook::NewEntry** method creates recipients for either an outgoing message or a modifiable container. There are three pairs of parameters that your client can set to specify information about the new entry:

Parameter pair	Description
cbEidContainer and *lpEidContainer*	Describes the entry identifier for the container into which the new entry should be placed.
cbEidNewEntryTpl and *lpEidNewEntryTpl*	Describes the entry identifier for the template to be used to create the new entry.
lpcbEidNewEntry and *lppEidNewEntry*	Describes the location for the entry identifier for the new entry.

To create a new entry for the recipient list of an outgoing message, set *cbEidContainer* to zero and *lpEidContainer* to NULL. The resulting entry is the same type of entry that is produced by a call to **IAddrBook::CreateOneOff**.

To insert a new entry into a particular container, set *lpEidContainer* to the container's entry identifier and *cbEidContainer* to the number of bytes in the container's entry identifier.

Most modifiable address book containers support one or more templates for creating entries of a particular type. Entering information into the template produces a recipient with an address that is correctly formatted for the type.

To specify the type of entry to be created, set *lpEidNewEntryTpl* to the entry identifier of the template to be used and *cbEidNewEntryTpl* to the count of bytes in this entry identifier. Obtain the template entry identifier from either:

- The PR_ENTRYID column in the container's one-off table, accessed by calling the container's **IMAPIProp::OpenProperty** method and specifying PR_CREATE_TEMPLATES as the property tag and IID_IMAPITable as the interface identifier.
- An address book provider's PR_DEF_CREATE_MAILUSER and PR_DEF_CREATE_DL properties which hold the entry identifiers for the provider's messaging user object and distribution list templates.

Note Do not confuse a new entry template's entry identifier with a different type of entry identifier called a template identifier. A template identifier is used only by providers to maintain entries copied from other providers; it is never used by clients and it is not used to create new entries.

To allow the user to determine the type of entry to be created, pass zero for *cbEidNewEntryTpl* and NULL for *lpEidNewEntryTpl*. When this occurs, **NewEntry** displays a common dialog box built from MAPI's one-off table, a hierarchical list of all of the templates supported by each address book provider in the profile.

When an address type has been determined, either through the setting of the *lpEidNewEntryTpl* parameter or a selection by the user from the one-off table display, **NewEntry** displays the corresponding template using its display table. All new entry templates support the PR_DETAILS_TABLE property.

To have **NewEntry** return the entry identifier of the created entry, pass a valid address for the *lpcbEidNewEntry* and *lppEidNewEntry* parameters. MAPI places the new entry identifier at the address pointed to by *lppEidNewEntry* and the byte count of the new entry identifier at the address pointed to by *lpcbEidNewEntry*.

Creating an Entry with IABContainer::CreateEntry

The **IABContainer::CreateEntry** method creates an entry in a particular container. Your client can use this method only with modifiable containers, meaning containers that have the AB_MODIFIABLE flag set in their PR_CONTAINER_FLAGS property. Address book providers with non-modifiable containers do not support this method. Specify the entry identifier of the template for creating an entry of the desired type in the *lpEntryID* parameter.

In the *ulCreateFlags* parameter, specify the type of duplicate entry checking required and whether or not new entries should replace existing ones. If **CreateEntry** fails to create a new object because of the duplicate entry checking imposed by the provider, do not expect to see an error or warning returned. Under these conditions, providers return a success code.

To create a distribution list in the PAB

1. Call **IAddrBook::GetPAB** to retrieve the entry identifier for the PAB.
2. Call **IAddrBook::OpenEntry** to open the PAB.
3. Call the PAB's **IMAPIProp::GetProps** method to retrieve its PR_DEF_CREATE_DL property.
4. Call the PAB's **IABContainer::CreateEntry** method to create a new distribution list. Pass the value of the PR_DEF_CREATE_DL. property in the *lpEntryID* parameter.
5. Call the returned object's **IUnknown::QueryInterface** method, passing IID_IDistList as the interface identifier, to determine if the returned object is a distribution list. Because **CreateEntry** returns an **IMAPIProp** pointer rather than the more specific **IMailUser** or **IDistList** pointer, your provider must check to make sure a distribution list object was created. If **QueryInterface** succeeds and returns a valid pointer, your provider can be sure that **CreateEntry** successfully created a distribution list object.
6. Call the distribution list's **IMAPIProp::SetProps** method to set its display name and other properties.
7. Call the distribution list's **IMAPIProp::SaveChanges** method to save it.
8. Call **IUnknown::Release** for the distribution list and the PAB.

Copying an Address Book Entry

To copy one or more recipients into a container, your client must first check that the container is modifiable. Containers that are modifiable set the AB_MODIFIABLE flag in their PR_CONTAINER_FLAGS property.

To copy one or more entries into a modifiable container, call the destination container's **IABContainer::CopyEntries** method. Because copying address book entries can be time-consuming, **CopyEntries** accepts four input parameters: an array of entry identifiers for the entries to be copied, a window handle, a progress indicator, and a bitmask of flags.

The window handle and progress indicator are used by the address book provider to show the status of the operation to the user. If your client wants to display progress, pass a window handle for the parent window of the progress indicator in the *ulUIParam* parameter and do not set the AB_NO_DIALOG flag in the *ulFlags* parameter. If your client has its own implementation of a progress indicator, pass a pointer to the implementation in the *lpProgress* parameter. If not, pass NULL. The address book provider will use MAPI's progress indicator implementation.

The bitmask of flags indicate whether or not your client wants to display a progress indicator and how duplicate entry checking should be handled. Set the AB_NO_DIALOG flag to suppress a progress indicator. Set the CREATE_CHECK_DUP_LOOSE flag to instruct the address book provider to loosely check for duplicates or the CREATE_CHECK_DUP_STRICT flag for stricter duplicate checking. Set the CREATE_REPLACE flag to have copied entries replace existing entries when the provider determines there are duplicates.

When copying into a Personal Address Book container, the provider copies some or all of the properties for each entry. Some PABs, such as the PAB supplied by MAPI, copy all of the properties. Other PABs eliminate properties. Because MAPI does not establish requirements for providers implementing container copy operations, your client cannot make assumptions as the number and type of properties that are copied with an address book entry.

Deleting an Address Book Entry

To remove one or more address book entries from a modifiable container, call the **IABContainer::DeleteEntries** method, passing an array of entry identifiers that represent the address book entries to be deleted.

DeleteEntries can return a warning, MAPI_W_PARTIAL_COMPLETION, to indicate that it couldn't delete one or more of the entries. Test for this return value with the HR_FAILED macro and call the the container's **IMAPIProp::GetLastError** method if more information about the problem is needed.

Preparing a Recipient with IAddrBook::PrepareRecips

Preparing recipients involves converting their short-term entry identifiers to long-term entry identifiers and possibly adding, changing, or reordering properties. Your client can prepare recipients that are part of a recipient list for a message or recipients that are unrelated to a message. Typically clients call **IAddrBook::PrepareRecips** directly to translate short-term entry identifiers into long-term entry identifiers for recipients that are included in the common address dialog box. For recipients that are associated with an outgoing message, recipient preparation is handled by the name resolution process.

To prepare a list of recipients, call **IAddrBook::PrepareRecips**. **PrepareRecips** accepts an **ADRLIST** structure and a list of property tags. The **ADRLIST** structure contains the recipients to be prepared while the property tag list represents properties that each recipient should support. **PrepareRecips** attempts to place the properties that are included in the property tag list at the beginning of the **ADRLIST**. If any of the properties in the list are missing from the **ADRLIST** structure, MAPI calls the address book provider to supply them. If your client only needs to check for long-term entry identifiers, pass NULL for the *lpSPropTagArray* parameter.

For an example, suppose your client is working with five recipients. All five recipients appear in the **ADRLIST** structure with the following properties in the following order:

1. PR_ENTRYID
2. PR_DISPLAY_NAME
3. PR_SEARCH_KEY
4. PR_EMAIL_ADDRESS
5. PR_ADDRTYPE

Three other properties are included in the **ADRLIST** structure for the first two recipients.

1. PR_ACCOUNT
2. PR_GIVEN_NAME
3. PR_SURNAME

Because your client needs all of the recipients to have as their first three properties PR_ADDRTYPE, PR_ENTRYID, and PR_HOME_TELEPHONE_NUMBER, it creates a property tag array with these properties and passes it and the **ADRLIST** structure to **PrepareRecips**. **PrepareRecips** calls each recipient's **IMAPIProp::GetProps** method to retrieve PR_HOME_TELEPHONE_NUMBER because it is not currently part of the **ADRLIST** structure. When **PrepareRecips** returns, the recipient list represents a merged list of recipients with the properties included in the **ADRLIST** structure appearing first for each recipient.

The recipient list for recipients 1 and 2 includes properties in the following order:

1. PR_ADDRTYPE
2. PR_ENTRYID
3. PR_HOME_TELEPHONE_NUMBER
4. PR_DISPLAY_NAME
5. PR_SEARCH_KEY
6. PR_EMAIL_ADDRESS
7. PR_ADDRTYPE
8. PR_ACCOUNT
9. PR_GIVEN_NAME
10. PR_SURNAME

The recipient list for recipients 3, 4, and 5 includes properties in the following order:

1. PR_ADDRTYPE
2. PR_ENTRYID
3. PR_HOME_TELEPHONE_NUMBER
4. PR_DISPLAY_NAME
5. PR_SEARCH_KEY
6. PR_EMAIL_ADDRESS
7. PR_ADDRTYPE

As an alternative to calling **IAddrBook::PrepareRecips** to work with properties, your client can call each recipient's **IMAPIProp::GetProps** method and, if necessary, its **IMAPIProp::SetProps** method. When only one recipient is involved, either technique is satisfactory. However, when multiple recipients are involved, calling **PrepareRecips** rather than the **IMAPIProp** methods saves time and, if your client is operating remotely, many remote procedure calls. **PrepareRecips** processes all recipients in a single call whereas **GetProps** and **SetProps** make one call for each recipient.

Retrieving Address Book Entry Properties

To access one or more properties of an address book entry, there are two approaches. The first approach involves the following tasks for each address book entry of interest:

1. Call **IAddrBook::OpenEntry**, passing the entry identifier of the target messaging user or distribution list.
2. Call the messaging user or distribution list's **IMAPIProp::GetProps** method with a list of the one or more properties to retrieve.

The second approach involves calling **IAddrBook::PrepareRecips**, passing an **ADRLIST** structure that holds all of the properties for all of the desired address book entries. Because one call to **PrepareRecips** can return information for multiple address book entries, it is the preferable strategy when your client is interested in more than one recipient.

Searching the Address Book

MAPI enables address book providers to implement two levels of search functionality:

- A basic level that matches a specified name with the PR_DISPLAY_NAME property of address book entries. This level allows users, for example, to view distribution lists with names beginning with Northwest or locate individual messaging users whose last name is Brown.
- An advanced level that matches on properties other than PR_DISPLAY_NAME. This level allows users, for example, to further narrow their searches, allowing users, for example, to find messaging users named Brown with a particular address type.

Because address book providers can support searching for each of their containers at the basic level, at both levels, or choose not to support it at all, your client should not expect searching to be implemented as a standard feature. To determine if a particular container supports searches, attempt to establish search criteria in a call to its **IMAPIContainer::SetSearchCriteria** method. If **SetSearchCriteria** returns MAPI_E_NO_SUPPORT, the container does not support searches.

In a container that supports searches, your client can retrieve established criteria by calling **IMAPIContainer::GetSearchCriteria**. Your client can also request that the user be prompted for search criteria before a container's contents table is displayed. To choose this option, set the AB_FIND_ON_OPEN flag of the container's PR_CONTAINER_FLAGS property. After the user enters the criteria, it is stored as a restriction and passed to the **SetSearchCriteria** method. Setting AB_FIND_ON_OPEN is particularly useful if your client is using an online service or any address book provider that has a slow link to its data.

To perform a basic search in an address book container

1. Call the container's **IMAPIContainer::GetContentsTable** method to open its contents table.
2. Choose a search technique that meets your client's needs. The choices include:
   - **IMAPITable::FindRow** to locate a specific row in the table.
   - **IMAPITable::SortTable** to order rows in the table.
   - **IMAPITable::Restrict** to limit the table view.
   - Property restriction using the PR_ANR property for resolving ambiguous names. Call **IMAPITable::Restrict** to impose this restriction.
   - **IABContainer::ResolveNames** to resolve ambiguous names.
3. Call **IMAPITable::QueryRows** to retrieve any rows that meet your applied search criteria. **QueryRows** can return zero or more matching rows.

The **FindRow**, **SortTable**, and **Restrict** methods are table methods that are available for any table that can be created, either by a client or a service provider. The PR_ANR property restriction and **IABContainer::ResolveNames** method are specific to address book providers and are used for resolving ambiguous names. Ambiguous names are entries in a recipient list that do not have a PR_ENTRYID property associated with them.

The PR_ANR restriction invokes an algorithm that separates a character string into words and matches those words with information in the address book using prefix-matching. The information used for the matching depends on the address book provider. All address book providers are required to support the PR_ANR restriction for their address book containers. For more information, see Implementing the PR_ANR Property Restriction.

IABContainer::ResolveNames performs PR_ANR restriction processing on multiple names without requiring the container's contents table to be open. Calling **ResolveNames** once to resolve multiple names can be much faster than invoking a PR_ANR restriction multiple times. However, address book providers are not required to support **ResolveNames**.

Using an Advanced Search Dialog Box

Some address book containers support an advanced searching capability that allows clients to search on properties other than PR_DISPLAY_NAME. Address book containers that support advanced searches have an container object property called PR_SEARCH. This container object provides access to a display table that describes the search dialog box, a dialog box used to enter and edit the advanced search criteria.

To perform an advanced search on an address book container

1. Call the container's **IMAPIProp::OpenProperty** method, specifying PR_SEARCH for the property tag and IID_IMAPIContainer for the interface identifier.
2. Call the search object 's **IMAPIProp::OpenProperty** method, specifying PR_DETAILS_TABLE for the property tag and IID_IMAPITable for the interface identifier.
3. Call the search object's **IMAPIProp::SetProps** to establish values for the properties to be used in the advanced search.
4. Call the search object's **IMAPIProp::SaveChanges** method to save the advanced search criteria.

This sequence of calls results in a restriction being available when a client calls the search object's **GetSearchCriteria** method.

Resolving a Name

When users compose messages, they enter the names of one or more recipients. Each of these names must go through a process known as name resolution, which involves associating an entry identifier to each name.

To resolve all of the names in a recipient list

1. Build an **ADRLIST** structure that contains an ADRENTRY structure for the properties of each recipient.
2. Pass the **ADRLIST** structure in a call to the **IAddrBook::ResolveName** method.

ResolveName begins processing by ignoring the entries in the **ADRLIST** that have already been resolved, as is indicated by the presence of an entry identifier in the corresponding **ADRENTRY** structure's **SPropValue** array. Next, **ResolveName** automatically assigns one-off entry identifiers to two types of recipients:

- Recipients with an address formatted as an Internet address
- Recipients with an address formatted as follows:

  ```
  displayname[address type:e-mail address]
  ```

For all remaining entries, **ResolveName** searches the address book for an exact match on the display name. **ResolveName** uses the PR_AB_SEARCH_PATH property to determine the set of containers to search and the search order. MAPI calls the **IABContainer::ResolveNames** method of every container to attempt to resolve all of the names. Because some containers do not support **ResolveNames**, if the container returns MAPI_E_NO_SUPPORT, MAPI applies a PR_ANR property restriction against its contents table. All address book containers are required to support name resolution with this restriction. Once all of the names are resolved, no further container calls are made. If all of the containers have been called, but ambiguous or unresolved names remain, MAPI displays a dialog box if possible to prompt for the user's help.

The PR_ANR restriction matches the value of the PR_ANR property against the display name in the **ADRLIST**. Limiting the view of a container's contents table with the PR_ANR property restriction causes the address book provider to perform a "best guess" type of search, matching against the property that makes sense for the provider. For example, one address book provider might always match names in the recipient list against PR_DISPLAY_NAME while another might allow an administrator to select the property.

To set a PR_ANR property restriction on an address book container's contents table

1. Create an **SRestriction** structure as shown in the following code:

   ```
   SRestriction SRestrict;

   SRestrict.rt = RES_PROPERTY;
   SRestrict.res.resProperty.relop = RELOP_EQ;
   SRestrict.res.resProperty.ulPropTag = PR_ANR;
   SRestrict.res.resProperty.lpProp->ulPropTag = PR_ANR;
   SRestrict.res.resProperty.lpProp->Value.LPSZ = lpszName;
   ```

2. Call the contents table's **IMAPITable::Restrict** method, passing the **SRestriction** structure as the *lpRestriction* parameter.

Handling a Message Store

Handling a message store is an important part of your client's tasks. These tasks can include:

- Opening one or more message stores.
- Performing maintenance on a folder.
- Validating the folder hierarchy.
- Handling user access to folders.
- Setting message store properties that define features.

Opening a Message Store

Your client will open one or more message stores during a typical session, either directly after logon or after a profile modification. With the Control Panel, MAPI provides the means for users to add message stores to the profile during an active session. If your client permits this, it can either ignore the newly added message stores until the next session or open them to make them immediately available during the current session.

To open a message store, pass the store's entry identifier to one of two methods:

IMAPISession::OpenMsgStore
IMAPISession::OpenEntry

Message store entry identifiers are published in the message store table, a summary list of dynamic information about each of the message stores in the profile. Most clients access the entry identifiers for the message stores they wish to open through this table. However, some clients know how to construct a message store entry identifier for a particular type of message store. If your client can construct a message store-specific entry identifier, it can pass this identifier directly to **OpenMsgStore** or **OpenEntry**, bypassing the message store table.

To use the message store table to open a message store

1. Call **IMAPISession::GetMsgStoresTable** to open the message store table.
2. Call **IMAPITable::SetColumns** to limit the table to a small column set that includes the following columns:

   PR_PROVIDER_DISPLAY or PR_DISPLAY_NAME
   PR_ENTRYID properties
   PR_MDB_PROVIDER
   PR_RESOURCE_FLAGS
3. Search for the row in the table that represents the message store. To look for a temporary message store or a store that is the default, primary, or secondary store, apply a bitmask restriction to PR_RESOURCE_FLAGS. Specify for example, STATUS_DEFAULT_STORE as the restriction mask to search for the session's default message store.

   To look for a message store by name, build a property restriction using any of the following properties:

   - Match PR_PROVIDER_DISPLAY with the general name for this type of message store. For example, PR_PROVIDER_DISPLAY might be set to "Personal Folders."
   - Match PR_MDB_PROVIDER with the specific name for this type of message store. For example, PR_MDB_PROVIDER might be set to "Microsoft Exchange Personal Folders."
   - Match PR_DISPLAY_NAME with the name for this particular message store. For example, PR_DISPLAY_NAME might be set to "My Messages for Fiscal Year 1995."
4. Pass the restriction in a call to **HrQueryAllRows**.
5. Retrieve the row from the table by calling **IMAPITable::QueryRows**.
6. Pass the entry identifier from the row to one of two methods to open the message store: **IMAPISession::OpenMsgStore** or **IMAPISession::OpenEntry**. Call **OpenMsgStore** if your client needs to specify a variety of special options for the message store.
7. Call **FreePRows** to free the **SRowSet** data structure returned by **IMAPITable::QueryRows**.
8. Release the message store table by calling its **IUnknown::Release** method.

To use an internal identifier to open a message store

1. Call the **WrapStoreEntryID** function to convert the internal identifier to a regular entry identifier.
2. Pass the converted entry identifier returned from **WrapStoreEntryID** to one of two methods to open the message store: **IMAPISession::OpenMsgStore** or **IMAPISession::OpenEntry**. Call

OpenMsgStore if your client needs to specify a variety of special options for the message store. **OpenEntry** allows clients to access session options only.

Validating a Message Store

Every message store that is opened in a session without the MAPI_NO_MAIL flag set should have a subtree of IPM folders. This subtree is more extensive if the message store is the default message store and less extensive if it is not. MAPI automatically creates the correct set of IPM folders for message stores when your IPM client calls **IMAPISession::OpenMsgStore** and assigns them default names and roles. MAPI is responsible for creating the folder subtree to avoid the incompatibilities that would inevitably occur if either clients or message store providers were responsible for the creation.

Your client can verify that the appropriate folders have been created and that they are valid by calling the API function **HrValidateIPMSubtree**. If the message store is the default, your client should pass the MAPI_FULL_IPM_TREE flag. When **HrValidateIPMSubtree** receives the MAPI_FULL_IPM_TREE flag, it checks for the following folders:

- Root folder for the IPM subtree
- Deleted Items folder in the IPM root folder
- Inbox folder in the IPM root folder
- Outbox folder in the IPM root folder
- Sent Items folder in the IPM root folder
- Folder views in the message store's root folder
- Common views in the message store's root folder
- Search folder in the message store's root folder

If the message store is not the default, your client has the option of setting the MAPI_FULL_IPM_TREE flag. When this flag is not set, **HrValidateIPMSubtree** checks for only the root folder for the subtree, the Deleted Items folder, and the root folder for message store search results.

During message store initialization your client might need to cache two message store properties: PR_VALID_FOLDER_MASK and PR_STORE_SUPPORT_MASK. These properties are bitmasks that describe, in the case of PR_VALID_FOLDER_MASK, the set of folders that have been validated and, in the case of PR_STORE_SUPPORT_MASK, the features that the message store supports.

Other properties that your client should store locally include the entry identifiers for the folders that the PR_VALID_FOLDER_MASK property describes as valid. Each of these special folders, except for the Inbox folder, has an entry identifier property associated with it. For example, the entry identifier for the Outbox folder is its PR_IPM_OUTBOX_ENTRYID property. Because these folders are the folders that your client will open frequently, it is a good idea to have their entry identifiers readily available.

Setting Message Store Features

Every message store supports the PR_RESOURCE_FLAGS property which describes the store's features. Some message stores can act as temporary stores or the default store for a session. Your client can identify a particular message store as the default by calling the message store's **IMAPISession::SetDefaultStore** method and passing the MAPI_DEFAULT_STORE flag. As a result of this call, **SetDefaultStore** sets the message store's PR_RESOURCE_FLAGS to STATUS_DEFAULT_STORE.

Your client can also use the **IMAPISession::OpenMsgStore** method to set message store features by setting two flags:

MDB_NO_MAIL
MDB_TEMPORARY

Setting the MDB_NO_MAIL flag indicates to MAPI that the message store will not be used for sending or receiving messages. MAPI does not inform the spooler about the existence of this message store. The MDB_TEMPORARY flag designates a message store as temporary, implying that it cannot be used to store permanent information. Temporary message stores do not appear in the message store table.

Handling User Access to Folders

Some folders support the PR_ACCESS property which describes the type of operations a user can perform. For example, one of the valid settings for PR_ACCESS is MAPI_ACCESS_DELETE, indicating that the folder can be removed. Another setting, MAPI_ACCESS_MODIFY, indicates that your client can write to the folder.

Although some message store providers include the PR_ACCESS property in the column set of their hierarchy tables or on the folder itself, other message store providers support it by calculating its value on demand. This is because calculating the value for PR_ACCESS is time consuming. When your client needs to determine whether or not a user can perform an operation on one or more folders, it is more practical to attempt the operation and return an error if necessary than to retrieve a folder's PR_ACCESS property first. Only prompt for PR_ACCESS when absolutely necessary.

Deleting a Message

Your client can delete messages when they are open while the user is reading them, or when they are closed while the user is viewing the contents table. To protect users from inadvertently removing messages, MAPI defines message deletion as a two step process. First, your client marks a message for deletion by moving it to the folder that has been designated as the Deleted Items folder, the folder whose entry identifier is stored in the PR_IPM_WASTEBASKET_ENTRYID property. Second, your client removes the message by calling the **IMAPIFolder::DeleteMessages** method.

When a user chooses to delete a message in a folder other than the Deleted Items folder, your client should mark it for deletion. Only when a user selects messages from within the Deleted Items folder should the messages be physically removed from the workstation. Your client can prompt the user to make sure that he or she really intended to perform the deletion.

To delete a message

1. Confirm with the user that the impending deletion is intentional.
2. Determine the parent of the folder to be deleted. If it is the Deleted Items folder or a subfolder within the Deleted Items folder, call **DeleteMessages** to remove the message.
3. If the folder is not contained within the Deleted Items folder, call **CopyMessages** with the MESSAGE_MOVE flag set to relocate the message to the Deleted Items folder.

Copying or Moving a Message

There are several ways to copy messages. The best way is to call the **IMAPIFolder::CopyMessages** method of the folder with the message to be copied. Your client can also call the **IMAPIFolder::CreateMessage** method of the folder to receive the copy and then copy all of the appropriate properties from the original message. However, because **CreateMessage** returns a new unique entry identifier for the copy, all links to the original are lost. Also, with **CreateMessage**, the message store provider generates object created notifications rather than object copied notifications.

To copy or move a message

1. Locate the entry identifiers for the source and destination folders.
2. Validate these entry identifiers.
3. Open these folders in read/write mode by calling **IMAPISession::OpenEntry** or **IMsgStore::OpenEntry** and setting the MAPI_MODIFY flag.
4. Check that the interface pointer returned from **OpenEntry** is an **IMAPIFolder** interface pointer.
5. Call **IMAPIFolder::CopyMessages**.
6. Release the interface pointers for the source and destination folders.

If your client copies messages from one message store to another and Unicode is not supported by both, information can be lost due to code page conversion. It is impossible to know ahead of time whether to copy the text as an ASCII string or as a Unicode string. If your client supports Unicode, try to perform a Unicode copy; if it fails with the error value MAPI_E_BAD_CHARWIDTH, resort to ASCII.

Handling a Transport Provider

Communication with a transport provider is accomplished through the **IMAPIStatus** interface − the interface made available through the session's status table. Your client may or may not communicate with a transport provider, depending on whether or not it includes configuration support in its feature set.

Sending or Receiving a Message on Demand

Clients typically rely on the MAPI subsystem – the MAPI spooler and the service providers – to handle the timing of message transmission and reception. However, your client can alter this timing by requesting the immediate sending or receiving of messages. The following procedures describe two techniques of how to send or receive messages on demand. The first procedure works with all of the transport providers simultaneously through the MAPI spooler's status object; the second procedure works on individual transport providers through the status object of each provider.

To send or receive messages on demand

1. Call **IMAPISession::GetStatusTable** to access the status table.
2. Call the status table's **IMAPITable::SetColumns** method to limit the column set to PR_ENTRYID and PR_RESOURCE_TYPE.
3. Build a property restriction using an **SPropertyRestriction** structure to match PR_RESOURCE_TYPE with MAPI_SPOOLER.
4. Call **HrQueryAllRows**, passing in the **SPropertyRestriction** structure, to retrieve the row that represents the status of the MAPI spooler.
5. Pass the PR_ENTRYID column to **IMAPISession::OpenEntry** to open the MAPI spooler's status object.
6. Call the MAPI spooler's **IMAPIStatus::FlushQueues** method, passing the FLUSH_NO_UI flag to suppress the user interface and either the FLUSH_DOWNLOAD or FLUSH_UPLOAD flag to flush the outgoing or incoming queues.
7. Release the status object and the status table, as well as the **SRowSet** structure that is allocated for the table.

To send or receive messages on demand and flush each transport provider individually

1. Call **IMAPISession::GetStatusTable** to access the status table.
2. Call the status table's **IMAPITable::SetColumns** method to limit the column set to PR_ENTRYID and PR_RESOURCE_TYPE.
3. Build a property restriction using an **SPropertyRestriction** structure to match PR_RESOURCE_TYPE with MAPI_TRANSPORT_PROVIDER.
4. Call **HrQueryAllRows**, passing in the **SPropertyRestriction** structure, to retrieve the row that represents the status of the MAPI spooler.
5. For each row returned from **HrQueryAllRows**:
   a. Pass the PR_ENTRYID column to **IMAPISession::OpenEntry** to open the transport provider's status object.
   b. Check that the transport status object supports the **FlushQueues** method by checking that its PR_RESOURCE_METHODS property has the STATUS_FLUSH_QUEUES flag set.
   c. If supported, call the transport object's **IMAPIStatus::FlushQueues** method. If unsupported; call the MAPI spooler's **IMAPIStatus::FlushQueues** method, passing the entry identifier of the transport in the *lpTargetTransport* parameter. Set the FLUSH_DOWNLOAD flag to flush the outgoing queues or the FLUSH_UPLOAD flag to flush the incoming queues.
   d. Release the status object and the status table, as well as the **SRowSet** structure that is allocated for the table.

The MAPI spooler honors the FLUSH_NO_UI flag as do most LAN transport providers. However, not all transport providers honor this flag, particularly those that use the Remote Access Service (RAS) because it is not designed to allow the suppression of a user interface and those that use the modem explicitly. It is possible for a client to be configured so that it can connect without requiring the interaction of a user, but it is difficult and requires intimate knowledge of the message services that the client will be using.

Setting Transport Order

The MAPI spooler assigns responsibility for outgoing messages based on one or more address types and unique identifiers, represented in **MAPIUID** structures, that a transport provider declares it can handle. Transport providers publish a list of address types and **MAPIUID**s through their **IXPLogon::AddressTypes** method, which MAPI calls directly after logon. Address types are stored in the PR_ADDRTYPE property.

When a transport provider returns an address type from **IXPLogon::AddressTypes**, it is implying that it can deliver messages to recipients that have their PR_ADDRTYPE property set to this address type. Likewise, when a transport provider returns a MAPIUID in the *lpppUIDArray* parameter, this indicates that the transport provider can deliver to recipients that are represented by entry identifiers that include the MAPIUID.

Most transport providers handle messages by address type. Transport providers that are tightly coupled with an address book provider and understand its entry identifier format can register to handle messages by MAPIUID, thereby improving performance. These tightly coupled transport providers can extract the recipient's e-mail address and other necessary information from the entry identifier without having to open it with an **IMAPISupport::OpenEntry** call.

MAPI maintains an order for transport providers, used when multiple transport providers have registered for the same address type or MAPIUID. Your client can override this order by:

- Calling **IMsgServiceAdmin::MsgServiceTransportOrder** and passing an ordered list of the MAPIUIDs of all of the active transport providers in the *lpUIDList* parameter.
- Using the **Delivery** property page of the Control Panel applet.

To retrieve a list of all of the address types that can be handled by one of the active transport providers, call **IMAPISession::EnumAdrTypes**. **EnumAdrTypes** returns an array of strings that describe address types supported by the transport providers that are active in the current session.

Reconfiguring a Transport Provider

Your client application can use a transport provider's status object to change some of the properties of the properties of the provider. The range of properties that your client can change depends on the properties that are included with the provider's property sheet, a dialog box implemented by each service provider, and how those properties are defined.

To reconfigure an active transport provider

1. Call **IMAPISession::GetStatusTable** to access the status table.
2. Locate the row for the transport provider to be reconfigured by creating a property restriction that matches PR_DISPLAY_NAME with the name of the target provider.
3. Call **IMAPITable::FindRow** to retrieve the appropriate row.
4. Check that the STATUS_SETTINGS_DIALOG and STATUS_VALIDATE_STATE flags are set in the target transport provider's PR_RESOURCE_METHODS property. These are optional methods for transport providers to support in their status objects.
5. If STATUS_SETTINGS_DIALOG is not set, the transport provider does not display a configuration property sheet. If STATUS_VALIDATE_STATE is not set, your client cannot perform dynamic reconfiguration.
6. If STATUS_SETTINGS_DIALOG is set, call **IMAPIStatus::SettingsDialog** to display the transport provider's property sheet and allow the user to make changes.
7. After the user has finished with the reconfiguration, call **IMAPIStatus::ValidateState** if STATUS_VALIDATE_STATE is set, passing CONFIG_CHANGED.

Handling Profiles

Some MAPI clients have nothing more to do with profiles than allowing their users to select one at logon time. Other clients provide a complete set of configuration features. Your client might need to create a profile or modify an existing profile with the interfaces supported by MAPI for profile and message service administration.

There are three approaches for creating profiles:

- Invoke the New Profile wizard, which allows users to select message services interactively. The message services that are selected must support Wizard-based configuration. If your client uses this approach, be aware that when the profile has been created, one or more message services might not be completely configured.
- Invoke the NEWPROF utility, a tool that reads from a template file called DEFAULT.PRF to populate a profile with a set of message services and service providers.
- Write C or C++ code, often the best choice for non-interactive client applications which require a specific set of message services.

To modify profiles, use the following interfaces:

- **IProfAdmin : IUnknown** for profile administration.
- **IProfSect : IMAPIProp** for access to profile section properties.
- **IMsgServiceAdmin : IUnknown** for message service administration.
- **IProviderAdmin : IUnknown** for administration of service providers within a message service.

New address book providers added to the profile do not show up until a new session is created. Address book providers can add or remove top-level containers dynamically by issuing notifications on their hierarchy table. MAPI receives these notifications and updates the address book.

Locating the Default Profile

The default profile is the profile that your client uses if another profile is not explicitly chosen. Default profiles are typically designated as such at installation time.

To locate the default profile

1. Call **MAPIAdminProfiles**.
2. Call **IProfAdmin::GetProfileTable** to access the profile table.
3. Build a property restriction with an **SPropertyRestriction** structure to match PR_DEFAULT_PROFILE with the value TRUE.
4. Call **IMAPITable::FindRow** to locate the row in the profile table that represents the default profile. The PR_DISPLAY_NAME column contains the name of the default profile.

Understanding Profile Order

When your client application is creating a profile, be careful as to the order in which information is entered. MAPI makes assumptions and assignments based on the order in which service providers and message services appear in the profile. Resources such as the default message store, default personal address book, and default search path for address book containers are usually configured based on the order in which they appear in the profile.

Keep the following issues in mind when creating profiles or a .PRF file for the NEWPROF utility.

MAPI resource	Configuration issue
Default message store	If not set explicitly, this is the first message store in a profile that is capable of being the default store.
Personal address book	If not set explicitly, this is the first container that is writeable and can contain names. Its PR_CONTAINER_FLAGS property is set to AB_RECIPIENTS and AB_MODIFIABLE.
Default address book directory	First container in the hierarchy that can contain names and is not the personal address book (PAB), unless the PAB is the only container that can hold names.
Default search path	Is the PAB followed by each directory that: • Has its PR_DISPLAY_TYPE property set to DT_GLOBAL. • Has names. • Does not have the AB_NOT_DEFAULT flag set in its PR_CONTAINER_FLAGS property. If there are no containers of type DT_GLOBAL, this is the PAB and the default directory.
Transport order	If not explicitly set, this is the same order that the transports were added to the profile, except that transports which set the STATUS_XP_PREFER_LAST flag are serviced last. The transport order is unimportant unless your profile contains two or more transports which handle the same e-mail address type.

These guidelines for ordering service providers and message services might sometimes conflict. If there is a conflict, your client must include C or C++ code to configure one or more of the provider orders. Your client can use the Delivery and Addressing pages of the Mail and Fax Control Panel applet to inspect a profile that it has created to determine if the providers have been ordered as expected.

If your client must run unattended, perhaps as a background application or a Windows NT service, some special cautions apply. For more information, see Writing an Automated Client and Windows NT Service Client Applications.

If your application requires a specific profile other than the default profile, you must save its name in your own configuration database or use a specific naming convention. MAPI does not expose any profile attributes other than the name and default flag in the profile table, and the default profile flag is reserved for the messaging client and related IPM applications.

Creating a Profile with NEWPROF

Using the NEWPROF utility to create a profile requires that your client application have available a .PRF file, or file with the .PRF extension. A default .PRF file called DEFAULT.PRF that contains the profile entries for the message services that are installed with MAPI is included with every MAPI installation. This file is divided into four sections:

- Section 1 includes the [General] section.
- Section 2 includes the [Service List] section.
- Section 3 includes sections that describe individual message services.
- Section 4 includes sections that map to profile properties.

Your client can create a new .PRF file using portions of DEFAULT.PRF as a model or modify DEFAULT.PRF directly, adding portions that pertain to your configuration.

To create a .PRF file

1. Assemble information from existing .PRF files. For each service you plan to use, copy the section that lists the name, type, and property identifier for each configuration property. The section must have the same name as the service. Services implemented by Microsoft are listed in DEFAULT.PRF.
2. Place the profile name in the ProfileName= line of the [General] section.
3. Create the [Service List] section after the [General] section. In this section, list each message service that you require and the name of the section that contains its configuration properties. For information about ordering services in profiles, see Understanding Profile Order.
4. For each service listed in the [Service List] section, create a section that lists the value for each configuration property. The section must have the same name as the service. This information appears in Section 3 of the file.
5. Invoke the NEWPROF utility. On 32-bit operating systems, NEWPROF can be run from a command line or batch file. On 16-bit Windows systems, NEWPROF cannot be run from an MS-DOS shell; it can be run by a Windows-based setup script, by the Run command on the File menu in Program Manager or File Manager, or by the **WinExec** system call.

Note References to Sections 1, 2, 3, and 4 in the preceding procedure refer to comments in the DEFAULT.PRF file distributed with the MAPI SDK.

Creating a Profile with Custom Code

When a client application opts to write code to create a profile, it must fully understand how the ordering of entries within a profile work and fully realize how much information is required. The code written to create a profile must add each message service and each service provider within the message service.

To create a profile with C or C++ code

1. Read the header file for each message service. Understand what properties you will need to configure and what values you will use.
2. Call the **MAPIAdminProfiles** function to obtain a profile administration object.
3. Call the profile administration object's **IProfAdmin::CreateProfile** method to create your profile.
4. Call the profile administration object's **IProfAdmin::AdminServices** method to obtain a message service administration object.
5. Use the message service administration object to add message services to the profile. For information about ordering message services in profiles, see Understanding Profile Order. For each message service to be added:
   a. Call the **IMsgServiceAdmin::CreateMsgService** method to create the new message service.
   b. Call the **IMsgServiceAdmin::GetMsgServiceTable** method to access the message service table.
   c. Call **HrQueryAllRows** to retrieve all rows from the table.
   d. Obtain the **MAPIUID** structure of the service you just created.
   e. Get the PR_SERVICE_UID column from the last row. This is the **MAPIUID** structure of the last service added. You may wish to check with an assertion that other properties of the service are as you expect.
   f. Call **IMsgServiceAdmin::ConfigureMsgService**, passing the **MAPIUID** structure of the service you just created and a property value array with its configuration properties.

To make the profile temporary, call the **IProfAdmin::DeleteProfile** method immediately after logging on to the profile. It will be deleted after you log off, and will not be visible to other applications in the meantime.

If your client must make configuration calls that require an **IMAPISession** interface, such as **IMAPISession::SetDefaultStore**, **IAddrBook::SetPAB**, or **IAddrBook::SetABSearchPath**, pass the MAPI_NO_MAIL flag to the **MAPILogonEx** function.

Copying a Profile

One way to create a profile is to copy from an existing profile and alter the necessary message services and service providers. Copying a profile involves using a profile administration object, provided by MAPI through the MAPIAdminProfiles function.

To copy a profile

1. Call **MAPIAdminProfiles**.
2. Call **IProfAdmin::GetProfileTable** to access the profile table.
3. Build a property restriction with an **SPropertyRestriction** structure to match PR_DISPLAY_NAME with the name of the profile to be copied.
4. Call **IMAPITable::FindRow** to locate the appropriate row in the profile table.
5. Call **IProfAdmin::CopyProfile**, passing the value of the PR_DISPLAY_NAME column as the *lpszOldProfileName* parameter.

Modifying a Profile

Modifying a profile can involve adding or deleting a message service or a service provider or manipulating a property of a message service or a service provider. All of these activities require that your client have access to a message service administration object or a provider administration object.

To modify a profile, your client can call

- **IMAPISession::AdminServices** or **IProfAdmin::AdminServices** to access a message service administration object.

  - Or -

- **IMAPISession::OpenProfileSection** or **IProviderAdmin::OpenProfileSection** to access a profile section created by your client.

A message service administration object allows your client to make changes to the active profile while in an active session.

To add a message service to a profile

1. Call **MAPIAdminProfiles** to access a profile administration object.
2. Call **IProfAdmin::AdminServices** to access a message service administration object.
3. Call **IMsgServiceAdmin::CreateMsgService** to add the message service to the current profile.

One technique for adding a service provider, specifically a message store provider, to a profile involves constructing an entry identifier for the provider. Because constructing an entry identifier requires knowledge of its format, this technique can only be used if the service provider has made its entry identifier format public.

With the newly constructed entry identifier, call **IMAPISession::OpenMsgStore**. MAPI automatically creates a profile section in the profile for the service provider, but does not add it to a message service.

When your client adds a service provider to a profile, the addition is not apparent until a new session is created.

Modifying a Message Service

Modifying a message service involves adding or deleting one or more service providers using a message service administration objects. Your client may or may not be allowed to modify the message services included in the profile; few providers allow it.

To add a service provider to a message service

1. Call **IMAPISession::AdminServices** to access a message service administration object.
2. Call **IMsgServiceAdmin::GetMsgServiceTable** to access the message service table.
3. Build a property restriction using an **SPropertyRestriction** structure that matches PR_DISPLAY_NAME or PR_SERVICE_NAME with the name of the message service your client needs to modify.
4. Call the message service table's **IMAPITable::FindRow** method to locate the row in the table that represents the targeted message service.
5. Call **IMsgServiceAdmin::AdminProviders** to retrieve an **IProviderAdmin** pointer. Pass the PR_SERVICE_UID column from the message service table row as the *lpUID* parameter.
6. Call **IProviderAdmin::GetProviderTable** to access the provider table.
7. Build a property restriction using an **SPropertyRestriction** structure that matches PR_DISPLAY_NAME or PR_PROVIDER_DISPLAY with the name of the service provider your client needs to add.
8. Call the provider table's **IMAPITable::FindRow** method to locate the row in the table that represents the targeted service provider.
9. Call **IProviderAdmin::CreateProvider** to add the provider to the message service. Pass the provider's PR_DISPLAY_NAME property as the *lpszProvider* parameter and the PR_SERVICE_UID property as the *lpUID* parameter.

Advanced Client Topics

This section discusses how to implement some of the more uncommon or difficult features in a client application.

Writing an Automated Client

An automated client application is an application that runs unattended, displaying no user interface. Many MAPI interface methods by default show a user interface. All of these methods have flags that allow a client to either allow or suppress this display. Although MAPI expects service providers to honor these flags, there are some providers that do not always meet these expectations. A legitimate reason for not honoring the flags is the reliance of the service provider on another service that does not allow user interface suppression. If you are developing an automated client, pay careful attention to the service providers your client is using and how they are configured. Do not assume that all of your client's calls to suppress a user interface will be successful.

Automated clients must have all of the necessary information available for proper configuration of each of the message services in the profile. There are two ways to supply configuration information at logon time:

- The service provider can retrieve information from the profile.
- The service provider can prompt the user for information.

Since the second option is unavailable to automated clients, these clients must use the first option. Clients must configure their profiles carefully to ensure that this option always works.

Using a profile created previously for another purpose is allowable, but will not work if the user has chosen not to cache credentials for one or more services. On Windows 95, caching credentials may not work due to password cache control. If the profile is created when the user is logged on to a network, and subsequently used when not logged on to a network, cached credentials will be unavailable.

Automated clients always set the MAPI_NO_MAIL flag in the **MAPILogonEx** function call to begin a MAPI session.

Writing a Remote Viewer

A remote viewer is a window in a client application that provides controlled access to messages stored on another computer. This controlled access might be with a modem or other type of slow link. Rather than retrieve a complete selection of available messages every time a user opens a folder, the remote viewer prompts for headers only first. The user then selects from the headers which of the real messages to display in full. Remote viewer clients can allow their users to delete messages before they are ever downloaded.

To retrieve the headers of messages stored remotely

1. Call **IMAPISession::GetStatusTable** to access the status table.
2. Limit the status table by calling **IMAPITable::Restrict** to only those rows that have their PR_RESOURCE_TYPE column set to MAPI_TRANSPORT_PROVIDER.
3. Establish the column set of the status table by calling **IMAPITable::SetColumns** to include the PR_ENTRYID, PR_RESOURCE_METHODS, PR_RESOURCE_TYPE, PR_PROVIDER_DISPLAY, and PR_STATUS_CODE columns.
4. Call **HrQueryAllRows** to retrieve all of the rows in the status table.
5. Pass the entry identifier for each row in the table in a call to **IMAPISession::OpenEntry**. Because this interface is marshalled from the MAPI spooler's process context to the client's process context, unlike interfaces typically obtained from address book or message store providers, issues concerning multithreading are of more importance.
6. Call the status object's **IUnknown::QueryInterface** method, passing IID_IMAPIFolder as the interface identifier, to retrieve the remote folder.

   The remote folder is not a complete folder implementation; it supports only a subset of folder methods and properties. One of the required methods, **IMAPIProp::GetProps**, supports the retrieval of the following properties:

PR_ACCESS	PR_ACCESS_LEVEL
PR_CONTENT_COUNT	PR_ASSOC_CONTENT_COUNT
PR_FOLDER_TYPE	PR_OBJECT_TYPE
PR_SUBFOLDERS	PR_CREATION_VERSION
PR_CREATION_TIME	PR_DISPLAY_NAME
PR_DISPLAY_TYPE	

   Remote folders must also support the **IMAPIProp::GetPropList**, **IMAPIContainer::GetContentsTable**, and **IMAPIFolder::SetMessageStatus** methods. Remote folder contents tables typically support the following columns:

PR_DISPLAY_TO	PR_ENTRYID
PR_HASATTACH	PR_IMPORTANCE
PR_INSTANCE_KEY	PR_MESSAGE_CLASS
PR_MESSAGE_DELIVERY_TIME	PR_MESSAGE_FLAGS
PR_MESSAGE_DOWNLOAD_TIME	PR_MESSAGE_SIZE
PR_MSG_STATUS	PR_OBJECT_TYPE
PR_NORMALIZED_SUBJECT	PR_PRIORITY
PR_SENDER_NAME	PR_SENSITIVITY
PR_SENT_REPRESENTING_NAME	PR_SUBJECT

7. Call the transport provider's **IMAPIStatus::ValidateState** method the first time that one of the transfer options are picked. Either the REFRESH_XP_HEADER_CACHE or PROCESS_XP_HEADER_CACHE process flag should be set as well as the SHOW_XP_SSESSION_UI flag to allow the user interface to be shown.

Note To mark a particular message header for downloading or deletion, a client calls **IMAPIFolder::SetMessageStatus** and sets either the MSGSTATUS_REMOTE_DOWNLOAD or MSGSTATUS_REMOTE_DELETE flags.

Writing a Hierarchy Viewer

A hierarchy viewer is a user interface component that is used for displaying folder and address book container hierarchy tables. Hierarchy viewers can display members of the hierarchy at different levels, expanding and contracting each level on demand.

If your client is using the 32-bit version of MAPI, it can implement its hierarchy viewer manually or with the standard tree control from COMCTL32.DLL.

The container property, PR_DEPTH, controls the level at which a hierarchy member is displayed. Entries that represent top-level address book containers or folders have their PR_DEPTH property set to zero. The value of this property is incremented sequentially for entries in sequential levels. That is, when a user selects a top-level container to expand, your client displays all containers with PR_DEPTH set to 1. When a user expands one of these subcontainers, all containers with PR_DEPTH set to 2 are made visible, and so on.

Hierarchy viewers support a different range of depths. Your client can limit its viewer to only one or two levels or support multiple levels, if displaying a expansive hierarchy is a priority.

The address book provides your client with a hierarchy viewer for all of the top-level containers in the address book.

To access the address book hierarchy viewer

1. Call **IAddrBook::OpenEntry**, passing a null entry identifier, to open the address book's root container.
2. Call the root container's **IMAPIContainer::GetHierarchyTable** method to access the hierarchy table of the MAPI address book.

To access the default message store's hierarchy viewer

1. Call **IMAPISession::GetMsgStoresTable** to access the message store table.
2. Build a restriction using the **SPropertyRestriction** structure to limit the table to only those rows that have a PR_DEFAULT_STORE property set to TRUE.
3. Call **IMAPITable::FindRow**, passing it the **SPropertyRestriction**, to locate the row representing the default message store.
4. Call **IMAPISession::OpenEntry**, passing in the PR_ENTRYID property from the default message store's row in the message store table.
5. Call the message store's **IMAPIProp::GetProps** method to retrieve the PR_IPM_SUBTREE_ENTRYID property.
6. Call the message store's **IMsgStore::OpenEntry** method, passing the PR_IPM_SUBTREE_ENTRYID property, to open the root folder of the message store's IPM subtree.
7. Call the IPM root folder's **IMAPIContainer::GetHierarchyTable** method to access its hierarchy table.

Using the MAPI Utilities

Occasionally, a client application will make only MAPI utility calls. Not all clients are messaging or configuration clients, which are clients that communicate with service providers and messaging subsystems. Clients using MAPI utilities only, such as some testing applications and the **ITableData** and **IPropData** interfaces, call the API function **ScInitMapiUtil** rather than the **MAPIInitialize** function. Clients that have not called **MAPIInitialize** and are using tables through the **ITableData** methods should be aware that table notifications will not work. Notifications require the use of the MAPI libraries.

ScInitMapiUtil enables clients to use utility functions that require MAPI allocators, but that do not ask for the allocators explicitly. When it is time to shut down, a client that has called **ScInitMapiUtil** at startup time needs to call **DeinitMapiUtil** to free resources. There should be no call made to **MAPIUninitialize**.

Converting to OLE Messaging Library Identifiers

Some clients use multiple client interfaces to take advantage of what more than one client API has to offer. Typically these clients use MAPI for some tasks and a simpler API such as the OLE Messaging Library for other tasks. If your client takes advantage of both the MAPI and the OLE Messaging Library client interfaces, it will need to convert between the two types of identifiers that are used for each interface. Whereas MAPI objects are accessed with binary entry identifiers, OLE Messaging Library objects are accessed with hexadecimal message identifiers.

To convert a MAPI entry identifier to an OLE Messaging Library message identifier

1. Call the message's **IMAPIProp::GetProps** method to retrieve its PR_ENTRYID property.
2. Call **HexFromBin** to convert the entry identifier to a hexadecimal string or use client code to perform the conversion.
3. Call the Win32 function **SysAllocString**, if necessary, to make the string compatible with Visual Basic.
4. Make the hexadecimal string available to the Visual Basic application.
5. Use the **Session.GetMessage** method in the OLE Messaging Library to open the message.

To reverse the process, converting an OLE Messaging Library message identifier to a MAPI entry identifier, call **FBinFromHex**. **FBinFromHex** converts a hexadecimal string to binary data.

Implementing a Progress Indicator

Many of the operations initiated by clients take a significant amount of time. One of the input parameters to these potentially lengthy operations is a pointer to a progress object, an object that implements the **IMAPIProgress** interface. Progress objects control the appearance and display of progress indicators and are implemented by clients and by MAPI. Your client can choose whether or not to implement a progress object; the MAPI implementation is available for service providers to use if your client elects not to supply an implementation.

Progress objects work with the following pieces of data:

- A global minimum value which, when your client's **IMAPIProgress::Progress** method is called, should be less than or equal to the value of the *ulValue* parameter. At the beginning of the operation, *ulValue* will be equal to this minimum value.
- A global maximum value which, when when your client's **IMAPIProgress::Progress** method is called, should be greater than or equal to the *ulValue* parameter. At the end of the operation, *ulValue* will be equal to this maximum value.
- A flags value which indicates whether the progress corresponds to a top or lower level item.
- A value that indicates the current level of progress for the operation.
- The number of the currently processed item relative to the total.
- The total number of items to be processed during the operation.

The minimum and maximum values represent the beginning and end of the operation in numeric form. Use 0 for the initial minimum value and 1000 for the initial maximum value, passing these values to service providers in the **IMAPIProgress::GetMin** and **IMAPIProgress::GetMax** methods. Service providers reset these values when they call **IMAPIProgress::SetLimits**.

The flags value is used by service providers to determine how they should set the other values. Initialize the flags value to MAPI_TOP_LEVEL and return this value in your client's implementation of **GetFlags** until the service provider resets it by calling **SetLimits**.

In your client's implementation of the **SetLimits** method, save local copies of each of the parameters: *lpulMin*, *lpulMax*, and *lpulFlags*. These values should be readily available when a service provider calls your client's **GetMin**, **GetMax**, or **GetFlags** methods.

To update the display of the progress indicator, service providers call your client's **IMAPIProgress::Progress** method. There are three parameters to this method: a value, a count, and a total. Use the first parameter, *ulValue*, to display the progress indicator. The *ulValue* parameter is the progres indicator and will be equal to global *ulMin* only at the very beginning of the operation and equal to global *ulMax* only at the completion of the operation.

Use the second and third parameters, *ulCount* and *ulTotal*, if available, to display an optional message such as "5 items completed out of 10." If the second and third parameters are set to 0, your client can choose whether or not to visually change the progress indicator. Some service providers set these parameters to zeroes to indicate that they are processing a subobject whose progress is monitored relative to a parent object. In this situation, it makes sense to change the display only when the parent object reports progress. Some service providers pass zeroes for these parameters every time.

Improving Client Performance

The following tips will help you create a client application with the best possible performance.

Avoid Certain Methods at Startup

To improve performance at startup time, avoid making the following calls:

IMAPISession::EnumAdrTypes	**IMAPISession::Logoff**
IMAPISession::GetStatusTable	**IMAPISession::MessageOptions**
IMAPISession::QueryDefaultMessageOpt	**IMAPISession::Release**
IMAPISession::QueryIdentity	**IMAPIStatus::ValidateState**

The call to **IMAPIStatus::ValidateState** affects performance only when made on either the MAPI spooler or the MAPI subystem. The reason that these methods slow start up processing is because they cannot complete until the MAPI spooler has finished its startup tasks.

Another task to avoid at startup is searching the message store. Make your **IMAPIContainer::SetSearchCriteria** call when startup processing has finished.

Get and Set Multiple Properties

By getting and setting as many properties as possible with the least number of calls, remote activity is curtailed and the overhead involved with each property is reduced. Although service providers try to collect properties before making a remote procedure call for the retrieval or modification, your client can optimize this effort by requesting multiple properties to begin with.

For example, if your client works with routing lists that describe future recipients with named properties belonging to particular property sets, process all of the recipients with two calls, one to **IMAPIProp::GetIDsFromNames** to retrieve the identifiers for all of the recipient properties and the other to **IMAPIProp::GetProps** to retrieve all of the values. The alternative, making a call to **GetIDsFromNames** followed by a call to **GetProps** for each recipient, is much less efficient.

Use a Table to Work with Properties

Many properties are available both from the objects that support them and as columns on tables. Whenever possible, retrieve these properties through the table.

The IMAPITable interface enables your client to access properties by calling:

1. **SetColumn** to include all of the properties that your client needs.
2. **QueryRows** to retrieve all of the rows of the table.

These two calls are usually sufficient for retrieving enough information to display to a user and frequently sufficient for any internal processing that your client performs, making a call to **OpenEntry** to open the object unnecessary.

There are only two exceptions:

- If the property is over 255 bytes. The **IMAPITable** interface might not return the entire property value, instead truncating it at 255 bytes. Think about this tradeoff, though. If your client is displaying this data to the user, 255 bytes may be enough for a textual field such as a comment.
- If your client needs a specific property from a single row in a table. In this case it is unnecessary to create a table with properties that will never be used. Most of the time your client will need the same properties for all rows.

Get and Set Properties with GetProps and SetProps

Whenever possible, try to retrieve or modify a property with the **IMAPIProp::GetProps** and **IMAPIProp::SetProps** methods. Unless the property your client is working with is very large, these methods should be adequate. The other alternative is to read from or write to a stream with the OLE interface **IStream**. Streams can handle very large properties successfully, but they are a greater drain on resources because they require the OLE libraries. Use the **IStream** interface only after your call to **IMAPIProp::GetProps** or **IMAPIProp::SetProps** fails.

Save Frequently Used Properties

Improve your client's performance by storing data that takes time and resources to get and is accessed frequently. Using extra memory can sometimes be a better option that repeated retrievals. Use caution and care when creating a cache for storing this data. Keep in mind that a poorly designed cache can negatively impact performance.

For example, most IPM clients need to display and open the IPM subtree of folders many times during a typical session. Your client can improve performance by storing the entry identifiers for each of these folders.

Defer Processing

Pass the MAPI_DEFERRED_ERRORS flag to method calls as much as possible. Many of the MAPI method calls have been optimized to accept this flag, causing the provider to either postpone the requested task until multiple tasks can be performed at once or your client can wait no longer for the results.

For example, if your client passes MAPI_DEFERRED_ERRORS to **IMsgStore::OpenEntry** to open a folder, the opening of the folder and a possible remote call can be postponed until your client makes another call such as the folder's **GetHierarchyTable** or **GetProps** methods. Both **GetHierarchyTable** and **GetProps** require the return of data from the service provider, a task that must be performed immediately.

Another way to defer processing is to simply not make a call. By being aware of the user and when the user can perceive a drain on resources or processing time, your client can determine when it makes sense to make calls. There is an opportunity to improve performance by making calls either at a time when the user will not notice or not making them at all.

For example, consider the situation when your client is receiving more than one notification per second from a message store that is moving a great number of messages. A progress indicator is displayed to indicate the percentage of the operation's completion. Users typically will not perceive this operation to be slow until a few seconds have passed. Therefore, if your client is updating the progress indicator, do not make any changes until at least four seconds after the initiation of the move operation. This will save time in the common cases when the operation is fast and inform users in a timely manner when the operation is slow.

Write Uncompressed Formatted Text

When preparing to send a message with formatted text, your client can either set the message's PR_RTF_COMPRESSED property to compressed or uncompressed text. Writing compressed text in the PR_RTF_COMPRESSED property is a very CPU intensive operation and can dramatically affect performance.

To improve the performance of sending formatted messages, your client can either:

1. Upgrade the CPU, a solution that is not always plausible.
2. Write uncompressed text in the PR_RTF_COMPRESSED property.

The procedure for setting PR_RTF_COMPRESSED with uncompressed text is the same as for setting it with compressed text, with one exception. When calling **WrapCompressedRTFStream,** set the **STORE_UNCOMPRESSED_RTF** flag in the *ulFlags* parameter. Setting uncompressed text has the disadvantage in that it increases the size of messages.

Avoid Using IStream::SetSize to Extend a Stream

When writing to streams, it is sometimes necessary to enlarge them because their initial size is no longer sufficient. Use the OLE method **IStream::Write** to accomplish this rather than **IStream::SetSize**. **IStream::Write** automatically extends the stream, making **IStream::SetSize** unnecessary. Calling **IStream::Write** without **IStream::SetSize** can be up to three times faster than making the **SetSize** call prior to **Write**.

Sort Tables after Setting Columns and Restrictions

When your client needs to limit the view of a sorted table, always make the following **IMAPITable** calls in the following order:

1. **IMAPITable::SetColumns** to define the column set.
2. **IMAPITable::Restrict** to impose the restriction.
3. **IMAPITable::SortTable** to perform the sort.

If the sorted table is categorized, make a call to **SetCollapsedState** if necessary after the **SortTable** call. This ordering of calls is important because most service providers sort a table as the last task to achieve the best performance. If, for example, your client needs a message store provider to categorize a folder contents table before a restriction is imposed, this categorization will be removed during the processing of the restriction. A second categorization will be necessary.

Call QueryRows for Small Tables

When your client is retrieving rows from a small table, call **IMAPITable::QueryRows** instead of first building a restriction. Creating a restriction impacts performance because the provider must first create a table, find the matching rows in the original table, and then copy the rows to the new table. If the total number of rows in the table is less than one hundred, it is probably more effective to read all of the rows and then call **IMAPITable::FindRow** to find the row that your client is looking for. This is a particularly good strategy if your client only occasionally retrieves this information.

A proper time to use a restriction is when the restricted or filtered information will be used over a longer period of time or used frequently. For instance, if your client always needs a view with unread messages, then a restriction is the proper call to use.

Open OLE Attachments with IStreamDocfile

When opening an OLE object attachment, use the **IStreamDocfile** interface rather than **IStream** or **IStorage**. **IStreamDocfile** provides direct access to the object using structured storage, eliminating the need to perform a copy operation and reducing overhead.

Name Folders with Character Strings

If your client has one or more folders that it accesses frequently during a session, consider assigning it a name with the **IMsgStore::SetReceiveFolder** method. Although **IMsgStore::SetReceiveFolder** is used primarily to establish special folders to receive incoming messages for particular message classes, it can also be used to associate any folder with a name. The name can be the same as the message class or it can be any character string, customized for your client's use. Associating a name with a folder decreases the time it takes to find and open the folder.

Service Provider Basics

The following topics apply to most, if not all, types of service providers. Details specific to one or more types of service providers are covered in the topics that relate to the particular service provider.

There are three common types of service providers: address book, message store, and transport. Developers can also create messaging hook providers and profile providers. Because messaging hook providers are active processes, they more closely resemble client extensions to the MAPI spooler than the other service providers. The MAPI spooler is the only process not driven directly by client actions. Another difference between messaging hook and other types of service providers is that messaging hook providers use a session object instead of a support object. MAPI provides the other types of service providers with a customized support object.

Your service provider should have three header files: one public header file and two internal files. Use the public header file for configuration and for documenting properties and their values. Include in one of the internal header files all of the necessary public MAPI headers; this header file should be included in all of your service provider source files. Use the other internal file to define resource identifiers.

Assign file names for your provider's executable files trhat are six characters or less in length. This is to allow a suffix to be appended to the end of the name to identify the platform.

Loading Service Providers

Your service provider's DLL will be loaded by MAPI either when a client makes a request that requires the services of your provider or as part of the standard logon process. At logon time, clients call the **IMAPISession::OpenAddressBook** method to load each of the address book providers included in the profile and the **IMAPISession::OpenMsgStore** method to load specific message store providers. There is no specific call that clients make to load transport providers; they are loaded when they are needed to send or receive messages.

To load your service provider, MAPI

1. Locates the name of your provider's DLL in the active profile.
2. Calls the Windows API function **LoadLibrary**.
3. Calls your service provider's DLL entry point function.

Service providers are required to register the name of their DLL in the MAPISVC.INF configuration file to ensure that it appears in the profile. When your service provider is added to a profile, either individually or as part of a message service, all of the [Service Provider] sections from MAPISVC.INF that apply to your provider are copied into the profile.

Because MAPI calls the Windows function **LoadLibrary** either every time it uses a service provider DLL, regardless of whether it has already been loaded, or only the first time, your service provider must not make assumptions about the number of times that it will be loaded. For every call to **LoadLibrary**, MAPI makes a call to the Windows API function **FreeLibrary** when a service provider DLL is no longer needed.

MAPI calls your service provider's DLL entry point function to initiate the logon process. The lengthy list of parameters passed to an entry point function include pointers to the standard memory allocators and version information. Service provider entry point functions ensure that a provider is using a version of the service provider interface (SPI) that is compatible with the version being used by MAPI and return pointers to newly created provider objects.

Address book providers that are part of a message service are loaded by MAPI before any of the other providers in the service.

About Provider DLL Entry Point Functions

Your service provider implements a DLL entry point function that MAPI calls during the load operation. Depending on the type of provider, this function will adhere to a different prototype. MAPI defines four different entry point function prototypes for service providers:

Address book providers	**ABProviderInit**
Message store providers	**MSProviderInit**
Transport providers	**XPProviderInit**
Messaging hook providers	**HPProviderInit**

Much of the functionality in these prototypes is identical across service provider types. Address book, message store, and transport DLL entry point functions initiate the process of configuration by performing two main tasks:

- Checking the version of the service provider interface (SPI) to make sure MAPI is using a version that is compatible with the version that your service provider is using.
- Instantiating a provider object appropriate for your type of service provider.

Messaging hook providers do not implement the same functionality as the other providers. Whereas the other providers instantiate provider and logon objects, messaging hook providers instantiate spooler hook objects, or objects that implement the **ISpoolerHook** interface.

Checking Service Provider Interface Versions

Because MAPI supports multiple versions of the Service Provider Interface (SPI) and every provider can support multiple versions, your DLL entry point function must check that MAPI's version is compatible with your version. Two input parameters are passed to your provider DLL entry point function for version checking. The *lpulMAPIVer* parameter contains the version of the SPI that MAPI is using while the *lpulProviderVer* parameter contains the version that your provider is using. These parameters are 32-bit unsigned integers composed of three parts:

- Bits 24-31 represent the major version.
- Bits 16-23 represent the minor version.
- Bits 0-15 represent the micro version.

Although the major version number rarely changes, the minor version number changes whenever MAPI is released and the SPI has changed. The micro version is the Microsoft internal build version; it is used to track changes during the development process.

MAPI defines the CURRENT_SPI_VERSION constant, documented in the MAPISPI.H header file, for indicating the present SPI version. Your provider's version check should fail with MAPI_E_VERSION if your provider is using a version of the SPI that is newer than the version that MAPI is using.

Service Provider Logon

Your service provider begins its logon process when MAPI calls your provider object's logon method. This call is made after your entry point function has completed. Logon methods are called **Logon** for address book and message store providers and **TransportLogon** for transport providers. Regardless of the type of service provider, all logon methods are passed a pointer to a support object and perform a basic set of tasks.

To implement a provider logon method

1. Increment the reference count on the support object by calling **IUnknown::AddRef**.
2. Call **IMAPISupport::OpenProfileSection** to access your provider's profile section.
3. Call the profile section's **IMAPIProp::SetProps** method to set the following properties:

PR_DISPLAY_NAME	PR_ENTRYID
PR_PROVIDER_DISPLAY	PR_RECORD_KEY

   Note Do not attempt to set the profile section's PR_RESOURCE_FLAGS or PR_PROVIDER_DLL_NAME properties. At logon time, these properties are read-only.

4. Verify configuration by checking that the appropriate properties are either stored in the profile or available from the user.
5. Call **IMAPISupport::SetProviderUID** to register a unique identifier, or **MAPIUID**, if your provider is an address book or message store provider. Transports register **MAPIUID**s when MAPI calls their **IXPLogon::AddressTypes** method.
6. Instantiate a logon object and return.

Your logon method will usually return one of the following values:

- S_OK to indicate a successful logon.
- MAPI_E_UNCONFIGURED to indicate that one or more of the configuration properties were unavailable.
- MAPI_E_USER_CANCEL to indicate that the user canceled the configuration dialog box, causing configuration properties to be unavailable.
- MAPI_E_FAILONEPROVIDER to indicate that your provider could not be configured, but that MAPI should allow it to be used regardless. Logon methods should return this value to report a non-fatal error, such as when the provider requires a password and cannot prompt the user for it because the user interface is disabled.

This list of tasks is the minimum set required. Your service provider might want to include additional functionality such as calling **IMAPISupport::ModifyStatusRow** to add a row in the status table.

Note To achieve the best performance at logon time, avoid calling either **IMAPISupport::PrepareSubmit** or **IMAPISupport::SpoolerNotify** for any reason. These calls require that the MAPI spooler complete its start up processing before they can complete.

Verifying Service Provider Configuration

Your service provider's logon method must check that all of the properties needed for full operation are set and set correctly. The number of required properties varies from provider to provider depending on the amount of user input required. Some service providers keep all of the necessary properties in the profile. Other service providers keep a partial set of properties in the profile and prompt the user for missing values. Still other providers do not store properties in the profile at all, relying on the user to supply all of the information needed for configuration.

To retrieve properties stored in the profile, call the **IMAPISupport::OpenProfileSection** method. **OpenProfileSection** opens the section of the profile that belongs to the service provider identified by a **MAPIUID** specified as an input parameter. MAPI passes back a pointer to the profile section object that can be used for further access.

To prompt the user for configuration property values, your service provider displays a dialog box. All logon methods allow a user interface display unless MAPI sets a flag on the call prohibiting the display. Logon methods fail with the error value MAPI_E_UNCONFIGURED when:

- The flag disallowing a user interface is set and not all of the configuration properties are available through the profile.
- The flag disallowing a user interface is not set, but the user did not supply all of the required information.

When your service provider fails configuration with MAPI_E_UNCONFIGURED, MAPI calls its DLL entry point function again. If the information cannot be located with the second call, depending on how important your service provider is, the session might terminate.

The following illustration shows the logic required for configuration in a service provider logon method.

{ewc msdncd, EWGraphic, groupx835 0 /a "MAPI_62.WMF"}

About Service Provider Logon Objects

Every address book, message store, and transport provider instantiates a logon object as part of its provider object's logon method implementation. Logon objects implement methods that help MAPI service client requests. Because the different types of service providers supply different types of services, there are different types of logon objects. Address book logon objects, which implement the **IABLogon : IUnknown** interface, and message store logon objects, which implement the **IMSLogon : IUnknown** interface, have the following features in common:

- Support for event notification (**Advise** and **Unadvise** methods).
- Entry identifier comparison (**CompareEntryIDs** method).
- Access to additional error information (**GetLastError** method).
- Access to objects implemented by the service provider (**OpenEntry** method).
- Access to status objects if available (**OpenStatusEntry** method).
- Logoff (**Logoff** method).

Address book logon methods also provide MAPI with a table of custom recipient templates through the **GetOneOffTable** method, access to a particular template through the **OpenTemplateID** method, and recipient list preparation through the **PrepareRecips** method.

A transport provider's logon object, which implements **IXPLogon**, is quite different from the logon objects implemented by the other types of service providers. It has only two features in common with the other logon objects: access to a status object through the **OpenStatusEntry** method and a logoff operation through the **TransportLogoff** method. Transport providers implement the following unique features in their logon objects:

- Registration for address types (**AddressTypes** method) and message options (**RegisterOptions** method).
- Control of message transmission (**StartMessage**, **EndMessage**, **SubmitMessage**).
- Internal state validation (**ValidateState** method).
- Ability to download or upload messages on demand (**FlushQueues** method).
- Ability to query for pending messages (**Poll** method).
- Idle state detection (**Idle** method).
- Interaction with the MAPI spooler (**TransportNotify** method).

Registering Service Provider Unique Identifiers

Address book, message store, and transport providers are represented by one or more unique identifiers, or **MAPIUID**s. A **MAPIUID** is a 16-byte identifier that contains a globally unique identifier, or GUID, and can be statically defined using a constant or created with the Microsoft utility UUIDGEN.EXE. The following constant is an example of a statically defined **MAPIUID** for an address book provider:

```
#define AB_UID_PROVIDER  { 0Xe3, 0x3c, 0x67, 0xa0, \
                           0xc8, 0x1f, 0x11, 0xce, \
                           0xb2, 0xe4, 0x0, 0xaa, \
                            0x0, 0x51, 0xe, 0x3b  }
```

If you are writing an address book or message store provider, call **IMAPISupport::SetProviderUID** to register a **MAPIUID** for each instantiated logon object and include this **MAPIUID** in the first 16 bytes of the **ab** member of every entry identifier that your provider creates. MAPI uses **MAPIUID**s to associate objects with service providers. When a client calls the **IMAPISession::OpenEntry** method to open an object, MAPI examines the **MAPIUID** portion of the entry identifier, matching it against the registered **MAPIUID**, to determine which logon object should receive the open request.

If you are writing a transport provider, you will register **MAPIUID**s when MAPI calls your provider's **IXPLogon::AddressTypes** method. MAPI uses the **MAPIUID**s registered by transport providers to assign responsibility for message delivery.

Although service providers typically register a single **MAPIUID**, your provider can register multiple **MAPIUID**s. If your address book or message store provider supports multiple logon objects, perhaps by permitting a user to add more than one instance of your provider to their profile, you might want to implement a different **MAPIUID** for each logon object. There are a few other reasons to support more than one **MAPIUID**:

- Your provider must support more than one version of its entry identifiers. Assign a different **MAPIUID** for each version.
- Your provider wants to distinguish between the types of objects it supports. For example, an address book provider might want to register one **MAPIUID** to use in the entry identifiers of its messaging user objects and a different **MAPIUID** to use for distribution lists.

When there are multiple logon objects that are concurrently active, it makes sense to have unique **MAPIUID**s for each one. This increases the accuracy with which MAPI matches entry identifiers to service providers and saves some work. When every logon object has its own unique identifier, MAPI can guarantee that any request it routes to a logon object can be handled by that object. When logon objects share **MAPIUID**s, MAPI routes the request to the first logon object that is identified by the **MAPIUID**. If one of your logon objects receives a request that it cannot process because it does not handle the entry identifier, pass the request on to your next logon object before returning an error.

Shutting Down Service Providers

Your service provider will be shut down when a client calls **IMAPISession::Logoff**. If your provider is a message store, a client call to **IMsgStore::StoreLogoff** will also initiate the shutdown process. **Logoff** ends the session and shuts down all active service providers whereas **StoreLogoff** shuts down one particular message store provider and has no affect on the session. There is no explicit way for shutting down a particular address book or transport provider; these providers can only be shut down at session's end.

Your DLL will be unloaded when MAPI calls the Win32 API function **FreeLibrary** after the last active client has called **MAPIUninitialize**. By this time, your service provider will have finished shutting down.

After a client has called **IMAPISession::Logoff** to end a session, MAPI calls your provider's logoff method in its logon object. For address book providers, MAPI calls **IABLogon::Logoff**; for message store providers, MAPI calls **IMSLogon::Logoff**; for transport providers, MAPI calls **IXPLogon::TransportLogoff**.

In your provider's logoff method

1. Release all open objects, including subobjects and status objects.
2. Call **IUnknown::Release** on the support object to decrement its reference count. The support object's **Release** method removes all of your provider's registered **MAPIUID**s and its row in the status table.

When your provider's logoff method returns, MAPI makes the following calls to your provider:

1. A call to your logon object's **IUnknown::Release** method.
2. A call to your provider object's **Shutdown** method (**IABProvider::Shutdown**, **IMSProvider::Shutdown**, or **IXPProvider::Shutdown**).
3. A call to your provider object's **IUnknown::Release** method.

In your **Shutdown** implementation, perform any necessary final cleanup tasks.

Shutting Down a Message Store Provider

If your provider is a message store provider, it can be shut down in two ways: when a client or the MAPI spooler calls **IMsgStore::StoreLogoff** or when a client calls **IMAPISession::Logoff**.

Your message store provider's implementation of **IMsgStore::StoreLogoff** calls **IMAPISupport::StoreLogoffTransports** to inform MAPI that it is being shut down and that any related transport providers should be logged off. Calling **StoreLogoff** causes a message store provider to be shut down in an orderly and controlled manner, under the control of the client.

When **IMsgStore::StoreLogoff** returns, its caller invokes the message store's **IUnknown::Release** method. Your implementation of **IMsgStore::Release** releases your provider's support object by calling its **IUnknown::Release** method.

MAPI performs the following tasks in its implementation of **IMAPISupport::Release**:

1. Removes all of the **MAPIUID**s registered by the message store provider.
2. Removes the message store provider's row from the status table.
3. Calls **IMSLogon::Logoff** to release all open objects, subobjects, and status objects.
4. Calls **IMSLogon::Release** to release the message store provider's logon object.

Some clients might omit the call to **IMsgStore::StoreLogoff**, initiating the shutdown of your message store provider with the call to the message store's **IUnknown::Release** method. A shutdown under these circumstances without the call to **StoreLogoff** is less orderly and controlled. Write your message store's **Release** method to handle this possibility and keep track of whether or not a call to **IMAPISupport::StoreLogoffTransports** has occurred. **StoreLogoffTransports** must be called once during the shutdown process. If you detect in your **Release** method that **StoreLogoffTransports** has not yet been called, invoke it with the LOGOFF_ABORT flag.

Invalidating Objects

The support object has a method called **MakeInvalid** which replaces an object's vtable with a vtable containing implementations for the three **IUnknown** methods: **AddRef**, **Release**, and **QueryInterface**. Your service provider can use the **IMAPISupport::MakeInvalid** method in the implementation of your logon object's logoff method to give MAPI the ultimate responsibility for freeing the memory associated with an object. Your service provider can free all of the resources connected with an object and then call **MakeInvalid** to invalidate all of the methods in its inherited interfaces. Calls to any of these methods will return MAPI_E_INVALID_OBJECT. Using **MakeInvalid** is an option that many service providers choose to ignore.

Using Support Objects

MAPI furnishes a support object for all service providers during logon. The purpose of the support object is to provide implementations for a fairly large number of methods commonly used by the providers. Each support object also contains contextual data specific to its own instance, such as the session the provider is running in, the profile section the provider is using, and error information for the session.

There are four different types of support objects: one for each major provider type (address book, message store, and transport) and one for configuration support when the message service entry point function is called. Messaging hook providers run within the same thread as the MAPI spooler and are given a session object instead of a support object.

Each type of support object has a different set of methods that are operational. Some methods, such as **OpenProfileSection**, are used in all four types. Other methods apply only to some of the types, for example **SpoolerNotify**, which is used only by message store and transport providers. A call to a method not supported for the calling provider type returns MAPI_E_NO_SUPPORT.

Support objects are implemented by MAPI and called by service providers. They are not accessible by clients. They expose the **IMAPISupport : IUnknown** interface, specified in the header file MAPISPI.H. The interface identifier is IID_IMAPISup, and the pointer type to a support object is LPMAPISUP. No MAPI properties are exposed by support objects.

Service providers and message services use support objects to accomplish the following tasks:

- Access a profile section for configuration
- Allocate and free memory
- Obtain a unique identifier for newly created objects
- Register message preprocessors
- Prepare message delivery reports
- Handle event notification
- Change provider status information

Using Support Objects During Logon

A provider is presented with a different support object every time MAPI logs it on, which is at least once per session. Address book and transport providers are logged on every time a client logs on to MAPI with a profile section requesting that provider. A message store provider is logged on whenever a client calls **IMAPISession::OpenMsgStore**; in the case of multiple logons in a session, the provider can choose either to retain and use each support object separately, or to discard the subsequent support objects and call **AddRef** on the one it already has. In the latter case MAPI takes care of discarding the unused support object.

MAPI logs on a service provider by calling the logon method of the provider object (**IABProvider::Logon**, **IMSProvider::Logon**, or **IXPProvider::TransportLogon**). One parameter of this call is a pointer to the new support object for the current logon. The logon method in turn instantiates a logon object, passing it the support object pointer. The logon object should use this pointer to call **AddRef** on the support object. When the client logs off, MAPI calls the logoff method of the logon object, which should call **Release** on the support object.

If a provider returns from a logon call with MAPI_E_UNCONFIGURED or a client requests provider configuration, MAPI instantiates a configuration support object and calls the message service entry point function for that provider. The support object, which is passed by pointer in this call, is valid only until the return from the entry point function; the message service should not call **AddRef** on it or retain it in any way.

A service provider normally has no need to call its message service's entry point function directly, since MAPI calls it on behalf of an unconfigured provider. However, a provider may need to reconfigure while actively running, for example during servicing of an **IMAPIStatus::SettingsDialog** call. In such a case the provider must call **IMAPISupport::GetSvcConfigSupportObj** to obtain a configuration support object. The provider then passes a pointer to this object in a call to the message service's implementation of the **MSGSERVICEENTRY** prototype.

Using Support Objects for Configuration

Following logon, the service provider's logon method calls **IMAPISupport::OpenProfileSection** to obtain a profile section object. The provider should avoid unnecessary interaction with the user during session startup, but it must be prepared to allow the user to alter some of the configuration properties in the profile section. Therefore, it should include MAPI_MODIFY in the *ulFlags* parameter in the **OpenProfileSection** call. This also creates a profile section if none exists. Note that the message service entry point function and the provider's logon method both call **OpenProfileSection**, but under different circumstances.

The profile section object exposes the **IProfSect : IMAPIProp** interface and furnishes several properties including PR_DEFAULT_PROFILE, PR_PROVIDER_DISPLAY, and PR_PROVIDER_DLL_NAME. The **IProfSect** interface is nontransactional, meaning its properties can be modified without waiting for a **SaveChanges** call. Because of this, operating a user dialog directly from the profile section permits the settings to change before the user even selects **OK** or **Cancel**. Therefore it is recommended that, once the profile section is opened, the relevant properties be copied to a property data object instantiated from the **IPropData : IMAPIProp** interface. The provider can perform the configuration itself using the property data object, and then copy the properties back to the profile section when the user has confirmed them.

If service provider configuration is to interact with the user, it is recommended that a property sheet be used. This maintains consistency with the Windows user interface. To prepare for using a property sheet, the provider or message service constructs a display table containing a row for each control that is to appear on the property sheet. The PR_CONTROL_STRUCTURE column in each row points to a structure that references the corresponding property either by tag or by name. One option available for creating this table is the **BuildDisplayTable** function.

Once a display table is built and populated, the provider or message service calls **IMAPISupport::DoConfigPropsheet**, which displays the property sheet based on the display table, performs the user dialog, and stores the new configuration settings in the object pointed to by the *lpConfigData* parameter. This object, supplied by the **DoConfigPropsheet** caller, exposes the properties referenced in the display table. It must be a property object, that is, it must derive from the **IMAPIProp : IUnknown** interface. A suitable candidate for this object is the property data object recommended earlier in preference to the profile section object. This object serves both purposes since **IPropData** derives from **IMAPIProp**.

Using Support Objects for Utility Services

The **IMAPISupport::GetMemAllocRoutines** method is available for determining the addresses of the memory allocation and deallocation functions without having to link with MAPI. Using **GetMemAllocRoutines** also makes it easier to trace memory leaks by surrounding the allocation function calls with debugging code. If the provider calls **GetMemAllocRoutines**, as is recommended, it should do so before calling the **CreateIProp** function, which requires the allocation function addresses as parameters.

When a provider or message service needs to create a new object such as an address book entry or a message, it should furnish a search key on this object. For a distribution list or messaging user object, MAPI specifies how the PR_SEARCH_KEY property should be constructed. For address book containers and messages, the provider should call **IMAPISupport::NewUID** to obtain a unique identifier it can use in building a search key. The provider's own hard-coded **MAPIUID** should be used only in a PR_ENTRYID and not in a search key.

A client application can sometimes release an object without releasing one or more of its affiliated objects. In such a case a provider may wish to render an unreleased object unusable. To do this, the provider frees all of the resources connected with the object and then calls **IMAPISupport::MakeInvalid** to invalidate the object's vtable. **MakeInvalid** replaces the vtable's **IUnknown** methods (**QueryInterface**, **AddRef**, and **Release**) with standard MAPI implementations and causes all other methods to return MAPI_E_INVALID_OBJECT. **MakeInvalid** also frees all the object's memory other than the vtable.

If a call to one of the support object's methods returns an error, the provider may elect to inform the user and possibly request a course of action. The **IMAPISupport::GetLastError** method populates a **MAPIERROR** structure with character strings and context useful for displaying information about the error. This method is most helpful when the error is MAPI_E_EXTENDED_ERROR; for other errors such as MAPI_E_CALL_FAILED, **GetLastError** may return NULL in the *lppMAPIError* parameter, meaning no additional information is available. Note that **GetLastError**, which is exposed in several interfaces, can only be used for errors that occurred in its own object. Providers cannot forward **GetLastError** calls to **IMAPISupport::GetLastError**.

Using Support Objects Among Service Providers

A provider sometimes needs to open an object belonging to another provider. For example, an incoming transport provider might receive a message in Rich Text Format and need to find out if a particular recipient can receive it in that format. The transport provider must open the recipient to check its PR_SEND_RICH_INFO property. The recipient belongs to an address book container that the transport provider does not normally have access to. So the transport provider obtains the recipient's entry identifier from the PR_ENTRYID column of the recipient table.

Next, the transport provider calls **IMAPISupport::OpenEntry**, in this example with NULL for the *lpInterface* parameter since it doesn't know yet whether the recipient is a distribution list or messaging user. The support object's method calls **IMAPISession::OpenEntry** to let the session determine which of its providers has access to the desired recipient. Finally, the session object calls the appropriate address book provider's **OpenEntry** method to open an interface to the recipient, a pointer to which is passed back through the **OpenEntry** calls to the transport provider.

A provider that has opened several objects from other providers may occasionally need to find out if two entry identifiers refer to the same object. For example, a short-term entry identifier may have to be compared with a long-term entry identifier to avoid redundant processing. In such a case the provider calls the **IMAPISupport::CompareEntryIDs** method, since entry identifiers cannot be compared directly.

Using Support Objects for Event Notification

A service provider can optionally support event notification, which allows other objects to register with the provider to be advised whenever an event of a selected type occurs within the provider. The object subscribing for notification is called the advise sink and must implement the **IMAPIAdviseSink : IUnknown** interface. Event notification can be complicated, so MAPI supplies three methods in the support object that implement the most difficult parts of the process. These methods work as a unit, and a provider must use all three or none of them.

The service provider calls **IMAPISupport::Subscribe** when a client registers for notification by calling the provider's **Advise** method. The provider must allocate a **NOTIFKEY** structure and create a unique notification key for the object that is to generate the event. For example, if a message store provider is asked to notify a client when a message is received in a particular folder, the provider creates a notification key for that folder. A pointer to the **NOTIFKEY** structure is one of the parameters the provider passes to **Subscribe**.

Another parameter to **Subscribe** is a pointer to the client's advise sink object, which was supplied in the call to **Advise**. **Subscribe** calls **AddRef** on the advise sink object, and MAPI retains the pointer until the subscription is terminated. Once **Subscribe** has returned, the provider has no further need to access the client's advise sink and can release its copy of the pointer.

Subscribe returns a nonzero connection number that the service provider in turn returns to the client. The connection number represents the link between the advise sink and the provider and remains valid until the client makes a successful call to **Unadvise**.

When the client is ready to terminate its subscription it calls the provider's **Unadvise** method. The provider then calls **IMAPISupport::Unsubscribe** to cancel the registration indicated by the connection number. **Unsubscribe** calls **Release** on the advise sink object, and MAPI relinquishes the link. The advise sink object itself is freed when all other references to it are released. As with **Advise** and **Unadvise**, calls to **Subscribe** and **Unsubscribe** must be paired. The provider must make one call to **Unsubscribe** for every call that is made to **Subscribe**.

When a subscribed event occurs, the service provider allocates one or more **NOTIFICATION** structures and calls **IMAPISupport::Notify** to ask MAPI to notify all registered advise sinks. Note that a separate **NOTIFICATION** structure is necessary for each subscribed event, even for multiple events of the same type. For example, if three clients are registered for table notification on a particular table and an operation adds five rows to that table, the provider populates five **NOTIFICATION** structures and calls **Notify** once. A batch notification such as this results in better performance than calling **Notify** five times. For each **Notify** call, MAPI calls the **IMAPIAdviseSink::OnNotify** method of every registered advise sink. If there are no registered advise sinks, MAPI ignores the call.

Considerations for Event Notification

The normal process of event notification is asynchronous, that is, the **OnNotify** methods registered for an event are called at some indeterminate time after **Notify** has returned to the calling service provider. Clients always expect such behavior, and in particular MAPI guarantees that the call during which an event happens will return to the client before any notification callback is made. Therefore a provider responding to an **Advise** call from a client must never set the NOTIFY_SYNC flag when it calls **Subscribe**.

For its own internal use a provider can request synchronous notification, where **Notify** does not return until all callbacks have been completed. This is requested by passing the NOTIFY_SYNC flag to the **Subscribe** method. If this option is specified, the provider must not make any changes to the advise sink object after calling **Subscribe**. Also, it must not call the **HrThisThreadAdviseSink** function and pass the wrapper object that it creates. **HrThisThreadAdviseSink** creates a thread-safe version of an advise sink to be used with asynchronous notification only.

If an advise sink registered for synchronous notification returns from **OnNotify** with the CALLBACK_DISCONTINUE flag set, MAPI sets the NOTIFY_CANCELED flag and returns from the **Notify** call without making any more callbacks. The provider should not initiate any more notifications for that advise sink.

Although **Advise** and **Unadvise** calls must be paired, as must **Subscribe** and **Unsubscribe** calls, these pairs do not themselves have to be paired. A provider's implementation of **Advise** and **Unadvise** can handle everything itself and not call MAPI. Similarly, a provider setting up internal notifications can call **Subscribe** and **Unsubscribe** purely on its own behalf without the client being involved. A provider implementing part of the notification process and calling the MAPI methods for the rest does not necessarily call **Subscribe** every time its **Advise** is called.

The MAPI support methods use notification keys to manage the connections between the advise sinks and the objects that generate the subscribed events. A notification key is a **NOTIFKEY** structure of binary data that identifies an object across processes. It is typically copied from the long-term entry identifier of the object that is expected to generate an event. If the client has supplied an entry identifier in the call to **Advise**, the provider can use it for the notification key. If the *lpEntryID* parameter to **Advise** is NULL, the provider should use the entry identifier of the outermost possible container object, such as the entire message store.

When the service provider sends a notification, it is recommended that all the unused members of the **NOTIFICATION** structure be set to zero. This technique for initializing the **NOTIFICATION** structure can help clients create smaller, faster, and less error-prone **OnNotify** implementations.

Implementing a Message Service

A message service is a grouping of one or more related service providers that simplifies provider installation and configuration for users. A message service supplies a layer between your service provider and a user, enabling the user to install and configure your service provider without detailed knowledge of the workings of your provider. It is recommended that all service providers be part of a message service.

To implement a message service

1. Design the message service, determining the number and type of service providers to be included.
2. Create a setup program to install the service providers in the message service.
3. Create an entry point program to support the configuration of the service providers in the message service.
4. Create a public header file containing the property tags and descriptions of valid values for any custom properties that the message service supports.

Designing a Message Service

The following issues are involved in message service design:

1. Determine how many service providers should be included in the message service.
2. Determine what type of service providers should be included in the message service.
3. Determine how many DLLs should contain the message service.
4. Determine how the message service DLL(s) should be named.

The number of service providers that should be included in your message service depends on the relationship between the providers that are possible candidates for the service. If the service providers are related, they should belong to the same message service.

Typically, there is only one provider of each type in a message service. For example, if a message store provider uses the same underlying messaging system as an address book and transport provider, create a message service to install and configure these three related providers. Alternatively, if a message store provider works independently, create a message service for this one provider. Unrelated service providers do not belong in the same message service. Use the profile for integrating unrelated service providers and message services.

Message services can be implemented in one or more DLLs. The number of DLLs that a message service uses depends on:

- The degree of complexity that you as the writer of the message service are willing to handle.
- The type of service providers in the message service.
- The relationship that the message service might have with another message service.

The simplest and recommended implementation uses one DLL. This single DLL contains the code to install and configure all of the service providers in the message service as well as the implementations of the service providers. More complex options include one DLL for a message service's installation and configuration code and another DLL for the implementations of the service providers or one DLL per provider implementation.

Do not include multiple service providers of the same type in a single message service DLL. This limitation is due to the fact that MAPI stores only one entry point per provider type. If it makes sense for your message service to include multiple providers of one type, they must either reside in separate DLLs or share an entry point function.

Two or more related message services can be implemented to share a DLL. Related message services are message services that are able to use the same installation and configuration code and the same DLL entry point function.

Assign a name to your message service that is six characters or less. The six character restriction allows MAPI to concatenate characters onto the end of the filename to indicate a 32-bit target platform. For example, if a message service that uses two DLLs, TEST and PROVS, is built for both a 16-bit platform and a 32-bit platform, MAPI would produce four different message service DLLs: TEST.DLL, PROVS.DLL, TEST32.DLL, and PROVS32.DLL. The DLLs without the suffix on the name are for the 16-bit platforms; the files with the suffix are for the 32-bit platforms.

Installing a Message Service

To allow the Control Panel to install your message service, create a SETUP.EXE program in a designated public directory. When users select the **Add** button and invoke the **Have Disk** dialog box to install your message service, they enter the name of this directory. The Control Panel runs the SETUP.EXE program in the specified directly and calls your message service entry point function with the MSG_SERVICE_INSTALL context.

Your message service's SETUP.EXE program should perform the following tasks:

- Copy message service files, such as the message service and service provider DLLs, from a CD or disk to the local drive. The files that need to be copied depend on your message service. Typically this is at least one DLL.
- Create a default profile if necessary.
- Add entries to the MAPISVC.INF configuration file.
- Add entries as appropriate to the WIN.INI initialization file for 16-bit services or the system registry for 32-bit services. For details about the entries that should appear in the WIN.INI file or the system registry, see About MAPI Installations.

Depending on the target workstation, a default profile might already exist. If a default profile does not yet exist, your installation program can either create one manually or with the NEWPROF utility or invoke one of MAPI's configuration applications, either the Profile Wizard or the Control Panel applet. Both of these applications present a series of dialog boxes that prompt the user for selections that affect the creation and settings of the profile.

Warning Because profiles are an expendable part of the MAPI architecture, make sure that your installation program does not store anything in the default profile that would be difficult to recreate. There are no utilities for profile recovery, for moving profiles from one machine to another, for off-line backup, or for individual or global restoration from backup copies.

About the MAPISVC.INF File

MAPI uses the MAPISVC.INF file to configure the MAPI subsystem, message services and service providers. Profiles are built from the information within the MAPISVC.INF file. When building a new profile or adding to an existing one, information that is required or helpful for configuring each message service and service provider is copied from MAPISVC.INF into the new or changed profile.

MAPISVC.INF acts as the central database for MAPI message service configuration information. The file contains information for MAPI, information for each of the message services installed on a computer, and information for the service providers that belong to each message service. Some of this information is mandatory; without it, MAPI cannot load or configure the message service or service provider in question. Message service and service provider implementors must explicitly add the required information as part of their installation process. For example, MAPI cannot load a service provider without information on its name and path. Therefore, MAPISVC.INF must contain an entry for the name and an entry for the full path of each service provider. Other configuration information is optional; its inclusion in MAPISVC.INF depends on the particular message service or service provider.

MAPISVC.INF is divided into linked hierarchical sections. The top level, divided into three sections called [Services], [Help File Mappings], and [Default Services], contains entries that apply to all profiles. Entries in the [Services] section link to subsequent MAPISVC.INF sections that are specific to individual message services.Entries in the message service sections in turn link to subsequent sections that are specific to individual service providers belonging to the message service.

The following illustration shows the organization of a typical MAPISVC.INF file. The sections are organized hierarchically, with the [Help File Mappings], [Default Services], and [Services] sections at the top of the hierarchy. The [Help File Mappings] section has one entry for every .HLP file provided by an installed message service. The [Default Services] section has one entry for each message service that should be added to the profile designated as the default. These are the message services that are loaded at session startup if the user has not explicitly selected any others.

The [Services] section contains one entry for every message service installed on the computer workstation. In this example, there are three message services: AB, MsgService, and MS. The name on the right hand side of the equal sign for each message service is the service's display name. Each message service has its own section elsewhere in the file that is linked to one or more service provider sections. There is one service provider section for every service provider that belongs to the message service. The AB and MS message services are single provider services whereas three service providers belong to the MsgService service.

{ewc msdncd, EWGraphic, groupx835 1 /a "MAPI_30.WMF"}

All MAPISVC.INF files have a similar organization with [Help File Mappings], [Default Services], and [Services] sections and the corresponding message service and service provider sections. Depending on the particular message services included, there can be more or less service provider sections and possibly some special sections.

Updating MAPISVC.INF

MAPI provides a skeletal version of the MAPISVC.INF file that contains the entries for the MAPI subsystem. Message service implementors add entries appropriate both for their service and the service providers that belong to their service as part of their installation program.

Service providers can add entries within their installation program to either the Message Service section or the Service Provider sections. The placement depends on the number of service providers in your message service. If your message service is a single provider service, you should store all of the entries in the section for the service provider rather than in the Message Service section. Accessing the Service Provider section is faster and more direct than accessing the Message Service section. If your message service is a multiple provider service, you can store entries in the Message Service section or in all of the Service Provider sections so that each service provider has access to the information. To avoid replication and the need to keep multiple copies synchronized, it is better to store the information once in the Message Service section.

A message service should include only public configuration data in the MAPISVC.INF file. Information that is private or requires extra protection, such as passwords or other credentials, should not be included in this file. Store information of this type in the profile as secure properties or don't store it at all. Secure properties have built-in protection features such as encryption.

To add entries to MAPISVC.INF using a temporary file, call the MERGEINI utility as follows where TMP.INI is the temporary file:

```
MERGEINI C:\MAPI\TMP.INI -m -q
```

The [Help File Mappings] Section in MAPISVC.INF

The [Help File Mappings] section contains entries that each map one message service to the file that provides Help for errors generated by the service. Entries in this section use the following format:

```
[Help File Mappings]
message service name=Help file name
```

The message service name is the name of the installed message service; the Help file name is the name of the file where the error information resides. The example following shows a typical [Help File Mappings] section that contains entries for three services: MAPI, the MsgService service, and the MS service.

```
[Help File Mappings]
MAPI=MAPI.HLP
MsgService=MYHELP.HLP
MS=STORE.HLP
```

The [Services] Section in MAPISVC.INF

The [Services] section lists the message services that are installed on a computer. Entries in this section use the following format:

```
[Services]
message-service section name=message service name
```

The message-service section name is a string defined by the message service that links this entry to a corresponding section for the service elsewhere in MAPISVC.INF. The message service name is the name of the installed service. The following section shows three message services: the Default Address Book, My Own Service, and the Message Store Service. These services are fictional, for illustration purposes only. Each message service implementor would substitute the appropriate entry for his or her message service in this section.

```
[Services]
AB=Default Address Book
MsgService=My Own Service
MS=Message Store Service
```

Each entry in this section has a corresponding section of its own where information for the message service is stored. For example, the corresponding section for the Default Address Book is called [AB].

The [Default Services] Section in MAPISVC.INF

The [Default Services] section lists all of the message services that are selected as default message services. These default message services are a subset of the message services listed in the [Services] section. When a profile configuration program creates a default profile, the message services in this section are automatically included.

The entries use the same format as entries in the [Services] section, as shown following:

```
[Default Services]
message-service section name=message service name
```

The following entries would be included in the [Default Services] section for the MAPISVC.INF shown in the earlier illustration:

```
[Default Services]
AB=Default Address Book
MsgService=My Own Service
```

Message Service Sections in MAPISVC.INF

MAPISVC.INF includes one message service section for each of the entries listed in the [Services] section. There are two types of entries in these sections: one for setting certain properties and the other for listing names of sections that are related to the message service being configured.

Property Entries

Entries that set properties use this format:

```
property tag=property value
```

The property tag can be a standard MAPI property tag if the configuration data represents one of the properties predefined by MAPI, or a nonstandard tag if the data does not represent a MAPI property. The nonstandard tag is made by combining the value for a property identifier with a property type. The result is an 8 digit hexadecimal number. The property value can be whatever makes sense for the property tag.

Message service sections can contain a variety of entries depending on the message service being configured. The following MAPI properties are typically included in a message services section in the listed format:

```
PR_DISPLAY_NAME=string
PR_SERVICE_DLL_NAME=name of DLL file
PR_SERVICE_ENTRY_NAME=name of entry point function
PR_SERVICE_SUPPORT_FILES=list of files
PR_SERVICE_DELETE_FILES=list of files
PR_RESOURCE_FLAGS=bitmask
```

The PR_DISPLAY_NAME string is the name of the message service that is shown in the user interface, the name that the user associates with the message service. The display name is an optional entry in MAPISVC.INF. Sometimes the display name will be made up of two parts; a part assigned by the message service and a part assigned by the user. If the user is responsible for assigning one of the parts, this is typically handled with a special dialog box known as a property sheet supplied by the message service under the control of a client application.

The information provided for the PR_SERVICE_DLL_NAME entry is the name of the DLL that contains the message service. The information provided for the PR_SERVICE_ENTRY_NAME entry is the name of the entry point function within that DLL that MAPI calls to configure the message service.

The files listed in the PR_SERVICE_SUPPORT_FILES entry are files that must be installed with the message service. Likewise, the files in the PR_SERVICE_DELETE_FILES entry must be removed when the message service is removed.

The PR_RESOURCE_FLAGS entry is a collection of options defined for the message service. For example, the SERVICE_SINGLE_COPY bit is set when the message service can only appear once in a given profile. The SERVICE_NO_PRIMARY_IDENTITY bit is set if the message service does not provide identity information.

Two examples of nonstandard property entries follow. The first entry specifies the path to the file used by the Default Address Book as the property value; the second entry specifies a numeric property value. Both entries have meaning specific to the AB message service.

```
6600001e=full path to file
66040003=integer
```

Message Service Section List Entries in MAPISVC.INF

There are two types of section list entries: one that lists service provider sections and one that lists miscellaneous message service-specific sections. These two types of entries appear in MAPISVC.INF using the following formats:

```
Providers=provider section1, provider section2, ...... provider sectionX
Sections=section name1, section name2, ......section nameX
```

Each section in the Providers entry maps to an individual section providing configuration information for a service provider that belongs to the message service. Each section in the Sections entry maps to a section that contains extra configuration information needed by the message service. Message service implementors define extra sections when they want to include special information that does not fit in the standard sections. Message services that have complicated configurations typically use the Sections entry to add extra information. Every message services section has a Providers entry with at least one section in the list; not all message service sections have a Sections entry.

Two examples of message service sections follow. The first section is for the Default Address Book service from the earlier illustration, a straightforward message service with a single service provider. The second section is for the MsgService service, a more complex sample message service with three service providers.

```
[AB]
PR_DISPLAY_NAME=Default Address Book
Providers=AB Provider
PR_SERVICE_DLL_NAME=AB.DLL
PR_SERVICE_SUPPORT_FILES=AB.DLL
PR_SERVICE_ENTRY_NAME=DABServiceEntry
PR_RESOURCE_FLAGS=SERVICE_NO_PRIMARY_IDENTITY

[MsgService]
PR_DISPLAY_NAME=My Own Service
Providers=MsgService Prov1, MsgService Prov2, MsgService Prov3
Sections=First_Special_Section, Second_Special_Section
PR_SERVICE_DLL_NAME=MYSERV.DLL
PR_SERVICE_SUPPORT_FILES=MYSERV.DLL, MYXXX.DLL, MYZZZ.DLL
PR_SERVICE_ENTRY_NAME=MyServiceEntry
PR_RESOURCE_FLAGS=SERVICE_SINGLE_COPY
66040003=00000000
```

The Sections entry in the [MsgService] section lists two additional sections, one called [First_Special_Section] and the other called [Second_Special_Section]. The data that might appear in extra sections is meaningful to the specific message service. These sections appear following to illustrate extra sections.

```
[First_Special_Section]
UID=13DB0C8AA05101A9BB000AA002FC45A
66020003=01000000
66000003=00040000
66010003=06000000
66050003=03000000
```

Service Provider Sections in MAPISVC.INF

MAPISVC.INF includes one service provider section for each of the entries listed in the Providers entry in the preceding message services section. Service provider sections are similar to message service sections in that both types of sections contain entries in this format:

```
property tag=property value
```

However, service provider sections and message service sections differ in that such property entries are the only type of entry included in service provider sections. There can be no additional or linked sections for service providers; all service provider information must be contained within the one section.

Some of the properties set in message service sections are also set in service provider sections because these properties make sense for both. The PR_DISPLAY_NAME property is an example. Both service providers and message services have a name that is used for display in the configuration user interface. Depending on the service provider, that name may or may not be the same. Other properties are specific to service providers.

Typical service provider sections include the following entries, all of which are required:

```
PR_DISPLAY_NAME=string
PR_PROVIDER_DISPLAY=string
PR_PROVIDER_DLL_NAME=name of DLL file
PR_RESOURCE_TYPE=long
PR_RESOURCE_FLAGS=bitmask
```

The PR_PROVIDER_DLL_NAME entry is similar to PR_SERVICE_DLL_NAME; it indicates the filename for the DLL that contains the service provider. Message service code may be stored with one of its service providers in the same DLL file or exist as a separate DLL. Note that no suffix is included in the entry regardless of the target platform; MAPI takes care of adding a suffix if necessary.

PR_RESOURCE_TYPE entry represents the type of service provider; service providers set it to the appropriate predefined constant. Valid values include MAPI_STORE_PROVIDER, MAPI_TRANSPORT_PROVIDER, and MAPI_AB_PROVIDER.

Another property entry that applies to both message services and service providers, the PR_RESOURCE_FLAGS entry indicates options. The settings for this property entry can differ depending on the service provider. For example, some message store providers might set PR_RESOURCE_FLAGS to STATUS_NO_DEFAULT_STORE if they can never operate as the default message store.

Three examples of service provider sections follow. The [AB Provider] section is the service provider section for the Default Address Book service. The [MsgService Prov1] and [MsgService Prov2] sections belong to My Own Service; the first is an address-book provider section and the second is a message-store provider section.

```
[AB Provider]
PR_DISPLAY_NAME=Default Address Book
PR_PROVIDER_DISPLAY=Default Address Book
PR_PROVIDER_DLL_NAME=AB.DLL
PR_RESOURCE_TYPE=MAPI_AB_PROVIDER
6600001e=C:\WINNT35\System32\DEFAB.TXT

[MsgService Prov1]
PR_DISPLAY_NAME=My Own Service
PR_PROVIDER_DISPLAY=My Own Address Book
PR_PROVIDER_DLL_NAME=MYXXX.DLL
```

```
PR_RESOURCE_TYPE=MAPI_AB_PROVIDER

[MsgService Prov2]
PR_DISPLAY_NAME=My Folders
PR_PROVIDER_DISPLAY=My Own Message Store
PR_RESOURCE_TYPE=MAPI_STORE_PROVIDER
PR_PROVIDER_DLL_NAME=MYZZZ.DLL
PR_RESOURCE_FLAGS=STATUS_NO_DEFAULT_STORE
66060003=00000000
66030003=00000000
34140102=78b2fa70aff711cd9bc800aa002fc45a
66090003=06000000
660A0003=03000000
```

Configuring a Message Service

Message service writers have a minimum set of tasks that they must perform to support message service configuration. In addition, they can provide support for a configuration application called the Profile Wizard. The Profile Wizard, implemented by MAPI, creates profiles with very little user intervention. Default settings are used whenever possible. Support for this application implies that the message service can be included in any profile that the Profile Wizard creates.

To support generic message service configuration

- Implement an entry point function that conforms to the **MSGSERVICEENTRY** prototype.
- Publish the name of your message service entry point function in the MAPISVC.INF configuration file.
- Create one or more property sheet dialog boxes for displaying configuration data.

To support the Profile Wizard

- Implement an entry point function that conforms to the **WIZARDENTRY** prototype.
- Implement a standard Windows dialog procedure.
- Enhance your message service entry point function to respond to additional events.

About Message Service Entry Point Functions

Message services entry point functions manage access to profile data and respond to configuration requests from clients. Although most message services will provide entry point functions and MAPI strongly recommends that they do, these functions are not strictly required. Message services can provide access to configuration data in other ways. However, using an entry point function standardizes and simplifies the processing of configuration tasks.

If your message service implements an entry point function, publish its name in the message service section of the MAPISVC.INF file as follows:

```
PR_SERVICE_ENTRY_NAME=<name of message service>
```

After MAPI has been initialized, your message service's entry point function will be called:

- When a client logs on to retrieve information for configuring your message service.
- When a client wants to view or change a configuration property.

MAPI expects all message service entry point functions to be able to store and retrieve properties from the profile sections that are associated with their message service. They can support this functionality interactively, programmatically, or both interactively and programmatically.

To support interactive configuration, your service's entry point function provides a property sheet that displays the properties involved in configuring your message service. As an option, your entry point function can also supply property sheets for each configurable provider. Some message services restrict users to a read-only view of configuration properties; other message services allow them to make changes.

To support programmatic configuration, your service's entry point function must be able to reconfigure the service itself or one of its members without user intervention. If your message service can be called by the Profile Wizard, it must support programmatic configuration in its entry point function. If your message service does not support the Profile Wizard, support for programmatic configuration is optional.

Implementing a Message Service Entry Point Function

Every message service entry point function follows the MSGSERVICEENTRY prototype. There are many input parameters and one output parameter, the MAPI error structure for reporting detailed error information. The input parameters include:

- An application handle.
- A pointer to an OLE memory allocator.
- A pointer to a support object.
- A window handle.
- An operation context.
- An array of property values and a count of the number of entries in the array.
- A pointer to a provider administration object.

The most significant of these parameters is the operation context, the property value array, and the provider administration object pointer. A message service entry point function is typically called after MAPI or a client has completed or is in the process of completing an operation. The operation context represents the particular operation.

Your message service's response to a particular operation context is similar to a response to event notification. With some operations, your entry point function will do little else except return S_OK. With other operations, there can be extensive processing required. There are seven possibilities for the operation context:

MSG_SERVICE_INSTALL
MSG_SERVICE_UNINSTALL
MSG_SERVICE_CREATE
MSG_SERVICE_CONFIGURE
MSG_SERVICE_DELETE
MSG_SERVICE_PROVIDER_CREATE
MSG_SERVICE_PROVIDER_DELETE

The MSG_SERVICE_INSTALL and MSG_SERVICE_UNINSTALL operations usually require very little work. The operation context will be set to MSG_SERVICE_INSTALL after the caller, typically the Control Panel or another configuration client, has run your setup program to install your message service. With the Control Panel, this occurs when a user selects the **Have Disk** dialog box through the **Add** button. The entry point function can either return immediately with S_OK or can perform post-installation processing if necessary.

When the operation context is set to MSG_SERVICE_UNINSTALL, typically the user of a client application has selected an option on a dialog box to remove your message service from the workstation. By the time the entry point function is called, either the user has finished removing your service, in which case any message service files or data can be deleted, or has canceled the removal. To determine whether or not the removal was successful, your provider must query the user. If the removal was canceled, another query is necessary to make sure that the user intentionally chose to cancel the operation. Your message service entry point function should return MAPI_E_USER_CANCEL for canceled service removals and redisplay the message service dialog box if the cancel was unintentional, giving the user another chance.

Your message service can also decide to cancel the uninstall operation by returning MAPI_E_USER_CANCEL regardless of whether the user has cancelled. Before deciding to cancel, your message service can:

- Query the user to ask if the message service should be removed.
- Check for related services before deciding if the message service should be removed.

- Decide to cancel the removal, explaining the reasons to the user and returning MAPI_E_USER_CANCEL unconditionally.

The MSG_SERVICE_CREATE, MSG_SERVICE_CONFIGURE, and MSG_SERVICE_DELETE operations apply to a profile that contains your message service. Your message service can identify the profile that is working with by examining one of the following two profile section properties:

PR_PROFILE_NAME
PR_SEARCH_KEY

In most situations, checking the profile name will be sufficient. The search key can be used for profile identification in those few situations where the name may not be unique enough, possibly because the profile has been deleted and a new one has been created with the same name. A profile section's search key is an arbitrarily sized binary identifier that is defined in the MAPIGUID.H header file as MUID_PROFILE_INSTANCE.

With the MSG_SERVICE_CREATE context, your message service is being added to a profile. With MSG_SERVICE_CONFIGURE, your message service is being configured; one or more of the entries in the profile might change. The entry point function's handling of the create and configure operations will depend on the settings of the flags parameter, *ulFlags*. The SERVICE_UI_ALWAYS flag must be set on a create operation because prompting the user through a dialog box is always necessary. If this flag is not set when the operation context is MSG_SERVICE_CREATE, the entry point function should fail.

The handling of the MSG_SERVICE_CONFIGURE operation is more complicated.

Perform programmatic configuration without a user interface if one of the following conditions is true:

- The property value array parameter contains a complete set of configuration properties.
- Neither the SERVICE_UI_ALWAYS nor SERVICE_UI_ALLOWED flags are set.

Perform interactive configuration with the aid of a user interface if one of the following conditions is true:

- The SERVICE_UI_ALLOWED flag is set and the property value array is empty.
- The SERVICE_UI_ALLOWED flag is set and the property value array does not contain a complete set of configuration properties.
- The entry point function does not support programmatic configuration.

The operation context is set to MSG_SERVICE_DELETE when your message service is removed from a profile. Deletion occurs on a per-instance basis meaning that other instances of your message service either in the same profile or in a different profile are unaffected. Your service can still be added to profiles. The service entry point function can handle this operation by returning immediately with a success code.

Note   There is not a special context value signifying a copy operation because when your message service is copied from one profile to another, your entry point function is not called. The configuration settings for the new service instance take on the values from the original service.

The MSG_SERVICE_PROVIDER_CREATE and MSG_SERVICE_PROVIDER_DELETE operations apply to a service provider that either will belong or already belongs your message service. When the context is set to MSG_SERVICE_PROVIDER_CREATE, the caller is attempting to add a service provider to your service. When the context is MSG_SERVICE_PROVIDER_DELETE, the caller is trying to delete a provider. Supporting these operations is optional; it is acceptable to handle them by returning MAPI_E_NO_SUPPORT. However, for services that do allow the dynamic addition or deletion of service providers, MAPI supplies a pointer to an **IProviderAdmin** interface implementation. The entry point function can handle the MSG_SERVICE_PROVIDER_CREATE operation with a call to **IProviderAdmin::CreateProvider** and the MSG_SERVICE_PROVIDER_DELETE operation with a call to **IProviderAdmin::DeleteProvider**.

Warning for 16-bit Message Services Be careful of stack consumption in your message service entry point function and methods or functions that it calls. Because this code can be called from the Control Panel applet, which has limited stack space, problems can occur if your entry point function's stack usage is too excessive.

Supporting the Profile Wizard

The Profile Wizard displays a series of dialog boxes for creating a profile. Each message service included in the profile is configured, using either default values or values entered by a user working with a property sheet provided by the message service. To enable your message service to be included in a Profile Wizard profile, your service must:

- Provide a standard entry point function that the Profile Wizard calls to invoke your service.
- Provide a standard Windows dialog procedure that will handle event processing.

About Profile Wizard Entry Point Functions

Profile Wizard entry point functions are written using the WIZARDENTRY prototype, defined as follows:

```
ULONG (STDAPICALLTYPE WIZARDENTRY)
    (HINSTANCE hProviderDLLInstance, LPTSTR FAR *lppcsResourceName,
    DLGPROC FAR *lpDlgProc, LPMAPIPROP lpMAPIProp,
    LPVOID lpMapiSupportObject);
```

The Profile Wizard passes in a handle to the message service DLL and a pointer to an **IMAPIProp** interface implementation. Access to the configuration properties for your message service is handled through this **IMAPIProp** implementation. Your message service should keep a reference to this implementation because since the Profile Wizard works with the most basic set of properties, your message service can use it to add extra properties. When the Profile Wizard has finished working with your message service, it calls **IMsgServiceAdmin::ConfigureMsgService** to inform your service.

The Profile Wizard entry point function manages the display of one or more of your message service's property sheets. Each property sheet can contain zero or more property pages. The Profile Wizard displays your service's property sheets within its property pages; the controls of your property sheets are created as children of the Profile Wizard's page. When the entry point function is called, your message service should reveal only the controls for the first page in the first property sheet. As the Profile Wizard moves from page to page, the function must hide all of the controls for the old page and expose the controls for the new one.

About **Profile Wizard Dialog Procedures**

Profile Wizard dialog procedures are standard Windows procedures that handle Window messages generated by the Profile Wizard during the display of your message service's property sheets.

The Profile Wizard creates a dialog frame with three buttons (**Back**, **Next** or **Finish**, and **Cancel**) and a fixed size area. It is into this area that your message service displays its property sheets. Any Windows messages or events that occur within this fixed area go directly to the Profile Wizard dialog procedure.

Because users of the Profile Wizard can move in a forward or backward direction through the pages of a service's property sheets, generating a WM_INITDIALOG message repeatedly, message services should keep the property sheets readily available.

The Profile Wizard is a single instance application. Therefore, your dialog procedure can be written as a single instance procedure and can take advantage of static variables for storing data.

Implementing Property Sheets

A property sheet is a dialog box for displaying the properties of an object. The properties can be read-only, enabling the user only to view them, or read/write, enabling the user to make changes. A property sheet contains one or more overlapping child windows called pages, each page contains control windows for setting a group of related properties. Users navigate from page to page using a tab that brings the selected page to the foreground of the property sheet.

Service providers are required to implement a property sheet that displays a minimal set of message service properties. Implementing property sheets for service provider properties is optional. Providers that allow these properties to be changed can either allow users of client applications such as the Control Panel applet to make the changes or implement the changes programmatically.

Service providers can create a property sheet using one of the following three techniques:

- Manually, as they would any dialog box.
- Using the property sheet common control provided in the Win32 SDK.
- Using a MAPI display table.

MAPI recommends that service providers use a display table for property sheet implementations. Creating a property sheet with a MAPI display table is simpler because it eliminates the need to work with the Windows user interface.

Service providers can call the **IMAPISupport::DoConfigPropSheet** method to display a property sheet based on a display table. Typically, a provider will need to display a property sheet:

- When a client calls the provider's **IMAPIStatus::SettingsDialog** method.
- When MAPI calls the provider object's logon method.
- When MAPI calls the provider's message service's entry point function.

To implement a property sheet

1. Call **IMAPISupport::OpenProfileSection** to open a section in the current profile. Pass a provider's **MAPIUID** or NULL to open your provider's section.
2. Call **CreateIProp** to create a property data object.
3. Call the profile section's **IMAPIProp::CopyTo** method to copy all of the section's properties to the property data object.
4. Create one or more **DTPAGE** structures that describe the controls for the property sheet.
5. Call **BuildDisplayTable** to create a display table.
6. Call **IMAPISupport::DoConfigPropSheet** to display a property sheet with the copied properties. Pass a pointer to the property data object as the *lpConfigData* parameter and a pointer to the display table as the *lpDisplayTable* parameter.
7. When all of the changes have been made in the property sheet, call the property data object's **IMAPIProp::CopyTo** method to copy the changed properties back to the profile section.

Displaying a Progress Indicator

Many of the operations that your service provider performs for clients can take a long time to complete. To inform clients of the progress of a lengthy operation, your service provider can show an indicator that displays graphically the finished portion of an operation at any given point from the start of the operation to its completion. Typically the progress indicator looks something like the following illustration, with each tic mark representing a percentage of the total operation to be completed.

{ewc msdncd, EWGraphic, groupx835 2 /a "MAPI.BMP"}

The following methods support lengthy operations and the display of a progress indicator:

- The **CopyMessages**, **CopyFolder**, **DeleteMessages**, **DeleteFolder**, **EmptyFolder**, and **SetReadFlags** methods in **IMAPIFolder**
- The **CopyProps** and **CopyTo** methods in **IMAPIProp**
- The **DoCopyProps**, **DoCopyTo**, **CopyFolder**, and **CopyMessages** methods in **IMAPISupport**
- **IMessage::DeleteAttach**
- **IABContainer::CopyEntries**

To display a progress indicator, MAPI defines a progress object. Progress objects implement the **IMAPIProgress** interface, an interface which includes methods for establishing the range of the indicator and creating the display. MAPI provides a progress object implementation as do some clients. Your service provider should use a client's implementation if one is supplied as an input parameter in the method performing the operation. If the client passes NULL instead of a pointer to a progress object, your provider should use MAPI's implementation. Service providers use the MAPI progress object by calling their support object's **IMAPISupport::DoProgressDialog** method.

Using Progress Objects

With the methods and data of a progress object, your provider can control how the indicator reports progress. Although a client or MAPI implements the progress object, most of the burden of insuring the correctness of the progress display falls on service providers. Service providers guarantee accuracy by specifying a particular order and value for the parameters that they pass to progress object methods.

Service providers pass the following parameters to progress objects:

- A bitmask of flags, set with **SetLimits** and retrieved with **GetFlags**
- A minimum value (local and global), set with **SetLimits** and retrieved with **GetMin**
- A maximum value (local and global), set with **SetLimits** and retrieved with **GetMax**
- A value that indicates the current percentage of completion of the operation, passed to **Progress**
- A count of the number of objects that have so far been processed, passed to **Progress**
- A count of the total number of objects involved in the operation, passed to **Progress**

All service providers begin their progress display processing with a call to **IMAPIProgress::GetFlags** to retrieve the present flags setting. Currently the flags can be set only to MAPI_TOP_LEVEL. Clients and MAPI initialize the flag to MAPI_TOP_LEVEL, relying on service providers to clear it when appropriate.

The flags value is set to MAPI_TOP_LEVEL while your provider is working with the top level object in the operation. The top level object is the object that is called by the client to begin an operation. In a folder copy operation, this is the folder being copied. In a folder delete operation, this is the folder being deleted. When your provider makes a call to process a lower level object, or subobject, it clears the flags value. In a folder copy operation, subobjects are the subfolders that are in the folder being copied.

MAPI allows service providers to differentiate between top level objects and subobjects with the MAPI_TOP_LEVEL flag so that all objects involved in an operation can use the same **IMAPIProgress** implementation to show progress, thereby causing the indicator display to proceed smoothly in a single positive direction. Whether or not the MAPI_TOP_LEVEL flag is set determines how service providers set the other parameters in subsequent calls to the progress object.

Because it can be nontrivial to set appropriate parameter values for a progress display at all levels of a multi-level operation, some service providers elect not to show progress for subobjects. To avoid showing progress for subobjects:

- Pass NULL for the *lpProgress* parameter in the call to process a subobject. For example, if copying folders, this is the call to a subfolder's **IMAPIFolder::CopyFolder** method.
- Write special code to determine how to interpret the *lpProgress* parameter. Because a NULL value for the *lpProgress* parameter can also mean that the client should display progress using MAPI's implementation, special code is necessary to determine when to ignore the *lpProgress* parameter and when to call **IMAPISupport::DoProgressDialog**.

Service providers call **IMAPIProgress::SetLimits** to set or clear the MAPI_TOP_LEVEL flag and to set local and global minimum and maximum values. The value of the flags setting affects whether the progress object understands the minimum and maximum values to be local or global. When the MAPI_TOP_LEVEL flag is set, these values are considered global and are used to calculate progress for the entire operation. Progress objects initialize the global minimum value to 1 and the global maximum value to 1000.

When MAPI_TOP_LEVEL is not set, the minimum and maximum values are considered local and are used internally by providers to display progress for lower level subobjects. Progress objects save the local minimum and maximum values only so that they can be returned to providers when **GetMin** and **GetMax** are called.

The other three parameters are input to the **IMAPIProgress::Progress** method. The first value, a number that indicates percentage of progress, is required. If the MAPI_TOP_LEVEL flag is set, your provider can also pass an object count and an object total. Some clients use these values to display a phrase such as "5 items completed out of 10" with the progress indicator. Progress on an operation can be reported strictly as a percentage or as a percentage and in terms of the number of items that have been processed out of the total to be processed. For example, if your provider is a message store that is copying 10 folders, the progress indicator can supply the user with additional information by displaying a phrase such as 1 of 10, 2 of 10, 3 of 10, and so on until the operation is complete.

Displaying Progress Step by Step

To display a progress indicator, call **IMAPIProgress::GetFlags** to retrieve the current flags setting.

If the MAPI_TOP_LEVEL flag is set

1. Set a variable equal to the total number of items to process in the operation. For example, if your provider is copying the contents of a folder, this value will be equal to the number of the subfolders in the folder plus the number of messages.
2. Set a variable equal to 1000 divided by the number of items.
3. If your provider is showing progress for subobjects, call the progress object's **IMAPIProgress::SetLimits** and pass the following values for the three parameters:
   - Set the *lpulMin* parameter to 0.
   - Set the *lpulMax* parameter to 1000.
   - Set the *lpulFlags* parameter to MAPI_TOP_LEVEL.
4. For each object to be processed:
   a. Call **IMAPIProgress::SetLimits** and pass the following values for the three parameters:
      - Set the *lpulMin* parameter to the variable set in step 2 multiplied by the current item - 1.
      - Set the *lpulMax* parameter to the variable set in step 2 multiplied by the current object.
      - Set the *lpulFlags* parameter to 0.

   b. Perform whatever processing should be done on this object. If this is a subobject and your provider wants to display progress on subobjects, pass a pointer to the progress object in the *lpProgress* parameter to the method.

   c. Call **IMAPIProgress::Progress** and pass the following values for the three parameters:
      - Set the *ulValue* parameter to variable set in step 2 multiplied by the current object.
      - Set the *ulCount* parameter to the current object.
      - Set the *ulTotal* parameter to the variable set in step 1, the total number of objects.

If the MAPI_TOP_LEVEL flag is not set

1. Call the progress object's **IMAPIProgress::GetMin** method to retrieve the minimum value for the display.
2. Call **IMAPIProgress::GetMax** to retrieve the maximum value for the display.
3. Set a variable equal to the total number of objects to be processed.
4. Set a variable equal to the result of subtracting the minimum value from the maximum value and then dividing by the total number of objects.
5. For each object to be processed:
   a. If your provider is showing progress for subobjects, call **IMAPIProgress::SetLimits** and pass the following values for the three parameters:
      - Set the *lpulMin* parameter to the minimum value plus the current item - 1 multiplied by the variable set in step 4.
      - Set the *lpulMax* parameter to the minimum value plus the current unit multiplied by the variable set in step 4.
      - Set the *lpulFlags* parameter to 0.

b. Perform whatever processing should be done on this object. If the object is a subobject, and your provider displays progress for subobjects, pass a pointer to the progress object in the *lpProgress* parameter to the method.
c. Call **IMAPIProgress::Progress** and pass the following values for the three parameters:

- Set the *ulValue* parameter to variable set in step 2 multiplied by the current object.
- Set the *ulCount* parameter to 0.
- Set the *ulTotal* parameter to 0.

The following code sample illustrates the logic required to show progress at all levels of an operation that copies the contents of a folder containing five subfolders.

```
lpProgress->GetFlags (lpulFlags);
ulFlags = *lpulFlags;

/* Folder in charge of the display. It contains 5 subfolders. */
if (ulFlags & MAPI_TOP_LEVEL)
{
    ulItems = 5                         // 5 subfolders in this folder
    ulScale = (ulMax / ulItems)         // 200 because ulMax = 1000
    lpProgress->SetLimits(0, ulMax, MAPI_TOP_LEVEL)

    for (i = 1; i <= ulItems; i++)      // for each subfolder to copy
    {
        lpProgress->SetLimits( (i - 1) * ulScale, i * ulScale, 0)
        CopyOneFolder(lpFolder(i), lpProgress)
        lpProgress->Progress( i * ulScale, i, ulItems)
    }
}
else
/* One of the subfolders to be copied. It contains 3 messages */
{
    lpProgress->GetMin(&ulMin);
    lpProgress->GetMax(&ulMax);
    ulItems = 3;
    ulDelta = (ulMax - ulMin) / ulItems;
    for (i = 1; i <= ulItems; i++)
    {
        lpProgress->SetLimits(ulMin + (i - 1) * ulDelta,
                              ulMin + i * ulDelta, 0)
        CopyOneFolder(lpFolder(i), lpProgress)

        /* Pass 0 for ulCount and ulTotal because this is not the */
        /* top level display and that information is unavailable  */
        lpProgress->Progress( i * ulDelta, 0, 0)
    }
}
```

Developing an Address Book Provider

An address book provider supplies recipient information to client applications, to message store and transport providers, and to MAPI. Recipient information is organized hierarchically into storage compartments known as containers. Every address book in the profile contributes one or more top level, or parent, containers to the MAPI address book, an integrated view of recipient information from all address book providers in a session. It is through the MAPI address book that clients and other service providers gain access to the data of an address book provider.

MAPI builds the integrated address book by:

1. Retrieving the top-level containers from each address book provider.
2. Retrieving each container's hierarchy table.
3. Copying each hierarchy table into an integrated hierarchy table. It is the integrated hierarchy table that is exposed to the client.

MAPI imposes few requirements on address book provider writers. The range of possible features you can implement as an address book writer is varied and flexible. For example, your provider could be limited to supplying a read-only view of a particular type of recipient information or implement a full set of features, perhaps allowing clients or providers to make additions or modifications to the recipient data and to impose search criteria for defining customized views.

Your provider's data can reside locally in a file or database or on a remote server. Some address book providers are meant to work with a particular messaging system, tightly coupled with a transport provider, while others can operate with any messaging system.

MAPI defines a special type of address book provider called a personal address book, or PAB, that implements a single modifiable container and can hold recipient information copied from other containers as well as information created directly. Although any address book provider can implement a PAB and multiple PABs can be added to a profile, only one of these providers can be designated to operate as the PAB during any one session.

Features for Address Book Providers

Address book providers can work with recipient information that is temporary or permanent, local or remote, understandable by one or more messaging systems, and formatted for a disk file or database table. There are a variety of features that an address book provider can implement, thereby adding value and improving interoperability with clients and other providers. However, few features are required. The only features required of all address book providers are:

- Support for logon and logoff
- Ability to create entry identifiers
- Provide status information to the status table
- Limited status object support
- Support for interactive and programmatic configuration

In addition to these few features, there are many other features that can or must be implemented, depending on your provider's characteristics. There are two additional sets of features that are required of some address book providers: one set for providers that have containers and another set for providers that allow recipients to be added to their containers or to a Personal Address Book (PAB).

The following table describes many of the common features that address book providers support and the objects, properties, or interface methods that are implemented to supply that support.

Feature	Requirement	What to implement
Logon support	Required of all providers	**IABProvider::Logon** method
Logoff support	Required of all providers	**IABProvider::Shutdown** method
Entry identifier creation	Required of all providers	PR_ENTRYID property for messaging user objects, distribution lists, and containers
Status table support	Required of all providers	**IMAPIStatus** interface Required properties for status table Call to **IMAPISupport::ModifyStatusRow**
Limited status object support	Required of all providers	**IMAPIStatus::ValidateState** fully Return MAPI_E_NO_SUPPORT from other **IMAPIStatus** methods
Configuration support	Required of all providers	Message service entry point function Call to **IMAPISupport::DoConfigPropsheet**
Access to objects	Required of providers with containers	**IABLogon::OpenEntry** method
Comparision of objects	Required of providers with containers	**IABLogon::CompareEntryIDs** method
Summary information about recipients	Required of providers with containers	PR_CONTAINER_CONTENTS property for containers
Hierarchical list of containers	Required of providers with containers	PR_CONTAINER_HIERARCHY

		property for containers
List of available templates for recipient creation	Required of providers that support recipient creation	**IABLogon::GetOneOffTable** PR_CREATE_TEMPLATES property for container
View of detailed recipient information	Required of providers with containers	PR_DETAILS_TABLE property for messaging users and distribution lists
View of detailed container information	Required of providers with containers	PR_DETAILS_TABLE property for containers
Grouping recipients into named unit	Optional	**IDistList** interface
Support for individual recipients	Required of providers with containers	**IMailUser** interface
Support for binding code to data in a host address book provider	Optional	PR_TEMPLATEID property for messaging users and distribution lists **IABLogon::OpenTemplateID** method
Prefix scrolling	Optional	Restrictions on container contents tables
Support for advanced searching in a container	Optional	PR_SEARCH property for containers
Name resolution	Required of providers with containers	PR_ANR property restriction

Implementing an Address Book Provider's DLL Entry Point Function

When a client application calls **MAPILogonEx** to begin a session using a profile that contains your address book provider, MAPI loads your provider and all others that are part of the profile. MAPI learns of the name of your provider's DLL entry point function by looking in the profile. There are several entries, some of which must appear in the MAPISVC.INF configuration file, that are included in the profile section of every address book provider. The following table lists these profile section entries and whether or not the MAPISVC.INF file must include them.

Profile section entry	**MAPISVC.INF requirement**
PR_DISPLAY_NAME=*string*	Optional
PR_PROVIDER_DISPLAY=*string*	Required
PR_PROVIDER_DLL_NAME=*DLL filename*	Required
PR_RESOURCE_TYPE=*long*	Required
PR_RESOURCE_FLAGS=*bitmask*	Optional

Your address book provider can place this information into a profile directly by calling its profile section's **IMAPIProp::SetProps** method or indirectly by modifying MAPISVC.INF. Profiles are built using the relevant information in MAPISVC.INF for the selected service providers or message services. For more information about the organization and contents of MAPISVC.INF, see About the MAPISVC.INF File.

The name of your address book provider's DLL entry point function must be **ABProviderInit** and it must conform to the **ABProviderInit** prototype. Perform the following tasks in your provider's DLL entry point function:

- Check the version of the service provider interface (SPI) to make sure MAPI is using a version that is compatible with the version that your address book provider is using.
- Instantiate an address book provider object.

Do not call either **MAPIInitialize** or **MAPIUninitialize** in this function.

The DLL entry point function instantiates a provider object and returns to MAPI a pointer to that object.

Implementing the IABProvider Interface

All address book providers must support a provider object, an object that implements the **IABProvider** interface. The **IABProvider** interface inherits directly from **IUnknown** and adds only two other methods: **Logon** and **Shutdown**. MAPI will call your provider's **IABProvider::Logon** method at the beginning of every session and whenever your provider is added to the current profile and the client supports dynamic reconfiguration. **Shutdown** is called when the session is ending.

Implementing IABProvider::Logon

When MAPI calls the **IABProvider::Logon** method, your address book provider begins its logon process.

To implement IABProvider::Logon

1. Initialize all of the output parameter pointers passed in by MAPI.
2. Call the support object's **IUnknown::AddRef** method to increment its reference count.
3. Call the support object's **IMAPISupport::OpenProfileSection** to open the section of the profile that contains configuration information about your provider. Pass NULL for the *lpUID* parameter and the MAPI_MODIFY flag if you intend to make changes.
4. Call the profile section's **IMAPIProp::GetProps** method to retrieve the properties that your provider needs for logon, such as the name of the data file or database table.
5. Check that the properties are all available and valid. If necessary and allowed, display a dialog box to prompt the user to make corrections or additions to invalid or missing information and call the profile section's **IMAPIProp::SetProps** to save any changes. Some of the common properties that should be available include:

   PR_DISPLAY_NAME
   PR_ENTRYID
   PR_PROVIDER_DISPLAY
   PR_RECORD_KEY

   Note Do not set PR_RESOURCE_FLAGS or PR_PROVIDER_DLL_NAME. At logon time, these properties are read-only.

6. If one or more configuration properties are unavailable, fail and return the value MAPI_E_UNCONFIGURED.
7. Call **IMAPISupport::SetProviderUID** to register a **MAPIUID**. Your provider can create a **MAPIUID** by:
   - Calling the **IMAPISupport::NewUID** method
   - Calling the UUIDGEN.EXE tool to define a GUID that your provider uses to include in one of its header files.
8. If desired, save a newly created **MAPIUID** in the current profile by calling the profile section's **IMAPIProp::SetProps** method.
9. Release the profile section by calling its **IUnknown::Release** method.
10. Instantiate a new logon object and set the contents of the *lppABLogon* parameter to the address of this new object.

Because it is possible for MAPI to call your **Logon** method several times during a session, it is wise to support this possibility in your implementation by being able to create multiple logon objects and keep track of the each object that is created. Supporting multiple **Logon** calls enables a user of a client application, for example, to log onto a session with different identities or use different delivery destinations.

Implementing IABProvider::Shutdown

MAPI calls your **IABProvider::Shutdown** method as one of the last tasks involved in shutting down a session. MAPI has released all of your provider's logon objects and, when your provider receives this call, it can assume that this is the last call it will receive. In your implementation of **IABProvider::Shutdown**, perform any final clean up that you feel is necessary. For example, your provider might call **MAPIDeinitIdle** if it has called **MAPIInitIdle** to use the idle utility during the session or the **IUnknown::Release** method of any objects that have yet to be released.

If your provider has no final clean up, its implementation can be made up of a single line of code:

```
return ResultFromScode(S_OK);
```

Implementing the IABLogon Interface

All address book providers support a logon object, an object that implements the **IABLogon** interface. The **IABLogon** interface is used to service requests from clients that MAPI receives as calls to the methods of the **IAddrBook** interface. Clients call the **IAddrBook** methods; MAPI calls the corresponding methods in your provider's **IABLogon** implementation.

Although the implementation of a logon object is required, your provider is not required to fully support all of the methods. That is, it is acceptable to return MAPI_E_NO_SUPPORT from method implementations that are not fully supported. For example, because support for template identifiers is optional, providers not supporting these identifiers return the MAPI_E_NO_SUPPORT value in their **IABLogon::OpenTemplateID** method.

Implementing IABLogon::OpenEntry

MAPI calls your provider's **IABLogon::OpenEntry** method when a client or provider has requested that one of your objects be opened. MAPI determines that the entry identifier representing the target object belongs to your provider by examining the **MAPIUID** portion of the entry identifier and matching it to the **MAPIUID** that your provider registered in the call to **IMAPISupport::SetProviderUID**. MAPI then calls your **OpenEntry** method. Your provider must respond by retrieving the corresponding object, a container, distribution list, or messaging user.

A NULL entry identifier indicates a request to open the address book provider's root container. Clients open the root container to access its hierarchy table and its recipients. Address book providers that only supply templates for creating one-off recipients do not support the **OpenEntry** call for the root container.

To implement IABLogon::OpenEntry

1. Check that the entry identifier is a valid identifier that your provider supports. If it is not a valid entry identifier, return MAPI_E_INVALID_ENTRYID.
2. Check the flag that is passed in with the *ulFlags* parameter. If MAPI has passed in MAPI_MODIFY and your provider does not allow its objects to be modified, fail and return the E_ACCESSDENIED error value.
3. Check that the interface requested in the *lpInterface* parameter is valid for the type of object your provider has been asked to open. If an invalid parameter has been passed in, fail and return the E_NOINTERFACE error value.
4. If the *cbEntryID* parameter is zero, this is a request to open your provider's root container. Create the root container and return a pointer to its **IABContainer** interface implementation.
5. If your provider implements several logon objects, each with its own registered **MAPIUID**, map the **MAPIUID** contained in the entry identifier with the appropriate logon object.
6. Determine which type of object the entry identifier represents: a messaging user, distribution list, or container belonging to your provider or a one-off messaging user or distribution list so that the appropriate value can be set for the *lpulObjectType* parameter.
7. Create the object of the appropriate type and set the following basic properties:

   PR_DISPLAY_TYPE

   PR_ENTRYID

   PR_OBJECT_TYPE

   PR_ADDRTYPE

   Calculate PR_EMAIL_ADDRESS and PR_DISPLAY_NAME from information in the entry identifier.
8. Return a pointer to the interface implementation for the object.

Implementing IABLogon::CompareEntryIDs

Your provider's **IABLogon::CompareEntryIDs** implementation compares the entry identifiers for two of your provider's objects. MAPI calls this method after determining that the two entry identifiers contain your provider's registered **MAPIUID**. Therefore, your **CompareEntryIDs** method need not check that the entry identifiers passed in for the *lpEntryID1* and *lpEntryID2* parameters belong to your provider.

Calling **IABLogon::CompareEntryIDs** is equivalent to retrieving the PR_RECORD_KEY for each of the two objects and comparing them directly.

To implement CompareEntryIds

1. Check the type of the entry identifiers passed in if your provider stores that information. For example, one entry identifier might belong to a messaging user while the other might belong to a distribution list. If the types do not match, set the contents of the *lpulResult* parameter to FALSE and return.
2. Compare the sizes of the two entry identifiers. If they are not the same, set the contents of the *lpulResult* parameter to FALSE and return.
3. Check that the size of the entry identifiers is the correct size for their type. If not, set the contents of the *lpulResult* parameter to FALSE and return the error value MAPI_E_UNKNOWN_ENTRYID.
4. Check if the entry identifiers are the same. If they compare equally, set the contents of the *lpulResult* parameter to TRUE and return. Otherwise, set it to FALSE before returning.
5. If your provider is comparing a short-term entry identifier with a long-term identifier, they should compare equally.

Implementing IABLogon::OpenStatusEntry

The **IABLogon::OpenStatusEntry** method is called to grant a client access to your provider's status object. The client calls **IMAPISession::GetStatusTable** to locate your provider's row in the status table. Once located, the client calls **IMAPISession::OpenEntry** to open the status object associated with the row. MAPI fulfills the open request by calling your provider's **IABLogon::OpenStatusEntry** method, causing your provider to open its status object and return to the client a pointer to its **IMAPIStatus** implementation.

To implement OpenStatusEntry

1. If your logon object has not yet created a status object:
   a. Call the support object's **IMAPISupport::OpenProfileSection** to access your provider's profile section.
   b. Create a new status object.
   c. Store a reference to the profile section in your provider's status object and call the profile section's **IUnknown::AddRef** to increment its reference count.
   d. Store a reference to the logon object in your provider's status object and call the logon object's **IUnknown::AddRef** to increment its reference count.
   e. Store a reference to the status object in your provider's logon object.
2. Call the status object's **IUnknown::AddRef** method to increment its reference count in the logon object.
3. Set the status object's PR_OBJECT_TYPE property to MAPI_STATUS.
4. Set the *lppMAPIStatus* output parameter to point to the status object and return.
5. Check the *ulFlags* input parameter. If it is set to MAPI_MODIFY and your provider supports read/write access to its status object, return a writeable object. However, if your provider does not support read/write access to its status object, do not fail. Return a status object that is read-only.

Implementing IABLogon::Advise and IABLogon::Unadvise

Your provider's **IABLogon::Advise** method is called by MAPI when a client calls **IAddrBook::Advise** to register for notifications on any one of your provider's containers, messaging users, or distribution lists. Because contents table notifications are the most important and most frequently supported form of address book notifications, supporting object and error notification is optional. However, address book providers are encouraged to support all types of object notification except for *fnevSearchComplete* as well as the *fnevCriticalError* event to add value.

Your **Advise** implementation can keep track of registrations itself or call **IMAPISupport::Subscribe** to take advantage of the MAPI implementation. **Subscribe** will generate a number to represent the connection between the client's advise sink and your provider. When the client no longer wants to receive the notification, it passes this connection number to your provider's **IABLogon::Unadvise** method. Again, your provider can implement the cancelation itself or call **IMAPISupport::Unsubscribe**.

For more information about notification in general, see About Notification.

Implementing IABLogon::PrepareRecips

A client calls MAPI's **IAddrBook::PrepareRecips** method to modify or rearrange a set of properties for one or more recipients. The recipients may or may not be part of the recipient list of an outgoing message. MAPI transfers this call to your address book provider's **IABLogon::PrepareRecips** method.

PrepareRecips has two important input parameters: a property tag array and an **ADRLIST** structure. The **ADRLIST** structure contains one **ADRENTRY** structure member for every recipient. Within the **ADRENTRY** structure, there is a property value array that specifies all of the properties for the recipient.

PrepareRecips performs four main tasks:

- Ensures that all recipients have a long-term entry identifier.
- Ensures that all recipients have the properties specified in the property tag array passed in by the client.
- Ensures that the properties from the property tag array appear before any other properties that existed prior to the call.
- Ensures that the order of the properties in each recipient's **ADRENTRY** structure in the **ADRLIST** is the same as in the property tag array.

To implement IABLogon::PrepareRecips

1. Check if there are entries in the *lpPropTagArray* parameter. If the property tag array is empty, there is no work to do. Return with a success code.
2. Process each recipient in the *lpRecipList* parameter. There is one **ADRENTRY** structure member for each recipient in the list. Ignore the following types of recipients:
   - Recipients without an entry identifier in the *rgPropVals* member of their **ADRENTRY** structure, that is, unresolved recipients.
   - Recipients with an entry identifier that does not belong to your provider. These recipients will be passed on to another address book provider.
3. Open the recipient and retrieve the properties already set for the recipient.
4. Merge the property value array specified in the *lpPropTagArray* with the array of properties returned from **GetProps**. If the same property exists in both property arrays, use the value from *lpPropTagArray*.
5. If the *lpPropTagArray* property value array is big enough to hold all of the necessary properties, just replace it with the merged array. If the *lpPropTagArray* property value array is not big enough, replace it with a newly allocated array. Make sure the new array has an updated value in each of its **cValues** members.

Note Never reallocate the **ADRLIST** structure that is passed into **PrepareRecips** or change its number of entries.

Implementing a Foreign Address Book Provider

A foreign provider is an address book provider that:

- Assigns template identifiers for its recipients.
- Supports the **IABLogon::OpenTemplateID** method.
- Supplies code for maintaining recipients that exist in the containers of other address book providers known as host providers. The code involves a property object, typically an **IMAPIProp** interface implementation, that wraps a property object from the host provider.

Acting as a foreign provider is an optional role; not all providers need to support template identifiers and their related code. Implement your provider as a foreign provider if you want to maintain control over recipients that host providers create using templates supplied by your provider.

The format that your provider uses for its entry identifiers can also be used for its template identifiers. Template identifiers must include your provider's registered **MAPIUID** to allow MAPI to successfully bind recipients to the appropriate providers.

MAPI calls your provider's **IABLogon::OpenTemplateID** method when a host provider calls **IMAPISupport::OpenTemplateID**. The host provider passes the template identifier of the recipient in the *lpTemplateID* parameter in its call to **IMAPISupport::OpenTemplateID**. MAPI determines that the template identifier belongs to your provider by matching the **MAPIUID** in the template identifier with the **MAPIUID** that your provider registered at logon time. MAPI then forwards the host provider's call to your provider through the **IABLogon::OpenTemplateID** method.

The host provider also passes a pointer to its property object implementation for the recipient in the *lpMAPIPropData* parameter, an interface identifier in the *lpInterface* parameter that corresponds to the type of interface implementation passed in *lpMAPIPropData*, and an optional flag, FILL_ENTRY. Your provider is expected to return in the *lppMAPIPropNew* parameter a pointer to a property object implementation of the type specified in *lpInterface*. The returned pointer can either be to the wrapped property object implemented by your provider or to the object supplied by the host provider in *lpMAPIPropData*. Your provider should return a wrapped property object pointer when:

- The recipient's display table contains list box controls.
- The e-mail address for the recipient must be assembled from data in multiple display table controls.
- Your provider issues display table notifications.

The FILL_ENTRY flag indicates to your provider that the host provider wants all of the properties of the recipient to be updated. Your provider is required to fulfill this request.

When a host provider calls your provider's **OpenTemplateID** method, your provider might:

- Periodically update the data for a copied entry.
- Keep a copied entry in sync with its original, such as when an address book entry is copied to the PAB.
- Implement functionality that cannot be implemented by the host provider, such as dynamically populating list boxes in the copied entry's details table from data on a server.
- Control the interaction among properties in a copied entry or instantiated template. For example, computing PR_EMAIL_ADDRESS from other properties displayed in the details table.

The first two items are examples of tasks that do not require your provider to supply a wrapped property object − an implementation of **IMAPIProp** that is based on the host provider's implementation. Your provider can just update the properties as necessary and return, setting the *lppMAPIPropNew* parameter to point to the pointer passed in by the host provider in the *lpMAPIPropData* parameter.

The second two tasks require that your provider return to the host provider a property object that wraps the host provider's object with additional functionality, such as the ability to display a property sheet for the entry. This property object will either be a messaging user or distribution list, depending on the type of object passed in by the host provider in the *lpMAPIPropData* parameter and indicated by the interface identifier in the *lpInterface* parameter. If the *lpMAPIPropData* parameter points to a messaging user, your provider's wrapped property object must be an **IMailUser** implementation. If *lpMAPIPropData* points to a distribution list, it must be an **IDistList** implementation.

Your provider's wrapped property object intercepts **IMAPIProp** method calls to perform context-specific manipulation of the host provider's recipient, the object it is wrapping. MAPI only has one requirement for wrapped property objects: all calls to **IMAPIProp::OpenProperty** requesting the PR_DETAILS_TABLE property should be passed to the host provider. Your provider's implementation can use the returned table to intercept display table notifications or to add its own if necessary.

The following list includes tasks that are typically implemented in the wrapped property object implemented by foreign providers:

- Preprocessing and postprocessing property values for the host recipient in **IMAPIProp::GetProps**.
- Handling details display table controls such as buttons and listboxes in **IMAPIProp::OpenProperty**.
- Validating or manipulating property values for the host recipient in **IMAPIProp::SetProps**.
- Computing required properties such as PR_EMAIL_ADDRESS and verifying that all of the necessary properties have been set before saving the host recipient in **IMAPIProp::SaveChanges**.

To implement IABLogon::OpenTemplateID

1. Check if the template identifier passed in with the *lpTemplateID* parameter is valid and is in a format that your provider recognizes. If it is not, fail and return MAPI_E_INVALID_ENTRYID.
2. Create an object of the type indicated by the template identifier, either a messaging user, distribution list, or one-off recipient.
3. Call the **IUnknown::AddRef** method in the host provider's property object, the object pointed to by the *lpMAPIPropData* parameter.
4. If the *ulTemplateFlags* parameter is set to FILL_ENTRY:
   a. If the new object is a messaging user or distribution list:
      1. Retrieve all of the properties of the new object, possibly by calling its **IMAPIProp::GetProps** method.
      2. Call the host provider's **IMAPIProp::SetProps** method to copy all of the retrieved properties to the host provider's property object.

   b. If the new object is a one-off recipient, call the host provider's **IMAPIProp::SetProps** method to set the following properties:
      - PR_ADDRTYPE to the address type handled by your provider.
      - PR_TEMPLATEID to the template identifier from the *lpTemplateID* and *cbTemplateID* parameters.
      - PR_DISPLAY_TYPE to DT_MAILUSER or DT_DISTLIST, as appropriate.
5. Set the contents of the *lppMAPIPropNew* parameter to point to either your provider's new object or the property object passed in with the *lpMAPIPropData* parameter, depending on whether your provider determines a wrapped object is necessary.
6. If a critical error occurs, such as a network failure or an out of memory condition, return the appropriate error value. This value should get propagated to the client with the appropriate **MAPIERROR** structure, a task performed by the host provider.

Implementing IABLogon::Logoff

Your provider's **IABLogon::Logoff** method is called whenever a client calls **MAPILogoff** or **IMAPISession::Logoff** to end a session. Your **Logoff** method can perform any tasks that relate to cleaning up resources. Some of those tasks might be:

- To terminate a connection with a remote server.
- To decrement the reference count on the logon object.
- To remove the logon object from the list of logon objects that your provider stores.
- In debug mode, issue traces to locate objects that have leaked memory.

Implementing Address Book Containers

Most address book providers support at least one container, some of them modifiable. Address book containers can supply contents and hierarchy tables, searching capabilities, and name resolution. Modifiable containers allow the deletion of entries such as messaging users, distribution lists, or other containers and the addition of entries from entries in other containers or from one-off templates.

Implementing IABContainer::CreateEntry

Your container's **IABContainer::CreateEntry** method is called to create a new messaging user or distribution list in the container using a one-off template from a one-off table. A one-off template allows the client to create a new recipient of a particular type. Most of the fields are editable. The template pointed to by the *lpEntryID* parameter might be one that your provider supplies or it might be a template from a foreign provider, if your provider supports foreign templates. Implementations of **CreateEntry** for providers that can create recipients from a foreign template are always more complex than implementations for providers that cannot.

To implement IABContainer::CreateEntry

1. Determine the type of entry identifier specified by the *lpEntryID* parameter.
2. If the entry identifier represents a template for a messaging user, distribution list, or address book container owned by your provider:
   a. Create and initialize the appropriate object. Your provider can set some initial properties if desired. These properties depend on the type of recipient being created.
   b. Return a pointer to the object's implementation in the contents of the *lppMAPIPropEntry* parameter.
3. If the entry identifier represents a template for a foreign provider:
   a. Call **IMAPISupport::OpenEntry** to open the foreign object.
   b. Call the object's **IMAPIProp::GetProps** method, passing NULL for the property tag array, to retrieve its properties.
   c. Edit the property value array returned from **GetProps** by changing the property tag to PR_NULL for all properties that will not apply to the new object and should not be transferred.
   d. Create an entry identifier for the new object.
   e. Create a new object of the appropriate type, either messaging user or distribution list.
   f. Initialize the new object by setting default properties.
   g. Check whether or not the foreign object supports the PR_TEMPLATEID property.
   h. If the foreign object supports PR_TEMPLATEID, call **IMAPISupport::OpenTemplateID** to retrieve a property object interface from the foreign provider and set the contents of the *lppMAPIPropEntry* parameter to the foreign property object implementation.
   i. If the foreign object does not support PR_TEMPLATEID, set the contents of the *lppMAPIPropEntry* parameter to your provider's implementation of the new object.
   j. Call the **IMAPIProp::SetProps** method of the object pointed to by the *lppMAPIPropEntry* parameter to set the appropriate properties from the foreign object.

Implementing IABContainer::CopyEntries

Your container's **IABContainer::CopyEntries** method is called when one or more recipients from the same or another container are to be copied into this container. **CopyEntries** has four input parameters: an array of entry identifiers representing the recipients to be copied, a window handle for the progress indicator, a progress object pointer, and a flags value. Your provider should display progress if the AB_NO_DIALOG flag is not set and use the progress object from the *lpProgress* parameter if it is not NULL. If *lpProgress* is NULL, call **IMAPISupport::DoProgressDialog** to use MAPI's progress object. For more information about displaying progress, see Displaying a Progress Indicator.

In addition to AB_NO_DIALOG to suppress a progress indicator, one of two other flags can be set to request a type of duplicate entry checking: CREATE_CHECK_DUP_LOOSE or CREATE_CHECK_DUP_STRICT. The CREATE_CHECK_DUP_LOOSE and CREATE_CHECK_DUP_STRICT flags are only suggestions as to how your provider determines duplicate entries and can be ignored. MAPI suggests that your provider implement support for these flags as follows:

Duplicate entry flag	**Suggested implementation**
CREATE_CHECK_DUP_LOOSE	Check if the display name in the entry to be created matches the display name of an entry already in the container.
CREATE_CHECK_DUP_STRICT	Check if both the display name and the search key in the entry to be created matches the display name and search key of a container entry.

The last flag, CREATE_REPLACE, indicates that if your provider determines that an entry to be created is a duplicate of an entry already in your container, that the new entry should replace the existing one.

If your provider is a personal address book, include the PR_DETAILS_TABLE property in every copy operation. Including the details display table of a copied recipient allows your container to display the details of the recipient rather than having to call the original container to create the display.

To implement IABContainer::CopyEntries

1. Determine if each entry identifier in the *lpEntries* parameter is in a format that your provider handles and if it is not, fail and return MAPI_E_INVALID_ENTRYID.
2. If an entry identifier represents a messaging user, distribution list, or container that your provider handles:
   a. Call your **IMAPISupport::OpenEntry** to open the corresponding recipient.
   b. Copy the recipient to your container.
3. If the entry identifier represents a foreign recipient:
   a. Call your container's **IABContainer::CreateEntry** method to create a new recipient.
   b. Set initial properties on the new recipient.
4. Call the new object's **IMAPIProp::SaveChanges** method to save it.
5. Update the container's contents table to reflect the new recipient.
6. Call **IMAPISupport::Notify** to send a table notification to registered clients.

Implementing IABContainer::DeleteEntries

Your container's **IABContainer::DeleteEntries** method is called to remove one or more recipients. **DeleteEntries** has two parameters: an array of entry identifiers representing the recipients to be deleted and a reserved flags value. Deleting a recipient affects the contents table of your container; in addition to deleting the recipient, your container must delete the contents table row that represents the recipient. When the row has been removed from the table, your container must issue a table notification to each registered client.

To implement IABContainer::DeleteEntries

1. Delete each recipient represented by the entry identifier from your container.
2. If your container's contents table is open:
   - Send an *fnevTableModified* notification with the **ulTableEvent** member set to TABLE_ROW_DELETED to registered clients for each deleted contents table row. If your provider uses the notification utility, call **IMAPISupport::Notify** to send these notifications.
   - If your provider supports object notifications, also send an *fnevObjectDeleted* notification.

Implementing Advanced Searching

Some address book containers support an advanced searching capability that allows clients to search on properties other than PR_DISPLAY_NAME. To support advanced searches, your provider must implement a special container that is accessible through the PR_SEARCH property of your other containers. PR_SEARCH contains a container object that provides access to a display table that describes the dialog box used to enter and edit the advanced search criteria.

To support advanced searching

1. Define a new string property to hold the advanced search criteria.
2. In the section of code in your container's **IMAPIProp::OpenProperty** method that handles the PR_SEARCH property:
   a. Check that the client is requesting the **IMAPIContainer** interface. If an inappropriate interface is being requested, fail and return MAPI_E_INTERFACE_NOT_SUPPORTED.
   b. Create a new search object that supports the **IMAPIContainer** interface.
3. At this point, a call will be made to your search container's **IMAPIProp::OpenProperty** method to retrieve its PR_DETAILS_TABLE property. Your provider must supply a display table, typically through a call to **BuildDisplayTable**, that describes the container's advanced search dialog box.
4. MAPI displays the search dialog box, allowing the user to enter the appropriate criteria. When the user has finished, MAPI calls the container's **IMAPIProp::SetProps** method to store the search criteria.
5. A call will be made to request your search container's contents table. Populate the contents table with all of the entries in the container that match the criteria.

Implementing Name Resolution

Address book providers are responsible for supporting name resolution, the process of associating an entry identifier with a display name. Clients initiate name resolution when they call **IAddrBook::ResolveName** to ensure that each member of an outgoing message's recipient list corresponds to a valid address.

Your provider can support name resolution in one of two ways:

- Support for the PR_ANR property restriction
- An implementation of **IABContainer::ResolveNames**

Supporting the PR_ANR property restriction is required. Implementing **IABContainer::ResolveNames** is optional and returning MAPI_E_NO_SUPPORT is acceptable.

Implementing IABContainer::ResolveNames

Your provider's implementation of **IABContainer::ResolveNames** attempts to locate an exact match for each unresolved display name in the **ADRLIST** structure passed in with the *lpAdrList* parameter. An unresolved display name is missing the PR_ENTRYID property in the property value array in its **aEntries** member of the **ADRLIST**. Any entries that have zero properties associated with them should be ignored.

The result of your provider's attempt at resolution is reported in the *lpFlagList* parameter, an array of flags that corresponds to the array of display names in *lpAdrList*. The flags are positional such that the first flag corresponds to the first **aEntries** member in the **ADRLIST**, the second flag corresponds to the second **aEntries** member, and so on.

There are three possible results for each unresolved entry:

- No match can be found. No container entries match the entry in the **ADRLIST** structure. Address book providers set the corresponding entry in the *lpFlagList* parameter to MAPI_UNRESOLVED.
- Several matches can be found. There are multiple container entries that match the entry in the **ADRLIST** structure. Address book providers set the corresponding entry in the *lpFlagList* parameter to MAPI_AMBIGUOUS. They do not change the number of entries in the **ADRLIST** structure.
- An exact match can be found. There is only one container entry that matches the entry in the **ADRLIST** structure. Address book providers set the corresponding member in the *lpFlagList* parameter to MAPI_RESOLVED and add the entry identifier to the array of properties associated with the **ADRLIST** entry.

Implementing the PR_ANR Property Restriction

All address book providers are required to support ambiguous name resolution, or the PR_ANR property restriction, on their container contents tables. To support PR_ANR restrictions, your provider must supply a table object that supports handling the PR_ANR property restriction in its implementation of **IMAPITable::Restrict**. To handle the PR_ANR restriction, perform a "best guess" type of search, matching against one or more particular properties that make sense for your provider. Your provider can choose to use the same property or properties every time, such as PR_DISPLAY_NAME or PR_ACCOUNT, or allow an administrator to choose from a list of acceptable properties.

Although most providers supply their own table object implementation, it is an option to customize the implementation supplied by MAPI through the **CreateTable** function. Because the MAPI implementation does not support restrictions, a provider must create a wrapper object to include a customized version of **Restrict** that intercepts the call.

Implementing a Host Address Book Provider

A host provider is an address book provider that includes recipients from other providers in its containers and relies on the implementation of the recipients by the other providers to partially control their maintenance. A host provider uses the template identifiers of these foreign recipients to bind the data for these recipients to code in the foreign provider. This binding process is initiated when your provider retrieves the PR_TEMPLATEID property of a recipient and passes it in a call to **IMAPISupport::OpenTemplateID**.

When your provider calls **IMAPISupport::OpenTemplateID**, MAPI matches the **MAPIUID** within the template identifier with a **MAPIUID** registered by a provider and calls the provider's **IABLogon::OpenTemplateID** method. The foreign provider might return a pointer to your provider's property object or its own property object implementation, an implementation that wraps your provider's object. The returned pointer is placed in the contents of the *lppMAPIPropNew* parameter.

Your provider can choose whether or not to call **IMAPISupport::OpenTemplateID** with the FILL_ENTRY flag set. Set this flag when the recipient is being created or when a long time has passed since your provider has refreshed the recipient's properties. A common use of the FILL_ENTRY flag is to keep a recipient in your provider in sync with the original. Implementing this type of synchronization schedule enhances performance. To keep a foreign recipient in sync:

1. Determine an appropriate interval for periodic updates.
2. Timestamp each call to **IMAPISupport::OpenTemplateID**.
3. Evaluate whether or not it is necessary to perform a full update based on the amount of time that has expired since the last call. If a full update is necessary, call **IMAPISupport::OpenTemplateID** with the FILL_ENTRY flag. If it is not necessary, do not set the flag on the call.

When a client makes a request for one of the copied recipient's properties, your provider can choose whether to handle the request itself or use the code supplied by the foreign provider. Your provider can expect the foreign provider to intercept most, if not all, calls to **IMAPIProp** except for **IMAPIProp::OpenProperty**. A call to **OpenProperty** requesting the PR_DETAILS_TABLE property is always forwarded back to your provider.

To access template identifier code

1. Open the recipient and call its **IMAPIProp::GetProps** method to retrieve the PR_TEMPLATEID property. If **GetProps** fails because PR_TEMPLATEID is unavailable, the foreign provider does not support a template identifier and related code for this recipient. Your provider will need to use its implementation of the recipient for all maintenance.
2. If the template identifier is returned from **GetProps**, pass it and a pointer to the recipient's **IMAPIProp** implementation in a call to the **IMAPISupport::OpenTemplateID** method. Set the FILL_ENTRY flag if most or all of the recipient's properties need to be updated, such as at creation time or if they have not been updated for a while.
3. If **OpenTemplateID** returns the foreign provider's **IMAPIProp** implementation, return to the client a pointer to this implementation.
4. If **OpenTemplateID** does not return an implementation, typically because the foreign provider is not in the profile, return to the client a pointer to your provider's **IMAPIProp** implementation. The client should be able to work with the object's properties using either interface.

Implementing Recipient Details

MAPI provides a common dialog box for showing recipient details. This dialog box is modal; a modeless version is provided for in the interface, but is unsupported. The details dialog box is created from a display table and an **IMAPIProp** implementation. The display table describes the appearance of the details display and the **IMAPIProp** implementation controls the data for the recipient. Your provider is responsible for supplying the display table and the **IMAPIProp** implementation for each recipient.

The easiest way to create the display table is to define a **DTPAGE** structure and call **BuildDisplayTable**. However, some providers, specifically read-only providers that allow the creation of one-off recipients, use **IPropData**. The **IMAPIProp** implementation can be any type of property object.

There are two methods for invoking this dialog box: **IAddrBook::Details** and **IMAPISupport::Details**. When your provider calls one of these methods to request details for a recipient, MAPI first opens the recipient by calling its container's **IMAPIContainer::OpenEntry** method. Next it calls the recipient's **IMAPIProp::OpenProperty** method to request the PR_DETAILS_TABLE property. PR_DETAILS_TABLE is the property that represents a recipient's details display table.

The **IPropData** interface can be used to monitor changes on display table controls, as described in the following procedure:

To monitor changes to a control

1. Before the user gains access to the control, call **IPropData::HrSetObjAccess** to set the control's access to IPROP_CLEAN.
2. Allow the user to work with the dialog box.
3. When the user has finished, call **IPropData::HrGetPropAccess** to retrieve the current access level of the control.
4. If the access level is IPROP_DIRTY, the user has modified the control. Your provider should:
   - Call **IPropData::HrSetPropAccess** to set the access level back to IPROP_CLEAN.
   - Call the property data object's **IMAPIProp::GetProps** method to retrieve the changed property and update it by calling **IMAPIProp::SetProps**.
5. If the access level is still IPROP_CLEAN, the control has not been modified.

For more information about creating display tables, see About Display Tables.

Implementing Control Objects

Control objects, or objects that support the **IMAPIControl** interface, are implemented by providers to add functionality to a button that appears on a MAPI dialog box. Control objects can only be implemented for buttons.

IMAPIControl has three methods: **GetLastError**, **GetState**, and **Activate**.

MAPI calls **IMAPIControl::GetState**:

- When the dialog box on which the button appears is first displayed.
- When a display table notification is issued for the button.

Your provider returns one of two values that reflects whether or not a user can interact with the button: MAPI_DISABLED or MAPI_ENABLED. This value is returned in the contents of the *lpulState* parameter. MAPI uses the returned value to determine whether to disable or enable the button.

When the user clicks the button, MAPI calls your provider's **Activate** method. **Activate** performs the task that has been associated with the button. This task can be anything appropriate for your provider, such as displaying a dialog box or updating a property. If the task is unsuccessful because the user has canceled it, **Activate** should return MAPI_E_USER_CANCEL. With other causes of failure, return the appropriate error value.

If the task is successful, and it is linked to a property change that is reflected in another control on the dialog box, call **ITableData::HrNotify** to issue a display table notification with the changed property's PR_CONTROL_ID in the TABLE_NOTIFICATION structure. Do not place the new property value in the structure; your provider must return the new value when **IMAPIProp::GetProps** is called. Although typically display table notifications cannot be used to disable or enable a control, buttons are the exception. MAPI will reflesh the changed control to respond to the notification.

MAPI calls your provider's **GetLastError** method when **Activate** returns an error other than MAPI_E_USER_CANCEL. If **GetLastError** places extended error information in the **MAPIERROR** structure that it returns in the contents of the *lppMAPIError* parameter, MAPI displays it for the user.

Implementing an Address Book Status Object

Address book providers are required to supply status information to clients and to the status table. To supply status information, your provider must:

- Call **IMAPISupport::ModifyStatusRow**.
- Implement **IABLogon::OpenStatusEntry**.
- Implement the methods of the **IMAPIStatus** interface.
- Support the properties that are required for status objects and the status table, including PR_RESOURCE_METHODS.

During logon, call **IMAPISupport::ModifyStatusRow** to have MAPI create a row in the status table for your provider. Pass a property value array containing the column information for the row and zero for the *ulFlags* parameter. If at some point later in the session your provider's status changes and it becomes necessary to update the column information, call **ModifyStatusRow** again with the STATUSROW_UPDATE flag set. Any information that is included in your provider's status table row must also be available through its status object.

MAPI will call **IABLogon::OpenStatusEntry** when a client has asked to open your provider's status object. Providers can support read-only status objects, read/write status objects, or both, depending on what is requested. Callers pass the MAPI_MODIFY flag to **OpenStatusEntry** to request a writeable status object. If your provider is like most providers and does not allow changes to be made to its status object, return a status object that is read-only. Do not fail **OpenStatusEntry**. Clients expecting to receive read/write status objects should verify that read/write access has been granted before attempting to make any changes.

All status objects implement the **IMAPIStatus** interface, but not all status objects fully support all four of the methods. MAPI requires address book providers to support **ValidateState** to allow clients to check provider status and strongly recommends them to support **SettingsDialog** to display configuration information in a property sheet. **FlushQueues** is for transport providers only and **ChangePassword** is optional. If your provider is password protected and clients can set the password, implement **ChangePassword**.

To inform clients about the methods that your provider's status object supports, set the appropriate flags in the PR_RESOURCE_METHODS property. PR_RESOURCE_METHODS is a required column entry in the status table. For example, if your provider supports **ValidateState** and **SettingsDialog** only, set PR_RESOURCE_METHODS to:

```
STATUS_VALIDATE_STATE | STATUS_SETTINGS_DIALOG
```

For more information about implementing the methods of **IMAPIStatus** and supporting status object properties, see Status Objects.

Implementing One-Off Tables

Your provider might implement one or more one-off tables. A one-off table is a summary list of one-off templates used to create recipients, either directly into a container or into the recipient list of an outgoing message. A one-off template is a form users employ for entering data relevant to a particular type of address. When the user is finished working with the template, your provider creates the new recipient and adds it to the message. Typically each template handles a single address type. However, it is possible for a template to handle multiple types or for multiple templates to handle the same type.

Your provider must support the **OpenEntry** method for each template that it includes in the one-off table. The implementation of **OpenEntry** should retrieve a display table for the template. MAPI uses the display table to make the template visible to the user.

Although most of the rows in one-off tables represent templates, some of the rows can be used to categorize, or group, templates. Whether or not a row in a one-off table represents a template is indicated by the value of its PR_SELECTABLE column. Rows that represent templates have the PR_SELECTABLE column set to TRUE; rows that do not represent templates have it set to FALSE.

MAPI defines three types of one-off tables:

- A one-off table that reflects the templates that an individual container supports
- A one-off table that reflects all of the templates that your provider supports
- A one-off table that reflects all of the templates that all of the providers in the profile support plus some that MAPI supports

The first two types are implemented by providers that support the creation recipients, either onto a message or into a container. Your provider can include the same set or a different set of templates in its one-off tables. The main difference between the two types is that your provider table should include templates for creating recipients that can be used on outgoing messages and your container table should include templates for creating recipients to be added to your container. A container may only support a restricted set of templates, but the provider one-off table should include every template the provider supports.

The third type of one-off table is implemented by MAPI; providers gain access to it by calling **IMAPISupport::GetOneOffTable**. MAPI's one-off table is the union of all of the provider tables; it includes every template supported by every provider in the profile. It also includes templates supported by MAPI. Your provider can use this table in place of the table requested for a container. However, the templates in this table can also be used for creating recipients for outgoing messages.

Implementing a Provider One-Off Table

MAPI calls your provider's **IABLogon::GetOneOffTable** method when the user of a client application adds a recipient to an outgoing message. Typically the types of addresses requested are unique to your messaging system. If your provider supports recipient creation, it must supply a one-off table that exposes templates for every type of supported recipient address. If your provider does not support recipient creation, return MAPI_E_NO_SUPPORT from the **GetOneOffTable** call.

MAPI will typically keep your provider's one-off table open for the lifetime of the session, releasing it only when a client calls either the subsystem's or address book's **IMAPIStatus::ValidateState** method. MAPI registers for notifications on this table so that if templates are added or deleted, these changes can be reflected back to the user.

To implement IABLogon::GetOneOffTable

1. Check the value of the flags parameter, *ulFlags*. If the MAPI_UNICODE flag is set and your provider does not support Unicode, fail and return MAPI_E_BAD_CHARWIDTH.
2. Check if your provider's one-off table has already been created. Because one-off tables are typically static, your provider never has to go through the creation process more than once. If a table already exists, return a pointer to it.
3. If a one-off table does not yet exist, call **CreateTable** to create one.
4. Set the following properties for the columns in your table rows:
   - PR_DISPLAY_NAME to the name of the type of recipient that the template can create.
   - PR_ENTRYID to the entry identifier for the one-off template.
   - PR_DEPTH to indicate the hierarchy level in the one-off table display.
   - PR_SELECTABLE to TRUE to indicate if the row represents a template and FALSE otherwise.
   - PR_ADDRTYPE to the type of address created by the template.
   - PR_DISPLAY_TYPE to DT_MAILUSER or another value that indicates the type of display for the template.
   - PR_INSTANCE_KEY to a unique binary value.
5. Call **ITableData::HrModifyRow** to modify the table directly.
6. Call **ITableData::HrGetView** to create an **IMAPITable** interface implementation to return to the caller.

Implementing a Container One-Off Table

To access the one-off table belonging to one of your containers, MAPI calls the container's **IMAPIProp::OpenProperty** method to open the PR_CREATE_TEMPLATES property with the **IMAPITable** interface. Your container is asked to return its one-off table when a client application is trying to add a recipient to the container. If the container allows any recipients, your provider can either return its own table implementation or call **IMAPISupport::GetOneOffTable** to return the MAPI implelmentation.

The set of templates in the container one-off table should reflect the type of recipients that the particular container can hold. Typically, this includes one or two templates, templates for creating an individual messaging user or a distribution list. The entry identifiers for these templates are held in the PR_DEF_CREATE_MAILUSER and PR_DEF_CREATE_DL properties. However, containers are by no means limited to these types of entries. They can hold other types of recipients or non-recipient entries such as directories.

Developing a Transport Provider

You should be familiar with the MAPI subsystem's architecture, the C++ language, and with writing dynamic-link libraries (DLLs) for the various Windows platforms before developing a transport provider.

For information about implementation of features necessary for TNEF and remote transport providers, see Developing a TNEF-Enabled Transport Provider and Developing a Remote Transport Provider.

What is a Transport Provider

A transport provider is a dynamic-link library (DLL) which acts as an intermediary between the MAPI subsystem and one or more underlying messaging systems. A messaging system is some specific mechanism by which messages are sent and received. Some examples of messaging systems are:

- A shared network file system that the transport provider writes messages to directly.
- A TCP/IP network interface that the transport provider uses to connect to a messaging server.
- An online service that users dial into.
- A host-based messaging or office automation system.
- A set of remote procedure calls to a messaging server.
- Anything that can be used to transfer data from one computer to another.

A transport provider DLL must conform to the interface specified by MAPI. As a transport provider developer, you will implement this interface in terms of the functionality present in the messaging system.

Types of Transport Providers

All transport providers support a range of standard features, such as:

- Providing proper security for the underlying messaging system.
- Sending and receiving messages from the underlying messaging system.
- Exposing the address types the transport providers support so the MAPI spooler and client applications can use them appropriately.
- Accepting delivery for specific recipients.

In addition, MAPI supports two specialized types of providers for specific messaging systems.

Transport type	Added functionality
Remote Transport	Enables interoperabilty with remotely connected clients.
TNEF Transport	Allows MAPI properties to be preserved on messaging systems that do not support them.

Remote transports are used by messaging clients that are using low-bandwidth network connections. The most common example is a client operating over a modem link. Remote transports implement the **IMAPIFolder : IMAPIContainer** interface in addition to the basic functionality implemented by all transports. The **IMAPIFolder** interface makes it possible for client applications to selectively download messages from a remote message server based on header information at a user's request. Remote transports make themselves known to clients as remote transports by registering themselves in the MAPI status table. For information on implementing remote transports, see Developing a Remote Transport Provider.

TNEF transports are used for translating messages between messaging systems that support different sets of MAPI properties. TNEF transports use the MAPI **ITnef:IUnknown** interface to convert any properties that the destination system cannot represent directly into a binary data stream that can be attached to the message. Later, another TNEF transport can use **ITnef** to decode the data stream and make the original MAPI properties available to client applications. Additionally, TNEF support is required if your transport needs to support custom message classes. For information on implementing TNEF transports, see Developing a TNEF-Enabled Transport Provider.

If your transport provider is not one of these types, you will have to implement it with the basic MAPI functions and networking functions available on your target platform.

Transport Provider's Role in the MAPI Subsystem

Transport provider dynamic-link libraries (DLLs) provide the interface between the MAPI spooler and that part of a messaging system responsible for message sending and receiving. The MAPI spooler and the transport provider work together to handle the responsibilities of sending a message or receiving a message. The MAPI spooler loads the transport provider DLL when it is first used and releases it when it is no longer needed. Multiple transports can be installed on the same system, but MAPI supplies the one spooler required.

Client applications do not typically communicate directly with the transport provider; rather, clients submit messages through a store provider, and the MAPI spooler sends outgoing messages to the appropriate transport provider and delivers incoming messages to the appropriate message store. The MAPI spooler does its work and makes its calls to transport providers when foreground applications are idle. After optionally displaying dialog boxes when the transport provider is first logged on, transport providers operate in the background unless called by the client to flush send and receive queues.

Transport providers have the following responsibilities in a MAPI messaging system:

- Register the address types they can accept with the MAPI spooler so that the MAPI spooler can submit messages to the appropriate transport provider depending on the destination address of the messages. One transport provider can register more than one address type. Transport providers can also register specific recipients' addresses with the MAPI spooler. Messages addressed to one of these addresses will be submitted to the transport provider that registered this address with the MAPI spooler. For more information, see Transport Provider-MAPI Spooler Operational Model.
- Deliver incoming messages to the MAPI spooler. Depending on the nature of the messaging system, a transport provider can either directly notify the MAPI spooler when a new message arrives, or can request that the MAPI spooler poll the transport provider periodically to check for new messages.
- Convert MAPI message properties to and from message properties native to the messaging system. For example, the transport provider might have to convert the sender's and recipients' addresses in an outgoing message to a form that is acceptable to the messaging system. Some messaging systems do not support all of the MAPI message properties. For information on preserving MAPI message properties when delivering messages to a messaging system, see Developing a TNEF-Enabled Transport Provider.
- Register message and recipient options specific to the transport provider.
- Perform any verification of credentials required by the messaging system.
- Access outbound messages using the message object passed to it by the MAPI spooler.
- Translate message format as required by the underlying messaging system.
- Notify the MAPI spooler which recipients of an outgoing message the transport provider has accepted responsibility for handling by setting the PR_RESPONSIBILITY property for those recipients.
- Inform the MAPI spooler when an incoming message needs to be handled.
- Pass incoming message data to the MAPI spooler by using message objects.
- Assign values to all required MAPI message properties on incoming messages.
- Delete the message from the underlying messaging system after delivery, if necessary.
- Provide status information for the MAPI spooler and client applications.

The following illustration shows a transport provider's role with respect to the other components of the MAPI architecture.

{ewc msdncd, EWGraphic, groupx837 0 /a "MAPI.WMF"}

Transport Provider-MAPI Spooler Operational Model

Transport provider initialization, startup, processing, shutdown and deinitialization are accomplished by a series of calls from the MAPI spooler to the transport provider. The calls are sequenced as follows:

1. The MAPI spooler calls the **XPProviderInit** function, passes a support object, gets the provider object, and checks that the provider and MAPI spooler support a compatible range of MAPI version numbers.
2. The MAPI spooler calls the **IXPProvider::TransportLogon** method of the XPProvider object. A session is established between the MAPI spooler and the transport with the credentials in the current section of the profile. The provider returns a logon object.
3. The MAPI spooler calls the **IXPLogon::AddressTypes** method. The transport returns a list of the Unique Identifiers (UIDs) and e-mail address types it will accept.
4. The MAPI spooler calls the **IXPLogon::RegisterOptions** method for recipient options. The transport returns a list of the available per-recipient options for any of the e-mail address types that it returned in the **IXPLogon::AddressTypes** call.
5. The MAPI spooler calls **RegisterOptions** for message options. The transport returns a list of the available per-message options for any of the e-mail address types that it returned in the **IXPLogon::AddressTypes** call.
6. The transport calls the **IMAPISupport::ModifyStatusRow** method to create its row in the MAPI status table.
7. The MAPI spooler calls the **IXPLogon::TransportNotify** method to enable message transmission and reception.
8. If requested by the transport provider in its return for the **TransportLogon** call, the MAPI spooler periodically calls the **IXPLogon::Idle** method. Idle processing is useful if the transport provider needs to poll the underlying messaging system for new messages or perform other low-priority tasks.
9. The MAPI spooler and transport send and receive messages (see Message Submission Model and Message Reception Model). The MAPI spooler services transport requests and calls on support, message, and attachment objects.
10. The MAPI spooler calls the **TransportNotify** method to disable message transmission and reception.
11. The MAPI spooler releases the logon and provider objects. See **IXPProvider::Shutdown**.

Message Submission Model

Message submission is accomplished by a series of calls from the MAPI spooler to the transport provider. The calls are sequenced as follows:

1. The MAPI spooler calls **IXPLogon::SubmitMessage**, passing in an **IMessage** instance, to begin the process.
2. The transport provider then places a reference value − a transport-defined identifier used in future references to this message − in the location referenced in **SubmitMessage**.
3. The transport provider accesses the message data by using the passed **IMessage** instance. For each recipient in the passed **IMessage** for which it accepts responsibility, the transport provider sets the PR_RESPONSIBILITY property, and then returns.
4. The transport provider can use the **IMAPISupport::StatusRecips** method to indicate if it recognizes any recipients that cannot be delivered to, or to create a standard delivery report. **StatusRecips** is a convenience for transport providers that have determined that some of the recipients cannot be delivered to or have received delivery information from their underlying messaging system that the user or client application might find useful.
5. The MAPI spooler's call to **IXPLogon::EndMessage** is the final responsibility hand-off for the message from the MAPI spooler to the transport provider.
6. The MAPI spooler can use **IXPLogon::TransportNotify** to cancel message processing during the **SubmitMessage** or **EndMessage** calls.

Message Reception Model

The transport provider controls whether whether the MAPI spooler must poll it for incoming mail or whether it performs a callback to the MAPI spooler when new mail arrives. The transport provider sets the SP_LOGON_POLL flag when it returns from **IXPProvider::TransportLogon** to request polling. Otherwise, the transport uses **IMAPISupport::SpoolerNotify** when incoming mail is available. After learning that incoming mail is available, the MAPI spooler opens a new message and asks the transport provider to store the received message properties into the message.

This process works as follows:

1. Available messages are indicated by either the transport provider calling **IMAPISupport::SpoolerNotify** or by the MAPI spooler calling **IXPLogon::Poll**.
2. The MAPI spooler calls **IXPLogon::StartMessage** to initiate the process.
3. The transport provider places a reference value in the location referenced in **StartMessage**. These reference values allow the transport provider and the MAPI spooler to keep track of which message is being processed when there are multiple messages to deliver.
4. The transport provider stores the message data into the passed **IMessage** instance.
5. The transport provider calls the **IMAPIProp::SaveChanges** method on the **IMessage** instance and returns from **StartMessage**.
6. The MAPI spooler calls **IXPLogon::TransportNotify** if it must stop message delivery.

Note If a transport provider must deliver a large number of messages and the transport provider is using **IMAPISupport::SpoolerNotify** instead of **IXPLogon::Poll**, care should be taken not to call **SpoolerNotify** too frequently in order not to deprive other transport providers of CPU time. The MAPI spooler does have logic to prevent this from happening, but in general the interval between **SpoolerNotify** calls should be longer than the time it takes your transport provider to process one message.

Also, the MAPI spooler may not process an incoming message immediately. The MAPI spooler may ask the transport provider to perform other tasks before it receives the incoming message.

Required Functionality for Transport Providers

All MAPI transport providers must implement certain features, the most important of which are as follows:

- Your transport provider must follow the general guidelines for working with MAPI and other service providers as described in MAPI Component Basics and Service Provider Basics.
- Your transport provider DLL must expose to MAPI its **XPProviderInit** initialization function.
- Your transport provider must expose to MAPI its implementation of the **IXPProvider** and **IXPLogon** interfaces.
- Your transport provider must expose to MAPI and client applications its implementation of the **IMAPIStatus** interface.
- Your transport provider must have a user interface for configuration, such as a wizard interface or service entry.

Working with MAPI and Other Providers

MAPI service providers of any kind must follow certain guidelines to work with other MAPI components. Each service provider must:

- Use the proper Provider and Logon objects to initialize your provider.
- Return a dispatch table of provider entry points to the messaging system upon initialization.
- Register a MAPI status table row for each resource owned by a provider and call the **IMAPISupport::ModifyStatusRow** method at appropriate times.
- Use the **IMAPISupport::NewUID** method to obtain valid Unique Identifiers (UIDs).
- Support the common MAPI interfaces on objects returned by your provider.
- Use the MAPI memory allocation functions to allocate memory returned to client applications and to release memory allocated by other parts of the MAPI subsystem.
- Maintain a profile section, if necessary, to store credentials to the underlying messaging system.
- Use the **IMAPISupport::RegisterPreprocessor** method to register any message preprocessing functions.
- Include the proper header files (including MAPISPI.H) that define common constants, structures, interfaces, and return values.
- Follow address format conventions for common address types.

Interacting with the MAPI Spooler

The methods in the **IXPLogon** interface are used by the MAPI spooler when calling the transport provider. It should be possible for most types of transport providers to implement most of these methods so that they return quickly. This is desirable because if a method takes a long time to return then it should be broken up with calls back to the MAPI spooler to release the CPU for other tasks. This is critical in non-preemptive multitasking environments like 16-bit Windows. Transport providers for 16-bit Windows platforms should take particular care to break up any operation that takes more than 0.2 seconds. This is also necessary in 32-bit Windows to provide time to handle system-wide messages such as shutdown notifications or plug-and-play device notifications.

The MAPI spooler does its work and makes its calls to transport providers when foreground applications are idle. After possibly displaying dialog boxes when the transport provider is first logged on (governed by flags passed from MAPI to the transport provider), transport providers operate in the background unless called by the client to flush send and receive queues. When flushing queues is the only time that a transport provider need not release the CPU, and then only if the user is informed that a potentially long action is in progress. The MAPI spooler typically requests that a transport provider flush its queues in response to a user action, so the transport provider typically does not need to do anything to ensure that the user is informed.

A transport provider can independently decide to flush a queue and use the STATUS_INBOUND_FLUSH and STATUS_OUTBOUND_FLUSH bits in the PR_STATUS_CODE property of its status row to inform the MAPI spooler that it wants an inordinate amount of attention so that it can get the job done. The status row is updated using the **IMAPISupport::ModifyStatusRow** method. In this case the transport provider should probably display a progress dialog or other interface to inform the user that a long action is occuring.

Since network activity often takes more than 0.2 seconds, transport providers should, whenever possible, use asynchronous network requests. This enables them to initiate a request, release the CPU by calling back to the MAPI spooler, and when the MAPI spooler again gives them control, to check to see if their network request has completed. If it has not yet completed, they again release the CPU by calling back to the MAPI spooler with the **IMAPISupport::SpoolerYield** method.

During message processing, between **IXPLogon::SubmitMessage** and **IXPLogon::EndMessage** and during **IXPLogon::StartMessage** the transport provider typically makes many calls on objects exposed to it by the MAPI spooler. As part of its handling of these objects, the MAPI spooler helps the transport provider behave appropriately as a background process by yielding on its own when appropriate. A transport provider requiring time-critical processing can declare a critical section to the MAPI spooler using the **IMAPISupport::SpoolerNotify** support object method. In this case, the CPU is released only on explicit **SpoolerYield** calls by the transport provider until the transport provider ends critical section processing with another call to **SpoolerNotify**. Note that this is not the same as a Win32 critical section. This should only be done when the transport provider needs real-time control of external resources such as reading incoming data from a FAX line. Since this raises the priority of the MAPI spooler process and can cause the workstation to be unresponsive for the duration of the operation, it is a good idea to notify the user that a potentially long action is underway and provide a progress indicator if possible.

Initializing the Transport Provider

The transport-spooler interface defines calls the MAPI spooler makes to a transport provider. Transport providers implement these routines in a DLL. The first direct entry point in the DLL used by the MAPI spooler must be the transport provider initialization function **XPProviderInit**.

MAPI uses the operating system routine **GetProcAddress** to get the address of the provider's initialization routine and then calls that routine. The name of the initialization routine is **XPProviderInit** for transport providers. It is different for other types of MAPI service providers so that one DLL can contain any combination of provider types, but only one provider of a particular type. However, one provider of a given type can implement multiple services of its type. For example, one transport provider can implement message transport functionality to multiple message services.

The MAPISPI.H header file has a type definition for the function prototype of the transport provider initialization function, and a predefined procedure name for it. By naming the initialization routines in your C and C++ files with the same names used by **GetProcAddress** and by using a straightforward export declaration in your DLL.DEF file, you automatically get type checking of the parameters on your initialization routine. See the sample transport provider source code for examples.

If a provider's initialization call succeeds but returns a service provider interface version number too small for MAPI to handle, MAPI immediately calls the **Release** method of the provider object and proceeds as if the initialization call had failed with MAPI_E_VERSION. This way MAPI and the provider jointly define the range of service provider interface version numbers they can handle, and if nothing matches then provider loading fails with a MAPI_E_VERSION return value.

The last step for the MAPI spooler in getting access to service provider resources is to log onto the transport provider. The MAPI spooler calls the **IXPProvider::TransportLogon** method of the **IXPProvider** object returned from **XPProviderInit**. This is the call where credentials ,if used, are checked and dialog boxes can be allowed.

If a process opens a second transport session on the same transport provider and MAPI session, the transport provider DLL should not create a second provider object. The first provider object should be used to log onto the second transport session. A transport provider should be programmed to support multiple transport sessions in a single provider object. A second provider object should only be created if different MAPI sessions are used in the same process.

Releasing the Transport Provider

When MAPI or the MAPI spooler finishes using a transport logon object

1. MAPI or the MAPI spooler calls the transport provider's **IXPLogon::TransportLogoff** method.
2. The transport provider invalidates the status object by calling the **IMAPISupport::MakeInvalid** method. Whether the transport provider invalidates message objects that are being sent or received at the time of the **TransportLogoff** call depends on the flags that were passed to **TransportLogoff**.
3. The transport provider calls the support object's **Release** method to remove the provider's row from the status table and remove from internal tables any Unique Identifiers (UIDs) that were set with the **IMAPISupport::SetProviderUID** method. It decrements the count of known logon objects active on this provider object. If the count reaches zero, MAPI calls the **IXPProvider::Shutdown** method and **Release** on the provider object. If this was the last known provider object using this DLL on this process, MAPI calls **FreeLibrary** on the DLL at some later time. Memory for the MAPI support object is freed and the support object **Release** method returns.
4. The **TransportLogoff** method returns S_OK.
5. MAPI or the MAPI spooler calls **Release** on the transport provider's logon object. The memory for the object is discarded.
6. MAPI or the MAPI spooler calls **FreeLibrary** on the provider DLL.

For robustness, the logon and provider objects should be able to handle final **Release** calls on themselves without first having their **TransportLogoff** or **Shutdown** methods called. If **Release** is called in such cases, transport providers should treat the calls as if **TransportLogoff** or **Shutdown** had been called with a zero argument followed by **Release**.

Implementing the FlushQueues Method

The MAPI spooler uses the **IXPLogon::FlushQueues** method to download and upload any pending messages to and from a transport provider. Typically, the MAPI spooler will flush the queues for all transport providers that are logged onto the session, starting with the first transport provider as set in the transport order section of the user's profile. Flushing queues is almost always the result of a direct request by the user, so the sending and receiving of messages while queues are flushing is synchronous to the MAPI spooler. Because these calls are synchronous, the transport provider should process them as quickly as possible.

Transport providers must handle the **FlushQueues** call as described in the following sequence of steps to enable proper message processing and to enable external resources such as modems to be used by other transport providers as part of the MAPI spooler's **FlushQueues** operation.

Step	Component	Implementation
1.	MAPI spooler	Calls the **IXPLogon::FlushQueues** method for the first transport provider listed in the transport order of the user's profile, passing the requested flags in the *ulFlags* parameter. **FlushQueues** is called once with all flags set for the entire upload and download operation.
2.	Transport Provider	Needs to do a number of things before returning from the **FlushQueues** call. If previously submitted messages are being deferred, the **IMAPISupport::SpoolerNotify** method should be called with the NOTIFY_SENT_DEFERRED flag set. Note that it is possible for the MAPI spooler to cancel a message that has been deferred before the transport provider has a chance to finish processing the message. If the transport provider uses an external resource such as a modem, the connection to the external resource should be established. The STATUS_OUTBOUND_FLUSH bit in the PR_STATUS_CODE property of the transport provider's status row must be set using the **IMAPISupport::ModifyStatusRow** method. The transport provider should then return S_OK for the **FlushQueues** call.
3.	MAPI spooler	Checks the transport provider's status row for the STATUS_OUTBOUND_FLUSH bit and calls **IXPLogon::SubmitMessage** for the first message in the queue.
4.	Transport provider	Handles the message and returns from the **SubmitMessage** call.
5.	MAPI spooler	If the transport provider returns S_OK from **SubmitMessage**, the MAPI spooler calls **IXPLogon::EndMessage** for the message as it does with regular message sending. If the transport provider returns a value other than S_OK from **SubmitMessage**, the MAPI

		spooler handles the value appropriately before calling **EndMessage**, or before calling **SubmitMessage** again.
6.	Transport provider	Returns from **EndMessage** with its message processing status in the *lpulFlags* parameter.
7.	MAPI spooler and transport provider	The **SubmitMessage**-**EndMessage** loop continues until all messages in the queue have been downloaded.
8.	MAPI spooler	Notifies the transport provider that it is has finished downloading messages by calling the transport provider's **IXPLogon::TransportNotify** method with the NOTIFY_END_OUTBOUND_FLUSH flag set.
9.	Transport provider	Should free any external resources used in sending outbound messages so they can be used by other transport providers to flush their queues. The STATUS_INBOUND_FLUSH bit in the PR_STATUS_CODE property of the transport provider's status row must be set using **ModifyStatusRow**.
10.	MAPI spooler	Checks the transport provider's status row for the STATUS_INBOUND_FLUSH bit and calls **IXPLogon::StartMessage** if it is set.
11.	Transport provider	Processes the message and returns from **StartMessage**. If the transport provider has other messages to upload, it should call **SpoolerNotify** with the NOTIFY_NEWMAIL flag set. If the transport provider has no messages to upload, it should call **IMAPIProp::SaveChanges** on the message the MAPI spooler passed in **StartMessage** and return.
12.	MAPI spooler	Continues calling **StartMessage** until **SaveChanges** is called on a message. After the transport provider has finished uploading, the MAPI spooler calls **TransportNotify** with the NOTIFY_END_INBOUND_FLUSH flag set.
13.	Transport provider	Clears the STATUS_INBOUND_FLUSH bit in the PR_STATUS_CODE property of its status row using **ModifyStatusRow** and releases all external resources that have been released so they are available for use by other transport providers.
14.	MAPI spooler	Calls **FlushQueues** for the next transport provider listed in the transport order of the user's profile.

If a client application calls **IMAPIStatus::FlushQueues** on a transport provider's status object, the transport provider should set the appropriate bit in its status row with **ModifyStatusRow**. The MAPI spooler then calls the provider's **IXPLogon::FlushQueues** method at the MAPI spooler's convenience. When the provider's **IXPLogon::FlushQueues** method is called as a result of a client application's **IMAPIStatus::FlushQueues** call, the operation occurs asynchronously to the client application.

Otherwise **IXPLogon::FlushQueues** works synchronously with the MAPI spooler.

For performance reasons, the MAPI spooler will only call a transport provider's **FlushQueues** method if the STATUS_INBOUND_FLUSH and STATUS_OUTBOUND_FLUSH flags are set in the transport provider's status row. Consequently, a transport provider can stop the **FlushQueues** operation at any time by clearing the STATUS_OUTBOUND_FLUSH and STATUS_INBOUND_FLUSH flags in its status row. If the MAPI spooler is shutting down and needs to end the **FlushQueues** operation, it calls **TransportNotify** with both the NOTIFY_END_INBOUND_FLUSH and NOTIFY_END_OUTBOUND_FLUSH flags set. The transport provider should release all external resources and return.

Setting Properties on Incoming Messages

The client applications within the MAPI subsystem expect a number of properties in any received message. When the transport provider brings a message into MAPI, it should set these properties, since it is either the only process with the necessary information to do so, or is at least the best source of the information.

Providers are encouraged to set the values for all of these properties in incoming messages.

Property Name	Description
PR_SUBJECT	Subject of the message.
PR_BODY	Plain text message text.
PR_RTF_COMPRESSED	Compressed RTF message text.
PR_MESSAGE_DELIVERY_TIME	Date and time the message was delivered.
PR_SENDER_NAME	Display name of the message originator.
PR_SENDER_ENTRYID	Address book entry of the message originator.
PR_SENDER_SEARCH_KEY	Address book search key of the message originator.
PR_CLIENT_SUBMIT_TIME	The time that the message was submitted to its messaging system by the sender's messaging client.
PR_SENT_REPRESENTING_NAME	Name of the representative delegate for sending.
PR_SENT_REPRESENTING_ENTRYID	Address book entry of the sending delegate.
PR_SENT_REPRESENTING_SEARCH_KEY	Address book search key of the sending delegate.
PR_RCVD_REPRESENTING_NAME	Name of the representative delegate for receiving.
PR_RCVD_REPRESENTING_ENTRYID	Address book entry of the receiving delegate.
PR_RCVD_REPRESENTING_SEARCH_KEY	Address book search key of the receiving delegate.
PR_REPLY_RECIPIENT_NAMES	List of delegated recipient display names, separated by a semicolon and space "; ".
PR_REPLY_RECIPIENT_ENTRIES	List of delegated recipients for a reply.
PR_MESSAGE_TO_ME	Indicates that the recipient was specifically named as a To recipient

	(not in a group).
PR_MESSAGE_CC_ME	Indicates that the recipient was specifically named as a Cc recipient (not in a group).
PR_MESSAGE_RECIP_ME	Indicates that the recipient was specifically named as a To, Cc or Bcc recipient (not in a group).

Providers which have no apparent mappings can set the PR_SENT_REPRESENTING group of properties to the same values as the PR_SENDER group, the PR_RCVD_REPRESENTING group to the same values as the PR_RECEIVED_BY group, and build the PR_REPLY_RECIPIENT group of properties based on the values of the PR_SENDER group. For example, PR_SENT_REPRESENTING_NAME can be set to the same value as PR_SENDER_NAME.

The PR_ENTRYID or PR_ENTRYLIST properties can be generated if necessary by calling the **IMAPISupport::CreateOneOff** method. The PR_SEARCH_KEY properties can be generated by concatenating the PR_ADDRTYPE property associated with a user, a colon ':', and the PR_EMAIL_ADDRESS property associated with the user, then folding the result to uppercase. The Windows API **CharUpperBuff** is a convenient function to use for this purpose. What is required of this process is to make a canonical form of the address that can be compared as a binary quantity. Note that this is not necessary if the transport provider is case-sensitive with respect to e-mail addresses.

Using the Support Object

MAPI provides an implementation of the **IMAPISupport** interface which the transport provider receives during logon. Transport providers can call the following **IMAPISupport** methods.

IMAPISupport Method	Description
CompareEntryIDs	Returns TRUE if two entry identifiers refer to the same support object.
CreateOneOff	Creates a custom recipient identifier.
DoConfigPropSheet	Displays the MAPI property sheet dialog box.
GetLastError	Returns information about the last error for the support object.
GetMemAllocRoutines	Returns the addresses of the MAPI memory management routines.
GetSvcConfigSupportObj	Creates a new support object.
IStorageFromStream	Converts an OLE **IStream** object to an **IStorage** object.
MakeInvalid	Invalidates an object derived from the **IUnknown** interface.
ModifyStatusRow	Sets values in the provider's row in the status table.
NewUID	Returns a new, unique MAPI identifier.
Notify	Notifies interested parties of changes to an object owned by the transport provider.
OpenAddressBook	Returns a pointer to the address book.
OpenEntry	Opens an object given its entry identifier.
OpenProfileSection	Returns a pointer to a given section of the profile.
RegisterPreprocessor	Registers a preprocessor callback for the provider.
SpoolerNotify	Requests servicing from the MAPI spooler.
SpoolerYield	Enables the MAPI spooler to give processor time to other applications or to Windows.
StatusRecips	Generates delivery and nondelivery reports.
Subscribe	Informs MAPI that the transport provider needs to be notified of changes to an object.
Unsubscribe	Cancels a previous subscription.

WrapStoreEntryID	Maps a message store's private identifier to an identifier useful to the rest of the messaging system.

Note that the **IMAPISupport** interface contains support methods for all types of service providers, many of which are not applicable to transport providers. The preceding list contains methods that transport providers can use.

Providing Status

Transport providers expose status and other dynamic information to client applications using the MAPI status table. When the MAPI spooler creates a session with a transport provider by calling the **IXPProvider::TransportLogon,** method, the provider creates a row in this table for each resource it owns. A transport provider, for instance, should create a row for each message queue it manages when it is being logged on. MAPI manages the table interface and notifies client applications of status changes. Transport providers must update rows in this table when changes occur and must support client calls to obtain information not exposed in the table.

Transport providers must support three operations on status table entries:

- Create a row in the status table by calling the **IMAPISupport::ModifyStatusRow** method during the **TransportLogon** call.
- Update a row in the status table by calling **ModifyStatusRow** as needed.
- Service requests from client applications for further status information.

The information that a transport provider gives MAPI appears as a single row in an **IMAPITable** object. When status changes, the transport provider should call **IMAPISupport::ModifyStatusRow** to inform MAPIof the change. MAPI broadcasts the notification to interested clients. Transport providers implement the **IMAPIStatus** interface to provide client applications with a way to query for more information than is contained in the status table.

Status properties

The following properties are used by transport providers to indicate various aspects of their status to the MAPI spooler and other MAPI components (such as MAPI clients). The transport provider should maintain accurate values for these properties in the MAPI status table.

Property	Meaning
PR_STATUS_CODE	A bitmask of values indicating the status of various aspects of the transport provider.
PR_STATUS_STRING	An ASCII message indicating the status of the transport provider.
PR_RESOURCE_TYPE	Each status object has a 32-bit resource type associated with it. The resource type for transport providers is MAPI_TRANSPORT_PROVIDER.
PR_RESOURCE_METHODS	Indicates which methods of the **IMAPIStatus** interface the transport provider implements.
PR_RESOURCE_FLAGS	Each status object has flags associated with it. The following flags are particularly important for transport providers: STATUS_DEFAULT_OUTBOUND STATUS_PRIMARY_IDENTITY STATUS_XP_PREFER_LAST Other flags are available.

Only the PR_STATUS_STRING property needs to be maintained dynamically by the transport provider. The PR_RESOURCE_TYPE, PR_RESOURCE_METHOD, and PR_RESOURCE_FLAGS properties are all read from the profile by MAPI and do not need to be supplied with the status row. The transport provider is only responsible for putting them in the MAPISVC.INF file when it is first installed.

Optional Features Transport Providers Can Implement

Optional features transport providers can implement include the following:

- Registering message and recipient options specific to the transport provider.
- Maintaining a profile, if necessary, to store configuration information and credentials to the messaging system.
- Performing any verification of credentials required by the messaging system.
- Supporting event notification for interested client applications with the **IMAPISupport::Notify** method.
- Displaying configuration property sheets and wizard dialog boxes to enable users to configure the transport provider's settings.
- Providing message delivery reports to client applications.

Implementing Message and Recipient Options with Transport Providers

When the MAPI spooler logs onto a transport provider, it calls the the transport object's **IXPLogon::RegisterOptions** method. This gives the transport provider a chance to inform the MAPI spooler about any message or recipient options it supports.

MAPI client applications call one or more special callback functions you define in a DLL (probably in the same DLL as your transport provider) to handle any message or recipient options defined in your transport provider at the time messages are composed or submitted. The transport provider's **RegisterOptions** method defines and returns some arbitrary data to MAPI; this data is passed to the callback functions later when the client calls the **IMAPISession::MessageOptions** method. Your transport provider can use this data to indicate to the callback what sort of option is being processed.

The transport provider's **RegisterOptions** method should return an array of **OPTIONDATA** structures describing the options supported by the transport provider. Each structure should describe one option.

Depending on how your transport provider uses the **lpszDLLName** and **ulOrdinal** members of the **OPTIONDATA** structure, you can implement your callbacks as one function which handles all options (and makes use of **lpbOptionsData**) or as many functions which handle one option each (and probably do not use **lpbOptionsData**).

For more information, see **IXPLogon::RegisterOptions**, **OPTIONCALLBACK**, and **OPTIONDATA**.

Implementing Security with Transport Providers

If the messaging system requires it, the transport provider is responsible for implementing an appropriate level of security for access to the messaging system. Each incoming or outgoing message sent through a transport provider by the MAPI spooler is handled in the context of a provider logon session. The transport provider can display a logon dialog box to the user that prompts for a user's credentials before establishing such a connection. Alternatively, the transport provider can store the user's previously entered credentials in the secure property range within a profile section and use them for access without prompting.

When implementing your transport provider's security, consider the following:

- With multiple installed service providers, there can be a multitude of names and passwords associated with a user.
- MAPI allows multiple sessions with multiple identities. Providers are encouraged to support multiple sessions but are not required to do so.
- Each session with a transport provider is associated by MAPI with a discrete section in the user's profile. The transport provider can use the **IMAPISupport::OpenProfileSection** method to gain access to this section, which can be used to store any information associated with this session, including credentials.
- With multiple installed transport providers, it is not necessarily true that the user only has a single e-mail address. A user can have a separate e-mail address for each installed transport provider, or a different address for each session on a single provider.

For more information on storing credentials in profile sections, see Message Services and Profiles, and the **IProfSect : IMAPIProp** interface.

Displaying Configuration Property Sheets with Transport Providers

Transport providers use the **IMAPISupport::DoConfigPropSheet** method to implement configuration property sheets. When calling **DoConfigPropSheet**, the transport provider passes in a pointer to an array of properties along with information about how to display them. MAPI then presents the properties to the user by means of a standard dialog box. You are strongly encouraged to use this property sheet mechanism when implementing your transport provider due to the benefit to the user of a consistent interface.

Transport providers can use the **BuildDisplayTable** function to simplify construction of a display table for use with **DoConfigPropSheet**. Transport providers targeted at the Win32 platform only can also use the property sheet API directly. To buffer changes to the properties, transport providers can use the **CreateIProp** function. This simplifies the handling of cancellations by the user while the user modifies the values stored in the properties. If necessary, a transport provider can also provide a wizard dialog box to simplify configuration tasks for the user.

Sending Message Delivery Reports with Transport Providers

Some underlying messaging systems support delivery reports and some do not. How the transport provider determines whether message delivery or nondelivery reports can be sent to client applications is an implementation detail specific to individual transport providers. If delivery reports can be sent to client applications, transport providers use the **IMAPISupport::StatusRecips** method to notify MAPI of successful or unsuccessful delivery for one or more recipients. MAPI then generates delivery or nondelivery reports corresponding to those recipients. Transport providers can also translate incoming delivery and nondelivery reports that are native to the messaging system into MAPI delivery and nondelivery reports by means of **StatusRecips**.

Developing a TNEF-Enabled Transport Provider

To promote interoperability between messaging systems that support different sets of MAPI features, MAPI provides the Transport Neutral Encapsulation Format (TNEF) as a standard way to transfer data. This format encapsulates MAPI properties not supported by an underlying messaging system into a binary stream that can be transferred along with the message when a transport provider sends it. The transport provider that receives the message can then decode the binary stream to retrieve all the properties of the original message and make them available to client applications. The operational model for TNEF is:

- Messaging clients submit and receive messages to a TNEF transport as normal.
- The transport separates the properties on outgoing messages into two categories: those that the underlying message system supports and those that it doesn't. The values of the properties that are supported by the underlying messaging system are translated into the required format.
- The transport uses MAPI's TNEF methods to encode any unsupported properties into a single data stream. The transport then turns that data stream into a special attachment on the outgoing message, using the underlying messaging system's attachment model, before sending the message.
- A TNEF enabled transport that receives such a message does two things. First, it translates the incoming message's properties – the ones supported by the underlying message system – into MAPI properties. Second, if the special attachment is present, it uses MAPI's TNEF methods to retrieve additional MAPI properties from the attachment before delivering the message to a client application.

MAPI supplies an implementation of the **ITnef** interface for use by MAPI transport providers when working with TNEF objects. The **OpenTnefStreamEx** function is used to create TNEF objects and associate them with a message. TNEF streams are built on top of the OLE **IStream** interface. For more information on OLE and the **IStream** interface, see the *OLE Programmer's Reference*.

Note You use **OpenTnefStreamEx** to create TNEF objects. The old **OpenTnefStream** function still exists for compatibility with old source code and should not be used in anything new.

The **ITnef** interface provides the following methods:

AddProps
EncodeRecips
ExtractProps
Finish
FinishComponent
OpenTaggedBody
SetProps

The MAPI TNEF implementation model supports:

- All MAPI properties without affecting other message properties.

  In order for MAPI messages to survive transport through a messaging system, all properties that cannot be encoded as properties of the messaging system must be encapsulated. Because it is almost never known at the time a message is sent whether or not a MAPI-compliant client will receive the message, the encapsulation scheme allows a transport provider to encode only those MAPI message properties that the messaging system does not natively support. This means that messages which use TNEF are not "opaque" to messaging systems that are not based on MAPI such as SMTP-based UNIX messaging systems. These systems receive the properties they support in whatever manner is typical for them, and other properties are received as an encoded TNEF data stream. The TNEF transport provider is responsible for differentiating between these two sets of properties and sending the supported set in the proper manner for the messaging system.

TNEF makes no assumptions as to the level of support provided by a messaging system. However, in the examples of TNEF usage included in this section, the assumption is made that the messaging system supports at least one single attachment aside from the message. In some cases, the attachment can only be supported through a uuencoded stream and transmitted as part of the message text. Only in very rare circumstances will the messaging system have so little support for message properties that full TNEF encoding of all properties is necessary.

- Workgroup Mail encapsulations.

  The Mail Client for Windows for Workgroups Version 3.1 implements a minimal TNEF encapsulation of some special properties. The MAPI TNEF scheme and underlying formats are backward compatible with this encapsulation mechanism.

- A mechanism for determining whether a TNEF stream on an incoming message belongs to the message based on the MAPI property PR_TNEF_CORRELATION_KEY.

  This property should be found both in the TNEF stream and in an appropriate message header. If the property has the same value in both places, or is missing in either place, the TNEF stream is assumed to belong to the message. Otherwise, the TNEF stream is ignored. TNEF enabled transports are responsible for choosing a value for this property on outbound messages and encoding it in an appropriate message header (for example, the Message-ID: header for SMTP messages) and in the TNEF stream.

TNEF Processing

The following series of actions describe how transports use TNEF methods to process outgoing and incoming messages.

To send a message that includes a TNEF stream

1. Process the message properties that are supported by the messaging system.
2. Mark the message in an implementation specific way so that the receiving transport provider can determine that the message requires TNEF processing. For example, a TNEF transport provider sending to an SMTP messaging system might add a custom header field like "X-CONTAINS-TNEF" to indicate that the message contains TNEF data.
3. Obtain a TNEF object and uses it to encapsulate the message properties not supported by the messaging system into a TNEF stream.
4. Encode the TNEF stream using the messaging system's attachment model. For example, if the underlying attachment model is to uuencode attachments and append them to the message text, then the transport provider must uuencode the TNEF stream into another attachment.

   The transport provider must also implement a method for recognizing which attachment contains the encoded TNEF stream when it receives a message. The standard way to mark this attachment is to give it an attachment filename of "WINMAIL.DAT". If your transport provider does this, any other TNEF-enabled transport providers that follow this convention will be able to interoperate with it.
5. Use **ITnef** interface methods to insert tags describing the positions of message attachments in the message text.
6. Access the tagged message text through OLE **IStream** methods, and send it to the messaging system.

To retrieve encapsulated properties

1. Write the properties supported by the messaging system into a new message, including the tagged message text that contains the encapsulated properties.
2. Decode the TNEF stream from the proper attachment.
3. Decode any other attachments and write them to new MAPI attachments on a message.
4. Open the TNEF stream for decoding using the **OpenTnefStreamEx** function.
5. Use the **ITnef::ExtractProps** method to decode the TNEF stream and write the encapsulated properties into the new message. Any encoded properties that are duplicates of nonencoded properties will overwrite them when the encoded properties are decoded.
6. Use the **ITnef::OpenTaggedBody** method to parse the message text to recover attachment positions from the tags in the message text.

Encoding a Message with TNEF

When a message is submitted, the transport provider can create a file that is used to contain the message during transmission. Next, an OLE **IStream** interface is wrapped around the file. The transport provider then uses **ITnef** methods to write the message properties to the stream in a tagged format that enables the properties to be easily decoded by the receiving transport providers.

To represent an entire message in a single file

1. Obtain a TNEF object by passing an **IStream** object and a message into the **OpenTnefStreamEx** function.
2. Get a list of all defined properties for the message by calling the **IMAPIProp::GetPropList** method.
3. Use **IMAPIProp** methods to exclude all properties supported by the messaging system. At an appropriate time write those properties to the messaging system in the format required by the messaging system.
4. Call **ITnef::AddProps** to encode the remaining properties, including all attachments.
5. Call the **ITnef::Finish** method to encode the message into the TNEF stream after all the requested properties are added.
6. Call the **ITnef::OpenTaggedBody** method to obtain the tagged message text. This tagged text is written out to the messaging system using methods from the OLE **IStream** interface.
7. Call the **IUnknown::Release** method to release the **ITnef** object.

To process an inbound TNEF message

1. Get a MAPI message object from the MAPI spooler and write message header properties into the new MAPI message.
2. Create and initialize an **IStream** object to contain the TNEF data from the inbound message.
3. Passe the MAPI message and the **IStream** object to the **OpenTnefStreamEx** function.
4. Decode the information in the TNEF data by calling the **ITnef::ExtractProps** method. It is important to note that anything decoded by **ExtractProps** will overwrite properties decoded from the incoming message's envelope. That is, extracted TNEF properties will overwrite the existing properties in a message.
5. Process the tagged message text by calling **ITnef::OpenTaggedBody** and the text is parsed to recover attachment positions.
6. Save the message by calling **IMAPIProp::SaveChanges**.
7. Release the TNEF object by calling the **IUnknown::Release** method.

Custom Processing with TNEF

Transport providers can use custom processing to process the properties on an attachment itself, transmit attachments separately, or transmit them through the messaging system's attachment model. TNEF uses a mechanism that enables the transport provider to send the attachments apart from the message and reconnect them on the receiving side.

Sending Messages with TNEF Custom Attachment Processing

To customize attachment processing when sending a message

1. Obtain a TNEF object by passing an **IStream** object and a message into the **OpenTnefStreamEx** function.
2. Get a list of all defined properties for the message by calling the **IMAPIProp::GetPropList** method.
3. Use **IMAPIProp** methods to exclude all properties supported by the messaging system. At an appropriate time write those properties to the messaging system in the format required by the messaging system.
4. Call the **ITnef::AddProps** method to add only the properties on the message − that is, none of the properties on the attachments − by setting the TNEF_PROP_MESSAGE_ONLY flag.
5. Call **ITnef::AddProps** with these items: the TNEF_PROP_EXCLUDE flag, a property tag array that contains the PR_ATTACH_DATA_BIN or PR_ATTACH_DATA_OBJ property, and an attachment identifier that specifies the attachment to be processed.
6. Use the **ITnef::SetProps** method to add the PR_ATTACH_TRANSPORT_NAME property tag with a unique string that identifies the attachment to the messaging system if the attachment has a filename that the messaging system cannot support. For example, multiple attachments with the same original filename, or a filename that is not a valid filename for the messaging system. This string will be used with a key number when writing the attachment tags in the tagged message text to associate an attachment with its data. See, TNEF Tagged Message Text.
7. Repeat the **AddProps** and **SetProps** calls for each attachment.
8. Call the **ITnef::Finish** method to encode the message into the TNEF stream after all the requested properties are added.
9. Obtain the tagged message text by calling the **ITnef::OpenTaggedBody** method. This tagged text is read using methods from the **IStream** interface, encoded using the messaging system's attachment model, and written out to the messaging system.
10. Call the **IUnknown::Release** method to release the **ITnef** object.
11. Write the remaining attachments to the messaging system through the messaging system's attachment model.

It is highly recommended that your transport provider use the method just described to process attachments. If that is not possible, a second method for customized attachment processing is available:

The transport provider ensures that the PR_ATTACH_TRANSPORT_NAME properties of all the attachments contain unique values that are valid attachment identifiers for the messaging system. The transport provider then uses a single call to **ITnef::AddProps** for each attachment, passing in the TNEF_PROP_CONTAINED flag.

Receiving Messages with TNEF Custom Attachment Processing

To receive a TNEF message with customized attachment processing

1. Import all the transmittable properties – those that the messaging system supports – from the incoming message into a new MAPI message. This includes the message text, which contains the TNEF data stream.
2. Identify and decode the special attachment that contains the TNEF stream.
3. Extract all the attachments from the incoming message into MAPI attachments on the new MAPI message. The recovered filenames, or other identifying markers on the attachments, should be placed into the PR_ATTACH_TRANSPORT_NAME property of the new attachments so that the **ExtractProps** method can later associate the correct attachment with the attachment tags encoded in the message text.
4. Create an OLE **IStream** object to wrap around the decoded TNEF stream and use that object along with the new MAPI message in a call to the **OpenTnefStreamEx** function.
5. Call the **ITnef::ExtractProps** method to recover the nontransmittable properties on the message from the TNEF data stream.
6. Call the **ITnef::OpenTaggedBody** method with the MAPI_CREATE and MAPI_MODIFY flags set. This call removes the attachment tags from the message text and converts them into attachment position information in the MAPI message.
7. Deliver the message through the MAPI spooler.

TNEF Tagged Message Text

Tagged message text is used by TNEF to resolve attachment positions in the parent message. This is done by adding a place holder into the message text at the position of the attachment. This place holder, or attachment tag, describes the attachment in such a way that TNEF knows how to resolve the attachment and its position. The tags are formatted as follows:

```
[[ <Object Title> : <KeyNum> in <Stream Name> ]]
[[ <File Name> : <KeyNum> in <Transport Name> ]]
```

The <*Object Title*> and <*File Name*> are variables containing values that are taken from the attachment itself. In cases where these values are not available, the title is defaulted by TNEF based on the attachment type.

The <*KeyNum*> variable contains the textual representation of the attachment key assigned to the attachment by TNEF. The base value of the key is passed into the **OpenTnefStreamEx** call. The base value must not be zero and should not be the same for every call to **OpenTnefStreamEx**. It should suffice to use pseudo random numbers based on the system time from whatever random number generator your run-time library provides, as long as you guarantee that they are never zero.

The <*Transport Name*> variable contains either the stream name passed into the **OpenTnefStreamEx** call or the value of the PR_ATTACH_TRANSPORT_NAME property.

Note The PR_ATTACH_TRANSPORT_NAME property and the <*Transport Name*> variable in a message text tag have nothing to do with the name of the transport provider you are implementing. These items represent the name of an attachment for the transport provider and messaging system.

The message text is tagged when a transport provider asks for a tagged message text by calling the **ITnef::OpenTaggedBody** method. When reading from the tagged text stream, TNEF replaces the single character that was in the message text at the index provided in the PR_RENDERING_POSITION property with the appropriate tag. When writing to the tagged text stream, TNEF checks the incoming data for tags, finds the associated attachment, and replaces the tag with a single space character.

Note that by using tagged message bodies, a transport provider can preserve the positioning of attachments regardless of most changes made to the message text by messaging systems.

Encoding Recipient Tables with TNEF

The encoding of a recipient table into the TNEF stream is rarely necessary since most messaging systems support recipient lists directly. In general, the recipient properties are transmitted in the message header. When inclusion of the recipient table is necessary, TNEF can encode the recipient table as a part of its usual processing. This is done during the initial phase of TNEF processing. The transport provider can include the message's recipient table by calling the **ITnef::AddProps** method with the PR_MESSAGE_RECIPIENTS property specified in the inclusion list. TNEF gets the recipient table from the message, queries the column set, and processes every row of the table into the TNEF stream.

An alternate method is available if the transport provider needs to modify the recipient table before it is encoded. The transport provider can construct the necessary table and then call the **ITnef::EncodeRecips** method. If NULL is passed in the *lpRecipTable* parameter, then the recipient table is taken directly from the message as described for **ITnef::AddProps**.

Developing a Remote Transport Provider

Some client applications benefit from having a transport provider with remote capabilities, including:

- Clients whose underlying messaging systems are slow.
- Clients with low bandwidth connections to their messaging servers.
- Clients with intermittant connections to messaging systems or networks.
- Clients that must, for whatever reasons, minimize the time spent connected to their message servers.

The most common situation where remote transport providers are used is when a client's connection to the messaging server is over a modem (low bandwidth) or is connected through a long-distance telephone call.

MAPI provides a definition of a remote interface that transport providers implement in order to satisfy clients' needs in these circumstances. That interface is the **IMAPIFolder** interface, which enables a user to preview the contents of the messaging server without downloading entire messages or folders. This is the same interface that message store providers implement. By having a transport provider implement a special MAPI folder, it can provide access to messaging systems with special connection requirements.

The following topics describe the remote transport architecture and provide information for implementing a remote transport provider.

Remote Transport Architecture

Your remote transport needs to translate user requests for message data into intelligently managed connections to the messaging server without requiring the user to manage the individual connections. This is accomplished through the MAPI remote architecture as shown in the following illustration.

{ewc msdncd, EWGraphic, groupx838 0 /a "MAPI.WMF"}

Client applications which are remote-enabled (remote viewers) communicate with the MAPI spooler and remote transport providers as usual for non-remote tasks such as logging on or flushing queues. In addition, the client application uses the remote transport's **IMAPIFolder** interface to perform remote tasks such as downloading message headers.

The remote viewer is a client application designed to work with remote transport providers. Remote viewer applications can check the MAPI status table for remote transport providers, those that have set the STATUS_REMOTE_ACCESS bit in their status table row, and adjust their user interface appropriately. Remote viewers are expected to utilize a special folder that contains header information about messages in the user's Inbox. The header information is stored locally and can be used to selectively mark messages for download or deletion.

A remote transport provider is similar to a message store because implements the **IMAPIFolder** interface; however, the folder so implemented is not identical to the generalized heirarchical folders that message stores implement. For a remote transport provider, the folder is an intermediary between the client application and the remotely accessed message store. The message headers and message identifiers in the folder do not necessarily correspond to messages that are stored locally on the user's computer or stored in any immediately accessible way.

The folder is expected not to have subfolders, although the presence of subfolders should not cause client application problems since they should not be displayed. Opening the folder does not cause the folder's contents to be downloaded. At the user's request, message headers can be downloaded and looked at off-line or at a later time. Remote transport providers should provide some sort of local cache or storage facility for downloaded message headers so that the folder can be populated with cached message headers when it is opened without connecting to the remote message store.

Advertising a Remote Transport

Remote viewer applications need a way to identify which transport providers support remote functionality. In order to advertize this ability to remote viewers, remote transport providers must set the STATUS_REMOTE_ACCESS bit in the PR_STATUS_CODE property and the STATUS_VALIDATE_STATE bit set in the PR_RESOURCE_METHODS property of the transport provider's row in the status table.

Remote transport providers must also include in the status row the PR_HEADER_FOLDER_ENTRYID property. This property indicates the entry identifier of the folder that is used for downloading by remote viewer applications. This entry identifier is registered with MAPI when the transport provider is initialized. When the user opens the folder, the remote viewer application calls the **IMAPISession::OpenEntry** method passing in the folder's entry identifier. MAPI then forwards the call to the transport provider which registered the folder's entry identifier. The transport should return a view of the contents of the folder. This folder's contents table should contain the list of messages in the folder.

Required Status Row Properties for Remote Transports

Remote transport providers must supply the following properties when their initial row in the MAPI status table is created:

- PR_DISPLAY_NAME
- PR_HEADER_FOLDER_ENTRYID
- PR_REMOTE_PROGRESS
- PR_REMOTE_VALIDATE_OK
- PR_RESOURCE_METHODS
- PR_RESOURCE_PATH
- PR_PROVIDER_DISPLAY
- PR_STATUS_CODE
- PR_STATUS_STRING

Additional Remote Transport Functionality

To support as many remote viewer requests as possible, your remote transport provider should be able to:

- Defer message delivery.
- Dynamically update the PR_REMOTE_PROGRESS and PR_REMOTE_PROGRESS_TEXT properties while a remote operation is in progress.
- Support the PR_REMOTE_VALIDATE_OK property.

Implementing the IMAPIFolder Interface for Remote Transports

Since the folder that remote transport providers implement is not a fully functional MAPI folder, remote transport providers do not have to implement all of the methods in the **IMAPIFolder** interface. You should derive a C++ class from the **IMAPIFolder** interface to implement the folder. Because **IMAPIFolder** inherits from the **IMAPIContainer**, **IMAPIProp**, and **IUnknown** interfaces, there are some methods from those interfaces that need to be implemented as well. This topic describes which methods remote transports need to implement, as well as any special considerations that are not described in the MAPI Programmer's Reference documentation for those methods. Methods from those interfaces that are not described here should be implemented as stubs which return MAPI_E_NO_SUPPORT.

IMAPIFolder is a pure virtual interface, containing no data members or pre-defined methods, only method declarations. You must implement basic functionality such as a constructor and destructor, reference counting, and declaring and defining any needed member variables. Your class's declaration should include data members for the folder's contents table, a pointer to the logon object and status object (the parent objects of the folder), and a reference counter.

Since the **IMAPIFolder** interface itself does not define any mechanism for filling the contents table with message headers, you will need to define custom methods to do this. The minimum implementation of this interface has read-only functionality. You will need to implement additional methods to give your implementation read-write functionality.

Implementing the IMAPIFolder Constructor for Remote Transports

The remote transport provider's **IMAPIStatus** object will call the folder's constructor with a pointer to the status object and a pointer to the transport provider's **IXPLogon** object. The constructor needs to perform the following MAPI-specific tasks:

- Store the status object pointer and logon object pointer in data members.
- Call the **AddRef** methods on the status object and logon object.
- Initialize the folder's contents table to NULL.
- Initialize the folder object's reference count mechanism.

As usual, the constructor should take care of initializing any implementation-specific data members.

Implementing the IMAPIFolder Destructor for Remote Transports

The folder's destructor needs to perform the following MAPI-specific tasks:

- If the folder still has a contents table, call the table's **Release** method and set the pointer to the table to NULL.
- Call the **Release** method on the status object and logon object, and set the pointers to those objects to NULL.

As usual, the destructor must also free any additional memory allocated by the folder.

Implementing IMAPIFolder::SetMessageStatus for Remote Transports

Your implementation of this method must follow the semantics described in the documentation of **IMAPIFolder::SetMessageStatus**. There are no special considerations with respect to this method for remote transport providers. Clients use this method to set the MSGSTATUS_REMOTE_DOWNLOAD and MSGSTATUS_REMOTE_DELETE bits to indicate that a particular message is to be downloaded or deleted from the remote message store.

Note You do not have to implement the related **GetMessageStatus** method. Clients must look in the folder's contents table to determine the status of a message.

Implementing the IMAPIContainer Interface for Remote Transports

The only method from the **IMAPIContainer** interface that must be fully implemented is **GetContentsTable**. The remaining methods can return MAPI_E_NO_SUPPORT.

Implementing IMAPIContainer::GetContentsTable for Remote Transports

Your implementation must return a pointer to an **IMAPITable** interface in the *ppTable* parameter passed into the **GetContentsTable** method. If your transport provider has an existing contents table, it suffices to return a pointer to it. If not, then this method must create a new **IMAPITable** object, populate the table with message headers (if any are available), and return a pointer to the new table. The **ITableData** method **HrGetView** is useful for generating a return value and storing the table pointer in the *ppTable* parameter. The contents table must support at least the following property columns:

PR_ENTRYID	PR_SENDER_NAME
PR_SENT_REPRESENTING_NAME	PR_DISPLAY_TO
PR_SUBJECT	PR_MESSAGE_CLASS
PR_MESSAGE_FLAGS	PR_MESSAGE_SIZE
PR_PRIORITY	PR_IMPORTANCE
PR_SENSITIVITY	PR_MESSAGE_DELIVERY_TIME
PR_MSG_STATUS	PR_MESSAGE_DOWNLOAD_TIME
PR_HASATTACH	PR_OBJECT_TYPE
PR_INSTANCE_KEY	PR_NORMALIZED_SUBJECT

Implementing the IMAPIProp Interface for Folder Objects for Remote Transports

Remote transport providers must implement only four methods from the **IMAPIProp** interface in addition to the usual stubs for the other methods in that interface. The required methods are **GetLastError**, **SaveChanges**, **GetProps**, and **GetPropList**.

Implementing IMAPIProp::GetLastError for Folder Objects for Remote Transports

As described in the documentation of **IMAPIProp::GetLastError**, this method takes an HRESULT representing the last error encountered and returns a **MAPIERROR** structure which contains information to display to the user regarding the error. The specifics of this implementation and what messages this method returns are up to you, since the particular error conditions which lead to various HRESULT values will be different for different transport providers.

Implementing IMAPIProp::SaveChanges for Folder Objects for Remote Transports

Whether you provide a functional implementation of this method is optional and will probably depend on other design choices in your implementation. If you wish to implement this method, do so according to the documentation of **IMAPIProp::SaveChanges**. Since folder objects are not transacted, at a minimum your implementation of **SaveChanges** must return S_OK without actually doing any work.

Implementing IMAPIProp::GetProps for Folder Objects for Remote Transports

The **GetProps** method must return the folder's property values for properties requested by the caller. As specified in the documentation of **IMAPIProp::GetProps**, your implementation needs to:

- Allocate a property value array to return to the caller and store its address in the property value pointer parameter passed in for that purpose.
- Copy the property tags from the folder's properties into the property tags in the property value array according to the array of property tags passed to **GetProps**.
- Ensure that the property type is set for all property tags passed to **GetProps**. The caller can pass in a property type of PT_UNSPECIFIED, in which case **GetProps** must set the correct property type for that property tag.
- Set the value of each property in the property value array according to its tag. For example, if the property tag requested by the caller is PR_OBJECT_TYPE, **GetProps** can set the value to MAPI_FOLDER.
- If the caller passes in any property tags that your implementation does not handle, you can set the property tag to PT_ERROR for those properties, and set the property value to MAPI_E_NOT_FOUND.
- Return S_OK if no errors occurred or MAPI_W_ERRORS_RETURNED if there were errors.

Your implementation of the **GetProps** method needs to support the following properties at a minimum:

PR_ACCESS	PR_ACCESS_LEVEL
PR_CONTENT_COUNT	PR_ASSOC_CONTENT_COUNT
PR_FOLDER_TYPE	PR_OBJECT_TYPE
PR_SUBFOLDERS	PR_CREATION_VERSION
PR_CREATION_TIME	PR_DISPLAY_NAME
PR_DISPLAY_TYPE	

Implementing IMAPIProp::GetPropList for Folder Objects for Remote Transports

The **GetPropList** method can be implemented exactly as specified in the documentation of **IMAPIProp::GetPropList** . There are no special concerns for remote transport providers. Your implementation should, of course, return the same list of properties as supported by the **GetProps** method.

Implementing the IUnknown Interface for Folder Objects for Remote Transports

The **IUnknown** interface is an OLE interface. Because MAPI adheres to the OLE component object model, your transport provider must implement a few methods from **IUnknown**.

Implementing IUnknown::QueryInterface for Folder Objects for Remote Transports

The semantics of the **QueryInterface** method that your implementation should follow are described in the OLE documentation. If the interface identifier passed into **QueryInterface** is one of IID_IMAPIFolder, IID_IMAPIContainer, or IID_IMAPIProp, then a pointer to the folder object should be passed back to the caller in the parameter provided for that purpose, the reference count of the folder must be incremented, and the method should return S_OK.

If the interface identifier passed in is IID_IUnknown, then a pointer to the transport provider's status object should be returned, according to the OLE common object model (COM) rules, and the status object's **AddRef** method should be called before **QueryInterface** returns S_OK.

If the interface identifier is anything else, **QueryInterface** should return E_NOINTERFACE.

Implementing IUnknown::Release for Folder Objects for Remote Transports

The **Release** method should do the following things:

- Decrement the reference count for the folder.
- If there are no remaining external references to the object − that is, the folder object contains its only remaining reference − and the transport provider caches the folder's contents table, this is the time to save the contents table to whatever external resource is appropriate.
- If the reference count is zero, the folder should delete itself.

Implementing the IMAPIStatus Interface for Remote Transports

You should derive a C++ class from the **IMAPIStatus** class to implement this interface. Because **IMAPIStatus** inherits from the **IMAPIProp** and **IUnknown** classes, there are some methods from those interfaces that need to be impemented as well. This topic describes which methods remote transports need to implement, as well as any special considerations that are not described in the MAPI Programmer's Reference documentation for those methods. Methods from those interfaces that are not described here should be implemented as stubs which return MAPI_E_NO_SUPPORT.

Implementing the IMAPIStatus Constructor for Remote Transports

The constructor for your implementation of the **IMAPIStatus** interface must do the following:

- Store the pointer to the **XPLogon** object passed in the first parameter to the constructor.
- Store the pointer to the profile section object passed in the second parameter to the constructor.
- Initialize the status object's reference count mechanism.
- If the profile section pointer is not NULL, call the **AddRef** method on the profile section object.

Implementing the IMAPIStatus Destructor for Remote Transports

The only thing that the destructor for your implementation of the **IMAPIStatus** interface must do is call the **Release** method on the profile section object if the profile section pointer is not NULL.

Implementing IMAPIStatus::ChangePassword for Status Objects for Remote Transports

The **ChangePassword** method is used to support provider specific passwords. It should be implemented as specified in the documentation for **IMAPIStatus::ChangePassword**. There are no special concerns for remote transport providers.

Implementing IMAPIStatus::FlushQueues for Status Objects for Remote Transports

The **FlushQueues** method for remote transport providers sets bits in the PR_STATUS_CODE property in the logon object's status row to control how queues are flushed. If a remote viewer passes in the FLUSH_UPLOAD flag, the **FlushQueues** method should set the STATUS_INBOUND_ENABLED and STATUS_INBOUND_ACTIVE bits. If a remote viewer passes in the FLUSH_DOWNLOAD flag, the **FlushQueues** method should set the STATUS_OUTBOUND_ENABLED and STATUS_OUTBOUND_ACTIVE bits. **FlushQueues** should then return S_OK. The MAPI spooler will then initiate the appropriate actions to upload and download messages.

Implementing IMAPIStatus::SettingsDialog for Status Objects for Remote Transports

If your transport provider has any settings, it should do the following:

- Open the transport provider's profile section.
- Get the transport provider's property settings from the profile.
- Display the property settings in a dialog box.
- If the dialog box allows editing of the property settings, check that the new settings are valid and store them back in the transport provider's profile section.
- Return S_OK, or any error values returned during the preceeding steps.

Implementing IMAPIStatus::ValidateState for Status Objects for Remote Transports

The **ValidateState** method is called by remote client applications to start remote processing for various actions. This method exists primarily to set status bits to communicate with the MAPI spooler rather than to actually do any work. Typically, the transport provider sets flags in its status row which indicate to the MAPI spooler what actions need to be initiated to complete the client's request. In this model of client-transport-spooler interaction, the actions requested by the client are asynchronous, in that **ValidateState** returns before the requested actions are complete. However, actions which do not necessarily involve the underlying messaging system, or which involve a transport-specific interface, can be synchronous. The client application passes in a bitmask of flags to specify which actions the remote transport provider should take. The flags are:

ABORT_XP_HEADER_OPERATION

If possible, the remote transport provider should cancel any operations involving downloading headers. To do this, the transport provider must set the following property values in the logon object's status row:

- Clear the STATUS_INBOUND_ENABLED and STATUS_INBOUND_ACTIVE bits in the PR_STATUS_CODE property to tell the MAPI spooler to cease the inbound flush process for this transport provider.
- Set the STATUS_OFFLINE bit in the PR_STATUS_CODE property.
- Set the PR_REMOTE_VALIDATE_OK property to TRUE.
- Set the PR_STATUS_STRING property to some string indicating the transport provider's status to the user.
- Return S_OK

However, if the operation in progress cannot be canceled, **ValidateState** should return MAPI_E_BUSY.

FORCE_XP_CONNECT

A remote transport provider should never establish a connection to a shared resource, for example, a modem or COM port, outside the context of the MAPI spooler-transport interaction involved in the **IXPLogon::FlushQueues** method. If **ValidateState** is called with this flag, your transport provider should do the following:

- Set some internal status flag to indicate that the remote connection needs to be established when the **FlushQueues** method is called.
- Set the proper values in the status table to cause the MAPI spooler to initiate the queue flushing process.
- When flushing of queues has completed, release the shared resource.
- Clear the STATUS_OFFLINE bit in the PR_STATUS_CODE property.
- Return S_OK.

FORCE_XP_DISCONNECT

The remote transport provider should release its connection to the messaging system resources. After doing so, it should:

- Set the STATUS_OFFLINE bit in the PR_STATUS_CODE property.
- Return S_OK.

PROCESS_XP_HEADER_CACHE

The remote transport provider should process remote messages and upload any messages that have been deferred. To do this the transport provider must set the following property values in the logon object's status row:

- Set the PR_STATUS_STRING property to a string indicating the transport provider's status to the user.

- Set the STATUS_OUTBOUND_ENABLED and STATUS_OUTBOUND_ACTIVE bits in the PR_STATUS_CODE property.
- Set the PR_REMOTE_VALIDATE_OK property in the transport provider's status row to FALSE.
- If another operation is in progress (such as downloading headers) when **ValidateState** is called, **ValidateState** should return MAPI_E_BUSY.
- Execute the code for processing the REFRESH_XP_HEADER_CACHE flag as well to satisfy requirements of the Microsoft Exchange client.

REFRESH_XP_HEADER_CACHE
The remote transport provider should retrieve any new message headers from the messaging system. To do this, the transport provider must:

- Set the PR_STATUS_STRING property to a string indicating the transport provider's status to the user.
- Set the STATUS_INBOUND_ENABLED and STATUS_INBOUND_ACTIVE bits in the PR_STATUS_CODE property.
- Clear the STATUS_OFFLINE bit in the PR_STATUS_CODE property.
- Set the STATUS_ONLINE bit in the PR_STATUS_CODE property.
- Set the PR_REMOTE_VALIDATE_OK property in the transport provider's status row to FALSE.
- Return MAPI_E_BUSY.

SHOW_XP_SESSION_UI
If your transport provider has any pieces of user interface for processing the message headers − such as a dialog box confirming downloading of messages − then that dialog box should be displayed. Otherwise, **ValidateState** can return MAPI_E_NO_SUPPORT.

If any flags other than these are passed in, **ValidateState** should return MAPI_E_UNKNOWN_FLAGS.

The client's call to the transport provider will often be to the **IMAPIStatus::ValidateState** method. It is important during the processing of **ValidateState** that the transport provider not perform any actions which allocate scarce system resources, such as a modem or COM port. The reason for this is because the MAPI spooler will, at times, need to flush queues on more than one transport provider. However, the client is allowed to call any transport provider's **ValidateState** method at any time. If your transport provider attempts to allocate a scarce resource during the processing of **ValidateState**, an error can result due to conflict with another transport provider that the MAPI spooler has instructed to flush its queues. If you allow all scarce resource allocations to happen under the direction of the MAPI spooler, such conflicts will be avoided. Your transport provider should support the PR_REMOTE_VALIDATE_STATE_OK property so that client applications can detect when your transport provider is busy or waiting for the MAPI spooler to initiate an action.

Implementing the IMAPIProp Interface for Status Objects for Remote Transports

Since remote transport providers, like other MAPI service providers, use the MAPI status table, they must implement some methods from the **IMAPIProp** interface to support client requests for property information.

Implementing IMAPIProp::GetLastError for Status Objects for Remote Transports

As described in the documentation of **IMAPIProp::GetLastError**, this method takes an HRESULT representing the last error encountered and returns a **MAPIERROR** structure which contains information to display to the user regarding the error. The specifics of this implementation and what messages this method returns are up to you, since the particular error conditions which lead to various HRESULT values will be different for different transport providers.

Implementing IMAPIProp::SaveChanges for Status Objects for Remote Transports

Because status objects – like folder objects – are not transacted, the **SaveChanges** method is essentially a no-op, and your implementation need do no real work. However, because some MAPI-compliant clients might have code to use the **SaveChanges** method on their status objects, your implementation should return S_OK instead of MAPI_E_NOT_SUPPORTED, in order to avoid returning an error where none is warranted.

Implementing IMAPIProp::GetProps for Status Objects for Remote Transports

The **GetProps** method must return the transport provider's property values for any property tags requested by the caller. As specified in the documentation of **IMAPIProp::GetProps**, your implementation needs to:

- Allocate a property value array to return to the caller and store its address in the property value pointer parameter passed in for that purpose.
- Copy the property tags from the status object's properties into the property tags in the property value array according to the array of property tags passed to **GetProps**.
- Ensure that the property type is set for all property tags passed to **GetProps**. The caller can pass in a property type of PT_UNSPECIFIED, in which case **GetProps** must set the correct property type for that property tag.
- Set the value of each property in the property value array according to its tag. For example, if the property tag requested by the caller is PR_OBJECT_TYPE, **GetProps** should set the value to MAPI_TRANSPORT_PROVIDER.
- If the caller passes in any property tags that your implementation does not handle, you can set the property tag to PT_ERROR for those properties, and set the property value to MAPI_E_NOT_FOUND.
- Return S_OK if no errors occurred or MAPI_W_ERRORS_RETURNED if there were errors.

Implementing IMAPIProp::GetPropList for Status Objects for Remote Transports

The **GetPropList** method can be implemented exactly as specified in the documentation of **IMAPIProp::GetPropList**. There are no special concerns for remote transport providers.

Implementing the IUnknown Interface for Status Objects for Remote Transports

The only method from the **IUnknown** interface that is special for MAPI is the **QueryInterface** method.

Implementing IUnknown::QueryInterface for Status Objects for Remote Transports

The semantics of the **QueryInterface** method that your implementation should follow are described in the *OLE Programmer's Reference*. If the interface identifier passed in is one of IID_IMAPIStatus, IID_IMAPIProp, or IID_IUnknown, then a pointer to the status object should be passed back to the caller in the parameter provided for that purpose, and your status object's reference count must be incremented.

If the interface identifier passed in is IID_IMAPIFolder, then your implementation should return a pointer to the folder that contains the available message headers and increment the reference count for that folder. If no folder is present, one should be created and then returned.

If the interface identifier is anything else, **QueryInterface** should return E_NOINTERFACE.

Background Processing and Allocating Scarce Resources

Transport providers operate largely at the direction of the MAPI spooler. This places remote transport providers under some obligations in order to interact with other MAPI service providers. Remote transport providers should strive to operate in the background in order not to cause client applications to appear inactive. Remote transport providers should also strive to minimize their use of scarce resources, such as modems and COM ports, in order to permit other MAPI service providers and other user applications to use those resources if necessary.

About Background Processing

Whenever possible, your transport provider should operate in the background. The transport provider's actions should be initiated by the MAPI spooler rather than directly by client calls to methods in the transport provider to avoid conflicts with other service providers (especially other transports) that might need to use resources that your transport provider also needs. The sequence of actions is typically like this:

- The client calls a method in the transport provider which requires some action.
- The transport provider sets some internal status flags which note the action requested by the client.
- The transport provider sets one or more special values in the MAPI status table which signal the MAPI spooler that the transport provider needs to perform the requested action.
- The MAPI spooler calls one or more methods in the transport provider to initiate the requested actions.

Guidelines for Allocating Scarce Resources

Transport providers that use scarce resources − such as a modem or COM port − to send and receive messages or message headers need to adhere to the following conventions in order to avoid resource conflicts with other transport providers:

- Transport providers can allocate scarce resources when the **TransportLogon** method is called and retain them until the **RegisterOptions** method returns, at which time the resources must be released.
- Transport providers can allocate scarce resources when the **SubmitMessage** method is called and retain them before returning from the subsequent call to the **EndMessage** method.
- Transport providers can allocate scarce resources for the duration of a call to the **StartMessage** method. The transport provider must release the resources before returning.
- Transport providers that need to keep a resource open across multiple method calls, such as while flushing queues, should signal the MAPI spooler of this need by setting the STATUS_INBOUND_ENABLED or STATUS_OUTBOUND_ENABLED bits in the transport provider's PR_STATUS_CODE property in the MAPI status table.

  When the MAPI spooler next makes a call to a method in the transport provider's logon object the transport provider can allocate the resource. The transport provider can keep the resource until the time that the action, such as flushing queues, is completed. In some cases, the transport provider will already hold the resource before calling the **IMAPISupport::ModifyStatusRow** method to update the transport provider's PR_STATUS_CODE property. This will cause some multi-transport operations to fail, but is sometimes unavoidable. The MAPI spooler will handle errors in such cases.

Developing a Message Store Provider

Like other MAPI service providers, message stores are dynamic-link libraries (DLLs) that present the services of some underlying storage mechanism to MAPI client applications and the MAPI spooler. The message store provider presents the underlying storage mechanism as a hierarchical set of folders and messages that MAPI clients and the MAPI spooler can use.

This diagram shows the basic MAPI message store architecture.

{ewc msdncd, EWGraphic, groupx839 0 /a "MAPI.WMF"}

You can implement a message store provider using any sort of underlying storage mechanism you like. However, performance concerns and the requirement that the underlying storage mechanism must be presented as a hierarchical collection of MAPI objects means that message stores are typically implemented using an existing database product which supports hierarchical storage of objects in the database and which has a programming interface or well defined file structure. For example, Access, SQL, and Oracle databases can be used as the underlying storage mechanism. Some database products have feature sets that make it easier to implement MAPI's features, so your choice of database product may be affected by the features that your message store provider needs to support.

Using an existing database as the underlying storage mechanism saves you work because it is usually easier to present database objects to MAPI clients as MAPI objects than to implement your own hierarchical storage mechanism. Doing this allows you to treat MAPI operations at a higher level than if you implement your own hierarchical storage mechanism. For example, searching for a message with a particular subject line becomes a fairly simple matter of constructing and submitting an appropriate database query rather than a matter of implementing complex routines to search your hierarchical storage mechanism.

Message store providers communicate with MAPI clients and the MAPI spooler to perform operations on folders and objects. The message store provider translates those operations into lower level operations on the underlying storage mechanism. The MAPI spooler typically communicates with the message store provider while sending and receiving messages. MAPI clients typically communicate with message store providers in order to manipulate the folder hierarchy and to read, edit, delete, and send messages.

Both the MAPI spooler and MAPI clients communicate with the message store provider to create new messages. Client applications do this when users compose a message. The MAPI spooler does this when it receives an incoming message. In either case, the new message is usually created in the Inbox folder of the message store, if there is one.

Message store providers make heavy use of MAPI tables, folders, messages, and properties. The implementation details for those objects are documented in Tables, Folders, Messages, and Properties. You should familiarize yourself with that material before attempting to implement a message store provider.

There are two important types of message store providers: those that can act as a user's default message store and those that cannot. A default message store is one in which client applications and the MAPI spooler can perform any messaging task, such as receiving messages or creating folders. A default message store provider must support several more features than the minimum number required for all message store providers.

About the Structure of Message Store Providers

A message store provider, when it is running in memory, is an **IMSProvider : IUnknown** object. The **IMSProvider** interface allows client applications and the MAPI spooler to log on to and off of the message store. The objects that client applications and the MAPI spooler use to access folders and messages in the message store are an **IMSLogon** object and an **IMsgStore** object. These objects are typically created when the message store is first logged on to, although the message store DLL's **MSProviderInit** entry point could also create them.

Because the **IMSLogon** and **IMsgStore** interfaces share some methods, it may be easier to create one class object that inherits from both of these interfaces. You can also implement these interfaces in separate objects, and write helper functions internal to your DLL that implement the shared methods which can then be called from the methods in the **IMSLogon** and **IMsgStore** interfaces.

The following illustration shows a high-level outline of the object hierarchy within a running message store.

{ewc msdncd, EWGraphic, groupx839 1 /a "MAPI.WMF"}

About Required and Optional Interfaces for Message Store Providers

MAPI defines a set of interfaces that relate to message store providers. Because of the wide range of features that a message store can choose to implement, some of these interfaces are required and some are not. The following table lists the MAPI interfaces related to message store providers, whether the interfaces are required or optional, and why they are used.

Interface	Status	Use
IMSProvider	Required	Methods for logging on and off a message store.
IMSLogon	Required	Methods for opening folders or messages, verifying the message store's identity, and handling notifications.
IMsgStore	Required	Methods for opening folders or messages, finding special folders, handling message submissions.
IMAPIFolder	Required	Methods for finding and manipulating messages and sub-folders.
IMessage	Required	Methods for manipulating attachments and setting some of a message's properties.
IMAPITable	Required	Used by other objects to present collections of data to various MAPI components.
IMAPIStatus	Required	Allows clients to validate the state of a message store, and perform some configuration tasks.
IAttach	Optional	Used by message objects if the store provider supports file attachments.
IStorage	Optional	OLE interface used by attachment objects if the store provider supports OLE object attachments.
IStream	Optional	OLE interface used by message and attachment objects.
IStreamDocfile	Optional	OLE interface used by some OLE 2.0 attachment objects.

The basic information you need to implement **IMAPIFolder**, **IMessage**, **IMAPIStatus**, and **IMAPITable** is documented elsewhere in the *MAPI Programmer's Reference*. This section contains supplementary information that is more directly related to message store providers. The rest of the MAPI interfaces should be implemented according to the information in this section and in the *MAPI Programmer's Reference*. See the OLE documentation in the Win32 Software Development Kit for details about implementing **IStorage**, **IStream**, and **IStreamDocFile**.

About Message Store Features

Message store providers are more complex than other MAPI service providers in that message store providers have a wider range of optional features they can implement. The list of required features for a message store provider is fairly short. However, a typical message store provider will support a number of optional features, since many of the optional features are very useful or required by most MAPI clients. The following table lists the major features that message store providers can implement, whether the feature is required or optional for all message store providers and for default message store providers.

Feature:	**All**	**Default**
Providing status with the MAPI status table	Required	Required
Implementing folder objects	Required	Required
Implementing message objects	Required	Required
Read and nonread reports	Required	Required
Progress Interface	Required	Required
Configuration Interface	Required	Required
Supporting associated contents tables for form and view support	Optional	Optional
Sending messages with the message store provider	Optional	Required
Receiving messages with the message store provider	Optional	Required
Supporting message attachments	Optional	Optional
Supporting Rich Text Format for messages	Optional	Optional
Providing notifications	Optional	Optional
Supporting searches	Optional	Optional
Tightly coupled message store/transport providers	Optional	Optional
Non-reuse of entry identifiers	Optional	Optional

Many of the optional features can be advertized to MAPI and client applications by setting various flags in the message store object's PR_STORE_SUPPORT_MASK property. The required features do not have flags associated with them. PR_STORE_SUPPORT_MASK is required on message store, folder, and message objects.

About Providing Status for Message Store Providers

Like all MAPI service providers, message store providers must support the **IMAPIStatus** interface. The methods in that interface are optional, however. Set the PR_RESOURCE_METHODS property in your message store provider's status table to indicate which methods are supported. At a minimum, your message store provider should implement the **IMAPIStatus::ValidateState** and **IMAPIStatus::FlushQueues** methods. For more information, see Status Objects.

Additionally, the message store provider must have a row in the MAPI status table. Whether the message store's identity properties are listed in that row is optional. For more information, see Identity Properties.

About Implementing Folders in Message Stores

A great deal of the information relating to message store providers' support for folders is covered in Folders. You should be familiar with that information before attempting to implement a message store provider. The Folders section is heavily oriented to the MAPI client's perspective; this topic covers additional information that is important from the message store provider's point of view.

About Exposing Folders in Message Stores

Every message store provider must present a top-level **IMAPIFolder** object to client applications. The top-level folder corresponds to the entire message store; it provides access to the folders that users see as the contents of the message store. In addition, the top-level folder is often used as the default receive folder for IPC messages and as the folder from which read reports are sent. Message store providers must also present an IPM subtree, a set of folders used to contain IPM messages, to client applications.

Client applications can open the top-level folder by calling **IMsgStore::OpenEntry** with 0 and NULL for the *cbEntryId* and *lpEntryId* parameters. In most cases, however, client applications open the set of folders containing IPM messages.

To get a list of folders in the message store's IPM folder tree

1. Use your MAPI session to call the **IMAPISession::OpenMsgStore** method.
2. Use the resulting message database pointer to call the **IMAPIProp::GetProps** method for the PR_IPM_SUBTREE_ENTRYID property.
3. Call the **IMsgStore::OpenEntry** method with the entry identifier to get an **IMAPIFolder** pointer.
4. Call the **IMAPIContainer::GetHierarchyTable** method to get a table of the contents of the folder.
5. Call the **IMAPITable::QueryRows** method to list the folders in the top-level folder.

MAPI clients use these folders to access other folder objects and message objects in the message store. **IMAPIFolder**, and its parent interface **IMAPIContainer**, contain the methods your message store provider must implement to populate folders with message objects and respond to clients' requests to operate on messages.

About Inbox and Outbox Folders in Message Stores

In order to be the default message store, a message store provider must implement an Inbox and an Outbox folder. They are typically stored within the IPM subtree of a message store. These folders are special in that they are designated as the folders that messages are delivered to and sent from, but no special functionality is required of them. Sending and receiving messages happens by way of defined call sequences between client applications, the MAPI spooler, and the message store provider. The Inbox and Outbox folders are simply folders that are used to hold messages during those call sequences. The important point is not that the folders are special, or even that they are named Inbox and Outbox; the important point is that the message store provider uses them as part of its support for sending and receiving messages.

To support receiving messages, the message store provider must implement the **IMsgStore::GetReceiveFolderTable** and **IMsgStore::GetReceiveFolder** methods. See About Receiving Messages with Message Store Providers for details.

To support sending messages, the message store provider must support the **IMsgStore::GetOutgoingQueue** method in addition to the other methods used by the MAPI spooler during the message sending process. A message store's outgoing queue does not have to correspond to an actual folder anywhere in the message store's folder tree. However, it is customary for a message store provider to show the contents of the outgoing message queue in the Outbox folder, if there is one. Doing so gives client applications a convenient way to indicate the status of messages that the user has sent, since an Outbox folder can be displayed along with all the other folders in a message store. See About Sending Messages with Message Store Providers for details.

About Special Folders in Message Stores

Special folders such as the Inbox, Outbox, and search-results folder, may be pre-created and protected by the message store provider. If the folders don't exist, MAPI will attempt to create them in the message store by calling the **HrValidateIPMSubtree** function. For more information, see About Special Folders.

About Implementing Messages in Message Stores

The **IMessage** interface is similar to the **IMAPIFolder** interface in that both interfaces derive from the **IMAPIProp** interface. Clients use the **IMAPIProp** methods to access the contents of a message. The **IMessage** interface supplies additional methods for manipulating messages, such as for adding attachments or modifying the recipients of a message. The methods in the **IMessage** interface serve to modify attributes of messages that are not stored directly in the message's properties.

About Creating and Storing Messages in Message Stores

How your message store provider creates and stores messages in the underlying storage mechanism depends heavily on the underlying storage mechanism itself. In general, you need only to write code to preserve the properties of a message and their values.

When creating a new message, the message store provider needs to create it with the required properties for messages. A list of these properties can be found in About Creating Messages. After that, client applications add any additional properties with **IMAPIProp** methods.

When saving a message to the underlying storage mechanism, the message store provider needs to iterate over the message's properties, and save them to the underlying storage mechanism such that they can be fully recovered if the message is later opened.

MAPI requires that the properties on **IMessage** objects are transacted, meaning that changes made to them do not become permanent until the **IMAPIProp::SaveChanges** method is called on the message object. The message store provider is responsible for implementing this behavior. Usually this is not difficult; it simply means holding properties in memory while they are being modified and committing them to the underlying storage mechanism when **SaveChanges** is called.

Some properties on message objects have special semantics for client applications with respect to the **SaveChanges** method:

- Some properties should be read/write before **SaveChanges** is called, but read-only afterwards. For example, PR_MESSAGE_FLAGS is set initially by the client application that creates the message (and thus is read/write) but can't be changed after the first call to **SaveChanges**.
- Some properties have special relations to properties on the folder they are in or to **IMAPIFolder** methods. For example, the PR_MESSAGE_FLAGS property is related to the flags used on the **IMAPIFolder::CreateMessage** call.
- Some properties may not be available until **SaveChanges** is called for the first time. For example, the PR_ENTRYID property may not be available until **SaveChanges** is called.
- Some properties can have special relationships to other properties on the message object. For example, the PR_BODY property is usually derived from the PR_RTF_COMPRESSED property in message store providers that support Rich Text Format messages.
- Some properties are used by more than one object type related to message stores. For example, the PR_STORE_SUPPORT_MASK property is required on folder and message objects as well as message store objects.

It is the responsibility of the message store provider to implement the correct semantics for such properties.

About Supporting Named Properties in Message Stores

Message objects can have properties in them that are not in the set of properties defined by MAPI. Such properties can be unnamed or named. Unnamed properties must reside in a range of property identifiers defined by MAPI. Named custom properties reside in a different range of property identifiers defined by MAPI. They are typically used by custom message types. Your message store provider must support named properties if it is to be used as the default message store. Supporting named properties means implementing the **IMAPIProp::GetNamesFromIDs** and **IMAPIProp::GetIDsFromNames** methods, and implementing one or more mapping signatures that identify what names go with what property identifiers. For more information, see About Defining New Properties and About Support for Named Properties.

Most message store providers that support named properties use a single mapping signature for all objects in the message store. This has two benefits. One is that it is simpler to implement mapping signatures if there is only one to keep track of. The second benefit is that if all objects in the message store use the same mapping signature, then client applications are assured that all property identifiers on messages in the message store actually refer to the same named property. This allows client applications to display columns for named properties in their folder view interface.

About Supporting Multi-Valued Properties in Message Stores

The requirements for supporting multivalued properties are documented in Properties. Multivalued property support is not required for message store providers but it is recommended, particularly for default message store providers. Some other features, such as folder form libraries, do require multivalued property support.

About Supporting Multiple Client Access to Messages in Message Stores

It is possible for multiple client applications to open a given message simultaneously. Message store providers do not have to follow any particular rules for governing such access. However, if the client applications modify the message and save their changes, the store provider should follow certain rules:

- Allow the first call to the **IMAPIProp::SaveChanges** method to proceed as if it were the only client that has the message open.
- On the subsequent **SaveChanges** calls by other clients, the message store provider should ignore the changes and return MAPI_E_OBJECT_CHANGED.
- Allow client applications to respond to a MAPI_E_OBJECT_CHANGED return code by calling **SaveChanges** again with the FORCE_SAVE flag. If a client application does this, the message store provider should replace the previous changes with the new ones.

Alternatively, the message store provider can detect the conflict and present an interface which allows the user to choose whether to keep the original message, overwrite the original message with the new changes, or save the new changes to another location.

About Displaying Progress for Message Store Providers

Message store providers, like all MAPI service providers, are required to display progress dialog boxes during long operations. This simply means making use of the **IMAPIProgress:IUnknown** objects that are supplied by MAPI and by client applications at the appropriate times. For more information, see **IMAPIProgress:IUnknown**.

About Providing Read and Nonread Reports for Message Store Providers

If a message store provider can receive messages, it is required to support read reports and nonread reports of messages received by the message store provider. If a received message contains the PR_READ_RECEIPT_REQUESTED property and that property's value is TRUE, then the message store should send a notification message to the sender when the user opens the message indicating that the message has been read. Similarly, if the user deletes the message before opening it, the message store should issue a reply to the sender indicating that the message was not read.

Issuing these reports is a matter of creating an **IMessage:IMAPIProp** object, filling out the relevant properties on the message, and submitting it to the MAPI spooler as if the message had originated with the user. The **IMAPISupport::ReadReceipt** method can be useful for this.

About Implementing a Configuration Interface for Message Store Providers

Message store providers are required to implement an interface that allows the user to configure the message store provider to run on that user's computer. Typically, the message store provider is configured when the message store provider is added to a user's profile. The message store provider's configuration interface generally handles tasks such as setting user names and passwords for protected message stores, choosing pathnames to necessary files, creating the underlying storage mechanism it will use, if necessary, etc.

The configuration interface you implement is accessed through additional entry points in your message service provider's DLL. For more information, see Configuring a Message Service. The message store provider's configuration interface is the only user interface that a message store provider must implement.

About Folder Associated Information Tables

MAPI defines the MAPI_ASSOCIATED flag for various MAPI components to use when dealing with folder-associated information tables. Each folder in a message store should have an associated contents table table along with its standard contents table. Client applications store special messages in a folder's associated contents table to hold forms and views. In fact, to support forms and views, your message store provider must implement associated contents tables.

To implement associated contents tables, your store provider must:

- Support the MAPI_ASSOCIATED flag in the **IMAPIContainer::GetContentsTable** method so client applications can get the folder's associated contents table instead of the standard contents table.
- Support the MAPI_ASSOCIATED flag in the **IMAPIFolder::CreateMessage** method so client applications can add messages to a folder's associated contents table.
- Set the MAPI_ACCESS_CREATE_ASSOCIATED bit in the PR_ACCESS property on folder objects.
- Support the DEL_ASSOCIATED flag in the **IMAPIFolder::EmptyFolder** method.
- Set the MSGFLAG_ASSOCIATED bit in the PR_MESSAGE_FLAGS property for messages in the associated contents table.
- Expose and respond to the PR_FOLDER_ASSOCIATED_CONTENTS property on folders.
- Maintain the PR_ASSOC_CONTENT_COUNT property on folders.

There is no bit in the PR_STORE_SUPPORT_MASK property to indicate whether your message store provider supports associated contents tables. If your message store provider does not support them, it should return MAPI_E_NO_SUPPORT when client applications call any of the above methods with the MAPI_ASSOCIATED flag.

About Sending Messages with Message Store Providers

Message store providers are not required to support outgoing message submissions, that is, the ability for client applications to use the message store provider to send messages. Client applications need to use a message store while sending messages because the message's data must be stored somewhere between the time that the user is finished composing it and the time that the MAPI spooler gives the message to a transport provider for submission to the underlying message system. If your message store provider does not support outgoing message submissions, it cannot be used as the default message store.

To support sending messages, your message store provider must:

- Implement an outgoing message queue.
- Support the **IMessage::SubmitMessage** method on message objects created within the message store.
- Support the **IMsgStore** methods that are specific to the MAPI spooler: **IMsgStore:FinishedMsg**, **IMsgStore:GetOutgoingQueue**, **IMsgStore:NotifyNewMail**, and **IMsgStore:SetLockState**.

The **SetLockState** method is important for proper interoperation between the MAPI spooler and client applications. When the MAPI spooler calls **SetLockState** on an outgoing message, the message store provider must not let client applications open the message. If a client application does try to open a message that is locked by the MAPI spooler, the message store provider should return MAPI_E_NO_ACCESS. The locked state of a message does not have to be persistent in case the store is shut down while the message is locked by the MAPI spooler.

Regardless of whether the MAPI spooler has locked an outgoing message, the message store provider should not allow a message in the outgoing message queue to be opened for writing. If a client application calls the **IMSgStore::OpenEntry** method on an outgoing message with the MAPI_MODIFY flag, the call should fail and return MAPI_E_SUBMITTED. If a client application calls **OpenEntry** on an outgoing message with the MAPI_BEST_ACCESS flag, the message store provider should allow read-only access to the message.

When a message is to be handled by the MAPI spooler, the message store provider sets the message's PR_SUBMIT_FLAGS property to SUBMITFLAG_LOCKED. The SUBMITFLAG_LOCKED value indicates that the MAPI spooler has locked the message for its exclusive use. The other value for PR_SUBMIT_FLAGS, SUBMITFLAG_PREPROCESS, is set when the message requires preprocessing by one or more preprocessor functions registered by a transport provider.

The following three procedures describe how the message store, transport, and MAPI spooler interact to send a message from a client application to one or more recipients.

The client application calls the **IMessage::SubmitMessage** method.

In SubmitMessage, the message store provider

1. Calls **IMAPISupport::PrepareSubmit**. If MAPI returns an error, the message store provider returns that error to the client.
2. Sets the MSGFLAG_SUBMIT bit in the PR_MESSAGE_FLAGS property of the message.
3. Makes sure that there is a column for PR_RESPONSIBILITY in the recipient table and sets it to FALSE to indicate that no transport has yet assumed responsibility for transmitting the message.
4. Sets the date-time of origination in the PR_CLIENT_SUBMIT_TIME property.
5. Calls **IMAPISupport::ExpandRecips** to:
   - Expand all personal distribution lists and custom recipients and replace all changed display names with their original names.
   - Remove duplicate names.
   - Check for any required preprocessing, and if preprocessing is required, set the NEEDS_PREPROCESSING flag and the PR_PREPROCESS property, a property reserved for

MAPI.
- Set the NEEDS_SPOOLER flag if the message store is tightly coupled with a transport and it cannot handle all of the recipients.

6. Performs the following tasks if the NEEDS_PREPROCESSING message flag is set:
   - Puts the message in the outgoing queue with the SUBMITFLAG_PREPROCESS bit set in the PR_SUBMIT_FLAGS property.
   - Notifies the MAPI spooler that the queue has changed.
   - Returns control to the client, and message flow continues in the MAPI spooler. The MAPI spooler performs the following tasks:
     1. Locks the message by calling **IMsgStore::SetLockState**.
     2. Performs the needed preprocessing by calling all of the preprocessing functions in the order of registration. Transport providers call **IMAPISupport::RegisterPreprocessor** to register preprocessing functions.
     3. Calls **IMessage::SubmitMessage** on the open message to indicate to the message store that preprocessing is complete.

Note The following two steps will occur in the client process if there was no preprocessing, and will occur when the MAPI spooler calls **SubmitMessage** if there was preprocessing.

7. Performs the following tasks if the message store is tightly coupled to a transport and the NEEDS_SPOOLER flag was returned from **IMAPISupport::ExpandRecips**:
   - Handles any recipients that it can handle.
   - Sets the PR_RESPONSIBILITY property to TRUE for any recipients that it handles.
   - Performs the following tasks if all recipients are known to this tightly-coupled store and transport:
     1. Calls **IMAPISupport::CompleteMsg** if the message was preprocessed or the message store provider wants the MAPI spooler to complete message processing which is recommended so that messaging hooks can be invoked. Message flow continues with the MAPI spooler as described in the following procedure.
     2. Performs the following tasks if the message was not preprocessed or the message store provider does not want the MAPI spooler to complete message processing:
        - Copies the message to the folder identified by the entry identifier in the PR_SENTMAIL_ENTRYID property, if set
        - Deletes the message if the PR_DELETE_AFTER_SUBMIT property has been set to TRUE.
        - Unlocks the message if it is locked
        - Returns to the client. Message flow is complete.
   - If the message was preprocessed or the provider wants the MAPI spooler to complete message processing (recommended so that messaging hooks can be invoked):
     1. The message store provider should call **IMAPISupport::CompleteMsg**.
     2. Message flow continues with the MAPI spooler. See About Sending Messages: MAPI Spooler Tasks.
   - If the message was not preprocessed or the provider does not want the spooler to complete

message processing:

    1. Copy the message to the folder identified by the entry identifier in the PR_SENTMAIL_ENTRYID property, if set.
    2. Delete the message if the PR_DELETE_AFTER_SUBMIT property has been set to TRUE.
    3. If the message is locked, unlock it.
    4. Return to the caller. Message flow is complete.

8. Performs the following tasks if the message store is not tightly coupled to a transport, not all of the recipients were known to the message store, or the NEEDS_SPOOLER flag is set:
    - Puts the message in the outgoing queue without setting the SUBMITFLAG_PREPROCESS bit in the PR_SUBMIT_FLAGS property.
    - Notifies the MAPI spooler that the outgoing queue has changed by generating a table notification.
    - Returns to the client, and message flow continues with a set of tasks performed by the MAPI spooler.

About Receiving Messages with Message Store Providers

Message store providers do not have to support incoming message submissions; that is, the ability for transport providers and the MAPI spooler to use the message store provider as a delivery point for messages. However, if your message store provider does not support incoming message submissions, it cannot be used as the default message store.

To support incoming message submissions, your message store provider must:

- Support the **IMsgStore::GetReceiveFolderTable** and **IMsgStore::GetReceiveFolder** methods so client applications can find incoming messages.
- Support the **IMsgStore::NotifyNewMail** method so that the MAPI spooler can inform the message store provider that a new message has arrived.
- Implement notifications so that client applications can register for new message notification. Strictly speaking, notifications are optional, but they are strongly recommended.

The sequence of method calls that occurs when an incoming message is delivered to a message store is:

1. The MAPI spooler calls **IMsgStore::OpenEntry** with the Inbox's EntryID to get an **IMAPIFolder** interface.
2. The MAPI spooler calls **IMAPIFolder::CreateMessage** to get a new message object.
3. The MAPI spooler passes the message object to the transport provider.
4. The transport provider fills in the message's properties with data from the underlying message system and calls the message object's **IMAPIProp::SaveChanges** method.
5. The spooler calls any registered hook providers. The message may be modified, moved to a different folder, deleted, etc.
6. The message store provider uses its notification method to inform registered client applications that a new message has arrived.
7. The MAPI spooler calls the message store's **IMsgStore::NotifyNewMail** method.

About Supporting Message Attachments for Message Store Providers

It is not required that your message store provider support message attachments. However, many client applications expect to be able to add attachments to messages. If your message store will be used to create or store IPM.Note messages, then your message store provider should support message attachments. Default message store providers are strongly encouraged to support message attachments. For more information, see About Message Classes, and About Default Message Stores.

There are five types of attachments that MAPI supports: file attachments, data attachments, message attachments, OLE object attachments, and links. The requirements for supporting each type are different. Client applications differentiate between the two types of attachments by means of the PR_ATTACH_METHOD property on attachment objects.

Supporting attachments means implementing the **IAttach:IMAPIProp** interface. The **IAttach** interface has no methods of its own; it only has methods inherited from the **IMAPIProp** interface. Since your message store provider must already implement properties for message objects, this greatly simplifies the task of supporting attachments. Implementing **IAttach** basically means providing a way for clients to access a table of properties for particular attachments on messages.

Data attachments are simply attachments where the contents of the attachment are stored directly in attachment's PR_ATTACH_DATA_BIN property. Data attachments exist primarily to allow client applications to attach files to a message when the sender and the recipient of the message do not have access to a common file server. See PR_ATTACH_METHOD for details.

Message attachments are attachments where the attachment sub-object is another **IMessage:IMAPIProp** object. Since message store providers already support the **IMessage** interface, supporting message attachments is not difficult.

Supporting OLE object attachments means implementing the OLE **IStorage**, **IStream**, and **IStreamDocfile** interfaces. Your message store provider must be able to convert OLE object data stored in the message into an active OLE object when a client application opens the object. For more information, see Attachment Properties.

Links come in two types: links to files and links to other messages. Links to files use the ATTACH_BY_REF_ONLY value for the PR_ATTACH_METHOD property along with PR_ATTACH_PATHNAME or PR_ATTACH_LONG_PATHNAME to specify the location of a file.

How one implements links to messages may be dependent on aspects of the local messaging system, and as such cannot be fully documented here. For example, sending a link to a message that is stored on a server-based message store is typically just a matter of sending the entry identifier of the linked message, providing that both the sender and recipient have access to that server. Other messaging system configurations present other requirements and challenges for implementing links to messages.

About Supporting RTF Text for Message Store Providers

Some client applications allow users to use Rich Text Format (RTF) text in their messages. If your message store provider needs to support RTF text in messages, it needs to handle the PR_RTF_COMPRESSED property in addition to the normal PR_BODY property. Primarily this means storing both properties, and making sure that the PR_BODY property contains a plain text version of the text in the PR_RTF_COMPRESSED property. The **RTFSync** function is useful for this.

There are two flags that can be set in the message store object's PR_STORE_SUPPORT_MASK property which tell client applications what they can expect from the message store provider with respect to the PR_BODY and PR_RTF_COMPRESSED properties on messages within the message store. The STORE_RTF_OK flag indicates that the store can generate the value of the PR_BODY property from the PR_RTF_COMPRESSED property dynamically, which relieves client applications from the burden of synchronizing them explicitly. The STORE_UNCOMPRESSED_RTF flag indicates that the message store provider can support uncompressed data in the PR_RTF_COMPRESSED property.

Message store providers that do not support RTF text need to delete the PR_RTF_IN_SYNC property when the PR_BODY property changes in order to inter-operate properly with client applications that do support RTF text.

About Providing Notifications for Message Store Providers

While notifications are optional, they are a very important part of a good message store provider. Client applications and the MAPI spooler rely on notifications from the message store provider to get good performance when submitting outgoing messages or receiving incoming messages. Client applications and the MAPI spooler can function without receiving notifications from the message store provider, but will not be able to inform users of changes in the message store without them. Typically, this means that users will be unable to see that a new message has arrived until their client application next opens the message store's receive folder.

Clients register for notifications by calling the **IMsgStore::Advise** method. The client passes in an **IMAPIAdviseSink** object, a bitmask indicating what type of notifications the client is interested in receiving, and an EntryID indicating which object in the message store the **Advise** request applies to. When relevant events occur within the object (for example, when a new message arrives in the receive folder within the message store), the message store provider or the object itself should call the **IMAPIAdviseSink::OnNotify** method for all of the **IMAPIAdviseSink** objects that have registered for that event type.

Even if your message store provider never notifies other MAPI components of changes in the message store, it should still implement **IMsgStore::Advise** to return MAPI_E_NO_SUPPORT. This informs other components not to expect notifications from the message store provider.

About Grouping and Restricting Tables in Message Store Providers

Client applications frequently allow users some control over how the contents of a folder are displayed. Typically, a user can choose to have messages grouped according to the value of one or more properties on the messages, or can choose to exclude messages matching some criteria from the list. This is done within the **IMAPITable** interface. Client applications can restrict the rows returned from the table to whatever criteria the user specifies. Therefore, a message store provider needs to implement the following **IMAPITable** methods.

IMAPITable method	Purpose
FindRow	Returns rows from a table that match the specified criteria.
QueryColumns	Returns the set of columns in a table or the set of currently used columns.
QueryRows	Returns one or more rows from a table starting from a given position.
Restrict	Applies a restriction to a table so that subsequent calls to **FindRow** only return rows that match the restriction.
SetColumns	Specifies which columns should be returned when rows are retrieved from the table.

Restrictions can be complex to implement; see About Restrictions for details. For more information on implementing tables, see Tables.

About Supporting Searches in Message Store Providers

Client applications frequently have some user interface components devoted to searching for messages within a message store. Search criteria are specified within the **IMAPIContainer** interface, by means of the **IMAPIContainer::SetSearchCriteria** and **IMAPIContainer::GetSearchCriteria** methods.

Client applications use the message store object's PR_FINDER_ENTRYID property to identify the root folder in the message store that contains folders for search results. The search-results folder is often a folder at the top-level of the message store that is not part of the IPM folder tree and is therefore hidden.

Whether your message store provider uses a permanent search-results folder or creates one when a client opens the entry identifier stored in the PR_FINDER_ENTRYID property is an implementation detail. It is somewhat easier for your message store provider to use a permanent folder that is created when the message store is created, because doing so avoids the complication of checking the entry identifier whenever any folder is opened to see whether to create a search-results folder. However, not all message store providers can do that; notably, read-only message stores or stores that provide a MAPI interface to a legacy database often are not allowed or are unable to create a permanent search-results folder in the underlying storage mechanism.

About Generating and Using Entry Identifiers in Message Store Providers

When a new folder or message is created in a message store, the message store provider has to assign that object an entry identifier so that client applications can refer to it. This brings up the option of re-using the defunct long-term entry identifiers of deleted objects or not. There is no requirement one way or the other for message store providers, although if it is feasable, it is better if a message store provider always generates new long-term entry identifiers for new objects rather than re-using old ones. It is fine to re-use short-term entry identifiers when the objects they refer to are deleted.

The reason for this is that client applications can cache entry identifiers, sometimes for long periods of time. If that happens and the message store provider does re-use entry identifiers, then it is possible for the entry identifier to refer to a different object when the client application opens the entry identifier than when it first obtained the entry identifier. If the message store provider does not re-use entry identifiers (or at least uses an entry identifier generation scheme that does not repeat for a very long time), then this problem cannot occur.

Similarly, it is good for message store providers to attempt to preserve entry identifiers for folders and messages and folders when they are moved within the message store. If the message store provider can do that, then references to objects in the store will not become invalid when the object is moved to a different location within the store.

About Default Message Stores

A default message store is one that client applications can use for general purpose messaging tasks. There are a number of optional features for message store providers that become required if the message store provider is to be used as the default message store. They are:

- Implementing the special folders: Inbox, Outbox, and search-results folder.
- Providing read and nonread reports.
- Allow incoming and outgoing message submissions.
- Allow creation of messages with an arbitrary message class.
- Support named and multivalued properties.
- Support the **IMSProvider::SpoolerLogon** method, even if the message store provider is tightly coupled with a transport provider.
- Support associated contents tables.
- Support notification of the MAPI spooler when there are messages in the outgoing message queue.

About Read-Only Message Stores

A read-only message store is one in which neither the MAPI client nor the MAPI spooler can create, modify or delete the objects in the message store. There are many reasons why you might want to implement a read-only message store. For example, a credit reporting firm could use a read-only store to allow its customers or employees to see but not change individual credit reports. Choosing to make a read-only message store has implications for the structure of the store provider, and for the store itself. For example, a read-only message store can't have an Outbox folder because then MAPI clients would request that new outgoing messages be created in that folder. Similarly, it is the store provider's responsibility to ensure the integrity of the underlying storage mechanism.

There are three flags that can be set in the message store's PR_STORE_SUPPORT_MASK property which support different levels of read-only access. the STORE_READONLY flag indicates that all **IMAPIProp** interfaces on objects within the message store are read-only. The STORE_MODIFY_OK flag indicates that existing messages in the message store may be modified, but new folders and messages may not be created. The STORE_CREATE_OK flag indicates that new messages and folders may be created, but indicates nothing about whether existing objects may be modified.

The fact that MAPI clients and the MAPI spooler may not be able to create, modify or delete objects in the message store does not mean that the contents of the underlying storage mechanism never change. Nor does it mean that your store provider never needs write access to the underlying storage mechanism. In some circumstances those two conditions may apply, but not in the general case of a read-only message store. What level of access your store provider requires and whether or not your store provider ever changes data in the underlying storage mechanism depends mainly on the specific nature of your store provider.

For example, if you are writing a store provider to give MAPI clients access to a database stored on a CD-ROM device, then the underlying storage mechanism cannot change and your store provider can only have read access to it. If, however, you are writing a store provider to give MAPI clients read-only access to a public folder database but the store provider needs to keep track of the read/unread status of messages for each user, then the store provider will need to write new data to the underlying storage mechanism. However, in neither example does the store provider ever have to create, modify, or delete folders or messages at the request of MAPI clients or the MAPI spooler.

The list of reasons that a store provider would need to write data to an underlying storage mechanism that is otherwise read-only is fairly short:

- To store the read/unread status of messages.
- To implement read/non-read notifications.
- To store views.
- To cache persistent indexes for user defined folder sort orders.
- To store what order a folder's contents are sorted in (supporting **IMAPIFolder::SaveContentsSort**).
- To store search criteria, search state, and results, if the message store provider support searches (supporting **IMAPIContainer::SetSearchCriteria**).

If your message store provider can never write data to the underlying storage mechanism, it will need to implement these features using a separate storage mechanism outside of the underlying storage mechanism. For example, a read-only message store provider could store the read/unread status of messages in the store in a file on the user's computer. This strategy presents additional difficulties, but may be the only feasable way for read-only message store providers to implement some features. For example, keeping the contents of the separate storage mechanism synchronized with the objects in the message store is more difficult than storing the read/unread status directly in the message store itself.

Searching presents an additional complication for read-only message store providers. Client applications use the folder specified in the message store object's PR_FINDER_ENTRYID property to locate the folder used for search results. Read-only message store providers often cannot install a permanent search-results folder into the message store. In that situation, the message store provider should store an entry identifier in the PR_FINDER_ENTRYID property that it can recognize when client applications open folders so that it can dynamically create a search-results folder instead of reading one from the underlying storage mechanism. However, since many read-only message store providers create all their folders dynamically, this is usually not too much of a burden.

The fact that your message store provider is read-only is advertized in the store provider object's PR_SUPPORT_MASK property. However, do not count on client applications to respect that property; your store provider's code should enforce the read-only status of the underlying storage mechanism.

About Supporting Forms and Views in Read-Only Message Stores

If your message store provider allows read-only access to the underlying storage mechanism, then client applications and the MAPI form manager will be unable to do certain things. Specifically, clients will be unable to add or modify custom views, and the MAPI form manager will be unable to install forms in the associated contents tables of the store's folders.

For many read-only message stores, that may not be a problem. If that is the case, then the message store provider does not need to support associated contents tables at all. However, if your message store provider must be read-only and it must also support a pre-defined set of views or forms, then it will need to support associated contents tables.

The most common strategy for doing this, since clients and the MAPI form manager cannot install the views or forms into the message store themselves, is for the message store provider to hard-code them into the message store. This means that the associated contents table or tables containing the views or forms will exist in the message store when it is created, before any client applications or the MAPI form manager ever access it. Then, when a client requests an associated contents table to get custom views from a form or the MAPI form manager requests an associated contents table in order to launch a form, the message store provider can provide one.

This requirement that the associated contents tables be created and populated when the message store itself is created implies that your message store provider will need to obtain information about the format of the special messages that client applications and the MAPI form manager use in order to store views and forms. What those formats are will depend on the client application and MAPI form manager being used, and so a description of them cannot be provided here. If you do not have access to that information, it is very likely that you will have to reverse engineer those formats from existing message stores that have views and forms installed into them. The MAPI SDK includes a utility called MDBVIEW.EXE which you can use to view the properties and their values of objects in message stores to help with this reverse engineering.

The danger in implementing forms and views by reverse engineering the form and view descriptor format of a MAPI client application is that if the application changes that format when it is upgraded, the forms and views in your message store will no longer function. This problem will persist until a universal format for form and view descriptors, defined by MAPI, is available. For this reason, reverse engineering form and view descriptors should only be done as a last resort.

About Tightly Coupled Message Store Providers

Message store providers can be "tightly coupled" with a transport provider. Tightly coupling MAPI service providers means implementing the two providers such that the store provider and transport provider can communicate to make the process of sending and receiving messages more efficient. The benefit of doing this is that performance improvements can result when two service providers can interact with each other directly rather than by means of the MAPI spooler. To tightly couple a message store provider to a transport provider, the transport must place the message store provider's entry identifier in the PR_OWN_STORE_ENTRYID property in the transport provider's row in the MAPI status table. This enables the MAPI spooler to connect the store provider to the transport provider.

There is no requirement that a message store provider ever be tightly coupled with any other service provider. The most common service provider to tightly couple with a message store provider is a transport provider. This is usually done so that sending and receiving messages can be accomplished without involving the MAPI spooler. For example, when the user submits an outgoing message, the combined message store provider and transport provider can send it directly. The combined service providers don't have to first notify the MAPI spooler that there is a new message to process, and then wait for the MAPI spooler to initiate the process of transferring the message from the message store provider to the transport provider. This has particular benefits when a server-based message store is being used by minimizing network traffic between the user's computer and the server.

In general, there are no well-specified procedures for tightly coupling service providers. However, there are some guidelines:

- Keep in mind that if the reason for tightly coupling service providers is performance, then they are taking other parts of the MAPI subsystem out of the processes that those parts would normally be involved in. This implies that the individual parts within the combined service provider should interact with each other in a way that simulates the interaction they would normally have with the parts of the MAPI subsystem that are not being used.
- When tightly coupled service providers do interact with other MAPI components, they must still interact with them in exactly the way they would if they were not tightly coupled. For example, if a user is using a combined message store provider/transport provider as their default message store but are using a separate transport provider to send messages, as can happen when a user takes his or her computer on the road and switches to a remote transport provider, the message store portion of the tightly coupled service provider must still interact with the MAPI spooler just as if it were a standalone message store provider.

About Loading Message Store Providers

When a client application opens a message store, MAPI loads the message store provider's DLL into memory. After loading the DLL, a very specific sequence of method calls occurs between the message store provider and MAPI. This method call sequence enables MAPI to get top level **IMSProvider:IUnknown** **IMSLogon:IUnknown**, and **IMsgStore:IMAPIProp** objects, and allows the message store provider to get a MAPI support object. After the call sequence, the message store provider should be ready to accept logons from client applications.

The call sequence when a message provider DLL is loaded is:

1. Client calls **IMAPISession::OpenMsgStore**.
2. If the message store is not already open, MAPI loads the store provider's DLL and calls the DLL's **MSProviderInit** entry point. If the message store is already open, MAPI skips steps 2 and 3, then uses the existing **IMSProvider** object to complete step 4.
3. **MSProviderInit** creates and returns an **IMSProvider** object.
4. MAPI calls **IMSProvider::Logon**, passing the client application's message store entry identifier.
5. **IMSProvider::Logon** creates and returns an **IMSLogon** object and an **IMsgStore** object, then calls the **IUnknown::AddRef** method on its **IMAPISupport:IUnknown** object. If the client application's message store entry identifier refers to a message store that is already open, the message store provider can return existing **IMSLogon** and **IMsgStore** objects, and does not need to call **AddRef** on its support object.
6. If the client application did not set the MAPI_NO_MAIL flag when it logged on and it did not set the MDB_NO_MAIL in step 1, then MAPI gives the message store's entry identifier to the MAPI spooler so that the MAPI spooler can log on to the message store.
7. MAPI returns the **IMsgStore** object to the client application.
8. The MAPI spooler calls **IMSProvider::SpoolerLogon**.
9. **IMSProvider::SpoolerLogon** returns the same **IMSLogon** object and **IMsgStore** object from step 5.

Note If the logon call to the message store provider fails because an incorrect password was supplied and the message store provider cannot display an interface to ask for the correct password, it should return MAPI_E_FAILONEPROVIDER from the **IMSProvider::Logon** method. This will allow client applications to ask the user for a password to try logging onto the message store provider again instead of causing MAPI to fail the provider for the entire session.

Developing MAPI Form Servers

This section details the process of creating form server executable and configuration files for creating custom MAPI forms. Before reading this section, you should familiarize yourself with the information in MAPI Form Architecture.

Developing a form server includes the following steps:

1. Deciding what information the form will carry and choosing a set of properties to hold that information.
2. Designing a user interface to allow users to interact with the form's properties.
3. Choose a message class and generate a unique message class identifier (CLSID) for the class.
4. Implement the required MAPI form interfaces, as well as any optional interfaces that your particular form server needs.
5. Write user interface code to handle the user's interaction with the form object and the properties the form uses.
6. Create a configuration file for the form.
7. Install the form on users' computers.

You will most likely perform steps 1-5 simultaneously rather than completing them in sequence. The process of developing a form server, like many programming projects, is not one in which there is a particularly well-defined starting point and ending point. For example, creating a configuration file is listed as the last step in this process, but your configuration file will probably be created incrementally and will become more complete as you add features to your form server.

Choosing a Form's Property Set

When you implement your form server, you need to have a property for each piece of information that your message class needs. These properties can be pre-defined MAPI properties, or they can be custom properties that you define. For details about working with properties, see Properties.

Your form's configuration file will contain a list of properties that your form server exposes for client applications to use, but this does not have to be the entire list of properties used by your form server. Client applications typically use the exposed properties to allow users to sort messages in a folder, or customize their interfaces in some way.

MAPI has a large set of pre-defined properties that suffice for most applications. However, there will be times when a custom message class needs a property that MAPI does not define. You can use custom properties to extend MAPI's pre-defined set of properties for whatever special information your form server needs to support.

Custom properties can be defined by:

- Choosing a name for the property and using the **IMAPIProp::GetIdsFromNames** function to obtain a property tag for it. The **IMAPIProp** interface through which you call this method comes from the **IMessage** pointer that is passed to the form server when the message is created. Note that the property name must be a wide-character string.
- Defining a custom property tag yourself. Custom property tags must be in the range 0x6800-0x7BFF. Properties in this range are message class specific.

For details about defining custom properties, see Properties.

Note Form servers that have a message text often use the PR_RTF_COMPRESSED property to store it. If your form server uses PR_RTF_COMPRESSED, it should also ensure that the PR_BODY property contains a text only version of the message text, in case the resulting message is read by a client that does not support RTF message text.

Choosing a Message Class

As described in About MAPI Message Classes, message classes are an important concept for establishing the relationship between types of custom messages, and by extension, between form servers themselves. Fortunately, choosing a message class string is fairly simple. The message class string of a message class is an arbitrary string, but should follow these conventions:

- The string should satisfy all the conventions described in the documentation for PR_MESSAGE_CLASS. Importantly, the string should be composed entirely of ANSI characters and be less than 256 characters long.
- If your form server is derived from an existing form server or is an extension of an existing form server, your message class string should be formed by adding a period and another word to the message class string of the form server your form is based on. For example, imagine that your are implementing a form for rescheduling a meeting, and your form is based on an existing form for scheduling meetings. If the meeting scheduling form's message class string is "IPM.Meeting", your message class string could be "IPM.Meeting.Reschedule".
- If your form is not based on any existing form, your message class string should still begin with either the IPM. or IPC. prefix, depending on whether the form is intended to be received by a person or by another piece of software. IPM. designates an interpersonal message that usually ends up in a user's Inbox, and IPC. designates an interprocess communication message that is not typically delivered to a user's Inbox.
- If your message class is intended to be human-readable, the message class string should start with IPM. A message class is generally considered human-readable if it uses any properties that contain plain text or RTF data. If your form uses the PR_BODY property, it should almost certainly use an IPM.-derived message class string. For example, if you are implementing a form for purchase orders, and your organization requires that purchase orders be approved by a manager, your message class string could be IPM.Purchase_Order. Forms that are designed for use with public folders or public folder applications are typically considered to be interpersonal because they are read by people even though they are not actually addressed to any person's e-mail address. The typical prefix for public folder message classes is IPM.post.
- If your message class is intended to be received by some other piece of software instead of by a person, the message class string should start with IPC. For example, if you are implementing a form for allowing people to automatically subscribe to mailing lists, your message class string could be IPC.Subscribe.
- Your message class string should never end with a period.

The message class string should be put in the **[Description]** section of the form's configuration file, in the **MessageClass** entry, like this:

```
MessageClass=IPM.Meeting.Reschedule
```

Once you have chosen an appropriate message class string, you should generate a class identifier for it. Class identifiers are generated with the UUIDGEN.EXE utility that is included with the Win32 SDK. The class identifier must be put in the configuration file's **CLSID** entry, along with the **MessageClass** entry, like this:

```
CLSID={88FFF551-B8C5-11ce-8DE0-00AA0060D242}
```

Your class identifier will almost certainly be different, of course. For details, see Creating a Configuration File.

When the form is installed on a user's computer, your installation process – whether it is a setup program or something else – must make a registry entry in the **HKEY_CLASSES_ROOT\CLSID\** section of the registry for the class identifier. This entry must be set to the message class string. For example, you would create a registry entry like this for the example class identifier above:

```
HKEY_CLASSES_ROOT\CLSID\{88FFF551-B8C5-11ce-8DE0-
     00AA0060D242}="IPM.Meeting.Reschedule"
```

For details, see Installing a Form into a Library.

About Form States

Form objects can be in one of five distinct states, depending on what methods have been called in them and whether any errors have occurred in performing those methods. The states are Uninitialized, Normal, No Scribble, Hands Off After Save, and Hands Off From Normal.

The states primarily relate to the status of the data in the form object. The different states reflect whether the data needs to be saved, whether the form object should allow modifications to the data, and what point in the process of saving the data the form is in. As such, the form states and transitions between them have more to do with your form server's implementation of **IPersistMessage** interface methods than any other. Knowledge of these states is very useful for proper implementation of the MAPI form interfaces that your form server must implement.

Each state is described in the following sections, along with the legal actions that cause transitions to other states. Any transitions not listed in the following sections are illegal. If your form objects make illegal transitions between states, they will not behave in the ways that messaging clients expect and could cause unpredictable client or form object behavior.

Note Some state transitions depend on information from previous states. Your form server will most likely have to implement a "dirty" flag in its form objects to indicate whether the values of the message's properties have been changed in order to facilitate later state changes.

About the Uninitialized State

The Uninitialized state is the initial state form objects should be in when they are first created. Form objects become initialized with message data when a client application calls the **IPersistMessage::InitNew** or **IPersistMessage::Load** method on the form object. Legal state transitions from the Unitialized state are described in the following table.

IPersistMessage method	Action	New State
InitNew	Load the form object with default data.	Normal
Load	Load the form object with data from the target message.	Normal
GetClassId	Return success, or else set the last error to and return E_UNEXPECTED.	Uninitialized
GetLastError	Return the last error.	Uninitialized
Other **IPersistMessage** methods or methods from other interfaces	Set the last error to and return E_UNEXPECTED.	Uninitialized

About the Normal State

The Normal state is where the form object spends most of its time, waiting for client applications to initiate an action such as saving changes or closing the form. Legal transitions from the Normal state are described in the following table.

IPersistMessage method	Action	New State
Save(*pMessage* == NULL, *fSameAsLoad* == TRUE) or **Save**(*pMessage* != NULL, *fSameAsLoad* == FALSE)	Recursively save any embedded OLE objects that are dirty. Save message data back to the message object. Store the *fSameAsLoad* flag for later use in the No Scribble state.	No Scribble
Save(*pMessage* != NULL, *fSameAsLoad* == TRUE)	This is the same as the previous case, except that this **Save** call is used in low memory situations and must not fail for lack of memory.	No Scribble
HandsOffMessage	Recursively invoke the **HandsOffMessage** method on embedded messages or the OLE **IPersistStorage::HandsOffStorage** method on embedded OLE objects. Release the message object and any embedded messages or objects.	Hands Off From Normal
SaveCompleted, **InitNew** or **Load**	Set last error to and return E_UNEXPECTED	Normal
GetLastError	Return the last error.	Normal
Other **IPersistMessage** methods or methods from other interfaces	Implement as described in the documentation for the **IPersistMessage** interface.	Normal

About the No Scribble State

The No Scribble state indicates that changes to a message are being saved. The actual saving of values stored in the form object's user interface happens when the form object's **IPersistMessage::Save** method is called by the client application. Legal transitions from the No Scribble state are described in the following table.

IPersistMessage method	Action	New State
SaveCompleted(*pMessage* == NULL)	If *fSameAsLoad* flag was TRUE on the **Save** call that caused the form to enter the No Scribble State and the message has been modified, then internally mark the changes as saved and call **IMAPIViewAdviseSink::OnSaved**.	Normal
SaveCompleted(*pMessage* != NULL)	Call the **IPersistMessage::HandsOffMessage** method (similar to the OLE **IPersistStorage::HandsOffStorage** method) followed by the normal **SaveCompleted** actions. If **SaveCompleted** was successful, enter the Normal state. Otherwise, enter the Hands Off After Save state.	Normal or Hands Off After Save
HandsOffMessage	Recursively invoke the **HandsOffMessage** method on embedded messages or the OLE **IPersistStorage::HandsOffStorage** method on embedded OLE objects. Release the message object and any embedded messages or objects.	Hands Off After Save
Save, **InitNew** or **Load**	Set the last error to and return E_UNEXPECTED.	No Scribble
GetLastError	Return the last error.	No Scribble
Other **IPersistMessage** methods or methods from other interfaces	Set the last error to and return E_UNEXPECTED.	No Scribble

About the Hands Off After Save State

The Hands Off After Save state is part of the process of saving the contents of a form to permanent storage. When in this state, the form object should refrain from making changes to the in-memory copies of values of the message's properties, since there may not be another opportunity to save those changes. Legal transitions from the Hands Off After Save state are described in the following table.

IPersistMessage method	Action	New State
SaveCompleted(*pMessage !=* NULL)	Open any embedded objects. The data in the message stored in *pMessage* is guaranteed to be the same as the message in the previous Save call. If the SaveCompleted call succeeds, enter the Normal state. Otherwise, set the last error to E_OUTOFMEMORY and stay in the Hands Off After Save state.	Normal or Hands Off After Save.
SaveCompleted(*pMessage* == NULL)	Set the last error to E_INVALIDARG or E_UNEXPECTED	Hands Off After Save
HandsOffMessage, **Save**, or **InitNew**	Set the last error to and return E_UNEXPECTED	Hands Off After Save
Load	Load the form object with data from the target message. This call can occur when the form object is going to the next or previous message in a folder.	Normal
GetLastError	Return the last error.	Hands Off After Save
Other **IPersistMessage** methods or methods from other interfaces	Set the last error to and return E_UNEXPECTED	Hands Off After Save

About the Hands Off From Normal State

The Hands Off From Normal state is very similar to the Hands Off After Save state. It is part of the process of saving the contents of a form to permanent storage. When in this state, the form object should refrain from making changes to the in-memory copies of values of the message's properties, since there may not be another opportunity to save those changes. Legal transitions from the Hands Off From Normal state are described in the following table.

IPersistMessage method	**Action**	**New State**
SaveCompleted(*pMessage* != NULL)	Replace the message object's message with *pMessage*, which is the replacement for the message revoked by the previous call to **HandsOffMessage**. The data in the new message is guaranteed to be the same as in the revoked message. The message should not be marked as clean, nor should the **IMAPIViewAdviseSink::OnSaved** be called after this call. If the **SaveCompleted** call succeeds, enter the Normal state. Otherwise, stay in the Hands Off From Normal state.	Normal or Hands Off From Normal.
SaveCompleted(*pMessage* == NULL)	Set the last error to E_UNEXPECTED.	Hands Off From Normal
HandsOffMessage, **Save**, **InitNew**, or **Load**	Set the last error to E_UNEXPECTED	Hands Off From Normal
GetLastError	Return the last error.	Hands Off From Normal
Other **IPersistMessage** methods or methods from other interfaces	Set the last error to E_UNEXPECTED	Hands Off From Normal

Writing Form Server Code

A form server can be thought of as three things: a Win16 or Win32 program that displays an interface and handles windows messages via the standard Windows message pump mechanisms, an object that registers its class factory with OLE and is activated by OLE automation methods, and a MAPI object that follows MAPI's rules for interactions with other MAPI components. Your code has to handle all three of those broad requirements simultaneously.

See the *OLE Programmer's Reference* for details about registering your form server's class factory. Handling windows messages and displaying an interface are standard Windows programming techniques that don't have any special requirements with respect to MAPI forms. Again, the *Win32 Software Development Kit* has details about Windows programming. This document contains what you need to know to implement the required and optional MAPI form interfaces so that they follow MAPI's rules for interactions with other MAPI components − primarily the MAPI Form Manager and messaging client applications.

All of the interfaces that you can use when implementing form servers are derived − either directly or indirectly − from the OLE base class **IUnknown**. This means that all your implementations of these interfaces will need to have **QueryInterface**, **AddRef**, and **Release** methods. You can save yourself a lot of work if you use multiple inheritance to implement all of the required interfaces in one new class of your own, so that all the interfaces you are using can share a single implementation of the required **IUnknown** methods. See the OLE documentation for the **IUnknown** class methods for details on the **QueryInterface**, **AddRef**, and **Release** methods. There are no special considerations with respect to MAPI form servers for these methods.

While not all of the MAPI form interfaces are mandatory for all form servers, the methods in any given interface are. That is, if you choose to implement a particular interface, you must implement all of the methods in the interface. This is different from the situation with some other MAPI components, such as message transports. Fortunately, the methods in the MAPI form interfaces are relatively straightforward so implementing all of them does not put a great burden on developers.

The MAPI form interfaces are independent of the type of development tool used to create a form server. This allows forms created using different development tools such as Microsoft Visual C++, Microsoft AppStudio and tools from other vendors. The only requirement is that all form servers must support the required MAPI form interfaces.

Note It is possible to implement form servers using a mix of languages. For example, you could use Visual Basic for the form's user interface and C++ for the underlying MAPI code. However, this document relates only to C++ development of form servers.

Not all of the MAPI interfaces relating to forms are required by all form servers. The optional interfaces allow you to implement some advanced form functions that are not needed by most form servers. The following table lists the interfaces, what they are for, and whether you must implement them.

Interface	Usage	Status
IMAPIForm	The primary interface that clients use when loading form servers, executing form verbs, and shutting down form servers. Also the OLE **IUnknown**- derived interface used to inform other OLE components what interfaces a form object implements.	Required
IPersistMessage	Used when loading messages into and saving messages	Required

	from form objects	
IMAPIFormAdviseSink	Used by form objects to keep track of messaging client status and to find out whether the form object is capable of displaying the next or previous message in a folder.	Optional
IClassFactory	OLE class factory interface used by form objects for compliance with the OLE class factory mechanism.	Required
IMAPIFormFactory	Used if your form server supports more than one type of form. In this case, the **IMAPIFormFactory** interface allows client applications to access the multiple **IClassFactory** interfaces (one per type of form that your form server supports) that your form server must also implement.	Optional

Declaring Form Interfaces

You can simplify the declarations of your implementations of MAPI form interfaces by using the MAPI_*interface*_METHOD macros, where *interface* is a form interface defined in the MAPIFORM.H header file. You are not required to use these macros, but if you do not you should take particular care that your declarations conform to the declarations in the MAPIFORM.H header file. For example, you could declare your form server's form object class like this:

```
class CMyForm : public IPersistMessage, public IMAPIForm,
                       public IMAPIFormAdviseSink
{
public:
     CMyForm(CClassFactory *);     // ctor takes a class factory object
     ~CMyForm(void);

// MAPI methods that need to be implemented.
     MAPI_IUNKNOWN_METHODS(IMPL);
     MAPI_GETLASTERROR_METHOD(IMPL);
     MAPI_IPERSISTMESSAGE_METHODS(IMPL);
     MAPI_IMAPIFORM_METHODS(IMPL);
     MAPI_IMAPIFORMADVISESINK_METHODS(IMPL);

// other implementation specific stuff:

};
```

Integrating MAPI Form Server Code with Windows Code

Recall that your form server is a Win32 or Win16 program. As such, there are some tasks related to loading your form server into memory and exiting cleanly. Like all Windows programs, the entry point for your form server is the **WinMain** function. This function is the appropriate place to perform the following tasks:

- Creating and registering a window class so that your form server can interact with other OLE components.
- Creating and registering a window class or classes for your form objects' user interfaces.
- Calling the **MAPIInitialize** function. **MAPIInitialize** handles the required OLE initialization for you as well. This must be done once per instance of your form server.
- Registering a global atom with a string representation of the form server's class identifier (CLSID). This atom should exist for the lifetime of the form server.
- Calling the OLE function **CoRegisterClassObject** to register your form server's class factory with OLE.
- Creating a main window to receive messages. This window probably does not need to be visible since the user will be interacting with the specific windows associated with individual form objects. However, during development, the main window can be a convenient place for debugging output or control of your form server.
- Creating a message loop which runs for the lifetime of the form server, translating and dispatching windows messages to active form objects.

When your form server exits, it should perform the following tasks:

- Call the OLE function **CoRevokeClassObject** to revoke your message class' OLE registration
- Call **MAPIUninitialize** to properly close the form server's connection to MAPI
- Delete the global atom containing the string representation of the class ID.

Implementing the IMAPIForm Interface for Form Servers

IMAPIForm is the primary interface for form objects. The following table lists the **IMAPIForm** methods that are required.

Method	Use
SetViewContext	Sets a form view context as the current view context for a form.
GetViewContext	Returns the current view context for a form.
ShutdownForm	Closes a form.
DoVerb	Requests a form object perform one of its verbs.
Advise	Registers a form viewer for notifications about changes to a form.
Unadvise	Removes a form viewer's registration for notification of form object changes.

Implementing IMAPIForm::SetViewContext for Form Servers

When the **SetViewContext** method is called, it must do the following:

- If the form object already has a view context, call the existing view context's **SetAdviseSink** method with a parameter of NULL, and release it.
- Store the new view context passed in by the caller.
- If the new view context is not NULL, call its **AddRef** method, call its **SetAdviseSink** method with a pointer to the form server's implementation of the **IMAPIFormAdviseSink** interface, and call its **GetViewStatus** method in order to get the view context's current set of status flags.
- Update any user interface elements that depend on the view context.

Implementing IMAPIForm::GetViewContext for Form Servers

The **GetViewContext** method needs to copy the form object's view context pointer into the view context pointer passed in by the caller, or NULL if there is no view context yet. It should return hrSuccess if there is a valid a view context pointer, or ResultFromScode(S_FALSE) otherwise.

Implementing IMAPIForm::ShutdownForm for Form Servers

The **ShutdownForm** method needs to perform the following tasks:

- Check that this method has not already been called. This is an unlikely event, but should be checked for anyway. If it has, return a status code based on E_UNEXPECTED.
- Check the form object's user interface to see if there are any changes to save. If there are, and if the *ulSaveOptions* parameter to this method indicates that changes should be saved, call the **SaveMessage** method in the form object's view context object.
- Call the form object's **AddRef** method. This is done to protect the form object because its internal data structures are referenced in some of the following actions.
- Call the **SetAdviseSink** method in the form object's view context object with a NULL parameter, and call the view context object's **Release** method.
- Call the **OnShutdown** method in any **IMAPIViewAdviseSink** that have registered for notification with the form object.
- Release any **IMessage** and **IMAPIMessageSite** objects held by the form object.
- Destroy the form object's user interface window, if one exists.
- Call the form object's **Release** method. This matches the earlier **AddRef** call.

Note After these actions have been completed, the only valid methods on the form object that may be called are those from the **IUnknown** interface.

Implementing IMAPIForm::DoVerb for Form Servers

The **DoVerb** method handles execution of the different verbs that the form object supports. This method is called with a verb number and a view context pointer. This method needs to perform the following tasks:

- If the view context pointer is not NULL, this indicates that the form object should use the specified view context while executing the verb rather than the one it already has. Make whatever changes to the form object's internal data structures are necessary for this.
- Execute whatever code is necessary for the verb indicated by the verb number.
- If necessary, restore the original view context.
- If an unknown verb number was passed in, return a result based on MAPI_E_NO_SUPPORT. Otherwise, return a result based on the success or failure of whatever verb was executed.
- Shut the form object down. It is the form object's responsibility to shut down after completing the **DoVerb** method.

Implementing IMAPIForm::Advise for Form Servers

The **Advise** method simply needs to store the **IMAPIViewAdviseSink** pointer parameter, call the advise sink object's **AddRef** method, and return a connection number in the ULONG pointer parameter. This method should return S_OK on success or E_OUTOFMEMORY on failure.

Typically, one implements a helper class to handle the advise sink and notification mechanisms, in which case this method is simply a wrapper around a call to some method in that support class for registering objects for notification.

Implementing IMAPIForm::Unadvise for Form Servers

The **Unadvise** method is called with a connection number as obtained by an earlier call to the **Advise** method. This method needs to call the **Release** method for the **IMAPIViewAdviseSink** object that was registered for that connection number in the earlier call to the **Advise** method. This method should return S_OK on success, or E_INVALIDARG if the connection number passed in does not correspond to a registered advise sink.

Implementing the IPersistMessage Interface for Form Servers

IPersistMessage is the interface that is used for loading and saving message data to and from form objects. The following table lists the **IPersistMessage** methods that are required.

Method	Use
GetClassID	Returns a form's message class identifier.
IsDirty	Checks a form for changes made since the form was last saved.
InitNew	Provides a form with a base message on which to build a new message.
Load	Loads a form from a specified message.
Save	Saves a revised form back to the message from which it was loaded or created.
SaveCompleted	Returns a message to a form after a save, submission, or other operation.
HandsOffMessage	Causes a message to release its message object.

Implementing IPersistMessage::GetClassID for Form Servers

The **GetClassID** method simply needs to copy the form server's class identifier (CLSID) in the class identifier pointer parameter, and return hrSuccess.

Implementing IPersistMessage::IsDirty for Form Servers

Callers use the **IsDirty** method to determine whether the message object has unsaved data. If it does, this method should return S_OK. If it does not, this method should return S_FALSE.

Implementing IPersistMessage::InitNew for Form Servers

The **InitNew** method is called when the user composes a new message of your form server's message class. This method should only be called when the form object is in the Uninitialized, Hands Off After Save, or Hands Off From Normal state. If this method is called while the form object is in any other state, it should return a result based on E_UNEXPECTED.

The clean or dirty status of a message typically controls whether messaging clients display a dialog asking the user whether to save changes. Therefore, new messages should always start out clean since the user has had no opportunity to make changes. However, if the constructor for your form object sets the value of any computed properties, and it is important for that computed value to be saved, then the form object starts out effectively neither clean nor dirty. It is not clean because it has useful data in it, but it is not dirty because you don't want users to see a Save Changes dialog box in the case where they create a new message of your form server's message class and then immediately close it. In this situation, you should implement the form's interface to detect this condition and respond appropriately. One option is to show the user a more appropriate dialog box to prompt if the message should be saved. If the user indicates that the message should be saved then the form object can then save its data, mark itself as clean, and exit normally.

If the form object has a message site object other than the one passed into this method, it should be released since it will not be used. After that, the pointers to the message site and message object passed to this method should be stored, and the **AddRef** method on those objects should be called.

The message flags and status should be set to something appropriate for your message class. Many message classes, for example, set the message's flags to MSGFLAG_UNSENT for new message objects.

If the form object has a valid user interface pointer, the user interface for the message object should be displayed.

If nothing has gone wrong with the above actions, the message object should move to the Normal state. At this point the message object should use its notification method to inform any interested parties that a new message has been created. Finally, **InitNew** should return S_OK.

Implementing IPersistMessage::Load for Form Servers

The **Load** method is called when an existing message is loaded into a form object so that the user can interact with it. This method should only be called when the form object is in the Uninitialized, Hands Off After Save, or Hands Off From Normal states. If this method is called while the form object is in any other state, it should return a result based on E_UNEXPECTED.

As with the the **InitNew** method, the **Load** method should release an existing message site and message object if the form object already has them since the ones passed into this method will be used instead. After releasing the existing objects, the **Load** method should store the message site pointer and message pointer passed in, and call the **AddRef** method on those objects.

At this point, the properties on the message object passed into this method can be copied to the form object.

Having loaded the properties, the form object should move into the Normal state and display its user interface. Finally, the **Load** method should use the form object's notification mechanism to inform any interested parties that the message has been loaded. The same notification can be used for loading a message as for creating a new message.

If no errors occur during this process, the **Load** method should return S_OK. If any errors occurred while reading the properties from the message object passed in, the error from the failed action should be returned instead.

Implementing IPersistMessage::Save for Form Servers

The **Save** method is called by client applications to instruct the form object to save changes to a form object's property values back to a message object – often the message object from which the form object was loaded or initialized. The **Save** method should only be called when the form object is in the Normal state. If this method is called while the form object is in any other state, it should return a result based on E_UNEXPECTED.

The caller passes a flag to the **Save** method to indicate whether the form object should save its data to the same message object that the form object was originally loaded from. If so, then either the message pointer passed into the **Save** method is NULL or is the same as the one that the form object already has. In either case, it is safe for the **Save** method to use its existing message object pointer as the one to save changes into.

If that flag indicates that the message object is a different one, then the **Save** method needs to copy everything in the original message object into the new message object before saving changes. The **IMAPIProp::CopyTo** method is useful for this.

At this point the **Save** method should copy the data from the form object – or its user interface – to the properties in the target message object and enter the No Scribble state.

If no errors occur during this process, the **Load** method should return S_OK. If any errors occurred while saving the properties to the target message object, the error from the failed action should be returned instead.

Note The form object should never make changes to the underlying message object's property values except during the **Save** method.

Implementing IPersistMessage::SaveCompleted for Form Servers

The **SaveCompleted** method is called by the client to inform the form object that the process of saving changes to a message object has been completed. The **SaveCompleted** method should only be called when the form object is in the Hands Off From Normal, Hands Off After Save, or the No Scribble state. If this method is called while the form object is in any other state, it should return a result based on E_UNEXPECTED.

If the form object is in the Hands Off From Normal or Hands Off From Save state and the message pointer parameter is NULL, **SaveCompleted** should return a result based on E_INVALIDARG.

At this point, the message should save its current state in a variable and enter the Normal state.

There are several possible actions that the **SaveCompleted** method can perform, depending on what the message pointer parameter contains, and what state the message is in:

- If the message pointer parameter is NULL and the form object saved its data to the same message object as it was originally loaded with, the form object should mark itself as clean, use its notification mechanism to inform interested parties that the form object's data has been saved, and return S_OK. If the message pointer parameter is NULL but the form object saved its data to a different message object, then this method should return S_OK and take no other action.
- If the form object was in the Hands Off From Normal state when this method was called, then it should release its current message object, replace it with the object in the message pointer parameter, call the **AddRef** method on the new message object, and return S_OK.
- If the form object was in the Hands Off After Save state when this method was called, the **SaveCompleted** method should use its notification mechanism to inform interested parties that the form object's data has been saved, should mark itself as clean, and return S_OK.
- If the form object was in the No Scribble state when this method was called, the **SaveCompleted** method should release its current message object, replace it with the message pointer parameter, call the **AddRef** method on the new message object, and then perform the same actions as in the previous case.

Implementing IPersistMessage::HandsOffMessage for Form Servers

The **HandsOffMessage** method is used to cause a form object to release the message object it is using. This method should only be called when the form object is in the Normal or No Scribble state. If this method is called while the form object is in any other state, it should return a result based on E_UNEXPECTED.

If the form object is in the Normal state it should enter the Hands Off From Normal state, otherwise it should enter the Hands Off After Save state. This method should then call the **Release** method on its message object, and return S_OK.

Implementing the IMAPIFormAdviseSink Interface for Form Servers

IMAPIFormAdviseSink is the interface that client applications – and other interested parties, if any – use to get notifications when certain events happen within a form object. The following table lists the **IMAPIFormAdviseSink** methods that are required.

Method	Use
OnChange	Notifies a form object about a change in a form viewer's status.
OnActivateNext	Identifies whether the message class of the next message to display can be handled by the current form object.

Implementing IMAPIFormAdviseSink::OnChange for Form Servers

The **OnChange** method is called to notify the form object about changes in a form viewer's status. The specific implementation of this method is entirely dependant on the specifics of your form. Most form objects use this method to alter their user interface − for example, enable or disable menu commands or buttons − to match the viewer status flags parameter.

Implementing IMAPIFormAdviseSink::OnActivateNext for Form Servers

The **OnActivateNext** method is called by client applications to determine whether a form object is capable of displaying the next message in a folder. Strictly speaking, it could be any other message, but this method is designed to make the process of reading multiple messages of the same message class more efficient by allowing client applications to re-use form objects whenever possible.

The **OnActivateNext** method receives a message class string, a message status code, message flags, as its parameters, and an LPPERSISTMESSAGE pointer to store its return value in. It can decide, based on the values of any of those parameters, how to respond.

Most form objects will use the message class string as the deciding factor; if it is the same as the form object's message class then the form object is able to handle the next message. The other common case is when the message class string is a substring of the form object's message class string (that is, the form object is a subclass of the next message, and thus may be able to handle the next message), although this case is not one that your form object is required to support. The **OnActivateNext** method can further restrict the decision based on the sent/unsent status of the next message, etc.

If your form object cannot or will not handle the next message, this method should return S_FALSE.

If your form object can handle the next message, this method should store NULL in the LPPERSISTMESSAGE pointer parameter and return S_OK.

If your form object can't handle the next message but can create a new form object that can handle the next message, it should:

- Call its class factory to create a new form object instance.
- Store that instance in the LPPERSISTMESSAGE pointer parameter
- Return S_OK.

A minimal implementation of this method would simply return S_FALSE in all cases, although this is highly discouraged because it results in degraded client application performance.

Implementing the IClassFactory Interface for Form Servers

IClassFactory is the OLE interface that client applications use to create new form objects of your form server's message class. The following table lists the **IClassFactory** methods that are required.

Method	Use
CreateInstance	Creates a new form object.
LockServer	Locks the form server in memory so that startup overhead can be avoided when multiple form objects are created.

The *OLE Programmer's Reference* has all the information necessary to implement these methods.

Implementing the IMAPIFormFactory Interface for Form Servers

IMAPIFormFactory is the interface that client applications use to create new form objects when your form server supports more than one message class – that is, more than one type of form object. The following table lists the **IMAPIFormFactory** methods that are required.

Method	Use
CreateClassFactory	Creates a new class factory for a form server.
LockServer	Locks the form server in memory so that startup overhead can be avoided when multiple form objects are created.

Implementing IMAPIFormFactory::CreateClassFactory for Form Servers

The **CreateClassFactory** method is called by form viewers to obtain a class factory object for form servers that implement multiple message classes. This method receives a class identifier (CLSID) as a parameter. Based on that parameter, this method can determine the specific kind of class factory object to return. You can have a single class factory implementation that creates appropriate class factory instances on demand, or one multiple class factory implementations, one for each message class.

Implementing IMAPIFormFactory::LockServer for Form Servers

The **LockServer** method is very similar to the **IClassFactory::LockServer** method. Essentially this method maintains a count of how many times it has been called and as long as that count is greater than 0, prevents the form server from being unloaded from memory. You can use the OLE function **CoLockObjectExternal** to implement this. See the *OLE Programmer's Reference* for details.

Creating A Configuration File

Configuration files exist to provide information about a form both to the form manager being used and to client applications. A configuration file is a file with a .CFG extension, and has a format similar to a Windows initialization file. It is a plain text file with a number of sections. Each section begins with a section name, enclosed in square brackets. Each section contains one or more lines which define values and settings relelvant to that section. Values have one of the following types: string, displayed string, platform string, pathname, integer, guid. The sections of a .CFG file are described below.

A configuration file contains an extensive specification for a form, including the properties published by the form for use by messaging clients, the verbs implemented by the form, and the platforms supported by the form.

Configuration File Format

A configuration file is formatted file created by form developers to define a form. Because configuration files are used by form managers to load forms, each form must be defined using a configuration file. Configuration files must have the .CFG filename extension. The file follows the general syntax of a Windows initialization file (.INI file). It is divided into named sections, and each section contains a series of entries and values. Values have one of the following types: string, displayed string, platform string, path, integer, or globally unique identifier (GUID). The sections of a configuration file are described in following topics. You can create a configuration file with any text editor or word processor that is capable of saving plain text files.

The [Description] Section in a Form Server Configuration File

The [Description] section lists all properties of the form that are associated with controls in the form's user interface,plus attributes that are used in locating the form. The **MessageClass**, **Clsid**, and **DisplayName** entries, which identify the name of the form's message class, its GUID, and the message class's display name, respectively, are required entries used to locate the form within the form library. The remaining entries are optional. The format of the [Description] section is:

[Description]
MessageClass = *string*
Clsid = *guid*
DisplayName = *displayed string*
SmallIcon = *path*
LargeIcon = *path*

;optional entries

Category = *displayed string*
Subcategory = *displayed string*
Comment = *displayed string*
Owner = *displayed string*
Number = *displayed string*
Version = *integer*
Locale = *string*
Hidden = *integer*
DesignerToolName = *string*
DesignerToolGuid = *clsid*
DesignerRuntimeGuid = *clsid*
ComposeInFolder = 0|1
ComposeCommand = *string*

The Category and Subcategory entries are used by form installers to set up the default categorization of forms within client application's user interface. For example a hierarchy could be set up where "Help Desk" is the category and "Software" and "Hardware" were the subcategories. This categorization can then be used by viewer applications to display messages in a more organized way. The Comment, Owner, and Number entries are all comment strings that appear in client application's user interface. These are form specific properties that can be used at the discretion of the form developer. For example, the Comment entry can be used to indicate the purpose of the form, the Owner entry used to indicate the person or organization responsible for maintaining the form, and the number used to track different version of the form. For the Comment entry, up to ten lines of comments can be included. The first line of comments uses the word "Comment" as the key, the second line of comments uses "Comment1" as the key, and so on through "Comment9."

The LargeIcon and SmallIcon entries are used to specify the path for the icon resources used to display icons in the client application's user interface, typically this is for table rows that include the PR_ICON or PR_MINI_ICON property columns. Icon file names can be specified as pathnames relative to the directory where the configuration file is installed. The Version entry is used to indicate the version number of the form. Locale is the three-letter language identifier of the destination form library. A list of these identifiers may be found in the *Win32 Programmer's Reference*.

The Hidden entry indicates whether the form should be displayed in a form library provider's user interface: 1 indicates that the file is hidden and 0 indicates that the form is visible. An example configuration file is shown following.

The ComposeInFolder entry controls whether the form is designed to be placed in the current folder or in the user's Inbox when the user saves the message while composing it: 1 indicates that the form should go in the current folder and 0 indicates that it should go in the Inbox.

The ComposeCommand entry is the string to be placed in the client application's compose menu. If this is not specified, the DisplayName entry will be used.

```
[Description]
MessageClass = IPM.Help
Clsid = {00020D31-0000-0000-C000-000000000046}
DisplayName = Help Desk Request Form

;optional entries

Category = Help Desk Requests
Subcategory = New Requests
Comment = Use this form to request network assistance
Owner = Help Desk
Number = 1
SmallIcon = C:\WINDOWS|EFORMS\HELPDESK\HDSMALL.ICO
LargeIcon = C:\WINDOWS|EFORMS\HELPDESK\HDLARGE.ICO
Version = 1.00
Locale = enu
Hidden = 0
ComposeInFolder = 0
ComposeCommand = &Help Desk Request
```

The [Extensions] Section in a Form Server Configuration File

The [Extensions] section lists the extended attributes of the form, typically a named property set, which are any attributes beyond the basic ones listed in the [Description] section of the configuration file. Extended attributes are properties returned from calls to the **GetProps** method of the **IMAPIFormInfo** object with the high bit set in the property tag. Client applications can determine a form's extended attributes, if any, by retrieving these tags. To do so, clients call the **IMAPIProp::GetIDsFromNames** method, passing in the names of the form's properties and call the **IMAPIProp::GetProps** method to get the properties.

Each entry in the [Extensions] section references a subsequent section that has a name with the syntax [**Extension**.*string2*].

[Extensions]
Extension.*string1* = *string2*

Each extension property section defines one extension attribute using the MAPI named property syntax. The property type must be either PT_LONG or PT_STRING8. Property sets containing named strings are not supported. The format of the [Extension] section is:

[Extension.*string2***]**
Type = *integer*
NmidPropset = *guid*
NmidInteger = *integer*
Value = *string* **|** *integer*

An example of an [Extensions] section and a subsequent related section is shown following.

```
[Extensions]
Extension.A = 1

[Extension.1]
Type = 30
NmidPropset = {00020D0C-0000-0000-C000-000000000046}
NmidInteger = 1
Value = 11220000
```

The [Platforms] Section in a Form Server Configuration File

The [Platforms] section lists the complete set of platforms supported by this form. Each platform entry consists of the prefix **Platform.***string*, where *string* is an arbitrary string code for the platform. Each string corresponds to the CPU entry of an individual [Platforms] sections. Each entry in a [Platforms] section defines a *platform string* that references a subsequent [**Platform.***platform string*] section as shown here.

[Platforms]

Platform.*string* = *platform string*

Following is an example of a [Platforms] section.

```
[Platforms]
Platform.1 = NTx86
Platform.2 = Win95
```

Each [**Platform.***platform string*] section contains the two required entries, **CPU** and **OSVersion**. The **CPU** entry specifies the processor, and the OSVersion entry specifies the operating system. Valid CPU entries are described in the following table.

CPU Entry	**Processor**
Ix86	Intel 80x86 and Pentium series processors, as well as equivalent processors from AMD, Cyrix, NextGen and other manufacturers.
MIPS	MIPS R4000 series processors.
AXP	Digital Equipment Corporation Alpha AXP processor.
PPC	Motorola Power PC series processors.
M68	Mororola 68x00 series processors.

Valid OSVersion entries are described in the following table.

OSVersion Entry	**Operating System**
Win3.1	Windows 3.1 and Windows for Workgroups 3.11.
WinNT3.5	Windows NT 3.5 or lower.
Win95	Windows 95.
WinNT4.0	Windows NT 4.0.
Mac7	Macintosh System 7.

Additionally, the [**Platform.***platform string*] section must contain either a **File** or **LinkTo** entry. The **File** entry lists the form server application executable file that the form library maintains and loads into a new subdirectory in the disk cache when the form is launched. If a **LinkTo** entry is used instead, it contains the name of a different platform string from which the **File** information is taken. This is useful if one version of a form supports multiple platforms.

The Registry entry is used whenever the File entry is used, it identifies the registry key for the form library where the executable file for the form server application is stored. Strings preceded by a backslash (\ ) are placed at the root of the registry. Strings not preceded by a backslash are placed in the HKEY_CLASSES_ROOT\CLSID\GUID\ registry key, where *GUID* is the GUID of the form. The characters "%d" can be used to indicate the pathname of the directory from which the configuration file has been read. This is useful for specifying other files with pathnames relative to the configuration file. Multiple File or Registry entries can be specified by using File or Registry as a prefix followed by any other text. The format for the [**Platform.***platform string*] section is:

[Platform.*platform string***]**
CPU = *string*
OSVersion = *string*
File = *path*
LinkTo = *string*
Registry = *string*

The following are two example [**Platform.***platform string*] sections, one using the **File** entry and one using the **LinkTo** entry.

```
[Platform.NTx86]
CPU = ix86
OSVersion = WinNT3.5
File = \helpdesk.exe
Registry = Local Server = %d\helpdesk.exe

[Platform.Win95]
CPU = ix86
OSVersion = Win95
LinkTo = NTx86
```

The [**Platform.***platform string*] section is ignored when adding a form to the local form library, when it is assumed that the installer has placed the files constituting the message class handler into accessible local storage as named in the handler's section in the OLE registry, and has done the OLE registration in the system's registry.

The [Properties] Section in a Form Server Configuration File

The [Properties] section lists the complete set of properties that the form uses and publishes; that is, the properties it creates in its custom messages that MAPI client applications can use when displaying columns, filtering contents tables, setting up search-results folders, and so on. Each entry in this property list references a subsequent [**Property.***string*] section as shown following.

[Properties]
Property.*string* = *string*

The format of a [**Property.***string*] section is:

[Property.*string***]**
Type = *integer*
NmidPropset = *guid*
NmidString = *string*
NmidInteger = *integer*
DisplayName = *string*
Flags = integer
SpecialType = 0|1
Enum1 = *string*

Each [**Property.***string*] section describes a single property. The Type entry specifies the MAPI property type, for example 3 (PT_I4), of the property. The NmidPropset entry is optional; together with either the NmidString entry or the NmidInteger entry, the NmidPropset entry gives the name of the property. NmidString gives the name of the property, while NmidInteger gives the ID of the property. NmidString and NmidInteger are mutually exclusive.

If set, NmidPropset should contain the name of the property set; if absent, NmidPropset is set to a default based on the following rule: If NmidInteger is present and its value is less than 0x8000, NmidPropset is set to PS_MAPI. If the value of NmidInteger is set to an integer greater than 0x8000, or if it is absent, NmidPropset is set to PS_PUBLIC_STRINGS.

The DisplayName entry contains the label for the property. The SpecialType entry, if present and nonzero indicates that this property is a special property. At present, the only special property type defined is SpecialType = 1, which indicates string enumerated properties. If SpecialType is set to 1, the Enum1 entry references the [**Enum1.***string*] section.

Following is an example of a [Properties] section and a [Properties.*string*] section.

```
[Properties]
Property.1 = Fire Hazard
Property.2 = Safe

[Property.Fire Hazard]
Type = 1
NmidPropSet = {E47F4480-8400-101B-934D-04021C007002]
NmidString = FireHazard
DisplayName = Fire Hazard
SpecialType = 1
Enum1 = HazardEnum
```

The Enum1 section in the preceeding example references to a subsequent [**Enum1.***string*] section describing an enumeration of a particular type. Such an enumeration associates the first property in the [**Property.***string*] section with an integer property, called the index. Such an enumeration also contains a list of the possible values that the display-index pair can assume. Specifying a property type for the enumeration is unnecessary because by definition an Enum1 entry always has the PT_I4 type. The format for the [**Enum1.***string*] section is:

[**Enum1.***string*]
NmidPropset = *guid*
NmidString = *string*
NmidInteger = *integer*
EnumCount = *integer*
Val.*integer***.Display** = *string*
Val.*integer***.Index** = *integer*

The following is an example property definition for an enumerated property named Fire Hazard with possible values of Low, Medium, and High.

```
[Properties]
Property1 = Fire Hazard

[Enum1.HazardEnum]
IdxNmidPropset={E47F4480-8400-101B-934D-04021C007002]
IdxNmidString=FireHazardEnum
EnumCount = 3
Val.1.Display = Low
Val.1.Index = 1
Val.2.Display = Medium
Val.2.Index = 2
Val.3.Display = High
Val.3.Index = 3
```

[**Enum1.***string*] sections can be used by applications for two purposes: to speed up the filtering of properties by using the index rather than the string and to sort by a different order than the alphanumeric order of the string values. For example, sorting could be done based on Low-Medium-High order rather than High-Medium-Low order.

The [Verbs] Section in a Form Server Configuration File

The [Verbs] section lists the complete set of verbs supported by the form. The format of the [Verbs] section is:

[Verbs]
Verb1 = *string*

Following is an example of a Verbs section.

```
[Verbs]
Verb1=1
Verb2=2
```

Each verb is defined in a separate [**Verb.***string*] section. A [**Verb.***string*] section describes a single verb offered by the form. The DisplayName entry in a [**Verb.***string*] section specifies the command name displayed in the user interface. The Code entry corresponds to the verb number passed in the **IMAPIForm::DoVerb** method. The syntax for the [**Verb.***string*] section is:

[**Verb.***string*]
DisplayName = *displayed string*
Code = *integer*
Flags = *integer*
Attribs = *integer*
Following is an example of a [**Verb.***string*] section.

```
[Verb.1]
DisplayName=Reply
code=1
Flags=0
Attribs=2

[Verb.2]
DisplayName=Delete
Code=2
Flags=0
Attribs=2
```

Verbs listed in this section are retrieved by a client using the **IMAPIFormInfo::CalcVerbSet** method. Verbs are activated by calling the form's **IMAPIForm::DoVerb** method and passing it the code number of the verb to be performed.

Form Server Interactions

This section describes interactions between client applications, the MAPI form manager, form servers, and form objects for a variety of common actions such as opening a message and composing a new message.

Retrieving Form Properties

To issue a query meaningful to a custom message type, an application needs to know the properties expected on that message. To get a list of properties used by a custom message class, a client application queries the MAPI form manager. The form manager gets this information from the appropriate form server's configuration file so that this information can be used by client applications without the overhead of activating the form server itself. To do this, the client application calls **IMAPIFormMgr::ResolveMessageClass**, like this:

```
IMAPIFormInfo *pfrminf = NULL;
hr = pfrmmgr->ResolveMessageClass("IPM.Demo", 0L, NULL, &pfrminf);
```

Note that the third argument to **ResolveMessageClass** is the folder containing the associated contents table that the query will search for form servers. "NULL" indicates that the form manager should search all available form containers. If the query is to run against a particular folder it is better to include the appropriate **IMAPIFolder** pointer instead.

Launching a Form to Read a Message

Form server implementors should expect the following sequence of method calls to their form server and form objects when a client application loads a message:

1. The client application opens the form manager with a call to **MAPIOpenFormMgr**.
2. The client application calls **IMAPIFormMgr::LoadForm**, which returns an object with **IMAPIForm**. The form manager may be released now if it won't be used for further form activations. Note that a call to **LoadForm** may take some time, since the form manager may have to install the form server's executable files before proceeding.
3. Optionally, the client application can prepare an **IMAPIViewContext** to control operations which may cause the form object to load the previous or next message in the folder. The client application can use **IMAPIForm::SetViewContext** to change the default view context that was set in the **LoadForm** call.
4. The client application calls **IPersistMessage::Load** to load message data into the form object.
5. The client application calls **IMAPIForm::DoVerb** to invoke the "open" verb, passing the the optional ViewContext interface pointer.

Launching a New Compose Form

Form server implementors should expect the following sequence of method calls to their form server and form objects when a client application opens a new message for composing:

1. The client application calls **IMAPIFormMgr::ResolveMessageClass** to get class information about the form server's message class.
2. The client application calls **IMAPIFormMgr::CreateForm** to get a new form object.
3. The MAPI form manager loads the form server, if it is not already in memory and gets an **IMAPIForm** object from the form server.
4. The client application takes the resulting **IMAPIForm** object and calls the **QueryInterface** method to get the object's **IPersistMessage** interface.
5. The client application calls **IPersistMessage::InitNew** to associate the form object with an **IMessage**, view context, and advise sink objects.
6. The client application calls **IMAPIForm::DoVerb** to invoke the "open" verb.

Launching a Form Server

The series of interactions that occurs when a form is loaded from persistant storage − that is, from a form library − to display a message is as follows:

1. The messaging client gets the message's message class, message flags and message status. This step is optional; if these pieces of data are not provided in step 2, the form manager will retrieve them.
2. The messaging client calls **IMAPIFormMgr::LoadForm** with the target message.
3. The form manager loads the form server from the appropriate form library. If the form server for the target message is not installed, the form manager installs the form's executable files as well.
4. The form manager calls **IUnknown::QueryInterface** on the form object to obtain the form object's **IMAPIForm** and **IPersistMessage** interfaces.
5. The form manager calls **IPersistMessage::Load** with the message site and message interfaces from the viewer object.
6. The form object calls back to the messaging client's **IMAPIMessageSite::GetSiteStatus** method.
7. The form manager calls the form object's **IMAPIForm::SetViewContext** with the view context interface from the messaging client.
8. The form object call back to the messaging client's **IMAPIViewContext::SetAdviseSink** method.
9. The form object calls back to the messaging client's **IMAPIViewContext::GetViewStatus** method.
10. The messaging client calls the form object's **IMAPIForm::Advise** method with the view context interfaces from the viewer object and the message site object.
11. The messaging client calls the form object's **IMAPIForm::DoVerb** method.
12. The form object creates its user interface, if necessary, and interacts with the user.

Installing a Form into a Library

The default MAPI form manager – the one supplied with the MAPI SDK – does not provide a user interface for installing forms in the various form libraries. Because of this, you will have to create a small application – or detailed set of instructions – that users can use to install the form.

If you implement an install application, the series of actions it must do to install a form into a folder's associated contents table are as follows:

1. Call the **MAPIOpenFormMgr** function to open the form manager.
2. Use **IMAPIFormMgr::OpenFormContainer** or **IMAPIFormMgr::SelectFormContainer** to select and open the target container for the form.
3. Use the **IMAPIFormContainer::InstallForm** function to install the form.

Steps 4-6 are for installation into a local form library:

4. Copy all files to the appropriate place on the local disk, if installation is to the local form library on the user's workstation. If necessary, modify the configuration file to reflect current pathnames of components. The configuration file can contain relative pathnames in which case this step may not be necessary.
5. Do the appropriate OLE registration to associate the message type with the form server being installed.
6. If the form was installed into the local form library, copy the form's icon (.ico) and configuration (.cfg) files into the %WINDOWS%\FORMS\CONFIGS directory for auto-restore in the event that the form library is corrupted or deleted. This step is recommended but not mandatory.

Note Installation to a local form library can be simplified by replacing steps 1 and 2 with a call to **MAPIOpenLocalFormContainer**.

Developing a Hook Provider or Preprocessor

MAPI defines two types of extensions that it uses to allow custom code to be inserted into the message transmission and reception processes – hook providers and preprocessors. A hook provider, also called a spooler hook, can be called after an outbound message reaches the transport providers and before an inbound message is delivered to the default folder for its message class. Preprocessors operate on outbound messages only and are called before and after transport providers handle the message.

Since hook providers and preprocessors act during the message sending and receiving processes, you should familiarize yourself with the roles that other MAPI components play in these processes. Specifically, you should read About Sending Messages, Service Provider Basics, Developing a Transport Provider, and Developing a Message Store Provider. You do not need to learn the material in these sections in any great detail in order to implement a hook provider or preprocessor, but you should have a basic understanding of the way those components interact to send and receive messages.

There are several things to consider when choosing whether to use a hook provider or a preprocessor to implement your custom code:

- Whether your component needs to be able to operate on inbound messages. If it needs to act on inbound messages, you must use a hook provider, since preprocessors are not called for inbound messages.
- Whether your component needs to operate before or after transport providers for outbound messages. Preprocessors are called before transport providers, hook providers are called after.
- What your component needs to do. Preprocessors are typically used to modify a message's contents or recipient list before sending. For example, a preprocessor can be used to automatically add a signature to outgoing messages. Hook providers are typically used to archive messages or automatically file them in different folders; that is, to manipulate the relationship between the message and its message store although they can also modify a message's content. Hook providers cannot, however, modify a message's recipients. Only preprocessors can cause a message to be sent to different recipients than those entered by the user sending the message.
- Ease of implementation. Hook providers are actual MAPI service providers, albeit simple ones, and must follow the guidelines for service providers. However, preprocessors have to be registered by means of a transport provider. If you happen to also have a transport provider that you can modify to register the preprocessor, then creating a preprocessor is probably easier than creating a hook provider. If not, however, you will have to create a minimal transport provider whose sole job is to register the preprocessor.

For an overview of the process of creating hook providers and preprocessors, see Using Message Filtering to Manage Messages.

Sending Messages using Hook Providers and Preprocessors

The following diagram illustrates the interaction between MAPI components involved in the message sending process.

{ewc msdncd, EWGraphic, groupx836 0 /a "MAPI.WMF"}

The steps in this process are:

1. The message store provider, on behalf of a client application, notifies the MAPI spooler that there is a new message to send.
2. The MAPI spooler calls each registered preprocessor to act on the message.
3. The MAPI spooler delivers the message to any transport providers required.
4. Transport providers transmit the message to the underlying messaging systems.
5. The MAPI spooler calls each preprocessor's **RemovePreprocessInfo** function.
6. The MAPI spooler calls hook providers after all transport providers have completed their work.

Receiving Messages using Hook Providers

The following diagram illustrates the interaction between MAPI components involved in the message receiving process.

{ewc msdncd, EWGraphic, groupx836 1 /a "MAPI.WMF"}

The steps in this process are:

1. The underlying messaging system delivers a message to a transport provider.
2. The transport provider communicates with the MAPI spooler to get a pointer to a new message object created by the message store provider.
3. The transport provider writes the inbound message's data to the new message object in the message store.
4. The MAPI spooler calls all hook providers to operate on the message.
5. The hook providers communicate with the message store provider to move the message to a different folder, delete it, and so on.

About Hook Providers

Hook providers are MAPI service providers, like message store providers, transport providers, and address book providers. However, hook providers are by far the simplest type of provider to implement. Hook providers are intended to perform sorting and archiving types of tasks with messages. Some typical hook provider tasks are:

- Sorting incoming messages into folders.
- Filtering out unwanted incoming messages.
- Saving copies of outgoing messages in special folders.

Hook providers generally change the relationship between a message and its message store, although they can also change the content of a message.

Hook providers control whether they are called for inbound messages, outbound messages, or both by setting the HOOK_INBOUND and HOOK_OUTBOUND flags in the PR_RESOURCE_FLAGS property of the hook provider's profile section in the MAPISVC.INF file. For more details, see About the MAPISVC.INF File. In the event that multiple hook providers are being used, the hook providers are called in the order that they are installed into a user's profile.

About Hook Provider Entry Points

Hook providers can expose the following entry points in their DLL.

Entry Point	Purpose
HPProviderInit	Called when the provider's DLL is loaded. See **HPProviderInit**.
ServiceEntry	Entry point called by MAPI from the system's Control Panel application. See **MSGSERVICEENTRY**.
WizardEntry	Entry point for MAPI configuration wizard. See **WIZARDENTRY**.

As with all DLLs, the entry points must be called by these names but can map to functions with arbitrary names in your DLL. The HPProviderInit and WizardEntry entry points are required, while the ServiceEntry entry point is optional.

About Implementing HPProviderInit for Hook Providers

The **HPProviderInit** entry point should create a message hook provider object, initialize it, and return a pointer to it in the *lppSpoolerHook* pointer parameter. Initializing the spooler hook object is usually accomplished by:

- Storing the MAPI session pointer, instance handle, and memory allocation routine pointers that are passed in by MAPI.
- Getting the hook provider's configuration information out of its profile section.
- Verifying that its configuration information is complete and correctly formatted.
- Connecting to any other MAPI service providers needed – such as store providers – by getting pointers to them from MAPI through the session pointer.
- Initializing any rules or filters that should be started when the hook provider is loaded.

For more details, see ISpoolerHook:IUnknown.

About Implementing ServiceEntry for Hook Providers

The ServiceEntry entry point is called by MAPI when the user configures the provider by means of the system's control panel applet. The ServiceEntry entry point is called after MAPI has gotten new configuration information from the user. The ServiceEntry entry point needs to incorporate that information into the hook provider's profile section.

The ServiceEntry entry point is passed a window handle it can use as the parent window for any user interface it needs to display. Usually the ServiceEntry entry point only needs to display a user interface to confirm or correct any bad configuration information entered by the user. In addition, the ServiceEntry entry point is passed pointers to memory allocation routines and a MAPI support object that can be used to access the hook provider's profile section.

For more details, see MSGSERVICEENTRY, and Using Support Objects for Configuration.

About Implementing WizardEntry for Hook Providers

The WizardEntry entry point is called by the MAPI profile wizard. The WizardEntry entry point is responsible for giving the configuration wizard a pointer to the function to be called to handle window events, and a pointer to the full resource name for the wizard dialog box implemented by the hook provider. The WizardEntry entry point must also support programmatic configuration, that is, it should be possible to call this entry point to configure the hook provider without causing any user interface to be displayed.

Note One of the window events you should expect is WIZ_QUERYNUMPAGES. Your event handling procedure should return the number of pages used to configure your hook provider.

For more information, see Supporting the Profile Wizard.

About Hook Provider Interfaces

Hook providers implement the **ISpoolerHook:IUnknown** interface. Aside from the **QueryInterface**, **AddRef**, and **Release** methods inherited from the **IUnknown** interface, **ISpoolerHook** has only two additional methods to implement: **InboundMsgHook** and **OutboundMsgHook**.

The MAPI spooler calls **InboundMsgHook** after a transport provider receives an inbound message, but before the message is delivered to the default receive folder for the message's message class. The MAPI spooler calls **OutboundMsgHook** after transport providers have delivered the message to any underlying messaging systems.

Implementing the InboundMsgHook and OutboundMsgHook Methods for Hook Providers

You have considerable latitude in how you implement the specifics of the **InboundMsgHook** and **OutboundMsgHook** methods. In general, these methods compare one or more properties of the input message against the rules the hook provider implements. Based on these comparisons, the message can be modified, moved to a different message store, moved to a different folder by returning a different folder entry identifier, or marked for deletion by the MAPI spooler by returning a null folder entry identifier.

In addition, these methods return one or both of the flags HOOK_CANCEL and HOOK_DELETE to indicate how the MAPI spooler should react to the **InboundMsgHook** method's actions. The HOOK_CANCEL flag indicates that no other hook providers should be called for the message. The HOOK_DELETE flag indicates that the MAPI spooler should delete the message; the hook provider should never use calls to the message store provider to delete the message directly but should set the HOOK_DELETE flag and allow the MAPI spooler to delete the message.

Hook providers can create additional messages based on the input message if desired. However, if the hook provider intends for an additional message to be submitted for sending, the message must be created using the default message store provider object provided by the MAPI spooler.

The **InboundMsgHook** and **OutboundMsgHook** methods are nearly identical as far as interaction with message objects and message stores. The only architectural difference is that the MAPI spooler calls them at different times. Many hook provider implementations can share substantial portions of code between these methods because they are so similar in function.

See **InboundMsgHook** and **OutboundMsgHook** for details on how to treat the flags and folder entry identifier parameters.

Interactions Between Hook Providers and the MAPI Spooler

The interactions between hook providers and the MAPI spooler are fairly simple:

1. The MAPI spooler receives an inbound or outbound message from a message store or transport provider.
2. The MAPI spooler calls the hook provider's **InboundMsgHook** or **OutboundMsgHook** method, whichever is appropriate.
3. The hook provider processes the message, optionally returning a different destination folder for the message or returning values that cause the MAPI spooler to delete the message.
4. The MAPI spooler calls any other hook providers − assuming that HOOK_CANCEL was not returned in step 3 − then delivers the message to its final destination or deletes it as appropriate.

About Preprocessors

Preprocessors are generally implemented along with a transport provider, usually in the same DLL.

Registering a Preprocessor

All preprocessors must be registered by transport providers. The transport provider for a preprocessor registers it along with its other logon processing at the time that the MAPI spooler logs on to the transport provider. To register a preprocessor, the transport provider calls its support object's **IMAPISupport::RegisterPreprocessor** function. It passes in the name of the DLL in which the preprocessor is implemented and the names of the functions in that DLL that implement the preprocessor.

The transport provider can also control what specific addresses or address types the preprocessor applies to. The preprocessor can apply to specific addresses by giving a specific **MAPIUID** structure to the **RegisterPreprocessor** method. The preprocessor can apply to specific address types by giving the string representation of the address type, for example SMTP. A preprocessor can apply to all address types by giving a null or empty string as the address type. The **RegisterPreprocessor** method must be called once for each **MAPIUID** or address type.

For more information, see **IMAPISupport::RegisterPreprocessor**.

Creating a Minimal Transport Provider

Since every preprocessor must be registered by a transport provider, you may need to create a minimal transport provider whose sole task is to register your preprocessor. A minimal transport provider does not advertise any address types to the MAPI spooler, so that the MAPI spooler will never call the transport provider to send or receive messages.

To create a minimal transport provider

1. Create a transport provider with the minimum required interfaces for transport providers. Many of the methods and interfaces used by transport providers can be implemented as stubs that do nothing or return MAPI_E_NO_SUPPORT. The minimal transport provider should initialize as recommended in the topic Required Functionality for Transport Providers. In response to the **IXPLogon::AddressTypes** call, the transport provider should return a single zero-length string in *lpppszAdrTypeArray* and NULL in *lpppUIDArray*.
2. Create a MAPISVC.INF fragment for the transport provider and the message service that contains it. See Implementing a Message Service.
3. Optionally, but strongly recommended, create a message service entry point for configuration. See About Message Service Entry Point Functions.
4. Optionally, create an online Help file linked to your configuration property pages, wizard pages, or both, providing full details about all configuration options.
5. Optionally, create a header file for custom programmatic configuration by MAPI clients.
6. Optionally, create a WizardEntry entry point for interactive configuration by users. See Supporting the Profile Wizard.
7. Optionally, create a .PRF file detailing configuration properties for the message service.

For more details about transport providers, see Developing a Transport Provider and the quick start topic Creating and Configuring a Profile.

Implementing the PreprocessMessage Function for Preprocessors

The MAPI spooler calls the **PreprocessMessage** function for messages that meet the preprocessor's criteria for **MAPIUID** or address type. **PreprocessMessage** can do several things with the input message:

- Modify the input message. To modify the input message, the **PreprocessMessage** function calls the **IMAPIProp::OpenProperty** method with the MAPI_MODIFY flag for one or more properties of the input message. It then modifies the values of those properties, and calls **IMAPIProp::SaveChanges** to make the changes permanent.
- Generate new messages to be sent as well. To generate new messages, the **PreprocessMessage** function must use its MAPI session pointer to access a message store provider through which it can create new message objects. New messages should be created using the default message store. These new message objects should be passed back to the MAPI spooler in the *lpppMessage* pointer, which is a pointer to an array of message object pointers. The original input message, however, should never be placed in this array.
- Prevent the input message from being sent. To prevent the input message from being sent, the **PreprocessMessage** function must remove all the recipients from the message's recipient list and set the PR_DELETE_AFTER_SUBMIT property on the message. This will prevent the MAPI spooler from submitting the message to any transport providers.
- Modify the recipient list of a message.

If multiple message preprocessors are being used, the individual **PreprocessMessage** functions are called in the same order as the transports. This order can be modified programmatically if your preprocessor needs to be called at some specific point in the transport order. See IMAPISession::AdminServices, IMsgServiceAdmin::MsgServiceTransportOrder and IXPLogon::AddressTypes for details.

Implementing the RemovePreprocessInfo Function for Preprocessors

After messages are sent by any transport providers, the MAPI spooler calls each preprocessor's **RemovePreprocessInfo** function. This gives the preprocessor a chance to undo any changes it made to the input message before the message is passed to any hook providers. For example, a preprocessor for a FAX-based transport can generate a bitmap representation of the input message for sending to receiving fax machines; there will be little reason for that bitmap to be archived by any hook providers, so the **RemovePreprocessInfo** function can take the bitmap out of the input message before any hook providers process the message.

If multiple message preprocessors are being used, the individual **RemovePreprocessInfo** functions are called in the opposite order as the **PreprocessMessage** functions. This is so that **RemovePreprocessInfo** implementations that depend on positions within the message can locate the information they should remove without interference by the actions of other preprocessors. For example, a preprocessor that adds an automatic closing or signature to a message's PR_BODY or PR_RTF_COMPRESSED properties will probably use a position or offset from the end of the property to locate the information it should remove. If another preprocessor modifies these properties later, then that preprocessor's **RemovePreprocessInfo** function should undo its actions before the first preprocessor undoes its actions.

Note There is no requirement that the **RemovePreprocessInfo** function actually undoes the effects of the associated **PreprocessMessage** function. However, if the **RemovePreprocessorInfo** does not undo the effects of the **PreprocessMessage**, then the **PreprocessMessage** function should take care to make its changes in such a way as to minimize interference with **RemovePreprocessInfo** implementations that do, or make an effort to inform the user that the transport provider which registers the preprocessor should be installed into the user's profile ahead of other transports.

Testing and Debugging

Testing strategies differ depending on whether you are developing a client or service provider. Because a client application requires one or more service providers to operate, clients should be tested in an environment with different sets of service providers.

Service providers, however, should be testing in isolation before integrating it with other providers. MAPI provides sample applications that are meant to test the features of a service provider of a particular type. ABVIEW.EXE, for example, exercises the features of an address book provider whereas MDBVIEW.EXE works with a message store provider. To test any service provider in isolation, use one of these sample applications and limit the entries in the profile the provider to be tested. Include any other service providers that are absolutely necessary. For example, to test a transport provider, your profile would contain that provider, the PAB, and a message store.

All clients and service providers can benefit from using a set of debugging macros provided by MAPI in the MAPIDBG utility. This utility is controlled by the entries in its initialization file, MAPIDBG.INI. MAPIDBG.INI is constructed in a similar way to other initialization files and the profile. It is divided into sections with each section containing several entries that affect the utility's behavior.

About MAPIDBG.INI

MAPIDBG.INI is a file that is used by MAPI's debugging utility to control various features of the MAPI subsystem. The file resides in the Windows directory and is organized in sections, similar to other Windows initialization files and MAPISVC.INF. Sections are labeled in the following format:

```
[SectionName] Section
```

There are several different sections as described in the following table.

Section name	Description
[General]	Controls debugging features common to all MAPI components.
[LocalHeapFailures]	Generates artificial allocation failures.
[Memory Management]	Controls debugging features inthe MAPI memory allocator.
[MAPIX]	Controls debugging features in the MAPI subsystem.
[Remoting]	Controls debugging features in the interface marshalling and remoting processes.
[Simple MAPI]	Controls debugging features in Simple MAPI.
[Spooler]	Controls debugging features in the MAPI spooler.
[TNEF]	Controls debugging features in the MAPI TNEF facility.

Every section has one or more entries, formatted as follows:

```
EntryName = <value>
```

Most entries have no effect on retail versions of MAPI; they add features only in the debug versions. However, there are a few entries that affect both versions; the detailed descriptions of each section explicitly point out these exceptions.

The [General] Section in MAPIDBG.INI

The entries in this section control debugging features common to all MAPI components.

Entry	Description
AssertBadBlocks = <**0**/**1**>	If this entry is set to 1, the **MAPIAllocateMore** and **MAPIFreeBuffer** functions assert when passed an invalid memory address. If it is set to 0, they fail without asserting. This is useful for detecting bugs such as freeing uninitialized pointers and freeing blocks more than once. The default value is 1 − asserts are generated.
AssertLeaks = <**0**/**1**>	If this entry is set to 1, MAPI asserts if any service provider DLL, **IMAPISession** interface, or **IProfAdmin** interface is not released before MAPI is shut down. The default value is 0 − asserts are not generated.
DebugTrace = <**0**/**1**>	This entry controls trace output from several of the macros defined in MAPIDBG.H, including **DebugTrace** and the **TraceSz** and **Assert** group of macros. If this entry is set to 1, traces are written to the debug port using the Win32 function **OutputDebugString**, and can be captured using a debugger or a monitoring utility such as DBWIN.EXE (for Win16) or DEB.EXE (for Win32). If this entry is set to 0, no traces are written. The default value is 1 − traces are enabled.
EventLog = <**0**/**1**>	On Windows NT only , if this entry is set to 1, all debug traces are written to the application event log instead of to the debug port. This is useful when debugging an NT service that uses MAPI. The default value is 0 − traces are written to the debug port, not the event log.
MemoryFillRandom = <**0**/**1**>	This entry is obsolete. Use the **FillMemory** entry in the [Memory Management] section.
RetainDLLs = <**0**/**1**>	If this entry is set to 1, MAPI does not call the Win32 function

	FreeLibrary to release provider DLLs although it does remove them from its internal data structures. This can occasionally be useful when pursuing bugs in provider startup and shutdown behavior. The default value is 0 − provider DLLs are freed.
TrapOnSameThread = <**0**/**1**>	On Win32, if this entry is set to 1, the message dialog boxes generated by the **Trap** and **Assert** group of macros are created on the calling thread. If it is set to 0, the message box is generated on a separate thread to prevent reentering the message loop on the calling thread. The default value is 0 − message dialog boxes are generated on a separate thread.
VerboseTNEF = <**0**/**1**>	If this entry is set to 1, the MAPI TNEF facility generates debug trace output for routine operations. If it is set to 0, TNEF generates traces only when errors occur. The default value is 1 − TNEF traces are verbose.
VirtualMemory = <**0**/**1**/**4**>	This entry is obsolete. Use the **VirtualMemory** entry in the [Memory Management] section.

The [Local Heap Failures] Section in MAPIDBG.INI

The entries in this section control features in the MAPI memory allocator that generate artificial allocation failures to test how other components handle out of memory conditions.

Entry	Description
AllocsToFirstFailure = <*number of allocations*>	This entry forces a memory allocation failure on the indicated attempt. Use it to test how well your client or provider handles memory allocation failures. The count includes blocks allocated by MAPI and by other components, not just by your own component.
FailureInterval = <*number of allocations*>	This entry forces repeated memory allocation failures at the specified interval after the count indicated by the **AllocsToFirstFailure** entry is reached. The count includes blocks allocated by MAPI and by other components, not just by your own component.
FailureSize = <*number of bytes*>	If this entry is nonzero, this entry forces every memory allocation request for the specified number of bytes or more to fail. This includes blocks allocated by MAPI and by other components, not just by your own component.

The [Memory Management] Section in MAPIDBG.INI

The entries in this section control debugging features in the MAPI memory allocator.

Entry	Description
VirtualMemory = <**0**/**1**/**4**>	If this entry is nonzero, memory allocations made through MAPI are surrounded by unaddressable memory on either side. This causes code that accesses memory outside the allocated block to fail immediately instead of corrupting memory and can be very helpful in isolating memory corruption bugs. If this entry is set to 4, the returned address is guaranteed to be aligned on a 4-byte boundary; if it is 1, the address of the returned memory block is unaligned. The default value is 0 − a normal heap is allocated.
AssertLeaks = <**0**/**1**>	If this entry is set to 1, MAPI asserts if any memory allocated using the MAPI allocators has not been freed at the time MAPI is shut down. This is usually at the last **MAPIUninitialize** call; if the application fails to uninitialize MAPI, it occurs when the MAPI DLL is unloaded. The default value is 1 − generates asserts.
DumpLeaks = <**0**/**1**>	If this entry is set to 1, MAPI outputs debug trace information regarding memory allocated using the MAPI allocators that were not freed by the time MAPI was shut down. The information includes a stack traceback (for Win32 only), size and location of the memory block, order in which the block was allocated, and the block's name (for internal MAPI allocations only). The default value is 1 − writes traces.
FillByte = <*0xNN*>	This entry specifies the hexadecimal value to use for filling memory in the **FillMemory** entry. The default value is 0xFE.
FillMemory = <**0**/**1**>	If this entry is set to 1, all memory blocks allocated by

	MAPI are filled with a fill byte after they're allocated and after they're freed. The default value is 1 – fills blocks.
SharedMemMaxSize = <*number of bytes*>	This entry limits the size of the MAPI shared memory area. It can be used to force allocation failures. The default value is 0 – the area is as big as necessary.

Anyone using the Microsoft RPC libraries who needs to use virtual allocation flags should set the **VirtualMemory** entry to 4 in both the [General] and [Memory Management] sections of MAPIDBG.INI. This includes anyone using EMSMDB.DLL or EMSABP.DLL. Microsoft RPC requires memory allocators to align memory at least as well as the operating system's allocator. The MAPI **Virtual Memory** allocator aligns on 4-byte boundaries when set to 4. Setting **VirtualMemory** to 1 does not do this in order to catch even single-byte overwrites immediately.

Enabling virtual allocation increases the MAPI subsystem's demand for system resources. On Win32, MAPI uses 64K of virtual address space for each allocation. On Win16, MAPI uses the **GlobalAlloc** function for each allocation, so it is possible to run out of selectors. Depending on the overall load your test scenario places on the system, you may need to narrow down the scenario enough to reproduce the problem without running out of system resources.

The [MAPIX] Section in MAPIDBG.INI

The entries in this section control debugging features in the MAPI core and profile provider.

Entry	Description
CheckNotifKeysOften = <**0/1**>	If this entry is set to 1, the MAPI notification support methods validate their lists of entries and registrations before and after making any changes to them. The default value is 0 − no checks are performed.
CheckNotifTasksOften = <**0/1**>	If this entry is set to 1, the MAPI notification support methods validate their list of active tasks before and after making any changes to it. The default value is 0 − no checks are performed.
DebugSpooler = <*path*>	If this entry is present, MAPI attempts to execute its value when it is time to launch the MAPI spooler instead of simply launching the executable. It can be used to launch a debugger, for instance. The default value is MAPISP32.EXE for Win32 or MAPISP.EXE for Win16.
DelaySpooler = <*seconds*>	This entry controls how long the MAPI spooler waits. The MAPI spooler process waits for a period of time immediately after being launched. This entry speeds up client startup by reducing contention between the client and MAPI spooler for CPU time and other resources. The default value is 15 seconds.
FlushRegistry = <**0/1**>	If this entry is set to 1, the MAPI profile provider flushes the registry whenever a writeable profile section is released and at certain other times. If it is 0, MAPI never flushes the registry. Unlike most entries, this works for retail MAPI as well as the debug builds. The default value is 0 − the registry is not flushed.
SkipSystem = <**0/1**>	If this entry is set to 0, MAPI tries the system directory first when loading the MAPI spooler or any service provider. If it is 1, MAPI loads those components in typical path order. Since most

	such components are installed in the system directory, it is faster to check there first. Unlike most entries, this works for retail MAPI as well as the debug builds. The default value is 0 – try the system directory first.
SpoolerAutoStartTimeout = <*seconds*>	This entry controls the length of time the client will wait for the MAPI spooler to start when it has been started automatically as the result of a client logon. The default value is 60 seconds.

The [Simple MAPI] Section in MAPIDBG.INI

The entries in this section control debugging features in Simple MAPI, including the long message identifier cache and the thunking layer.

Entry	Description
DumpCacheContents = <**0**/**1**>	If this entry is set to 1, the entire contents of the message identifier cache are dumped to debug trace each time the cache is accessed or modified. If it is 0, the cache is not dumped. The default value is 0 − the cache is not dumped.
MessageIDCacheGrow = <*number of entries*>	This entry controls the number of entries by which the cache grows when it is in runaway mode. When it is in circular buffer mode, its size does not change. The default value is 50 entries. This entry is found in EXCHNG.INI (for Win16) or EXCHNG32.INI (for Win32).
MessageIDCacheSize = <*number of entries*>	This entry controls the initial size of the cache and the size to which it is reset when returning to circular buffer mode. The default value is 10 entries. This entry is found in EXCHNG.INI (for Win16) or EXCHNG32.INI (for Win32).
TraceEntry = <**0**/**1**>	If this entry is set to 1, the arguments to each Simple MAPI call are dumped to debug trace. The default value is 0 − calls are not traced.
TraceMessageIDCache = <**0**/**1**>	If this entry is set to 1, normal operations in the message identifier cache generate debug trace output. If it is 0, only errors generate trace output. The default value is 0 − no traces are generated on normal operations.
TracePacket = <**0**/**1**>	If this entry is set to 1, debug traces are generated for errors encountered on the send side of the Simple MAPI thunk layer. The default value is 0 − no traces are generated for thunk errors.
TraceServer = <**0**/**1**>	If this entry is set to 1, debug traces are generated for errors and certain operations on the receive side of the Simple MAPI thunk layer. The default value is 0 − no traces are generated for thunk server errors.
TraceVerbose = <**0**/**1**>	If this entry is set to 1, debug traces

are generated for routine operations in the Simple MAPI thunk layer. The default value is 0 – no traces are generated for routine thunk operations.

The [Spooler] Section in MAPIDBG.INI

The entries in this section control debugging features in the MAPI spooler.

Entry	Description
AlwaysRebuild = <**0/1**>	If this entry is set to 1, the MAPI spooler rebuilds its merged outgoing queue on every outgoing queue notification. If it is 0, the queue is only rebuilt when opening a message store for the first time. The default value is 0 − rebuild the outgoing queue rarely.
AssertSessionLeaks = <**0/1**>	If this entry is set to 1, the MAPI spooler asserts if it finds a reference count of greater than 1 when freeing memory for one of its **IMAPISession** objects. The default value is 0 − no assert for leaked spooler-side sessions.
LowPrioritySpooling = <**0/1**>	If this entry is set to 1, the MAPI spooler gives its main thread below-normal base priority by calling the Win32 function **SetThreadPriority** and setting the THREAD_PRIORITY_BELOW_NORMAL value. If it is 0, the MAPI spooler runs at normal priority. The MAPI spooler continues to adjust its priority when flushing and performing other foreground operations regardless of this entry's value. The default value is 1 − the MAPI spooler runs at below-normal priority.
NonPersistanceTimeout = <*seconds*>	This entry controls the length of time the MAPI spooler waits before exiting. The MAPI spooler exits after the last client session has logged off, but not immediately. The default value is 20 seconds.
SafeMode = <**0/1**>	If this entry is set to 1, then during its startup the MAPI spooler aborts submission on all messages in the outgoing queue and restarts the submission process from scratch. Submission is not aborted during subsequent rebuilds of

	the outgoing queue. If it is 0, the submission state is left intact. The default value is 0 − do not abort submission when the outgoing queue changes.
TraceHeartbeat = <**0**/**1**>	If this entry or **TraceEverything** is set to 1, the MAPI spooler generates debug trace output during its normal idle time processing. The default value is 0 − no traces are generated for normal idle processing.
TraceHooks = <**0**/**1**>	If this entry or **TraceEverything** is set to 1, the MAPI spooler generates debug trace output for normal operations involving MAPI spooler hook providers. The default value is 0 − no trace output is generated for normal hook operation.
TraceUploads = <**0**/**1**>	If this entry or **TraceEverything** is set to 1, the MAPI spooler generates debug trace output for normal message transmission steps through message transports. The default value is 0 − no trace output is generated for normal message transmission.
TraceDownloads = <**0**/**1**>	If this entry or **TraceEverything** is set to 1, the MAPI spooler generates debug trace output for normal message download steps from transport providers. The default value is 0 − no trace output is generated for normal downloading of messages.
TraceOutgoingQueues = <**0**/**1**>	If this entry or **TraceEverything** is set to 1, the MAPI spooler generates debug trace output for normal notifications and other operations on the merged outgoing queue. The MAPI spooler maintains a merged outgoing queue that combines messages from outgoing queues of all open message stores that support message submission. The default value is 0 − no trace output is generated for normal outgoing queue processing.
TracePreprocessors = <**0**/**1**>	If this entry or **TraceEverything**

	is set to 1, the MAPI spooler generates debug trace output when calling preprocessors. The default value is 0 − no trace output is generated for normal preprocessing operations.
TraceService = <**0**/**1**>	If this entry or **TraceEverything** is set to 1, the MAPI spooler generates debug trace output when queuing, dequeueing, and processing service requests on behalf of MAPI and transport providers. The default value is 0 − no trace output is generated for normal service queue processing.
TraceVerbose = <**0**/**1**>	If this entry or **TraceEverything** is set to 1, the MAPI spooler generates debug trace output in all categories for events that would otherwise be considered too frequent or insignificant to warrant trace output. The default value is 0 − the MAPI spooler does not generate trace output..
TraceYields = <**0**/**1**>	If this entry or **TraceEverything** is set to 1, the MAPI spooler generates debug trace output when explicitly yielding the CPU to other processes, either on its own behalf or on behalf of a transport provider. The default value is 0 − no trace output is generated when yielding.
TraceEverything = <**0**/**1**>	If this entry is set to 1, all the trace output requested by **TraceHeartbeat**, **TraceHooks**, **TraceUploads**, **TraceDownloads**, **TraceOutgoingQueues**, **TracePreprocessors**, **TraceService**, **TraceVerbose**, and **TraceYields** is generated, and the values of those entries are ignored. You cannot selectively turn off trace categories when **TraceEverything** is 1. The default value is 0 − entries for the individual categories control trace output.

The [TNEF] Section in MAPIDBG.INI

The entry in this section controls debugging features in the MAPI TNEF facility.

Entry	Description
ForceUncompressedRTF = <**0/1**>	If this entry is set to 1, TNEF assumes that all RTF text is uncompressed and checks the stream explicitly. If it is 0, TNEF assumes the text is compressed. The default value is 0.

Tracing Memory Leaks

When leak tracing is enabled by setting the **DumpLeaks** entry in the [Memory Management] section to 1, MAPI produces debug trace output for each leaked block of memory. For each leaked block, MAPI lists:

- The name of the block and the heap it was allocated in, if available.
- The address of the block.
- The order in which the block was allocated, starting with 1 for the first block.
- The size of the block.
- The stack trace of the routine that allocated it (up to 20 frames, Win32 only).

One technique for turning the stack traceback into a usable symbolic trace is to save the debug output to a text file, get your application back into the WINDBG debugger in a steady state, and convert the hexadecimal numbers to symbols using the 'list near' command. This command prints the nearest symbols before and after a given address. It is helpful to use a macro in your preferred editor to convert the MAPI trace, which looks like this:

```
Memory leak 'Proxy/Stub Object' in MAPIX Internal Heap @ 004E0770,
    Allocation #18, Size: 48
[0] 6C4C3B2F
[1] 6C4C401D
[2] 77CF7AB7
[3] 77D30E30
[4] 77CC9076
```

into a string of "list near" commands that looks like this (it gives you the closest symbol before and after the address):

```
> ln 6C4C3B2F; ln  6C4C401D; ln 77CF7AB7; ln 77D30E30; ln 77CC9076
```

When run in the WINDBG command window, this string produces a list of symbols like the following (there are two symbols for each address, the first is almost always the one you want):

```
MAPI32!operator new(unsigned int)+0x2f
MAPI32!operator delete(void *)-0x31
MAPI32!StdPSFactory::CreateProxy(IUnknown *, const _GUID &,
    IRpcProxyBuffer * *, void * *)+0xbd
MAPI32!StdPSFactory::CreateStub(const _GUID &, IUnknown *, IRpcStubBuffer *
*)-0xe3
OLE32!?
CreateInterfaceProxy@CRemoteHdlr@@AAEPAVCPSIX@@ABU_GUID@@PAPAXPAJ@Z+0x7d
OLE32!?CreateInterfaceStub@CRemoteHdlr@@AAEPAVCPSIX@@ABU_GUID@@PAJ@Z-0xc1
OLE32!_IEnumUnknown_RemoteNext_Proxy@16+0xf
OLE32!_IEnumUnknown_RemoteNext_Thunk@4-0x6
OLE32!?AddRef@CRemoteHdlr@@UAGKXZ+0
OLE32!?GetRH@CStdIdentity@@AAEPAUIRemoteHdlr@@XZ-0x11
```

MAPI does not expose an API to trigger an allocation dump.

It is most efficient to address memory leaks as they occur instead of waiting until there is a large number.

MAPI Interfaces

The documentation for each interface consists of an introductory section that includes a brief description of the interface's purpose followed by an "At a Glance" table, which contains the following information:

Specified in header file:	The header file where the interface is defined and that must be included when you compile your source code.
Object that supplies this interface:	The object implementing the interface.
Corresponding pointer type:	The pointer type for the object implementing the interface.
Implemented by:	A list of the components that must provide an implementation of the interface.
Transaction model:	If non-transacted, changes take effect immediately; if transacted, changes do not take effect until **SaveChanges** is called.
Called by:	A list of the components that typically call the methods of the interface.

Following the "At a Glance" table is another table that lists all the methods of this interface in vtable order. A *vtable* is an array of function pointers created by the compiler containing one function pointer for each method of a MAPI object. The methods are listed in the same order that they are declared. Methods inherited from other interfaces are not shown in the "Vtable Order" table but can be used in the same way as documented in the interface that defines them.

Following the "Vtable Order" table, the interface's methods are then covered in alphabetical order. For each method, the documentation includes a brief purpose statement for the method and its syntax followed by this information:

Heading	**Content**
Parameters	A description of each parameter in the method.
Return Values	A description of the unique values that the method can return. These are the values that callers should check for in their code.
Remarks	A description of why and how the method is used.
See Also	Cross-references to other topics in the *MAPI Programmer's Reference.*

IABContainer : IMAPIContainer

The **IABContainer** interface provides access to address book containers. MAPI and client applications call the methods of the **IABContainer** interface to perform name resolution and to create, copy, and delete recipients. Although address book providers must supply an implementation for all methods of this interface, this implementation can return MAPI_E_NO_SUPPORT to indicate a lack of support for a particular operation. For example, an address book container that does not perform name resolution returns MAPI_E_NO_SUPPORT from its **IABContainer::ResolveNames**.

At a Glance

Specified in header file:	MAPIDEFS.H
Object that supplies this interface:	Address book container
Corresponding pointer type:	LPABCONT
Implemented by:	Address book providers
Transaction model:	Non-transacted
Called by:	MAPI and client applications

Vtable Order

CreateEntry	Creates a new recipient in the address book container; the container must support modification.
CopyEntries	Copies one or more recipients into the address book container; the container must support modification.
DeleteEntries	Removes one or more entries from the address book container; the container must support modification.
ResolveNames	Matches recipients in a list with entries in the address book container.

Required Properties

PR_CONTAINER_FLAGS	Read/write
PR_DISPLAY_NAME	Read/write
PR_ENTRYID	Read-only
PR_OBJECT_TYPE	Read-only
PR_RECORD_KEY	Read-only

IABContainer::CopyEntries

The **IABContainer::CopyEntries** method copies one or more recipients, typically messaging users or distribution lists, into the address book container.

HRESULT CopyEntries(
  LPENTRYLIST *lpEntries***,**
  ULONG *ulUIParam***,**
  LPMAPIPROGRESS *lpProgress***,**
  ULONG *ulFlags*
)

Parameters

lpEntries
  Input parameter pointing to an array of **ENTRYLIST** structures containing the entry identifiers of the entries to copy.

ulUIParam
  Input parameter containing the handle of the parent window for any dialog boxes or windows this method displays. The *ulUIParam* parameter must be zero if the AB_NO_DIALOG flag is set in the *ulFlags* parameter.

lpProgress
  Input parameter pointing to a progress object or NULL. If *lpProgress* is NULL, progress should be displayed using the progress object supplied by MAPI through the **IMAPISupport::DoProgressDialog** method. The *lpProgress* parameter is ignored if AB_NO_DIALOG is set in *ulFlags*.

ulFlags
  Input parameter containing a bitmask of flags that controls how the copy operation is performed. If no flags are set, the entries are copied. The following flags can be set:

  AB_NO_DIALOG
    Suppresses display of progress information. If this flag is not set, progress information is displayed.

  CREATE_CHECK_DUP_LOOSE
    Indicates a loose level should be used for duplicate entry checking, which returns more matches than setting a strict level with the flag CREATE_CHECK_DUP_STRICT. For example, a provider can define a loose match as any two entries having the same display name, while defining a strict match as any two entries having the same display name and messaging address.

  CREATE_CHECK_DUP_STRICT
    Indicates a strict level should be used for duplicate entry checking, which returns fewer matches than setting a loose level with the flag CREATE_CHECK_DUP_LOOSE.

  CREATE_REPLACE
    Indicates that if the entry to be created is a duplicate of one already in this container, the new entry should replace the existing one.

Return Values

S_OK
  The copy operation succeeded.

MAPI_W_PARTIAL_COMPLETION
  The copy operation succeeded, but one or more of the recipients could not be copied. To test for this warning, use the **HR_FAILED** macro. When this warning is returned, the call should be handled as successful.

Remarks

Address book containers that support the **IABContainer::CopyEntries** method must also support modification. Modifiable containers set the AB_MODIFIABLE flag in their PR_CONTAINER_FLAGS property.

The **IABContainer::CopyEntries** method copies recipients into this container. A call to **IABContainer::CopyEntries** is functionally equivalent to making the following three calls for each recipient to be copied:

1. **IABContainer::CreateEntry** to create the new recipient.
2. The new messaging user or distribution list's **IMAPIProp::SaveChanges** method to perform a save.
3. The new messaging user or distribution list's **IUnknown::Release** method to release the container's reference.

Address book providers must support all the *ulFlags* flags; however, a provider is free to determine what the semantics of CREATE_CHECK_DUP_LOOSE and CREATE_CHECK_DUP_STRICT mean within its implementation's context. Providers that cannot determine whether an entry is a duplicate should allow the entry to be copied.

If either the CREATE_CHECK_DUP_LOOSE or CREATE_CHECK_DUP_STRICT flag is set, and the implementation does not copy the entry because it is a duplicate, the MAPI_W_PARTIAL_COMPLETION warning is not returned. MAPI_W_PARTIAL_COMPLETION is only returned when a nonduplicate entry cannot be copied.

If either the CREATE_CHECK_DUP_LOOSE or the CREATE_CHECK_DUP_STRICT flag, along with the CREATE_REPLACE flag, is not set, then the entry is copied even if it is a duplicate.

The CREATE_REPLACE flag checks for CREATE_CHECK_DUP_LOOSE or CREATE_CHECK_DUP_STRICT. The personal address book does not implement CREATE_REPLACE. Providers are not required to support the flag, which means they can ignore it. If CREATE_REPLACE is set without other flags it has no meaning. To consider another combination, if CREATE_CHECK_DUP_STRICT is used without CREATE_REPLACE, the check for duplicates is carried out and if there is a duplicate there is no replacement. This is what happens when a user makes an addition to the personal address book through the standard user interface.

No support for the flags means the provider can ignore the flags, that is, simply not check for duplicates. The flags are intended as a suggestion from the client to avoid unnecessary duplications. If the client requires addition of an entry, it will pass no flags to check for duplicates, and the entry will be added.

Personal address book rules for duplicate checking define "loose" to mean that original display names are equal; and "strict" to mean that original display names and search keys are equal. In the personal address book, the original display name is defined as the transmittable display name, if available. Otherwise, it is simply the display name. Combined with the PR_SEARCH_KEY property, these criteria usually eliminate duplicates in most applications whether automated or user driven.

For more information on using the **HR_FAILED** macro, see Using Macros for Error Handling.

See Also

ENTRYLIST structure, **IABContainer::CreateEntry** method, **IMAPIProgress : IUnknown** interface, **IMAPIProp::SaveChanges** method

IABContainer::CreateEntry

The **IABContainer::CreateEntry** method creates a new entry in the address book container. The new entry can be a messaging user, a distribution list, or another container.

HRESULT CreateEntry(
  ULONG *cbEntryID***,**
  LPENTRYID *lpEntryID***,**
  ULONG *ulCreateFlags***,**
  LPMAPIPROP FAR * *lppMAPIPropEntry*
)

Parameters

cbEntryID
  Input parameter containing the size, in bytes, of the entry identifier pointed to by the *lpEntryID* parameter.

lpEntryID
  Input parameter pointing to the entry identifier of a template for creating new entries of a particular type.

ulCreateFlags
  Input parameter containing a bitmask of flags that controls how entry creation is performed. The following flags can be set:

  CREATE_CHECK_DUP_LOOSE
    Indicates a loose level should be used for duplicate entry checking, which returns more matches than setting a strict level with the flag CREATE_CHECK_DUP_STRICT. For example, a provider can define a loose match as any two entries having the same display name, while defining a strict match as any two entries having the same display name and messaging address.

  CREATE_CHECK_DUP_STRICT
    Indicates a strict level should be used for duplicate entry checking, which returns fewer matches than setting a loose level with the flag CREATE_CHECK_DUP_LOOSE.

  CREATE_REPLACE
    Indicates that if the entry to be created is a duplicate of one already in this container, the new entry should replace the existing one.

lppMAPIPropEntry
  Output parameter pointing to a pointer to the newly created entry.

Return Values

S_OK
  The new entry was successfully created.

Remarks

The **IABContainer::CreateEntry** method creates a new entry in the container and returns a pointer to the newly created object. The entry identifier in the *lpEntryID* parameter represents a template, published in the container's one-off table.

Implementations of **IABContainer::CreateEntry** which allow creation of arbitrary entries from other providers' templates are necessarily more complicated than those that do not allow it. Such an must provide storage for some or of the properties associated with the entries. For example, **CreateEntry** can provide storage for the PR_DETAILS_TABLE property so that details can be made available without requiring the foreign provider to translate.

If **CreateEntries** supports the creation of entries with template identifiers, meaning that the entries support the PR_TEMPLATEID property, it must perform the following tasks:

1. Call **IMAPISupport::OpenTemplateID**. **IMAPISupport::OpenTemplateID** allows the foreign provider's code for the entry to bind to the new entry being created. Foreign providers support this binding process to maintain control over entries created from their templates into the containers of host providers.
2. Perform any necessary initialization and populate the new object with all of the properties from the entry in the foreign provider, the object returned in the *lppMAPIPropNew* parameter from **OpenTemplateID**.

If **IMAPISupport::OpenTemplateID** succeeds, the calling provider should initialize the new entry for offline use as it would for any other entry from a foreign provider, but it should copy the properties to the bound interface returned by **OpenTemplateID** rather than directly to the property object.

If **IMAPISupport::OpenTemplateID** returns an error, the calling provider should fail the **CreateEntry** call and not allow the entry to be created. Allowing an entry with a template identifier to be created without successfully binding the related code in the foreign provider to the new entry's data could invalidate assumptions made about the data content by the foreign provider's code.

Address book containers that support the **IABContainer::CreateEntry** method must be modifiable. Modifiable containers set the AB_MODIFIABLE flag in their PR_CONTAINER_FLAGS property.

Although MAPI's Personal Address Book does not support the CREATE_REPLACE flag, address book providers are encouraged to support all of the the *ulFlags* flags. The flags are intended as a suggestion from the client to avoid unnecessary duplications. Providers have the option to avoid checking for duplicates and choose to either always create a new entry. Providers that cannot determine whether or not an entry is a duplicate should allow the creation to proceed.

Providers are free to determine the semantics of the CREATE_CHECK_DUP_LOOSE and CREATE_CHECK_DUP_STRICT flags. Personal address book rules for duplicate checking define "loose" to mean that original display names are equal; and "strict" to mean original display names and search keys are equal. In MAPI's Personal Address Book, the original display name is defined as the transmittable display name, if available. Otherwise, it is simply the display name. Combined with the PR_SEARCH_KEY property, these criteria usually eliminate duplicates in most applications whether automated or user driven.

The CREATE_REPLACE flag checks for the setting of either the CREATE_CHECK_DUP_LOOSE or CREATE_CHECK_DUP_STRICT flag. When CREATE_REPLACE is set without one of the other flags being set, it has no meaning. For example, if CREATE_CHECK_DUP_STRICT is set by itself, the provider for duplicates is carried out and if there is a duplicate there is no replacement. When CREATE_REPLACE is either not supported or not set, a duplicate entry prohibits the creation from occurring.

When **CreateEntry** returns, an entry identifier for the new entry will not necessarily be accessible until after the new entry's **IMAPIProp::SaveChanges** method has been called. Whether or not the entry identifier is available depends on the provider's implementation.

Although duplicate checking flags are passed in **CreateEntry**, the duplicate checking operation does not occur until **SaveChanges** is called.

Error values such as MAPI_E_COLLISION, which can occur when an entry is created, are returned for the subsequent **SaveChanges** call, not for **CreateEntry**.

See Also

IABContainer::CopyEntries method, **IMAPIProp::OpenProperty** method, **IMAPIProp::SaveChanges** method, PR_CREATE_TEMPLATES property

IABContainer::DeleteEntries

The **IABContainer::DeleteEntries** method removes one or more entries from an address book or other container that enables the user to delete entries. The removed entries are typically messaging users, distribution lists, or containers.

HRESULT DeleteEntries(
  LPENTRYLIST *lpEntries***,**
  ULONG *ulFlags*
)

Parameters

lpEntries
  Input parameter pointing to an array of **ENTRYLIST** structures containing entry identifiers for the entries being deleted.

ulFlags
  Reserved; must be zero.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

MAPI_W_PARTIAL_COMPLETION
  The call succeeded, but one or more of the entries could not be deleted. To test for this warning, use the **HR_FAILED** macro. When this warning is returned, the call should be handled as successful. For more information on using the **HR_FAILED** macro, see Using Macros for Error Handling.

IABContainer::ResolveNames

The **IABContainer::ResolveNames** method resolves entries in the address book container.

HRESULT ResolveNames(
  LPSPropTagArray *lpPropTagArray***,**
  ULONG *ulFlags***,**
  LPADRLIST *lpAdrList***,**
  LPFlagList *lpFlagList*
)

Parameters

lpPropTagArray
  Input parameter pointing to an **SPropTagArray** structure containing an array of property tags indicating the properties required for each name being resolved. To request the default set of property columns for the container's contents table be returned in the address list, pass NULL in the *lpPropTagArray* parameter.

ulFlags
  Input parameter containing a bitmask of flags that controls the type of the text in returned strings. The following flag can be set:

  MAPI_UNICODE
    Indicates the returned strings of the default column set are in Unicode format. If the MAPI_UNICODE flag is not set, the strings are in ANSI format.

lpAdrList
  Input-output parameter that on input points to an **ADRLIST** structure containing the list of entries whose names need to be resolved. On output, the *lpAdrList* parameter returns the list of resolved names.

lpFlagList
  Input-output parameter containing an array of flags; each flag corresponds to an entry in the *lpAdrList* parameter and provides the name-resolution status for that particular entry. The flags in the *lpFlagList* parameter are in the same order as the entries in *lpAdrList*. The following flags can be set:

  MAPI_AMBIGUOUS
    Indicates that the corresponding entry is resolved, but that it did not resolve to a unique entry identifier. If this flag is returned, other containers should ignore this entry in further resolution.

  MAPI_RESOLVED
    Indicates that the corresponding entry has been resolved to a unique entry identifier. If this flag is returned, other containers should ignore this entry in further resolution.

  MAPI_UNRESOLVED
    Indicates that the corresponding entry is unresolved. Another container can attempt to resolve this entry.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

MAPI_E_BAD_CHARWIDTH
  Either the MAPI_UNICODE flag was set and the implementation does not support Unicode, or MAPI_UNICODE was not set and the implementation only supports Unicode.

MAPI_E_NO_SUPPORT
  The address book provider does not support bulk name resolution using this method.

Remarks

The **IABContainer::ResolveNames** method attempts to match unresolved names from the array of entries in the *lpAdrList* parameter to recipients in the address book container and assign entry identifiers. An unresolved name is a recipient that does not have an entry identifier associated with it and has its corresponding flag in the *lpFlagList* parameter set to MAPI_UNRESOLVED.

Address book providers are not required to support name resolution with the **IABContainer::ResolveNames** method. Instead, they can support it with the PR_ANR property restriction. Address book providers that choose to rely on the PR_ANR restriction for name resolution can return MAPI_E_SUPPORT from their **IABContainer::ResolveNames** implementation. For more information, see Implementing the PR_ANR Property Restriction.

An address book container's **ResolveNames** method is called as a result of a client calling **IAddrBook::ResolveName**. MAPI calls the **IABContainer::ResolveNames** method for each address book container in the address book. The container attempts to uniquely match each entry in the recipient list passed in the *lpAdrList* parameter. Entries that resolve to unique recipients in the container have their corresponding flag in the *lpFlagList* parameter set to MAPI_RESOLVED and the PR_ENTRYID property added to their corresponding **ADRENTRY** structure in the *lpAdrList* parameter. The entry identifier stored in the PR_ENTRYID property can be short-term or long-term.

Entries that do not match any of the container's recipients have their corresponding flag set to MAPI_UNRESOLVED and are passed onto the next container by the **IAddrBook::ResolveName** method. Entries that match multiple recipients have their flag set to MAPI_AMBIGUOUS and their **ADRENTRY** structure untouched. MAPI displays these ambiguous names in a dialog box that prompts the user to resolve the ambiguity.

If it is not possible to return all of the properties requested for an unresolved entry in the *lpPropTagArray* parameter, and these properties don't exist in the **ADRENTRY** structure passed in, the address book provider should set the property type of each unavailable property to PT_ERROR. Any property columns that are already included for an entry should be retained if that entry is resolved.

If a provider is replacing an **ADRENTRY**, it should first free the **ADRENTRY** using the **MAPIFreeBuffer** function and then allocate the replacement **ADRENTRY** using the **MAPIAllocateBuffer** function.

MAPI requires certain properties to submit a message and will call providers to get those properties as part of its implementations of the **IAddrBook::PrepareRecips** and **IMAPISupport::ExpandRecips** methods. Providers implementing **ResolveNames** can eliminate the MAPI callbacks for **PrepareRecips** and **ExpandRecips**, and thus improve performance for message submission, by returning the following property columns when they resolve entry names:

PR_ADDRTYPE
PR_DISPLAY_NAME
PR_EMAIL_ADDRESS
PR_ENTRYID
PR_OBJECT_TYPE
PR_SEARCH_KEY
PR_TRANSMITTABLE_DISPLAY_NAME

Clients can also use the returned property columns in their calls to the **IMessage::ModifyRecipients** method.

See Also

ADRENTRY structure, **ADRLIST** structure, **IAddrBook::PrepareRecips** method, **IAddrBook::ResolveName** method, **IMAPISupport::ExpandRecips** method, **IMessage::ModifyRecipients** method, PR_ANR property, **SPropertyRestriction** structure

IABLogon : IUnknown

The **IABLogon** interface is used to access resources in an address book provider.

At a Glance

Specified in header file:	MAPISPI.H
Object that supplies this interface:	Address book logon object
Corresponding pointer type:	LPABLOGON
Implemented by:	Address book providers
Called by:	MAPI

Vtable Order

GetLastError	Returns information about the last error that occurred for an address book object.
Logoff	Logs a client application off an address book provider.
OpenEntry	Opens a container or recipient object and returns a pointer to the object to provide further access.
CompareEntryIDs	Compares two entry identifiers to determine if they refer to the same object.
Advise	Registers an object for notifications about changes within an address book.
Unadvise	Removes an object's registration for notification of address book changes previously established with a call to the **IABLogon::Advise** method.
OpenStatusEntry	Opens a status object.
OpenTemplateID	Allows run-time binding of one address book provider's code to data for an entry in another address book provider, so the entry's properties can later be updated.
GetOneOffTable	Returns a table of templates for custom recipient addresses that can be used to create recipients for a message.
PrepareRecips	Prepares a recipient list for later use by the messaging system.

IABLogon::Advise

The **IABLogon::Advise** method registers an advise sink object for notifications about changes within containers supported by this address book provider.

HRESULT Advise(
  ULONG *cbEntryID*,
  LPENTRYID *lpEntryID*,
  ULONG *ulEventMask*,
  LPMAPIADVISESINK *lpAdviseSink*,
  ULONG FAR * *lpulConnection*
)

Parameters

cbEntryID
Input parameter containing the size, in bytes, of the entry identifier pointed to by the *lpEntryID* parameter.

lpEntryID
Input parameter pointing to the entry identifier of the object about which notifications should be generated.

ulEventMask
Input parameter containing an event mask of the types of notification events occurring for the object about which MAPI will generate notifications. The mask filters specific cases. Each event type has a structure associated with it that holds additional information about the event. The following table lists the possible event types along with their corresponding structures.

Notification event type	Corresponding structure
`fnevCriticalError`	**ERROR_NOTIFICATION**
`fnevObjectCreated`	**OBJECT_NOTIFICATION**
`fnevObjectDeleted`	**OBJECT_NOTIFICATION**
`fnevObjectModified`	**OBJECT_NOTIFICATION**
`fnevObjectCopied`	**OBJECT_NOTIFICATION**
`fnevObjectMoved`	**OBJECT_NOTIFICATION**

lpAdviseSink
Input parameter pointing to the advise sink object to be called when an event occurs for the object about which notification has been requested.

lpulConnection
Output parameter pointing to a variable that upon a successful return holds the connection number for the notification registration. The connection number must be nonzero.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

MAPI_E_INVALID_ENTRYID
The service provider is not able to use the entry identifier passed in *lpEntryID*.

MAPI_E_NO_SUPPORT
The service provider either does not support changes to its objects or does not support notification

of changes.

MAPI_E_UNKNOWN_ENTRYID
A service provider that could handle the *lpEntryID* entry identifier could not be found.

Remarks

Address book providers implement the **IABLogon::Advise** method to register an advise sink object to receive notifications relating to changes in one or more of their containers. Advise sinks typically call **IAddrBook::Advise** and MAPI forwards this call to the address book provider active for the session that is responsible for the object indicated by the entry identifier in *lpEntryID*. Whenever a change occurs to the indicated object, the provider checks to see what event mask bit was set in the *ulEventMask* parameter and thus what type of change occurred. If a bit is set, then the provider calls the **IMAPIAdviseSink::OnNotify** method for the advise sink object indicated by the *lpAdviseSink* parameter to report the event. Data passed in the notification structure to the **OnNotify** routine describes the event.

The call to **OnNotify** can occur during the call that changes the object, or at any following time. On systems that support multiple threads of execution, the call to **OnNotify** can occur on any thread. For a way to turn a call to **OnNotify** that might happen at an inopportune time into one that is safer to handle, a client application should use the **HrThisThreadAdviseSink** function.

To provide notifications, the address book provider implementing **Advise** needs to keep a copy of the pointer to the *lpAdviseSink* advise sink object; to do so, it calls the **IUnknown::AddRef** method for the advise sink to maintain its object pointer until notification registration is canceled with a call to the **IABLogon::Unadvise** method. The **Advise** implementation should assign a connection number to the notification registration and call **AddRef** on this connection number before returning it in the *lpulConnection* parameter. Service providers can release the advise sink object before the registration is canceled, but they must not release the connection number until **Unadvise** has been called.

After a call to **Advise** has succeeded and before **Unadvise** has been called, clients must be prepared for the advise sink object to be released. A client should therefore release its advise sink object after **Advise** returns, unless it has a specific long-term use for it.

For more information on the notification process, see About Notification.

See Also

HrThisThreadAdviseSink function, **IABLogon::Unadvise** method, **IMAPIAdviseSink::OnNotify** method, **NOTIFICATION** structure

IABLogon::CompareEntryIDs

The **IABLogon::CompareEntryIDs** method compares two entry identifiers belonging to this address book provider to determine if they refer to the same object.

HRESULT CompareEntryIDs(
  ULONG *cbEntryID1***,**
  LPENTRYID *lpEntryID1***,**
  ULONG *cbEntryID2***,**
  LPENTRYID *lpEntryID2***,**
  ULONG *ulFlags***,**
  ULONG FAR * *lpulResult*
)

Parameters

cbEntryID1
  Input parameter containing the size, in bytes, of the entry identifier pointed to by the *lpEntryID1* parameter.

lpEntryID1
  Input parameter pointing to the first entry identifier to be compared.

cbEntryID2
  Input parameter containing the size, in bytes, of the entry identifier pointed to by the *lpEntryID2* parameter.

lpEntryID2
  Input parameter pointing to the second entry identifier to be compared.

ulFlags
  Reserved; must be zero.

lpulResult
  Output parameter pointing to a variable where the returned result of the comparison is stored; this variable is TRUE if the two entry identifiers refer to the same object, and FALSE otherwise.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

Remarks

Address book providers implement the **IABLogon::CompareEntryIDs** method to compare two entry identifiers for a given entry within one or more of their address book containers to determine whether they refer to the same object. If the two entry identifiers refer to the same object, then **CompareEntryIDs** sets the *lpulResult* parameter to TRUE; if they refer to different objects, **CompareEntryIDs** sets *lpulResult* to FALSE.

CompareEntryIDs is useful because an object can have more than one valid entry identifier; such a situation can occur, for example, when trying to compare a short-term entry identifier with a long-term entry identifier.

IABLogon::GetLastError

The **IABLogon::GetLastError** method returns a **MAPIERROR** structure containing information about the previous error that occurred in an address book object.

HRESULT GetLastError(
  HRESULT *hResult***,**
  ULONG *ulFlags***,**
  LPMAPIERROR FAR * *lppMAPIError*
)

Parameters

hResult
: Input parameter containing the result returned for the last call for the address book object that returned an error.

ulFlags
: Input parameter containing a bitmask of flags that controls the type of the returned strings. The following flag can be set:

  MAPI_UNICODE
  : Indicates the strings in the **MAPIERROR** structure returned in the *lppMAPIError* parameter are in Unicode format. If the MAPI_UNICODE flag is not set, the strings are in ANSI format.

lppMAPIError
: Output parameter pointing to a pointer to the returned **MAPIERROR** structure containing version, component, and context information for the error. The *lppMAPIError* parameter can be set to NULL if there is no **MAPIERROR** structure to return.

Return Values

S_OK
: The call succeeded and has returned the expected value or values.

MAPI_E_BAD_CHARWIDTH
: Either the MAPI_UNICODE flag was set and the implementation does not support Unicode, or MAPI_UNICODE was not set and the implementation only supports Unicode.

Remarks

Address book providers implement the **IABLogon::GetLastError** method to retrieve information to display in a message to the user regarding the last error returned from a method call for the address book object.

To release all the memory allocated by MAPI, client applications need only call the **MAPIFreeBuffer** function for the returned **MAPIERROR** structure.

The return value from **GetLastError** must be S_OK for an application to make use of the **MAPIERROR** structure. Even if the return value is S_OK, a **MAPIERROR** structure might not be returned. If the implementation cannot determine what the last error was, or if a **MAPIERROR** structure is not available for that error, **GetLastError** returns a pointer to NULL in *lppMAPIError* instead.

See Also

MAPIERROR structure, **MAPIFreeBuffer** function

IABLogon::GetOneOffTable

The **IABLogon::GetOneOffTable** method returns a table of one-off templates for creating recipients to be added to the recipient list of an outgoing message.

HRESULT GetOneOffTable(
  ULONG *ulFlags*,
  LPMAPITABLE FAR * *lppTable*
)

Parameters

ulFlags
: Input parameter containing a bitmask of flags that controls the type of the returned strings. The following flag can be set:

  MAPI_UNICODE
  : Indicates the returned strings are in Unicode format. If the MAPI_UNICODE flag is not set, the strings are in ANSI format.

lppTable
: Output parameter pointing to a pointer to the returned table.

Return Values

S_OK
: The call succeeded and has returned the expected value or values.

MAPI_E_BAD_CHARWIDTH
: Either the MAPI_UNICODE flag was set and the implementation does not support Unicode, or MAPI_UNICODE was not set and the implementation only supports Unicode.

MAPI_E_NO_SUPPORT
: The address book provider doesn't have any custom recipient lists.

Remarks

MAPI calls the **IABLogon::GetOneOffTable** method to make available one-off templates for creating recipients. The following properties must be available as columns in the one-off table returned by **GetOneOffTable**:

PR_ADDRTYPE
PR_DEPTH
PR_DISPLAY_NAME
PR_DISPLAY_TYPE
PR_ENTRYID
PR_INSTANCE_KEY
PR_SELECTABLE

MAPI keeps the one-off table open, and changes to this table should be handled through table notifications. If the address book provider changes the table, it should call the **IMAPIAdviseSink::OnNotify** method with the appropriate event mask for any clients that have registered for changes on that table.

Setting the MAPI_UNICODE flag in the *ulFlags* parameter does the following:

- Sets the string type to Unicode for data returned for the initial active columns of the one-off table by the **IMAPITable::QueryColumns** method. The initial active columns for the table are those columns **QueryColumns** returns before the provider that contains the table calls the **IMAPITable::SetColumns** method.

- Sets the string type to Unicode for data returned for the initial active rows of the one-off table by the **IMAPITable::QueryRows** method. The initial active rows are those rows **QueryRows** returns before the provider that contains the table calls **SetColumns**.
- Controls the property types of the sort order returned by the **IMAPITable::QuerySortOrder** method before the provider that contains the one-off table calls the **IMAPITable::SortTable** method.

See Also

BuildDisplayTable function, **IABContainer::CreateEntry** method, **IAddrBook::NewEntry** method, **IMAPISupport::GetOneOffTable** method

IABLogon::Logoff

The **IABLogon::Logoff** method logs off an address book provider.

HRESULT Logoff(
  ULONG *ulFlags*
)

Parameters

ulFlags
  Reserved; must be zero.

Return Value

S_OK
  The call succeeded and has returned the expected value or values.

Remarks

Address book providers implement the **IABLogon::Logoff** method to support the logoff process. **IABLogon::Logoff** is called by MAPI after a client has called **IMAPISession::Logoff** to end a session. **IABLogon::Logoff** performs the following tasks:

- Releases all open objects, such as any subobjects or the status object.
- Releases the provider's support object.

For more information about the logoff process of address book providers, see Shutting Down Service Providers.

See Also

IABProvider::Logon method

IABLogon::OpenEntry

The **IABLogon::OpenEntry** method opens a container or recipient object and returns a pointer to the object to provide further access. A recipient can be either a messaging user or a distribution list.

HRESULT OpenEntry(
  ULONG *cbEntryID***,**
  LPENTRYID *lpEntryID***,**
  LPCIID *lpInterface***,**
  ULONG *ulFlags***,**
  ULONG FAR * *lpulObjType***,**
  LPUNKNOWN FAR * *lppUnk*
)

Parameters

cbEntryID
  Input parameter containing the size, in bytes, of the entry identifier pointed to by the *lpEntryID* parameter.

lpEntryID
  Input parameter pointing to the entry identifier for the container or recipient object to open.

lpInterface
  Input parameter pointing to the interface identifier (IID) representing the interface to be used for further access to the opened object. Passing NULL indicates the provider should return the standard interface for the object, such as **IMailUser** for messaging users and **IDistList** for distribution lists.

ulFlags
  Input parameter containing a bitmask of flags that controls how the object is opened. The following flags can be set:

  MAPI_BEST_ACCESS
    Indicates the object should be opened with the maximum network permissions allowed for the user and the maximum client application access. For example, if the client has read/write access, the object is opened with read/write access; if the client has read-only access, the object is opened with read-only access. The client can retrieve the access level by getting the property PR_ACCESS_LEVEL.

  MAPI_DEFERRED_ERRORS
    Indicates the call is allowed to succeed even if the underlying object is not accessible to the calling process. If the object is not accessible, some subsequent call to the object might return an error.

  MAPI_MODIFY
    Requests read/write access. By default, objects are created with read-only access, and clients should not work on the assumption that read/write access has been granted.

lpulObjType
  Output parameter pointing to a variable where the type of the opened object is stored.

lppUnk
  Output parameter pointing to a variable where the pointer to the opened object is stored.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

MAPI_E_NO_ACCESS
  An attempt was made to modify a read-only object or an attempt was made to access an object for which the user has insufficient permissions.

MAPI_E_NOT_FOUND

There is not an object associated with the entry identifier passed in *lpEntryID*.

MAPI_E_UNKNOWN_ENTRYID

The entry identifier passed in the *lpEntryID* parameter is in an unrecognizable format. This value is typically returned if the address book provider that contains the object is not open.

Remarks

MAPI calls the **IABLogon::OpenEntry** method to open a container or recipient. An address book provider only receives an **OpenEntry** call for an object with an entry identifier that MAPI has determined belongs to that provider. The provider returns a pointer that provides further access to the opened object. The default behavior is to open the object as read-only, unless the call sets the MAPI_MODIFY or MAPI_BEST_ACCESS flag in the *ulFlags* parameter. If an address book provider does not allow modification for the object requested, then it should return the value MAPI_E_NO_ACCESS.

The *lpInterface* parameter indicates which interface should be used for the opened object. Passing NULL in *lpInterface* indicates the standard MAPI interface for that type of object should be used. An address book provider can implement another interface for the object by passing the interface identifier for the interface that should be returned.

If MAPI passes NULL for *lpEntryID*, it indicates the provider should open the root container in its container hierarchy.

When the entry to be opened supports a template identifier through its PR_TEMPLATEID property, the provider calls **IMAPISupport::OpenTemplateID,** passing the template identifier for the *lpTemplateID* parameter and a zero value for the *ulTemplateFlags* parameter**. IMAPISupport::OpenTemplateID** passes this information to the foreign provider in a call to the foreign provider's **IABLogon::OpenTemplateID** method. This set of calls binds code in the foreign provider to data in the host provider. If **IMAPISupport::OpenTemplateID** returns an error, usually because the foreign provider is unavailable or not included in the profile, the provider should try to continue by treating the unbound entry as read-only.

For more information about opening foreign address book entries, see Implementing a Foreign Address Book Provider or Implementing a Host Address Book Provider.

IABLogon::OpenStatusEntry

The **IABLogon::OpenStatusEntry** method opens the provider's status object.

HRESULT OpenStatusEntry(
  LPCIID *lpInterface*,
  ULONG *ulFlags*,
  ULONG FAR * *lpulObjType*,
  LPMAPISTATUS FAR * *lppMAPIStatus*
)

Parameters

lpInterface
: Input parameter pointing to the interface identifier (IID) representing the interface to be used for further access to the opened status object. Passing NULL indicates that the provider should return the standard interface, or **IMAPIStatus**.

ulFlags
: Input parameter containing a bitmask of flags that controls how the status object is opened. The following flag can be set:

  MAPI_MODIFY
  : Requests read/write access. By default, objects are created with read-only access, and client applications should not work on the assumption that read/write access has been granted.

lpulObjType
: Output parameter pointing to the type of the opened object.

lppEntry
: Output parameter pointing to a pointer to the opened object.

Return Value

S_OK
: The call succeeded and the status object has been opened.

Remarks

Address book providers implement the **IABLogon::OpenStatusEntry** method to open their status object. Clients can use this status object to, for example, reconfigure options for the provider or validate the provider's state.

See Also

IMAPIStatus : IMAPIProp interface, **IMAPIStatus::SettingsDialog** method, **IMAPIStatus::ValidateState** method

IABLogon::OpenTemplateID

The **IABLogon::OpenTemplateID** method allows the implementing address book provider to bind code code within its implementation to data for one of its entries within a host provider's address book container. This binding occurs at run time.

HRESULT OpenTemplateID(
  ULONG *cbTemplateID***,**
  LPENTRYID *lpTemplateID***,**
  ULONG *ulTemplateFlags***,**
  LPMAPIPROP *lpMAPIPropData***,**
  LPCIID *lpInterface***,**
  LPMAPIPROP FAR * *lppMAPIPropNew***,**
  LPMAPIPROP *lpMAPIPropSibling*
)

Parameters

cbTemplateID
: Input parameter containing the size, in bytes, of the template identifier pointed to by the *lpTemplateID* parameter.

lpTemplateID
: Input parameter pointing to the template identifier. This value is stored in the PR_TEMPLATEID property of an entry in the implementing provider's container.

ulTemplateFlags
: Input parameter containing a bitmask of flags used to indicate how to open the entry represented by the template identifier. The following flag can be set:

  FILL_ENTRY
  : Indicates that the host provider is creating a new entry in its container based on the entry represented by the template identifier. The implementing provider should either perform specific initialization of the host provider's entry, pointed to by the *lpMAPIPropData* parameter, or bind its own **IMAPIProp** implementation by returning it through the *lppMAPIPropNew* parameter.

lpMAPIPropData
: Input parameter pointing to a property object, an **IMAPIProp** implementation, in the host provider's container for an entry that is based on the entry in the implementing provider's container that is represented by the *lpTemplateID* parameter.

lpInterface
: Input parameter pointing to the interface identifier (IID) specifying the interface to be returned for the bound property object. If this parameter is NULL, the standard messaging user interface, **IMailUser**, should be returned.

lppMAPIPropNew
: Output parameter pointing to the bound property object, the **IMAPIProp** implementation supplied by the implementing provider.

lpMAPIPropSibling
: Reserved; must be NULL.

Return Values

S_OK
: The call succeeded and has returned the expected value or values.

MAPI_E_UNKNOWN_ENTRYID
: The template identifier passed is not recognized by the address book provider.

Remarks

An address book provider implements the **IABLogon::OpenTemplateID** method to bind code to data supported by a host address book provider. This binding occurs at run time in response to the host provider retrieving its entry's PR_TEMPLATEID property and passing it in a call to **IMAPISupport::OpenTemplateID**.

An address book provider only needs to implement **OpenTemplateID** and related code to maintain control over copies of its entries that exist in the containers of host providers. **OpenTemplateID** is called with the value of an entry's PR_TEMPLATEID property when the entry is created or when it is opened by MAPI.

An address book provider should implement **OpenTemplateID** and the related property object to handle the following situations:

- To periodically update the data for a copied entry so that it stays in sync with the original.
- To implement functionality that the host provider cannot implement, such as dynamically populating a list box that appears in the entry's details table from data on a server.
- To control interaction between properties in the host provider's entry and the original entry, such as computing PR_EMAIL_ADDRESS from edit controls in the details display that hold various components of the adddress.

In general, a property object implemented for a host provider should intercept all of the methods to perform context-specific manipulation of the relevant properties. If the FILL_ENTRY flag is passed in the *ulFlags* parameter, the address book provider must set all properties for the entry.

Address book providers that return a new property object in the *lppMAPIPropNew* parameter must call the **IUnknown::AddRef** method of the host provider's property object to maintain a reference. All calls through the bound object, the **IMAPIProp** implementation returned in *lppMAPIPropNew*, should be routed to their corresponding method on the host property object after they are dealt with by the bound object.

The property identifiers of any named properties passed through the bound object are in the host provider's identifier name space. The implementing provider **GetNamesFromIDs** method should determine the names of the properties so that it can perform any tasks specific to the implementation. Similarly, properties that the implementing provider passes to the property object must also be in the foreign provider's identifier name space. For example, if **IABLogon::OpenTemplateID** sets a named property, the property should use a property identifier belonging to the host provider. If necessary, the implementing provider should use **GetIDsFromNames** to create an appropriate identifier.

If an address book provider doesn't recognize the entry identifier passed in *lpTemplateID*, it should return MAPI_E_UNKNOWN_ENTRYID.

For more information on working with address book template identifiers, see Implementing a Foreign Address Book Provider.

See Also

IMAPISupport::OpenTemplateID method, **IPropData : IMAPIProp** interface, PR_TEMPLATEID property.

IABLogon::PrepareRecips

The **IABLogon::PrepareRecips** method prepares a recipient list for later use by the messaging system.

HRESULT PrepareRecips(
  ULONG *ulFlags***,**
  LPSPropTagArray *lpPropTagArray***,**
  LPADRLIST *lpRecipList*
)

Parameters

ulFlags
  Reserved; must be zero.

lpPropTagArray
  Input parameter pointing to an **SPropTagArray** structure containing an array of property tags indicating the properties that require updating, if any. The *lpPropTagArray* parameter can be NULL.

lpRecipList
  Input parameter pointing to an **ADRLIST** structure holding the list of recipients.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

MAPI_E_NOT_FOUND
  The requested object does not exist.

Remarks

MAPI calls the **IABLogon::PrepareRecips** method to ensure that each recipient in the address list pointed to by *lpRecipList* has a long-term entry identifier and that all of the properties requested in the *lpPropTagArray* parameter exist in the property value array included in the address list's **ADRENTRY** structure member for the recipient. To do so, **PrepareRecips** converts recipients' short-term entry identifiers to long-term entry identifiers and updates the recipients from the list that belong to this address book provider. If necessary, **PrepareRecips** also retrieves a recipient's long-term entry identifier along with any additional properties requested.

Within the property value array associated with a recipient's **ADRENTRY** structure in the address list, the requested properties are ordered first, followed by any additional properties that were already present for the entry. If one or more of the requested properties are not recognized by an address book provider, the provider should set their property types to PT_ERROR and their property values either to MAPI_E_NOT_FOUND or to another value giving a more specific reason why the properties are not available.

Like the **ADRLIST** structure as a whole , each **SPropValue** property value structure passed in *lpPropTagArray* must be separately allocated using the **MAPIAllocateBuffer** and **MAPIAllocateMore** functions such that it can be freed individually. If the provider must allocate additional space for any **SPropValue** structure, for example to store the data for a string property, it can use **MAPIAllocateBuffer** to allocate additional space for the full property-tag array, use the **MAPIFreeBuffer** function to free the original property-tag array, and then use **MAPIAllocateMore** to allocate any additional memory required.

See Also

ADRLIST structure, **IMAPIProp::GetProps** method, **IMessage::ModifyRecipients** method, PR_ENTRYID property, PT_ERROR property type, **SPropValue** structure, **SRowSet** structure

IABLogon::Unadvise

The **IABLogon::Unadvise** method cancels a notification registration represented by a specified connection number.

```
HRESULT Unadvise(
    ULONG ulConnection
  )
```

Parameters

ulConnection

Input parameter containing the connection number associated with an active notification registration. The value of *ulConnection* must have been returned by a previous call to **IABLogon::Advise**.

Return Value

S_OK

The call succeeded and has returned the expected value or values.

Remarks

MAPI calls the **IABLogon::Unadvise** method to cancel a registration to be notified of changes to one or more address book provider objects. **Unadvise** cancels the registration by releasing the pointer to the advise sink object passed in the *lpAdviseSink* parameter in the previous call to **IABLogon::Advise**. Generally, the provider calls the advise sink's **IUnknown::Release** method during the **Unadvise** call, but if another thread is in the process of calling the advise sink's **IMAPIAdviseSink::OnNotify** method, the **Release** call is delayed until **OnNotify** returns.

See Also

IABLogon::Advise method, **IMAPIAdviseSink::OnNotify** method

IABProvider : IUnknown

The **IABProvider** interface provides a method to log on to an address book provider object and a method to invalidate an address book provider object.

At a Glance

Specified in header file:	MAPISPI.H
Object that supplies this interface:	Address book provider object
Corresponding pointer type:	LPABPROVIDER
Implemented by:	Address book providers
Called by:	MAPI

Vtable Order

Shutdown	Closes an address book provider object in an orderly fashion.
Logon	Logs MAPI read/write one instance of an address book provider.

IABProvider::Logon

The **IABProvider::Logon** method logs MAPI on to one instance of the address book provider.

HRESULT Logon(
  LPMAPISUP *lpMAPISup*,
  ULONG *ulUIParam*,
  LPTSTR *lpszProfileName*,
  ULONG *ulFlags*,
  ULONG FAR * *lpulcbSecurity*,
  LPBYTE FAR * *lppbSecurity*,
  LPMAPIERROR FAR * *lppMAPIError*,
  LPABLOGON FAR * *lppABLogon*
)

Parameters

lpMAPISup
: Input parameter pointing to the current MAPI support object for the address book provider.

ulUIParam
: Input parameter containing the handle of the parent window for the logon dialog box that **Logon** displays if permitted. The *ulUIParam* parameter contains the value of the parameter of the same name passed to MAPI in the preceding call to the **MAPILogonEx** function.

lpszProfileName
: Input parameter pointing to a string containing the name of the session profile.

ulFlags
: Input parameter containing a bitmask of flags that controls how the logon is performed. The following flags can be set:

  AB_NO_DIALOG
  : Indicates the provider should not display a dialog box during logon. If this flag is not set, the provider can display a dialog box to prompt the user for missing configuration information.

  MAPI_DEFERRED_ERRORS
  : Indicates the provider does not have to complete the logon process before returning.

  MAPI_UNICODE
  : Indicates the passed-in strings are in Unicode format. If the MAPI_UNICODE flag is not set, the strings are in ANSI format.

lpulcbSecurity
: Input-output parameter pointing to the size, in bytes, of the security credentials structure pointed to by the *lppbSecurity* parameter. On input, the value must be nonzero; on output, the value must be zero. In both cases the pointers must be valid.

lppbSecurity
: Input-output parameter pointing to a pointer to the buffer containing security credentials. On input, the value must be nonzero; on output, the value must be zero. In both cases the pointers must be valid.

lppMAPIError
: Output parameter pointing to a pointer to the returned **MAPIERROR** structure. The *lppMAPIError* parameter can be set to NULL if there is no **MAPIERROR** structure to return.

lppABLogon
: Output parameter pointing to a pointer to the provider's logon object.

Return Values

S_OK
: The call succeeded and has returned the expected value or values.

MAPI_E_FAILONEPROVIDER
  Indicates that the provider cannot logon, but that MAPI can continue to log on the other providers in the message service to which the provider belongs.

MAPI_E_UNCONFIGURED
  Indicates that the provider does not have enough information to complete the logon. MAPI calls the provider's message-service entry function.

MAPI_E_UNKNOWN_CPID
  Indicates the server is not configured to support the client's code page.

MAPI_E_UNKNOWN_LCID
  Indicates the server is not configured to support the client's locale information.

MAPI_E_USER_CANCEL
  The user canceled the operation, typically by clicking the **Cancel** button in the logon dialog box.

Remarks

When a client calls the **IMAPISession::OpenAddressBook** method, MAPI loads all of the address book providers in the profile into memory by calling each provider's **IABProvider::Logon** method.

IABProvider::Logon calls **IMAPISupport::SetProviderUID** to register the provider's unique identifier, or **MAPIUID**. A provider's **MAPIUID** represents the provider in every entry identifier for every one of its objects. MAPI uses the **MAPIUID** to match an object with its provider. For example, when a client calls **IMAPISession::OpenEntry** to open a messaging user, **OpenEntry** examines the **MAPIUID** portion of the entry identifier passed in and matches it with a **MAPIUID** registered by an address book provider.

If a client logs read/write an address book provider more than once, the provider can register one or more **MAPIUIDs**. If the provider registers a single **MAPIUID** every time MAPI calls its **IABProvider::Logon** method, the provider must be able to perform the routing rather than leaving it up to MAPI. For more information on using multiple logon objects, see Loading Service Providers.

MAPI passes **IABProvider::Logon** a support object in the *lpMAPISup* parameter to provide access to methods needed by the provider. Each provider instance receives a difference support object instance. A provider must call the **IUnknown::AddRef** method for the support object if the **IABProvider::Logon** call is successful; otherwise, the provider is unloaded.

The local name of the user's profile in *lpszProfileName* is provided as a convenience for address book providers. It can be used in error dialog boxes, logon screens, or other user interfaces to show the user the name of the profile. For a provider to keep and use this profile name, it must copy the name to storage that it has allocated. The profile name is displayed in the character set of the user's client as indicated by the presence or absence of the MAPI_UNICODE flag in the *ulFlags* parameter.

The address book provider should create a logon object and return a pointer to it in the *lppABLogon* parameter. MAPI uses this logon object on several subsequent calls, for example when it calls the provider's **IABLogon::OpenEntry** and **IABLogon::Logoff** methods.

An address book provider that needs to logon to an underlying messaging system or directory service can use the MAPI support method **IMAPISupport::OpenProfileSection** for saving and retrieving security credential sets for this particular logon session. If a provider finds that all required information is not in the current profile, it should return MAPI_E_UNCONFIGURED so that MAPI calls the provider's message service entry function.

If a provider requires a password during logon, a logon dialog box is displayed, unless the AB_NO_DIALOG flag is set in *ulFlags*. If the user cancels the logon process, typically by clicking the Cancel button in a dialog box, a provider should return MAPI_E_USER_CANCEL.

When MAPI logs read/write a provider, the MAPI spooler also logs read/write the provider. The values in the *lpulcbSecurity* and *lppbSecurity* parameters are copied from the MAPI spooler's logon. These values are used to allow multiple logons to share the same security context. On output, the values for the *lpulcbSecurity* and *lppbSecurity* parameters must be allocated with the **MAPIAllocateBuffer** function. The address book provider can ignore *lpulcbSecurity* and *lppbSecurity*.

Typically when an address book provider cannot logon, MAPI disables the message service to which the failing provider belongs. That is, MAPI will not try to logon any of the other providers belonging to the service for the rest of the session's lifetime. However, two errors cause MAPI to continue the logon with other providers: MAPI_E_FAILONEPROVIDER and MAPI_E_UNCONFIGURED. With the MAPI_E_FAILONEPROVIDER and MAPI_E_UNCONFIGURED, MAPI does not disable the message service to which the provider belongs. **Logon** should return MAPI_E_FAILONEPROVIDER if it encounters an error that does not warrant disabling the entire service for the life of the session.

If a provider returns MAPI_E_UNCONFIGURED from its logon, MAPI will call the provider's message service entry point function and then retry the logon. MAPI passes MSG_SERVICE_CONFIGURE as the *ulContext* parameter, to give the service a chance to configure itself. If the client has chosen to allow a user interface on the logon, the service can present its configuration property sheet.

See Also

IABLogon::Logoff method, **IABLogon::OpenEntry** method, **IMAPISupport::OpenProfileSection** method, **IMAPISupport::SetProviderUID** method, **MSGSERVICEENTRY** function prototype

IABProvider::Shutdown

The **IABProvider::Shutdown** method logs off an instance of the address book provider.

HRESULT Shutdown (
  ULONG FAR * *lpulFlags*
)

Parameters

lpulFlags
  Reserved; must be a pointer to zero.

Return Value

S_OK
  The call succeeded and has returned the expected value or values.

Remarks

When MAPI calls an address book provider's **IABLogon::Shutdown** method, the provider does whatever it needs to do to shut down. **Shutdown** is only called after all of the provider's logon objects have been released.

IAddrBook : IUnknown

The **IAddrBook** interface supports access to the MAPI address book and includes operations such as displaying common dialog boxes, opening containers within the address book, and performing name resolution.

At a Glance

Specified in header file:	MAPIX.H
Object that supplies this interface:	Address book object
Corresponding pointer type:	LPADRBOOK
Implemented by:	MAPI
Called by:	Client applications, service providers

Vtable Order

OpenEntry	Opens a container or recipient object and returns a pointer to the object to provide further access.
CompareEntryIDs	Compares two entry identifiers to determine if they refer to the same object.
Advise	Registers an object for notifications about changes within the address book.
Unadvise	Removes a registration for notification of address book changes previously established with a call to the **IAddrBook::Advise** method.
CreateOneOff	Creates an entry identifier in the special format reserved for one-off recipients.
NewEntry	Displays a dialog box for creating new entries within a container or the recipient list of a message.
ResolveName	Initiates the name resolution process to assign entry identifiers to recipients.
Address	Displays the common address dialog used to browse and modify the address book.
Details	Displays a dialog box showing details for a particular entry in the address book.
RecipOptions	Displays a dialog box enabling a user to change options for a particular recipient.
QueryDefaultRecipOpt	Returns the available recipient options for a particular messaging address type and sets them to their defaults.
GetPAB	Returns the entry identifier for the container designated as the Personal Address Book (PAB).
SetPAB	Designates a particular container to be the Personal Address Book.
GetDefaultDir	Returns the entry identifier for the address book container designated to provide the initial data for the address dialog box display.

SetDefaultDir	Designates a particular container to provide the initial data for the address dialog box display.
GetSearchPath	Returns a list of ordered entry identifiers representing containers that **IAddrBook::ResolveName** uses in the name resolution process.
SetSearchPath	Establishes an order for one or more containers to be used by **IAddrBook::ResolveName**..
PrepareRecips	Prepares a recipient list for later use by the messaging system.

IAddrBook::Address

The **IAddrBook::Address** method displays the common address dialog box.

HRESULT Address(
  ULONG FAR * *lpulUIParam***,**
  LPADRPARM *lpAdrParms***,**
  LPADRLIST FAR * *lppAdrList*
)

Parameters

lpulUIParam
  Input-output parameter containing the handle of the parent window of the dialog box. On input, a window handle must always be passed. On output, if the **ulFlags** member of the *lpAdrParms* parameter is set to DIALOG_SDI, then the window handle of the modeless dialog box is returned.

lpAdrParms
  Input-output parameter pointing to an **ADRPARM** structure that controls the presentation and behavior of the address dialog box.

lppAdrList
  Input-output parameter that is the address of an array of ADRLIST structures containing recipient information. On input, this parameter can be NULL or point to a valid array. On output, this parameter points to a valid array of recipient information.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

Remarks

When a client application or service provider calls the **IAddrBook::Address** method, it passes a list of recipients, possibly empty, in the *lppAdrList* parameter, and **Address** returns in *lppAdrList* an **ADRLIST** structure holding an updated list of recipients. Clients can use the updated list for an outgoing message by calling the message's **IMessage::ModifyRecipients** method.

Each recipient in the **ADRLIST** structure is represented by an **ADRENTRY** structure. The **ADRENTRY** structures are organized in the **ADRLIST** by the type of recipient they hold, as indicated by the recipient's PR_RECIPIENT_TYPE property. The possible types are MAPI_TO for direct recipients, MAPI_CC for carbon copy recipients, and MAPI_BCC for blind carbon copy recipients.

ADRLIST structures can hold resolved and unresolved recipients. The difference between a resolved and unresolved recipient is that a resolved recipient always has an entry identifier. The **rgPropVals** member of its **ADRENTRY** structure contains as one of its property values a value for the PR_ENTRYID property. Resolved recipients include at least the following properties:

  PR_ENTRYID
  PR_RECIPIENT_TYPE
  PR_DISPLAY_NAME
  PR_ADDRTYPE
  PR_DISPLAY_TYPE

When a user creates a recipient list for an outgoing message by directly entering recipients, the recipients that are created are unresolved. The client fills the **rgPropVals** member of the **ADRENTRY** structure for each recipient with only the PR_DISPLAY_NAME and PR_RECIPIENT_TYPE properties. Recipient lists that are created by selecting from address book container entries contain resolved recipients.

In addition to resolved and unresolved recipient entries, **ADRENTRY** structures can be NULL, that is, the **cValues** member is zero. This is the case, for example, when the dialog box presented by **IAddrBook::Address** is used to remove a recipient from the list.

The **ADRLIST** structure holding the address list must be separately allocated using the **MAPIAllocateBuffer** function. If the **Address** method needs to pass a larger **ADRLIST** structure on output than was passed in on input, or if NULL is passed in *lppAdrList* on input, then **Address** allocates a larger buffer for the **ADRLIST** structure it returns using **MAPIAllocateBuffer** and returns this buffer's address in *lppAdrList*. **Address** frees the old buffer by using the **MAPIFreeBuffer** function.

Each **SPropValue** property value structure is separately allocated by using **MAPIAllocateBuffer**. The **Address** method allocates additional property value structures and frees old ones as appropriate.

Address returns immediately if the DIALOG_SDI flag is set in the **ADRPARM** structure in the *lpAdrParms* parameter.

See Also

ADRENTRY structure, **ADRLIST** structure, **ADRPARM** structure, **FreePadrlist** function, **FreeProws** function, **IMAPITable::QueryRows** method, **IMessage::ModifyRecipients** method, **MAPIAllocateBuffer** function, **MAPIAllocateMore** function, **MAPIFreeBuffer** function, **SPropValue** structure, **SRowSet** structure

IAddrBook::Advise

The **IAddrBook::Advise** method registers a client or service provider to receive notifications about changes to one or more entries in the address book.

HRESULT Advise(
  ULONG *cbEntryID*,
  LPENTRYID *lpEntryID*,
  ULONG *ulEventMask*,
  LPMAPIADVISESINK *lpAdviseSink*,
  ULONG FAR * *lpulConnection*
)

Parameters

cbEntryID
 Input parameter containing the size, in bytes, of the entry identifier pointed to by the *lpEntryID* parameter.

lpEntryID
 Input parameter pointing to the entry identifier of the address book container, messaging user, or distribution list that will generate a notification when a change occurs of the type or types described in the *ulEventMask* parameter.

ulEventMask
 Input parameter containing one or more notification events that the caller is registering to receive. Each event is associated with a particular notification structure containing information about the change that occurred. The following table lists the valid values for *ulEventMask* and their corresponding structures.

Notification event	Corresponding structure
`fnevCriticalError`	**ERROR_NOTIFICATION**
`fnevObjectCreated`	**OBJECT_NOTIFICATION**
`fnevObjectDeleted`	**OBJECT_NOTIFICATION**
`fnevObjectModified`	**OBJECT_NOTIFICATION**
`fnevObjectCopied`	**OBJECT_NOTIFICATION**
`fnevObjectMoved`	**OBJECT_NOTIFICATION**
`fnevTableModified`	**TABLE_NOTIFICATION**

lpAdviseSink
 Input parameter pointing to the advise sink object to be called when an event occurs for the object about which notification has been requested.

lpulConnection
 Output parameter pointing to a nonzero connection number that represents the notification registration.

Return Values

S_OK
 The call succeeded and has returned the expected value or values.

MAPI_E_INVALID_ENTRYID
 The address book provider responsible for the entry identifier passed in *lpEntryID* could not register a notification for the entry.

MAPI_E_NO_SUPPORT
 Notification is not supported by the address book provider responsible for the object identified by the entry identifier passed in the *lpEntryID* parameter.

MAPI_E_UNKNOWN_ENTRYID
  The entry identifier passed in *lpEntryID* can not be handled by any of the address book providers in the profile.

Remarks

Client applications and service providers call the **IAddrBook::Advise** method to register for notifications with one or more address book entries. MAPI forwards this call to the address book provider that is responsible for the object indicated by the entry identifier in the *lpEntryID* parameter. Whenever a change of the type requested in the *ulEventMask* parameter occurs to the indicated object, the provider calls the **IMAPIAdviseSink::OnNotify** method for the advise sink object indicated by the *lpAdviseSink* parameter to report the event. The **NOTIFICATION** structure passed in the *lpNotifications* parameter to **OnNotify** contains data that describes the event.

Depending on the address book provider, the call to **OnNotify** can occur immediately following the change to the registered object or at some later time. On systems that support multiple threads of execution, the call to **OnNotify** can occur on any thread. Clients can request that these notifications occur on a particular thread by calling the **HrThisThreadAdviseSink** function to create the advise sink object that is passed to **Advise**.

Because an address book provider can release the advise sink object passed in by clients at any time after the successful completion of the **Advise** call and before an **Unadvise** call to cancel the notification, clients should release their advise sink objects when **Advise** returns.

For more information on the notification process, see About Notification.

See Also

HrThisThreadAdviseSink function, **IAddrBook::Unadvise** method, **IMAPIAdviseSink::OnNotify** method, **NOTIFICATION** structure

IAddrBook::CompareEntryIDs

The **IAddrBook::CompareEntryIDs** method compares two entry identifiers belonging to a particular address book provider to determine if they refer to the same address book object.

HRESULT CompareEntryIDs(
  ULONG *cbEntryID1*,
  LPENTRYID *lpEntryID1*,
  ULONG *cbEntryID2*,
  LPENTRYID *lpEntryID2*,
  ULONG *ulFlags*,
  ULONG FAR * *lpulResult*
)

Parameters

cbEntryID1
  Input parameter containing the size, in bytes, of the entry identifier pointed to by the *lpEntryID1* parameter.

lpEntryID1
  Input parameter pointing to the first entry identifier to be compared.

cbEntryID2
  Input parameter containing the size, in bytes, of the entry identifier pointed to by the *lpEntryID2* parameter.

lpEntryID2
  Input parameter pointing to the second entry identifier to be compared.

ulFlags
  Reserved; must be zero.

lpulResult
  Output parameter pointing to the result of the comparison. The contents of *lpulResult* is set to TRUE if the two entry identifiers refer to the same object, and FALSE otherwise.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

MAPI_E_UNKNOWN_ENTRYID
  One or both of the entry identifiers passed in with the *lpEntryID1* or *lpEntryID2* parameters are not recognized by any address book provider.

Remarks

Client applications and service providers call the **IAddrBook::CompareEntryIDs** method to compare two entry identifiers to determine whether they refer to the same object. **CompareEntryIDs** is useful because an object can have more than one valid entry identifier; such a situation can occur, for example, after a new version of an address book provider is installed.

MAPI passes this call onto to the address book provider responsible for the entry identifiers, determining the appropriate provider by matching the **MAPIUID** in the entry identifiers with the **MAPIUID** registered by the provider.

If the two entry identifiers refer to the same object, then **CompareEntryIDs** sets the contents of the *lpulResult* parameter to TRUE; if they refer to different objects, **CompareEntryIDs** sets it to FALSE. In either case, **CompareEntryIDs** returns S_OK. If **CompareEntryIDs** returns an error, which can occur if no address book provider has registered a **MAPIUID** that matches the one in the entry identifiers, clients and providers should not take any action based on the result of the comparison. They should instead take the most conservative approach to the action being performed.

IAddrBook::CreateOneOff

The **IAddrBook::CreateOneOff** method associates a one-off entry identifier with a recipient.

HRESULT CreateOneOff(
  LPTSTR *lpszName*,
  LPTSTR *lpszAdrType*,
  LPTSTR *lpszAddress*,
  ULONG *ulFlags*,
  ULONG FAR * *lpcbEntryID*,
  LPENTRYID FAR * *lppEntryID*
)

Parameters

lpszName
Input parameter pointing to a string containing the display name of the recipient. The *lpszName* parameter can be NULL.

lpszAdrType
Input parameter pointing to a string containing the address type of the recipient, such as FAX, SMTP, or X500. The *lpszAdrType* parameter cannot be NULL.

lpszAddress
Input parameter pointing to a string containing the messaging address of the recipient. The *lpszAddress* parameter cannot be NULL.

ulFlags
Input parameter containing a bitmask of flags that provides information about the creation. The following flags can be set:

MAPI_UNICODE
Indicates the passed-in strings are in Unicode format. If the MAPI_UNICODE flag is not set, the strings are in ANSI format.

MAPI_SEND_NO_RICH_INFO
If a client sets this flag, MAPI sets the recipient's PR_SEND_RICH_INFO property to FALSE. If this flag is not set, in most cases MAPI sets PR_SEND_RICH_INFO to TRUE. The one exception is when the recipient's address is interpreted to be an Internet address. In this case, MAPI sets PR_SEND_RICH_INFO to FALSE.

lpcbEntryID
Output parameter pointing to the size, in bytes, of the entry identifier pointed to by the *lppEntryID* parameter.

lppEntryID
Output parameter pointing to a pointer to the newly created entry identifier.

Return Values

S_OK
The one-off entry identifier was created successfully.

Remarks

Clients call the **IAddrBook::CreateOneOff** method to create an entry identifier for a special type of recipient known as a one-off. One-offs are recipients that do not as of yet belong to a specific address book provider. They are created typically with a template that allows the user of a client application to enter the particular information that is associated with this type of address book provider.

The PR_SEND_RICH_INFO flag controls whether or not formatted text in the RTF format is sent along with each message. Although most transport providers by default send messages with TNEF

(Transport Neutral Encapsulation Format), some do not regardless of how the recipient sets its PR_SEND_RICH_INFO property. This is not an issue for messaging clients that work with IPM messages, but because TNEF is typically used to send custom properties for custom message classes, not supporting it can be a problem for form-based clients or clients that require custom MAPI properties.

See Also

IMAPISupport::CreateOneOff

IAddrBook::Details

The **IAddrBook::Details** method displays a modal dialog box showing details about a particular address book entry.

HRESULT Details(
  ULONG FAR * *lpulUIParam*,
  LPFNDISMISS *lpfnDismiss*,
  LPVOID *lpvDismissContext*,
  ULONG *cbEntryID*,
  LPENTRYID *lpEntryID*,
  LPFNBUTTON *lpfButtonCallback*,
  LPVOID *lpvButtonContext*,
  LPTSTR *lpszButtonText*,
  ULONG *ulFlags*
)

Parameters

lpulUIParam
: Output parameter containing the handle of the parent window for the dialog box.

lpfnDismiss
: Input parameter pointing to the address of a function based on the **DISMISSMODELESS** function prototype. This function is called when the modeless variety of the details dialog box is dismissed. However, because MAPI does not support a modeless details dialog box, this parameter is ignored.

lpvDismissContext
: Input parameter containing data that is passed to the function specified by the *lpfnDismiss* parameter. However, because MAPI does not support a modeless details dialog box, this parameter is ignored.

cbEntryID
: Input parameter containing the size, in bytes, of the entry identifier pointed to by the *lpEntryID* parameter.

lpEntryID
: Input parameter pointing to the entry identifier for the object for which details are displayed.

lpfButtonCallback
: Input parameter pointing to a pointer to a button callback function that adds a button to the dialog box. The callback function is based on the **LPFNBUTTON** function prototype.

lpvButtonContext
: Input parameter pointing to data used as a parameter for the button callback function.

lpszButtonText
: Input parameter pointing to a string containing text to be applied to the added button if that button is extensible. The *lpszButtonText* parameter should be NULL if an extensible button is not needed.

ulFlags
: Input parameter containing a bitmask of flags that controls the type of the text for *lpszButtonText*. The following flag can be set:

  MAPI_UNICODE
  : Indicates the passed-in strings are in Unicode format. If the MAPI_UNICODE flag is not set, the strings are in ANSI format.

Return Values

S_OK
: The call succeeded and has returned the expected value or values.

Remarks

Client applications call the **IAddrBook::Details** method to display a modal dialog box giving details on a particular entry in an address book. The *lpfButtonCallback*, *lpvButtonContext*, and *lpButtonText* parameters can be used to add a button the client has defined to the dialog box. When the button is chosen, MAPI calls the callback function pointed to by *lpfButtonCallback*, passing both the entry identifier of the button and the data in *lpvButtonContext*. If an extensible button is not needed, *lpszButtonText* should be NULL. The callback function pointed to by *lpfButtonCallback* is based on the **LPFNBUTTON** function prototype.

See Also

IAddrBook::Address method, **IAddrBook::Details** method, **LPFNBUTTON** function prototype

IAddrBook::GetDefaultDir

The **IAddrBook::GetDefaultDir** method returns the entry identifier for the address book container that is displayed initially to the user.

HRESULT GetDefaultDir(
  ULONG FAR * *lpcbEntryID***,**
  LPENTRYID FAR * *lppEntryID*
)

Parameters

lpcbEntryID
  Output parameter pointing to the size, in bytes, of the entry identifier pointed to by the *lppEntryID* parameter.

lppEntryID
  Output parameter containing the address of a pointer to the entry identifier of the default directory.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

Remarks

Client applications and service providers call the **IAddrBook::GetDefaultDir** method to retrieve the default directory of the address book. The default directory is the directory the user sees displayed in the address book when the address book is first opened. If a default directory has not been set by a call to the **IAddrBook::SetDefaultDir** method, MAPI determines the default directory. MAPI does so by locating the first address book that contains names and that is not the Personal Address Book. If there is no such other address book, then the Personal Address Book is set as the default directory.

Calls made to **GetDefaultDir** return in *lppEntryID* a pointer to the entry identifier of the default directory. MAPI allocates memory for this entry identifier with the **MAPIAllocateBuffer** function, and the calling client or provider must release it with the **MAPIFreeBuffer** function when done.

To set the default directory, a client or provider calls the **IAddrBook::SetDefaultDir** method. Clients and providers need not call the **IMAPIProp::SaveChanges** method to make directory changes permanent.

See Also

IAddrBook::SetDefaultDir method, **MAPIAllocateBuffer** function, **MAPIFreeBuffer** function, PR_CONTAINER_FLAGS property

IAddrBook::GetPAB

The **IAddrBook::GetPAB** method returns the entry identifier of the container designated as the Personal Address Book (PAB).

HRESULT GetPAB(
  ULONG FAR * *lpcbEntryID*,
  LPENTRYID FAR * *lppEntryID*
)

Parameters

lpcbEntryID
  Output parameter pointing to the size, in bytes, of the entry identifier pointed to by the *lppEntryID* parameter.

lppEntryID
  Output parameter containing the address of a pointer to the entry identifier of the Personal Address Book. The *lppEntryID* parameter contains zero if no container has been designated as the Personal Address Book.

Return Values

S_OK
  The call succeeded and has returned the expected value.

Remarks

Client applications call the **IAddrBook::GetPAB** method to retrieve the entry identifier of the container designated as the Personal Address Book. If a Personal Address Book has not been established in the profile, MAPI selects the first container in the address book hierarchy that allows modifications to assume the role.

See Also

MAPIAllocateBuffer function, **MAPIFreeBuffer** function, PR_CONTAINER_FLAGS property

IAddrBook::GetSearchPath

The **IAddrBook::GetSearchPath** method returns an ordered list of entry identifiers of containers to be included in the name resolution process initiated by **IAddrBook::ResolveName**.

HRESULT GetSearchPath(
  ULONG *ulFlags*,
  LPSRowSet FAR * *lppSearchPath*
)

Parameters

ulFlags
  Input parameter containing a bitmask of flags that controls the type of the strings returned in the search path. The following flag can be set:
  MAPI_UNICODE
    Indicates the returned strings are in Unicode format. If the MAPI_UNICODE flag is not set, the strings are in ANSI format.

lppSearchPath
  Output parameter containing the address of a pointer to an ordered list of container entry identifiers. **GetSearchPath** stores the ordered list in an **SRowSet** structure. If there are no containers in the address book hierarchy, zero is returned in the **SRowSet**.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

Remarks

Client applications and service providers call the **IAddrBook::GetSearchPath** method to get the search path that is used to resolve names with **IAddrBook::ResolveName**. Typically clients call the **IAddrBook::SetSearchPath** method to establish a container search path in the profile before they call **GetSearchPath** to retrieve it. However, calling **SetSearchPath** is optional.

If **IAddrBook::SetSearchPath** has never been called, **GetSearchPath** builds a path by working through the address book's hierarchy tables. The order of the default search path established by **GetSearchPath** is as follows:

1. First container with read/write access, usually the Personal Address Book.
2. Every container that has a PR_DISPLAY_TYPE property set to the DT_GLOBAL flag. Such a setting indicates PR_DISPLAY_TYPE contains a global address list that holds recipients.
3. If there is no container with the DT_GLOBAL flag set for its PR_DISPLAY_TYPE property, the default container is added to the path if the default container is not the same as the first container with read/write access.

If **IAddrBook::SetSearchPath** has been called, **GetSearchPath** builds a path using the address book containers that have been stored in the profile. **GetSearchPath** validates this path before returning it to the caller.

After the first call to **SetSearchPath**, subsequent calls to **SetSearchPath** must be used to modify the search path returned by **GetSearchPath**. In other words, the calling client or provider does not receive the default search path after the first call to **SetSearchPath**.

See Also

IAddrBook::SetSearchPath method, **SRowSet** structure

IAddrBook::NewEntry

The **IAddrBook::NewEntry** method displays a dialog box for creating new recipients, either within a container or a message.

HRESULT NewEntry(
  ULONG *ulUIParam*,
  ULONG *ulFlags*,
  ULONG *cbEIDContainer*,
  LPENTRYID *lpEIDContainer*,
  ULONG *cbEIDNewEntryTpl*,
  LPENTRYID *lpEIDNewEntryTpl*,
  ULONG FAR * *lpcbEIDNewEntry*,
  LPENTRYID FAR * *lppEIDNewEntry*
)

Parameters

ulUIParam
  Input parameter containing the handle of the parent window for the dialog box.

ulFlags
  Reserved; must be zero.

cbEIDContainer
  Input parameter containing the size, in bytes, of the entry identifier pointed to by the *lpEIDContainer* parameter.

lpEIDContainer
  Input parameter pointing to the entry identifier of the container where the new recipient is to be added. If the *cbEIDContainer* parameter is zero, the **IAddrBook::NewEntry** method returns a recipient entry identifier and a list of templates as if the **IAddrBook::CreateOneOff** method was called.

cbEIDNewEntryTpl
  Input parameter containing the size, in bytes, of the entry identifier pointed to by the *lpEIDNewEntryTpl* parameter.

lpEIDNewEntryTpl
  Input parameter pointing to a one-off template to be used to create the new recipient. If the *cbEIDNewEntryTpl* parameter is zero, passing NULL in the *lpEIDNewEntryTpl* parameter displays a dialog box enabling the user to select from a list of one-off templates.

lpcbEIDNewEntry
  Output parameter pointing to a variable where is returned the size, in bytes, of the entry identifier pointed to by the *lppEIDNewEntry* parameter.

lppEIDNewEntry
  Output parameter pointing to a pointer to the new recipient's entry identifier.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

Remarks

Client applications and service providers call the **IAddrBook::NewEntry** method to display a dialog box to create a new recipient to be inserted directly into a container or added to the recipient list of an outgoing message.

To create a recipient to be inserted into a modifiable container, a client or provider passes:

The container's entry identifier in the contents of the *lpEIDContainer* parameter

Zero in the *cbEIDContainer*, and NULL for the rest of the parameters.n a modifiable address book and not get its entry identifier back,

To add a custom recipient address directly to the open message and not to a modifiable container, a client or provider passes zero in *cbEIDContainer* and NULL in *lpEIDContainer*. To display a dialog box enabling the user to select a template for adding custom recipients to a modifiable container, the client or provider passes zero in *cbEIDNewEntryTpl* and NULL in *lpEIDNewEntryTpl*.

To open a specific custom-recipient dialog box directly, so that users enter custom recipients in their Personal Address Books using a predetermined template, a client or provider uses the following series of calls. First, it calls the **IAddrBook::OpenEntry** method and passes in either the entry identifier of a modifiable container or zero; passing zero opens the root folder of the address book container. Next, the client or provider calls the **IMAPIProp::OpenProperty** method and passes the PR_CREATE_TEMPLATES property in the *ulPropTag* parameter so it can open PR_CREATE_TEMPLATES. Doing so returns a table object that lists the types of objects that can be created in the address book container. The client or provider finds in this table the entry identifier for the template with which new entries should be created. Then, the client or provider calls **NewEntry** and passes NULL in *lpEIDContainer* and the entry identifier for the entry-creation template to use in the *lpEIDNewEntryTpl* parameter.

Calls made to **NewEntry** return the entry identifier of the new custom recipient address in the *lppEIDNewEntry* parameter, unless NULL was passed in *lppEIDNewEntry*. The calling client or provider is responsible for freeing the returned entry identifier by calling the **MAPIFreeBuffer** function.

See Also

IAddrBook::OpenEntry method, **IMAPIProp::OpenProperty** method, PR_CREATE_TEMPLATES property

IAddrBook::OpenEntry

The **IAddrBook::OpenEntry** method opens a container or recipient and returns a pointer to the object to provide further access. A recipient can be either a messaging user or a distribution list.

HRESULT OpenEntry(
  ULONG *cbEntryID*,
  LPENTRYID *lpEntryID*,
  LPCIID *lpInterface*,
  ULONG *ulFlags*,
  ULONG FAR * *lpulObjType*,
  LPUNKNOWN FAR * *lppUnk*
)

Parameters

cbEntryID
  Input parameter containing the size, in bytes, of the entry identifier pointed to by the *lpEntryID* parameter.

lpEntryID
  Input parameter pointing to the entry identifier for the object to be opened.

lpInterface
  Input parameter pointing to the interface identifier (IID) identifying the interface to be used to access the open object. Passing NULL indicates that the standard interface should be used, such as **IMailUser** for a messaging user and **IDistList** for a distribution list. The *lpInterface* parameter can also be set to an identifier for an appropriate interface for the object.

ulFlags
  Input parameter containing a bitmask of flags that controls how the object is opened. The following flags can be set:

  MAPI_BEST_ACCESS
    Indicates the object should be opened with the maximum network permissions allowed for the user and the maximum client application access. For example, if the client has read/write access, the object is opened with read/write access; if the client has read-only access, the object is opened with read-only access. The client can retrieve the access level by getting the property PR_ACCESS_LEVEL.

  MAPI_DEFERRED_ERRORS
    Indicates the call is allowed to succeed even if the underlying object is not accessible to the calling implementation. If the object is not accessible, some subsequent call to the object might return an error.

  MAPI_MODIFY
    Requests read/write access. By default, objects are created with read-only access, and clients should not work on the assumption that read/write access has been granted.

lpulObjType
  Output parameter pointing to a variable where the type of the opened object is stored.

lppUnk
  Output parameter pointing to a variable where the pointer to the opened object is stored.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

MAPI_E_NO_ACCESS
  An attempt was made to modify a read-only object or an attempt was made to access an object for which the user has insufficient permissions.

MAPI_E_NOT_FOUND
: The object indicated by *lpEntryID* does not exist.

MAPI_E_UNKNOWN_ENTRYID
: The object indicated by the *lpEntryID* parameter is not recognized. This value is typically returned if the address book provider that contains the object is not open.

Remarks

Client applications and service providers call the **IAddrBook::OpenEntry** method to open an address book object. MAPI forwards the call to the appropriate address book provider, based on the **MAPIUID** included in the entry identifier passed in the *lpEntryID* parameter. The address book provider opens the object as read-only unless the MAPI_MODIFY or MAPI_BEST_ACCESS flag in the *ulFlags* parameter is set. However, these flags are suggestions. If the address book provider does not allow modification for the object requested, then it returns MAPI_E_NO_ACCESS.

The *lpInterface* parameter indicates which interface should be used for the opened object. Passing NULL in *lpInterface* indicates the standard MAPI interface for that type of object should be used. Because the address book provider might return a different interface that the one suggested by the *lpInterface* parameter, the caller should check the value returned in the *lpulObjType* parameter to determine that the object type returned is what was expected. Commonly, after the client or provider checks the type of the object, it then casts the pointer in the *lppUnk* parameter into a more appropriate object pointer.

IAddrBook::PrepareRecips

The **IAddrBook::PrepareRecips** method prepares a recipient list for later use by the messaging system.

HRESULT PrepareRecips(
  ULONG *ulFlags***,**
  LPSPropTagArray *lpSPropTagArray***,**
  LPADRLIST *lpRecipList*
)

Parameters

ulFlags
  Reserved; must be zero.

lpSPropTagArray
  Input parameter pointing to an **SPropTagArray** structure containing an array of property tags indicating the properties that require updating, if any. The *lpSPropTagArray* parameter can be NULL.

lpRecipList
  Input parameter pointing to an **ADRLIST** structure holding the list of recipients.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

Remarks

Client applications and service providers call the **IAddrBook::PrepareRecips** method to ensure that all recipients in the *lpRecipList* parameter have long-term entry identifiers and that they have all the properties requested in the *lpSPropTagArray* parameter. Recipients' short-term entry identifiers are converted to long-term entry identifiers. If necessary, recipients' long-term entry identifiers are retrieved from the appropriate address book provider along with any additional properties requested.

Within an individual recipient entry, the requested properties are ordered first, followed by any additional properties that were already present for the entry. If one or more of the requested properties are not handled by the appropriate address book provider, their property types will be set to PT_ERROR and their property values either to MAPI_E_NOT_FOUND or to another value giving a more specific reason why the properties are not available.

Like the **ADRLIST** structure as a whole, each **SPropValue** property value structure passed in *lpSPropTagArray* must be separately allocated using the **MAPIAllocateBuffer** and **MAPIAllocateMore** functions such that it can be freed individually.

See Also

ADRLIST structure, **IMAPIProp::GetProps** method, **IMessage::ModifyRecipients** method, PR_ENTRYID property, PT_ERROR property type, **SPropValue** structure, **SRowSet** structure

IAddrBook::QueryDefaultRecipOpt

The **IAddrBook::QueryDefaultRecipOpt** method returns the available recipient options for a particular address type.

HRESULT QueryDefaultRecipOpt(
  LPTSTR *lpszAdrType***,**
  ULONG *ulFlags***,**
  ULONG FAR * *lpcValues***,**
  LPSPropValue FAR * *lppOptions*
)

Parameters

lpszAdrType
  Input parameter pointing to a string containing the address type for which the options dialog box should be displayed, such as FAX, SMTP, or X500. The *lpszAdrType* parameter must not be NULL.

ulFlags
  Input parameter containing a bitmask of flags that controls the type of the passed-in strings. The following flag can be set:

  MAPI_UNICODE
    Indicates the passed-in strings are in Unicode format. If the MAPI_UNICODE flag is not set, the strings are in ANSI format.

lpcValues
  Output parameter pointing to a variable containing the number of returned property values in the *lppOptions* parameter.

lppOptions
  Output parameter pointing to a pointer to **SPropValue** structures containing available recipient options and their defaults.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

Remarks

Client applications and service providers call the **IAddrBook::QueryDefaultRecipOpt** method to return the recipient options available for a particular messaging address type. *Recipient options* are the properties of a recipient that govern its behavior after a client submits a message including that recipient. Recipient options are usually, but not always, specific to a particular address type.

Clients call the **IAddrBook::QueryDefaultRecipOpt** method to get the set of default recipient options for a particular recipient type. These options are registered by transport providers using the **IXPLogon::RegisterOptions** method. If a client must display a dialog box to enable the user to select recipient options, it should call the **IAddrBook::RecipOptions** method. In addition to being applied to recipients, such options can be applied to an entire message as a default for all the message's recipients; for more information on this functionality, see **IMAPISession::MessageOptions**.

See Also

IAddrBook::RecipOptions method, **IMAPISession::MessageOptions** method, **IMAPISession::QueryDefaultMessageOpt** method

IAddrBook::RecipOptions

The **IAddrBook::RecipOptions** method displays a dialog box enabling a user to change options for a particular recipient.

HRESULT RecipOptions(
  ULONG *ulUIParam***,**
  ULONG *ulFlags***,**
  LPADRENTRY *lpRecip*
)

Parameters

ulUIParam
  Input parameter containing the handle of the parent window for the dialog box.

ulFlags
  Reserved; must be zero.

lpRecip
  Input parameter pointing to the **ADRENTRY** structure for the recipient whose options are to be displayed or set.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

MAPI_W_ERRORS_RETURNED
  The call succeeded overall, but there are no recipient options for this type of recipient. To test for this warning, use the **HR_FAILED** macro. When this warning is returned, the call should be handled as successful.

Remarks

Client applications and service providers call the **IAddrBook::RecipOptions** method to display a dialog box to get settings for recipient options from the user. *Recipient options* are the properties of a recipient governing its behavior after a client submits a message including that recipient. Recipient options are usually, but not always, specific to a particular address type. The **IAddrBook::RecipOptions** method returns a new **ADRENTRY** structure containing the recipient options selected by the user.

To retrieve the set of default recipient options without presenting a dialog box to the user, call the **IAddrBook::QueryDefaultRecipOpt** method instead. If any recipient options are changed on a **RecipOptions** call, MAPI frees the old **SPropValue** property value structures for those options within the recipient's **ADRENTRY** structure and allocates new ones.

The PR_DISPLAY_NAME, PR_ADDRTYPE and PR_ENTRYID properties must be present for any recipient entry. Other useful properties for recipient-option **ADRENTRY** structures are PR_SEARCH_KEY and PR_EMAIL_ADDRESS.

If there are no recipient options available for the address type indicated in the *lpRecip* parameter, the warning MAPI_W_ERRORS_RETURNED is returned, indicating there was an error returned for the **RecipOptions** call. Calling the **IMAPIProp::GetLastError** method returns a text string describing the warning.

In addition to being applied to specific recipients, recipient options can be applied to an entire message, as a default for all the message's recipients; for more information on this functionality, see **IMAPISession::MessageOptions**.

For more information on using the **HR_FAILED** macro, see Using Macros for Error Handling.

See Also

ADRENTRY structure, **IAddrBook::QueryDefaultRecipOpt** method, **IMAPISession::MessageOptions** method, **IMAPISession::QueryDefaultMessageOpt** method, **SPropValue** structure

IAddrBook::ResolveName

The **IAddrBook::ResolveName** method perform name resolution, assigning entry identifiers to recipients in a recipient list.

HRESULT ResolveName(
  ULONG *ulUIParam*,
  ULONG *ulFlags*,
  LPTSTR *lpszNewEntryTitle*,
  LPADRLIST *lpAdrList*
)

Parameters

ulUIParam
: Input parameter containing the handle of the parent window for a dialog box that is shown if necessary and allowed to prompt the user to resolve ambiguity.

ulFlags
: Input parameter containing a bitmask of flags that controls whether a name-resolution dialog box can be displayed. The following flag can be set:

  MAPI_DIALOG
  : Displays a dialog box to prompt the user for additional name-resolution information. If this flag is not set, no dialog box is displayed.

lpszNewEntryTitle
: Input parameter pointing to a string containing control title text; the title is for the control in the name-resolution dialog box that prompts the user to enter a recipient entry. The title string contents vary depending on the entry type. The *lpszNewEntryTitle* parameter can be NULL.

lpAdrList
: Input parameter pointing to an **ADRLIST** structure containing a list of recipient names and associated properties. This **ADRLIST** structure can be the one created by **IAddrBook::Address**.

Return Values

S_OK
: The call succeeded and has returned the expected value or values.

MAPI_E_AMBIGUOUS_RECIP
: The recipient matched more than one entry identifier. Usually, this value is returned when a name-resolution dialog box could not be displayed because the MAPI_DIALOG flag was not set in the *ulFlags* parameter.

MAPI_E_NOT_FOUND
: The name doesn't match any recipients.

Remarks

Client applications and service providers call the **IAddrBook::ResolveName** method to initiate the name resolution process. An unresolved entry is an entry that does not yet have an entry identifier, or PR_ENTRYID property.

ResolveName goes through the following process for each unresolved entry in the address list passed in the *lpAdrList* parameter*:*

1. If the address type of the recipient adheres to the format of an SMTP address (i.e. displayname@address.addresslist), **IAddrBook::ResolveName** assigns it a one-off entry identifier.
2. For each container in the PR_AB_SEARCH_PATH property, **IAddrBook::ResolveName** calls **IABContainer::ResolveNames**. **IABContainer::ResolveNames** tries to match the display name of each unresolved recipient with a display name belonging to one of its entries.

3. If a container does not support **IABContainer::ResolveNames**, **IAddrBook::ResolveName** restricts the container's contents table using a PR_ANR property restriction. This restriction causes the container to perform a "best guess" type of search to locate a matching recipient. All containers must support the PR_ANR property restriction.
4. When a container returns a recipient that matches multiple names, **IAddrBook::ResolveName** displays a dialog box, if the MAPI_DIALOG flag is set, allowing the user to select the correct name.
5. If all of the containers in the PR_AB_SEARCH_PATH property have been called and no match has been found, the recipient remains unresolved.

If one or more recipients are unresolved, **IAddrBook::ResolveName** returns MAPI_E_NOT_FOUND. If one or more recipients had ambiguous resolution that could not be resolved with a dialog box because the MAPI_DIALOG flag was not set, **IAddrBook::ResolveName** returns MAPI_E_AMBIGUOUS_RECIP. When some of the recipients are ambiguous and some cannot be resolved, **IAddrBook::ResolveName** can return either error value.

If a name cannot be resolved, the client can create a one-off recipient with a specially formatted address and entry identifier. For more information about the format of one-off entry identifiers, see About One-Off Entry Identifiers. For more information about the format of one-off addresses, see About One-Off Addresses.

See Also

ADRLIST structure, **IABContainer::ResolveNames** method, **IAddrBook::Address** method

IAddrBook::SetDefaultDir

The **IAddrBook::SetDefaultDir** method establishes a container as the default address book container, the container that the user initially sees.

HRESULT SetDefaultDir(
  ULONG *cbEntryID***,**
  LPENTRYID *lpEntryID*
)

Parameters

cbEntryID
  Input parameter containing the size, in bytes, of the entry identifier pointed to by the *lpEntryID* parameter.

lpEntryID
  Input parameter pointing to the entry identifier of the default address book container.

Return Values

S_OK
  The call succeeded and has returned the expected value.

Remarks

Client applications and service providers call the **IAddrBook::SetDefaultDir** method to set a new default address book container. The default container is the container that the user sees displayed in the address book when the address book is first opened. **SetDefaultDir** sets the default container in the profile, where it is saved. The container remains as the default either until another call to **SetDefaultDir** is made, in the same session or in another session, or the container is removed.

See Also

IAddrBook::GetDefaultDir method, **IAddrBook::GetSearchPath** method, **IMAPISession::Logoff** method, **MAPILogonEx** function

IAddrBook::SetPAB

The **IAddrBook::SetPAB** method designates a particular container to be the Personal Address Book (PAB).

HRESULT SetPAB(
  ULONG *cbEntryID***,**
  LPENTRYID *lpEntryID*
)

Parameters

cbEntryID
  Input parameter containing the size, in bytes, of the entry identifier pointed to by the *lpEntryID* parameter.

lpEntryID
  Input parameter pointing to the entry identifier of the Personal Address Book. If NULL is passed, the **IAddrBook::SetPAB** method returns MAPI_E_INVALID_ENTRYID.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

Remarks

Client applications and service providers call the **IAddrBook::SetPAB** method to designate a particular container to be the Personal Address Book. The Personal Address Book is the container where new entries are added. Information on which container is the Personal Address Book is saved between instances of a session. This functionality means that, after a call to the **IMAPISession::Logoff** method, subsequent calls to the **MAPILogonEx** function during the same session return the same Personal Address Book as previously set, as long as that container still exists.

Clients and providers need not call the **IMAPIProp::SaveChanges** method to make the Personal Address Book change permanent.

See Also

IAddrBook::GetPAB method, **IAddrBook::GetSearchPath** method, PR_CONTAINER_FLAGS property

IAddrBook::SetSearchPath

The **IAddrBook::SetSearchPath** method sets the search path that is used to resolve names with **ResolveNames** methods.

HRESULT SetSearchPath(
  ULONG *ulFlags*,
  LPSRowSet *lpSearchPath*
)

Parameters

ulFlags
  Reserved; must be zero.

lpSearchPath
  Input parameter pointing to the **SRowSet** structure used to hold information about the search path.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

MAPI_E_MISSING_REQUIRED_COLUMN
  The **SRowSet** structure did not contain the PR_ENTRYID property as the first column in the row set, which it must.

Remarks

Client applications and service providers call the **IAddrBook::SetSearchPath** method to save changes made to the container search order that is used to resolve names with **ResolveNames** methods. This search path is saved between instances of a session. This functionality means that, after a call to the **IMAPISession::Logoff** method, subsequent calls to the **MAPILogonEx** function during the same session return the same search path as previously set, as long as that search path still exists.

Clients and providers need not call the **IMAPIProp::SaveChanges** method to make the search path changes permanent.

See Also

IAddrBook::GetDefaultDir method, **IAddrBook::GetPAB** method, **IAddrBook::GetSearchPath** method, PR_CONTAINER_FLAGS property

IAddrBook::Unadvise

The **IAddrBook::Unadvise** method removes an object's registration for notification of address book changes previously established with a call to the **IAddrBook::Advise** method.

HRESULT Unadvise(
  ULONG *ulConnection*
)

Parameters

ulConnection
  Input parameter containing the number of the registration connection returned by a call to **IAddrBook::Advise**.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

Remarks

Client applications call the **IAddrBook::Unadvise** method to release the pointer to the advise sink object passed in the *lpAdviseSink* parameter in the previous call to **IAddrBook::Advise**, thereby canceling a notification registration. As part of discarding the pointer to the advise sink object, the object's **IUnknown::Release** method is called. Generally, **Release** is called during the **Unadvise** call, but if another thread is in the process of calling the **IMAPIAdviseSink::OnNotify** method for the advise sink object, the **Release** call is delayed until the **OnNotify** method returns.

See Also

IAddrBook::Advise method, **IMAPIAdviseSink::OnNotify** method

IAttach : IMAPIProp

The **IAttach** interface has no unique methods of its own; methods inherited from the **IMAPIProp** interface can be called through **IAttach** for use with attachment objects. For more information about using attachment objects, see About Message Attachments.

At a Glance

Specified in header file:	MAPIDEFS.H
Object that supplies this interface:	Attachment object
Corresponding pointer type:	LPATTACH
Implemented by:	Message store providers
Transaction model:	Transacted
Called by:	Client applications

Vtable Order

No unique methods

Required Properties

PR_OBJECT_TYPE	Read-only
PR_ATTACH_METHOD	Read/write
PR_RENDERING_POSITION	Read/write

IDistList : IMAPIContainer

The **IDistList** interface is used to provide access to distribution lists in modifiable address book containers. **IDistList** can create, copy, and delete distribution lists, in addition to performing name resolution. The methods of the **IDistList** interface are identical to those of the **IABContainer** interface and are not redocumented here. For further information on using the methods shown in the **IDistList** vtable section, following, see the reference entries for the parallel methods of **IABContainer**. For more information on working with distribution list objects, see Objects and Interfaces.

At a Glance

Specified in header file:	MAPIDEFS.H
Object that supplies this interface:	Distribution list object
Corresponding pointer type:	LPDISTLIST
Implemented by:	Address book providers
Called by:	Client applications

Vtable Order

CreateEntry	Creates a new distribution list in an address book container that supports modification.
CopyEntries	Copies one or more distribution lists into an address book container that supports modification.
DeleteEntries	Removes one or more distribution lists from an address book container that supports modification.
ResolveNames	Resolves entries in an address book container.

Required Properties

PR_ADDRTYPE	Read/write
PR_DISPLAY_NAME	Read/write
PR_ENTRYID	Read-only
PR_OBJECT_TYPE	Read-only
PR_RECORD_KEY	Read-only

IMailUser : IMAPIProp

The **IMailUser** interface has no unique methods of its own; methods inherited from the **IMAPIProp** interface can be called through **IMailUser** for use with messaging user objects.

At a Glance

Specified in header file:	MAPIDEFS.H
Object that supplies this interface:	Messaging user object
Corresponding pointer type:	LPMAILUSER
Implemented by:	Address book providers
Called by:	Client applications

Vtable Order

No unique methods

Required Properties

PR_ADDRTYPE	Read/write
PR_DISPLAY_NAME	Read/write
PR_DISPLAY_TYPE	Read-only
PR_EMAIL_ADDRESS	Read/write
PR_ENTRYID	Read-only
PR_OBJECT_TYPE	Read-only
PR_RECORD_KEY	Read-only
PR_SEARCH_KEY	Read-only

IMAPIAdviseSink : IUnknown

The **IMAPIAdviseSink** interface is used to implement an advise sink object for handling notification. Typically, a client application has a different advise sink object for each notification registration – that is, for each call to an **Advise** method. For more information, see **IMAPIAdviseSink::OnNotify**.

At a Glance

Specified in header file:	MAPIDEFS.H
Object that supplies this interface:	Advise sink object
Corresponding pointer type:	LPMAPIADVISESINK
Implemented by:	Client applications
Called by:	Service providers and MAPI

Vtable Order

OnNotify	Supplies information about a change of a particular type.

IMAPIAdviseSink::OnNotify

The **IMAPIAdviseSink::OnNotify** method supplies information about one or more events for which the client has registered. Clients register for notification events by calling an **Advise** method. When the event occurs, the **IMAPIAdviseSink::OnNotify** method is called.

ULONG OnNotify(
  ULONG *cNotif*,
  LPNOTIFICATION *lpNotifications*
)

Parameters

cNotif
  Input parameter containing the number of **NOTIFICATION** structures pointed to by the *lpNotifications* parameter.

lpNotifications
  Input parameter pointing to the **NOTIFICATION** structures that describe the events that have occurred.

Return Value

S_OK
  The call succeeded and has returned the expected value or values.

Remarks

Service providers call **Notify** to request that MAPI call the **IMAPIAdviseSink::OnNotify** method of a client's advise sink object when changes occur to an object for which the client has requested notifications.

Each notification describes a separate event. The *lpNotifications* parameter points to one or more **NOTIFICATION** structures. There is a different type of **NOTIFICATION** structure for each type of event. The following table lists the constants that MAPI defines to represent event types and the corresponding notification structures.

Notification event type	Corresponding structure
`fnevCriticalError`	**ERROR_NOTIFICATION**
`fnevNewMail`	**NEWMAIL_NOTIFICATION**
`fnevObjectCreated`	**OBJECT_NOTIFICATION**
`fnevObjectDeleted`	**OBJECT_NOTIFICATION**
`fnevObjectModified`	**OBJECT_NOTIFICATION**
`fnevObjectCopied`	**OBJECT_NOTIFICATION**
`fnevSearchComplete`	**OBJECT_NOTIFICATION**
`fnevTableModified`	**TABLE_NOTIFICATION**
`fnevStatusObjectModified`	**STATUS_OBJECT_NOTIFICATION**
`fnevExtended`	**EXTENDED_NOTIFICATION**

The client should not modify or free the **NOTIFICATION** structure passed to **OnNotify**. The data in the structure is valid only until **OnNotify** returns.

The notifications generated by a single MAPI call, such as a call to the **IMAPIFolder::CopyMessages** method, can be delivered in one or multiple calls to **OnNotify** depending upon the provider implementation and on memory constraints. The notifications generated by multiple MAPI calls can also be combined and delivered in one call to **OnNotify**, depending upon the provider implementation.

As an example of advise sink usage, consider an advise sink associated with a dialog box with which a user browses the contents of a folder in a message store. Commonly, such an advise sink object has internal data structures that reference the dialog box on the screen and that describe the contents of the dialog box. The **OnNotify** method of the advise sink object typically sends a Windows message to the dialog box indicating how it should update itself.

During an **Advise** call, the **IUnknown::AddRef** method is called to update the reference count for the advise sink object. The client retains the notification connection number returned in the *ulConnection* parameter by **Advise**. When the client no longer requires this particular type of notification, it calls the **Unadvise** method, which then calls the **IUnknown::Release** method of the advise sink object. Generally, **Release** is called during the **Unadvise** call, but if another thread is in the process of calling **OnNotify** on the advise sink object, the **Release** call is delayed until the **OnNotify** method returns.

The timing of a call to **OnNotify** depends on a provider's implementation. It can occur during the MAPI call that caused the event, or it can occur at some later time. On systems that support multiple threads of execution, calls to **OnNotify** can occur in a different execution thread than the **Advise** call that registered the notification. For a way to turn a call to **OnNotify** that might happen at an inopportune time into one that is safer to handle, a client should call the **HrThisThreadAdviseSink** function.

For more information about setting up and stopping notifications, see the reference entries for the **Advise** and **Unadvise** methods for any of the following interfaces: **IABLogon**, **IAddrBook**, **IMAPIForm**, **IMAPISession**, **IMAPITable**, **IMsgStore**, and **IMSLogon**. For more general information about the notification process, see About Notification.

See Also

HrAllocAdviseSink function, **HrThisThreadAdviseSink** function, **IMAPISupport::Notify** method, **NOTIFICATION** structure

IMAPIContainer : IMAPIProp

The **IMAPIContainer** interface manages high-level operations on container objects such as address books, distribution lists, and folders. The **IMAPIFolder**, **IABContainer**, and **IDistList** interfaces are derived from **IMAPIContainer**.

At a Glance

Specified in header file:	MAPIDEFS.H
Object that supplies this interface:	Folder, address book container, or distribution list
Corresponding pointer type:	LPMAPICONTAINER
Implemented by:	Message store, address book, and remote transport providers
Transaction model:	Not specified, abstract class
Called by:	Client applications

Vtable Order

GetContentsTable	Returns a pointer to the container's contents table.
GetHierarchyTable	Returns a pointer to the container's hierarchy table.
OpenEntry	Opens an object in the container and returns a pointer to an interface for further access.
SetSearchCriteria	Sets the search criteria for a particular search-results folder within the container.
GetSearchCriteria	Obtains the search criteria for a particular search-results folder within the container.

Required Properties

PR_CONTAINER_HIERARCHY	Read-only
PR_CONTAINER_CONTENTS	Read-only
PR_CONTAINER_FLAGS	Read/write

IMAPIContainer::GetContentsTable

The **IMAPIContainer::GetContentsTable** method returns a pointer to the container's contents table.

HRESULT GetContentsTable(
  ULONG *ulFlags***,**
  LPMAPITABLE FAR * *lppTable*
)

Parameters

ulFlags
  Input parameter containing a bitmask of flags that controls how the contents table is returned. The following flags can be set:

  MAPI_ASSOCIATED
    Indicates the service provider should return the associated contents table rather than the standard contents table. This flag is used only with folders. The messages that are included in the associated contents table are created with the MAPI_ASSOCIATED flag set. Client applications can use the associated contents table to retrieve forms and views.

  MAPI_DEFERRED_ERRORS
    Indicates the call is allowed to succeed even if the underlying object is not accessible to the calling client. If the object is not accessible, some subsequent call to the object might return an error.

  MAPI_UNICODE
    Indicates the returned strings are in Unicode format. If the MAPI_UNICODE flag is not set, the strings are in ANSI format.

lppTable
  Output parameter pointing to a pointer to the contents table.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

MAPI_E_BAD_CHARWIDTH
  Either the MAPI_UNICODE flag was set and the implementation does not support Unicode, or MAPI_UNICODE was not set and the implementation only supports Unicode.

MAPI_E_NO_SUPPORT
  The container has no contents and cannot provide a contents table.

Remarks

Use the **IMAPIContainer::GetContentsTable** method to get a pointer to the **IMAPITable** implementation for the contents table of a container. A contents table contains summary information about objects within the container. The set of columns included in the table are always the default set of columns for a contents table.

Address book container contents tables typically contain the following columns:

PR_ADDRTYPE
PR_DISPLAY_NAME
PR_DISPLAY_TYPE
PR_ENTRYID
PR_INSTANCE_KEY
PR_OBJECT_TYPE
PR_RECORD_KEY

Folder contents tables typically contain the following columns:

PR_CLIENT_SUBMIT_TIME	PR_DISPLAY_TO
PR_ENTRYID	PR_HASATTACH
PR_INSTANCE_KEY	PR_LAST_MODIFICATION_TIME
PR_MAPPING_SIGNATURE	PR_MESSAGE_CLASS
PR_MESSAGE_DELIVERY_TIME	PR_MESSAGE_FLAGS
PR_MESSAGE_SIZE	PR_MSG_STATUS
PR_NORMALIZED_SUBJECT	PR_OBJECT_TYPE
PR_PARENT_ENTRYID	PR_PRIORITY
PR_RECORD_KEY	PR_SENDER_NAME
PR_SENSITIVITY	PR_STORE_ENTRYID
PR_STORE_RECORD_KEY	PR_SUBJECT

Providers that implement **GetContentsTable** for their containers must also:

- Support calls to the containers' **IMAPIProp::OpenProperty** method for the PR_CONTAINER_CONTENTS property.
- Return PR_CONTAINER_CONTENTS from calls to the containers' **IMAPIProp::GetProps** or **IMAPIProp::GetPropList** method.

The string and binary columns of contents tables can be truncated. Typically providers return 255 characters. Because clients cannot know ahead of time whether or not a table they are using includes truncated columns, they should assume that a column is truncated if the length of the column is either 255 or 510 bytes. Clients can retrieve the full value of a truncated column if necessary from the object directly by callings the object's **IMAPIProp::GetProps** method.

Depending on the provider's implementation, restrictions and sorting operations can apply to an entire string or to the truncated version of that string.

Setting the MAPI_UNICODE flag in the *ulFlags* parameter does the following:

- Sets the string type to Unicode for data returned for the initial active columns of the contents table by the **IMAPITable::QueryColumns** method. The initial active columns for a contents table are those columns **QueryColumns** returns before the provider that contains the table calls the **IMAPITable::SetColumns** method.
- Sets the string type to Unicode for data returned for the initial active rows of the contents table by the **IMAPITable::QueryRows** method. The initial active rows for a contents table are those rows **QueryRows** returns before the provider that contains the table calls **SetColumns**.
- Controls the property types of the sort order returned by the **IMAPITable::QuerySortOrder** method before the provider that contains the contents table calls the **IMAPITable::SortTable** method.

See Also

IMAPIProp::GetPropList method, **IMAPIProp::GetProps** method, **IMAPIProp::OpenProperty** method, **IMAPITable : IUnknown** interface, PR_CONTAINER_CONTENTS property

IMAPIContainer::GetHierarchyTable

The **IMAPIContainer::GetHierarchyTable** method returns a pointer to the container's hierarchy table.

HRESULT GetHierarchyTable(
  ULONG *ulFlags*,
  LPMAPITABLE FAR * *lppTable*
)

Parameters

ulFlags
  Input parameter containing a bitmask of flags that controls how information is returned from the table. The following flags can be set:

  CONVENIENT_DEPTH
    Fills the hierarchy table with containers from one or more levels. The PR_DEPTH property in each table row indicates the depth, relative to the container, of the given child container. The container's immediate child containers are at depth zero. Child containers within the zero depth child containers are at depth one and so on. If the flag CONVENIENT_DEPTH is not set, the hierarchy table contains only the container's immediate child containers.

  MAPI_DEFERRED_ERRORS
    Indicates the call is allowed to succeed even if the underlying object is not accessible to the calling client application. If the object is not accessible, some subsequent call to the object might return an error.

  MAPI_UNICODE
    Indicates the returned strings are in Unicode format. If the MAPI_UNICODE flag is not set, the strings are in ANSI format.

lppTable
  Output parameter pointing to a pointer to the hierarchy table.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

MAPI_E_BAD_CHARWIDTH
  Either the MAPI_UNICODE flag was set and the implementation does not support Unicode, or MAPI_UNICODE was not set and the implementation only supports Unicode.

MAPI_E_NO_SUPPORT
  The container has no child containers and does not support a hierarchy table.

Remarks

Use the **IMAPIContainer::GetHierarchyTable** method to get a pointer to the **IMAPITable** interface implementation of a container's hierarchy table. A hierarchy table holds summary information about the child containers within the container. Folder hierarchy tables hold information about child folders; address book hierarchy tables hold information about child address book containers and distribution lists.

Address book hierarchy tables include the following columns:

PR_CONTAINER_FLAGS	PR_DEPTH
PR_DISPLAY_NAME	PR_DISPLAY_TYPE
PR_ENTRYID	PR_INSTANCE_KEY
PR_OBJECT_TYPE	PR_AB_PROVIDER_ID

Folder hierarchy tables include the following columns:

PR_DEPTH	PR_DISPLAY_NAME
PR_COMMENT	PR_ENTRYID
PR_INSTANCE_KEY	PR_STATUS
PR_SUBFOLDERS	PR_FOLDER_TYPE
PR_SUBJECT	

Providers that implement **GetHierarchyTable** for their containers must also:

- Support calls to the containers' **IMAPIProp::OpenProperty** method for the PR_CONTAINER_HIERARCHY property.
- Return PR_CONTAINER_HIERARCHY from calls to the containers' **IMAPIProp::GetProps** or **IMAPIProp::GetPropList** method.

The string and binary columns of hierarchy tables can be truncated. Typically providers return 255 characters. Because clients cannot know ahead of time whether or not a table they are using includes truncated columns, they should assume that a column is truncated if the length of the column is either 255 or 510 bytes. Clients can retrieve the full value of a truncated column if necessary from the object directly by callings the object's **IMAPIProp::GetProps** method.

Depending on a provider's implementation, restrictions and sorting operations can apply to an entire string or to the truncated version of that string.

Setting the MAPI_UNICODE flag in the *ulFlags* parameter does the following:

- Sets the string type to Unicode for data returned for the initial active columns of the hierarchy table by **IMAPITable::QueryColumns**.
- Sets the string type to Unicode for data returned for the initial active rows of the hierarchy table by the **IMAPITable::QueryRows** method. The initial active rows for a hierarchy table are those rows **QueryRows** returns before the provider that contains the table calls **IMAPITable::SetColumns**.
- Controls the property types of the sort order returned by the **IMAPITable::QuerySortOrder** method before the provider that contains the hierarchy table calls **IMAPITable::SortTable**.

See Also

IMAPIProp::GetPropList method, **IMAPIProp::GetProps** method, **IMAPITable : IUnknown** interface, PR_CONTAINER_HIERARCHY property

IMAPIContainer::GetSearchCriteria

The **IMAPIContainer::GetSearchCriteria** method obtains the search criteria for the container.

HRESULT GetSearchCriteria(
  ULONG *ulFlags*,
  LPSRestriction FAR * *lppRestriction*,
  LPENTRYLIST FAR * *lppContainerList*,
  ULONG FAR * *lpulSearchState*
)

Parameters

ulFlags
Input parameter containing a bitmask of flags that controls the type of the passed-in strings. The following flag can be set:

MAPI_UNICODE
Indicates the passed-in strings are in Unicode format. If the MAPI_UNICODE flag is not set, the strings are in ANSI format.

lppRestriction
Output parameter pointing to a pointer to an **SRestriction** structure defining the search criteria. If a client application passes NULL in the *lppRestriction* parameter, **IMAPIContainer::GetSearchCriteria** does not return an **SRestriction** structure.

lppContainerList
Output parameter pointing to a pointer to an array of entry identifiers representing containers to be included in the search. If a client passes NULL in the *lppContainerList* parameter, **GetSearchCriteria** does not return an array of entry identifiers.

lpulSearchState
Output parameter containing a pointer to a bitmask of flags used to indicate the current state of the search. If a client passes NULL in the *lpulSearchState* parameter, **GetSearchCriteria** does not return any flags. The following flags can be set for *lpulSearchState:*

SEARCH_FOREGROUND
Indicates the search runs at high priority relative to other searches. If this flag is not set, the search runs at normal priority relative to other searches.

SEARCH_REBUILD
Indicates the search is in the CPU-intensive mode of its operation, attempting to bring in all current messages that match the criteria. If this flag is not set, the CPU-intensive part of the search's operation is over. This flag only has meaning if the search is active (that is, if the SEARCH_RUNNING flag is set).

SEARCH_RECURSIVE
Indicates the search looks in specified containers and all of their child containers for matching entries. If this flag is not set, only the containers explicitly included in the last call to the **IMAPIContainer::SetSearchCriteria** method are searched.

SEARCH_RUNNING
Indicates that the search is active and that the container's contents table is being updated to reflect changes in the message store. If this flag is not set, the search is inactive and the contents table is static.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

MAPI_E_BAD_CHARWIDTH
Either the MAPI_UNICODE flag was set and the implementation does not support Unicode, or

MAPI_UNICODE was not set and the implementation only supports Unicode.

MAPI_E_NOT_INITIALIZED

The client did not set any search criteria before **GetSearchCriteria** was called.

Remarks

Use the **IMAPIContainer::GetSearchCriteria** method to obtain the search criteria for a particular container that supports searches. Search criteria is created using by calling a container's **IMAPIContainer::SetSearchCriteria** method. **GetSearchCriteria** is primarily used with search-results folders, created when clients set the FOLDER_SEARCH flag on a call to **IMAPIFolder::CreateFolder** call. Address book providers only need to implement **GetSearchCriteria** if they provide the advanced search capabilities associated with the PR_SEARCH property. For more information on implementing the advanced search feature for address book containers, see Implementing Advanced Searching.

When a client is done with the data structures returned by **GetSearchCriteria** in *lppRestriction* and *lppContainerList*, it should call the **MAPIFreeBuffer** function to release the structures – one call to release each structure.

See Also

IMAPIContainer::SetSearchCriteria method, **IMAPIFolder::CreateFolder** method, **MAPIFreeBuffer** function, PR_SEARCH property

IMAPIContainer::OpenEntry

The **IMAPIContainer::OpenEntry** method opens an object within the container, returning an interface pointer for further access.

HRESULT OpenEntry(
  ULONG *cbEntryID***,**
  LPENTRYID *lpEntryID***,**
  LPCIID *lpInterface***,**
  ULONG *ulFlags***,**
  ULONG FAR * *lpulObjType***,**
  LPUNKNOWN FAR * *lppUnk*
)

Parameters

cbEntryID
  Input parameter containing the size, in bytes, of the entry identifier pointed to by the *lpEntryID* parameter.

lpEntryID
  Input parameter pointing to the entry identifier of the object to open.

lpInterface
  Input parameter pointing to the interface identifier (IID) representing the interface to be used for further access to the object. Passing NULL results in the container returning the standard interface for the object, such as **IMessage** for a message or **IMAPIFolder** for a folder.

ulFlags
  Input parameter containing a bitmask of flags that controls how the object is opened. The following flags can be used:

  MAPI_BEST_ACCESS
    Indicates the object should be opened with the maximum network permissions allowed for the user and the maximum client application access. For example, if the client has read/write access, the object is opened with read/write access; if the client has read-only access, the object is opened with read-only access. The client can retrieve the access level by getting the property PR_ACCESS_LEVEL.

  MAPI_DEFERRED_ERRORS
    Indicates the call is allowed to succeed even if the underlying object is not accessible to the calling client. If the object is not accessible, some subsequent call to the object might return an error.

  MAPI_MODIFY
    Requests read/write access. By default, objects are created with read-only access, and clients should not work on the assumption that read/write access has been granted.

lpulObjType
  Output parameter pointing to the opened object's type.

lppUnk
  Output parameter pointing to a pointer to the opened object.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

MAPI_E_NO_ACCESS
  An attempt was made to modify a read-only object or an attempt was made to access an object for which the user has insufficient permissions.

MAPI_E_NOT_FOUND

The object indicated by *lpEntryID* does not exist.

MAPI_E_UNKNOWN_ENTRYID
The entry identifier in the *lpEntryID* parameter is not of a format recognized by the container.

Remarks

Use the **IMAPIContainer::OpenEntry** method to open one of the objects within a container and obtain a pointer to an interface implementation to use for further.

Service providers by default open objects with read-only access unless the either the MAPI_MODIFY or MAPI_BEST_ACCESS flag is set in the *ulFlags* parameter. When one of these flags is set, service providers attempt to return a modifiable object. If the provider does not allow modification for the object requested, then it returns the value MAPI_E_NO_ACCESS.

If a client passes NULL for *lpEntryID*, the provider opens the top-level container in the container's hierarchy. Clients should check the value in the *lpulObjType* parameter to verify the returned object has the expected object type. If necessary, the client should cast the pointer returned in the *lppUnk* parameter to a pointer of the appropriate type.

IMAPIContainer::SetSearchCriteria

The **IMAPIContainer::SetSearchCriteria** method sets the search criteria for the container.

HRESULT SetSearchCriteria(
  LPSRestriction *lpRestriction*,
  LPENTRYLIST *lpContainerList*,
  ULONG *ulSearchFlags*
)

Parameters

lpRestriction
: Input parameter pointing to an **SRestriction** structure defining the search criteria. If a client application passes NULL in the *lpRestriction* parameter, then the search criteria used most recently for this container are used again. A client should not pass NULL in *lpRestriction* for the first search within the container.

lpContainerList
: Input parameter pointing to an array of entry identifiers representing containers to be included in the search. If a client passes NULL in the *lpContainerList* parameter, then the entry identifiers used most recently for this container are included in the new search. A client should not pass NULL in *lpContainerList* for the first search within a container.

ulSearchFlags
: Input parameter containing a bitmask of flags that controls how the search is performed. The following flags can be set:

BACKGROUND_SEARCH
: Indicates the search runs at normal priority relative to other searches. This flag cannot be set at the same time as the FOREGROUND_SEARCH flag.

FOREGROUND_SEARCH
: Indicates the search runs at high priority relative to other searches. This flag cannot be set at the same time as the BACKGROUND_SEARCH flag.

RECURSIVE_SEARCH
: Searches specified containers and all of their their child containers. This flag cannot be set at the same time as the SHALLOW_SEARCH flag.

RESTART_SEARCH
: Indicates a search that is inactive should be restarted. In addition to being set to restart a search, this flag must be passed on the first call to the **IMAPIContainer::SetSearchCriteria** method in order to initiate a search. This flag cannot be set at the same time as the STOP_SEARCH flag.

SHALLOW_SEARCH
: Indicates the search only looks in the specified containers for matching entries. This flag cannot be set at the same time as the RECURSIVE_SEARCH flag.

STOP_SEARCH
: Stops an ongoing search, if any. This flag cannot be set at the same time as the RESTART_SEARCH flag.

Return Values

S_OK
: The call succeeded and has returned the expected value or values.

Remarks

Use the **IMAPIContainer::SetSearchCriteria** method to set the search criteria for a particular container that supports searches. **SetSearchCriteria** can only be used with search-results folders, created when clients set the FOLDER_SEARCH flag on a call to **IMAPIFolder::CreateFolder** call. Address book providers establish search critieria by applying restrictions to the container's contents table. See Implementing Advanced Searching for more information about setting search criteria on address book containers.

A client accomplishes a search by making the following calls:

1. **IMAPIFolder::CreateFolder** to create the search-results folder. The FOLDER_SEARCH flag must be set.
2. **IMAPIContainer::SetSearchCriteria** to define search criteria for the created search-results folder.
3. **IMAPIContainer::GetContentsTable** to browse the results of the search.

A search-results folder only contains links to the messages that meet the search criteria; the actual messages are still stored in their original locations. The only unique data contained in the search-results folder is a contents table view consisting of the merged contents of the message store after the search restriction has been applied. A search operation only works on this merged contents table; it does not search through other search-results folders. The search results only return the messages that match the search criteria; the folder hierarchy is not returned.

To include message attachments or recipients in a search, clients use a subobject restriction that includes a property restriction for either the PR_MESSAGE_RECIPIENTS or PR_MESSAGE_ATTACHMENTS property. Clients can use subobject restrictions to locate, for example, messages containing file attachments with the extension .MSS. The appropriate **SSubRestriction** structure would include PR_MESSAGE_ATTACHMENTS for the **ulSubObject** member and an **SPropertyRestriction** structure for the **lpRes** member that specifies the following property restriction `PR_ATTACH_EXTENSION = MSS`. Service providers that do not support subobject restrictions return MAPI_E_TOO_COMPLEX.

If a client sets the FOREGROUND_SEARCH flag in the *ulSearchFlags* parameter, the provider performs the indicated search at high priority, possibly degrading the client's performance. If the client does not set FOREGROUND_SEARCH, the provider performs the search in the background. In either case, MAPI immediately returns control to the calling client after finishing the search.

A client can use **SetSearchCriteria** to change the search criteria of a search already in progress – it can specify new restrictions, new lists of folders to search, and a new search priority (for example, a background search can be made high priority). Changes in search priority do not cause an existing search to restart, but other changes to search criteria can.

When a client is through using a search-results folder, it can delete the folder or leave it for later use. If a client does delete the search-results folder, the provider only deletes the message links contained in the folder; it does not delete any messages from their parent folders.

Providers should support open, copy, move, and delete operations on the items within their search-results folders. For more information on supporting these operations, see the **IMAPIContainer::OpenEntry** method, the **IMAPIFolder: IMAPIContainer** interface, and About Search-Results Folders. These operations apply to items within the search-results folder, not the search-results folder itself. Items cannot be created within, or copied into, search-results folders.

See Also

IMAPIContainer::GetContentsTable method, **IMAPIContainer::OpenEntry** method, **IMAPIFolder::CreateFolder** method, **IMAPIFolder : IMAPIContainer** interface, **SPropertyRestriction** structure, **SRestriction** structure, **SSubRestriction** structure

IMAPIControl : IUnknown

The **IMAPIControl** interface is used to enable and disable a button and perform tasks when a button is enabled and selected. The **IMAPIControl** interface is used to control custom buttons on dialog boxes that are defined with display tables, such as configuration property sheets.

For more information on working with display tables and control objects, see About Display Tables.

At a Glance

Specified in header file:	MAPIDEFS.H
Object that supplies this interface:	Control object
Corresponding pointer type:	LPMAPICONTROL
Implemented by:	Service providers
Called by:	MAPI

Vtable Order

GetLastError	Returns a **MAPIERROR** structure containing information about the last error that occurred for a control object.
Activate	Performs a task in response to the button being pressed.
GetState	Retrieves a value that indicates whether a button is enabled or disabled.

IMAPIControl::Activate

The **IMAPIControl::Activate** method performs a task, such as displaying a dialog box or starting a programmatic operation, when an enabled button is selected by a user of a client application.

HRESULT Activate(
  ULONG *ulFlags***,**
  ULONG *ulUIParam*
)

Parameters

ulFlags
  Reserved; must be zero.

ulUIParam
  Input parameter containing the handle of the parent window for the dialog box.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

Remarks

Service providers implement **IMAPIControl::Activate** to handle the processing involved when a user selects an enabled button. MAPI calls **Activate** only when the button is enabled after it has been selected. In the *ulUIParam* parameter, MAPI passes the handle for the parent window for the dialog box or property sheet. MAPI calls **IMAPIControl::GetState** to determine if the button is enabled before making the call to **Activate**.

See Also

IMAPIControl::GetState method

IMAPIControl::GetLastError

The **IMAPIControl::GetLastError** method returns a **MAPIERROR** structure containing information about the last error that occurred for a control object.

HRESULT GetLastError(
  HRESULT *hResult***,**
  ULONG *ulFlags***,**
  LPMAPIERROR FAR * *lppMAPIError*
)

Parameters

hResult

Input parameter containing the result returned for the last call for the control object that returned an error.

ulFlags

Input parameter containing a bitmask of flags that controls the type of the returned strings. The following flag can be set:

MAPI_UNICODE

Indicates the strings in the **MAPIERROR** structure returned in the *lppMAPIError* parameter are in Unicode format. If the MAPI_UNICODE flag is not set, the strings are in ANSI format.

lppMAPIError

Output parameter pointing to a pointer to the returned **MAPIERROR** structure containing version, component, and context information for the error. The *lppMAPIError* parameter can be set to NULL if there is no **MAPIERROR** structure to return.

Return Values

S_OK

The call succeeded and has returned the expected value or values.

MAPI_E_BAD_CHARWIDTH

Either the MAPI_UNICODE flag was set and the implementation does not support Unicode, or MAPI_UNICODE was not set and the implementation only supports Unicode.

Remarks

Use the **IMAPIControl::GetLastError** method to retrieve information to display in a message to the user regarding the last error returned from a method call for the control object.

To release all the memory allocated by MAPI, clients need only call the **MAPIFreeBuffer** function for the returned **MAPIERROR** structure.

The return value from **GetLastError** must be S_OK for an implementation to make use of the **MAPIERROR** structure. Even if the return value is S_OK, a **MAPIERROR** structure might not be returned. If the implementation cannot determine what the last error was, or if a **MAPIERROR** structure is not available for that error, **GetLastError** returns a pointer to NULL in *lppMAPIError* instead.

See Also

MAPIERROR structure, **MAPIFreeBuffer** function

IMAPIControl::GetState

The **IMAPIControl::GetState** method retrieves a value indicating whether the button is disabled or enabled.

HRESULT GetState(
  ULONG *ulFlags***,**
  ULONG FAR * *lpulState*
)

Parameters

ulFlags
  Reserved; must be zero.

lpulState
  Output parameter containing a value that indicates the state of the button control. One of the following values can be set:

  MAPI_DISABLED
    Indicates the button is disabled and cannot be clicked.

  MAPI_ENABLED
    Indicates the button is enabled and can be clicked.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

Remarks

MAPI calls the **IMAPIControl::GetState** method to retrieve the state of a button. If enabled, the button can respond to a mouse click or key press. If disabled, the button is displayed as grayed and is incapable of responding to a mouse click or key press.

See Also

IMAPIControl::Activate method

IMAPIFolder : IMAPIContainer

The **IMAPIFolder** interface is used to perform operations on the contents of a folder such as creating, copying, and deleting its messages and subfolders.

At a Glance

Specified in header file:	MAPIDEFS.H
Object that supplies this interface:	Folder object
Corresponding pointer type:	LPMAPIFOLDER
Implemented by:	Message store providers
Transaction model:	Non-transacted
Called by:	Client applications

Vtable Order

CreateMessage	Creates a new message within the folder.
CopyMessages	Copies or moves one or more of the folder's messages.
DeleteMessages	Deletes one or more of the folder's messages.
CreateFolder	Creates a new subfolder within the folder.
CopyFolder	Copies or moves one of the folder's subfolders.
DeleteFolder	Deletes one of the folder's subfolders.
SetReadFlags	Sets or clears the read flags for the messages in the folder.
GetMessageStatus	Obtains the status associated with a message in the folder.
SetMessageStatus	Sets the status associated with a message in the folder.
SaveContentsSort	Sets the default sort order for the folder's contents table.
EmptyFolder	Deletes all items from the folder without deleting the folder itself.

Required Properties

PR_DISPLAY_NAME	Read/write
PR_ENTRYID	Read-only
PR_FOLDER_TYPE	Read/write
PR_OBJECT_TYPE	Read-only
PR_PARENT_ENTRYID	Read-only
PR_RECORD_KEY	Read-only
PR_STORE_ENTRYID	Read-only
PR_STORE_RECORD_KEY	Read-only

IMAPIFolder::CopyFolder

The **IMAPIFolder::CopyFolder** method copies or moves one of the folder's subfolders.

HRESULT CopyFolder(
  ULONG *cbEntryID*,
  LPENTRYID *lpEntryID*,
  LPCIID *lpInterface*,
  LPVOID *lpDestFolder*,
  LPTSTR *lpszNewFolderName*,
  ULONG *ulUIParam*,
  LPMAPIPROGRESS *lpProgress*,
  ULONG *ulFlags*
)

Parameters

cbEntryID
Input parameter containing the size, in bytes, of the entry identifier pointed to by the *lpEntryID* parameter.

lpEntryID
Input parameter pointing to the entry identifier of the subfolder to copy or move.

lpInterface
Input parameter pointing to the interface identifier (IID) representing the interface to be used to access the destination folder indicated by the *lpDestFolder* parameter. Passing NULL results in the service provider returning the standard folder interface, **IMAPIFolder**. Clients must pass NULL. Service providers and MAPI can set the *lpInterface* parameter IID_IUnknown, IID_IMAPIProp, IID_IMAPIContainer, or IID_IMAPIFolder.

lpDestFolder
Input parameter pointing to the open destination folder where the folder identified in the *lpEntryID* parameter is to be copied or moved.

lpszNewFolderName
Input parameter pointing to the name of the newly created or moved folder. If the client passes NULL in the *lpszNewFolderName* parameter, the name of the newly created or moved folder is the same as that of the original.

ulUIParam
Input parameter containing the handle of the parent window for any dialog boxes or windows that this method displays. The *ulUIParam* parameter is ignored unless the client sets the FOLDER_DIALOG flag in the *ulFlags* parameter and passes NULL in the *lpProgress* parameter.

lpProgress
Input parameter pointing to a progress object for displaying a progress indicator. If NULL is passed in *lpProgress*, the service provider displays a progress indicator using MAPI's progress object implementation. The *lpProgress* parameter is ignored unless the FOLDER_DIALOG flag is set in *ulFlags*.

ulFlags
Input parameter containing a bitmask of flags that controls how the copy or move operation is accomplished. The following flags can be set:

COPY_SUBFOLDERS
Indicates all subfolders are included in the copy operation. This functionality is optional for copy operations and is implied for move operations.

FOLDER_DIALOG
Displays a progress indicator while the operation proceeds.

FOLDER_MOVE

Indicates that the folder is to be moved rather than copied. If this flag is not set, the folder is copied.

MAPI_DECLINE_OK
: Informs the message store provider that if it implements **IMAPIFolder::CopyFolder** by calling its support object's **IMAPISupport::DoCopyTo** or **IMAPISupport::DoCopyProps** method, it should immediately return MAPI_E_DECLINE_COPY.

MAPI_UNICODE
: Indicates the passed-in strings are in Unicode format. If the MAPI_UNICODE flag is not set, the strings are in ANSI format.

Return Values

S_OK
: The call succeeded and has returned the expected value or values.

MAPI_E_BAD_CHARWIDTH
: Either the MAPI_UNICODE flag was set and the implementation does not support Unicode, or MAPI_UNICODE was not set and the implementation only supports Unicode.

MAPI_E_COLLISION
: The name of the folder being moved or copied is the same as that of a subfolder in the destination folder. Folder names must be unique.

MAPI_E_DECLINE_COPY
: The provider implements this method by calling a support object method and the caller has passed the MAPI_DECLINE_OK flag.

MAPI_E_FOLDER_CYCLE
: The source object directly or indirectly contains the destination object. Significant work might have been performed before this condition was discovered so the source and destination object might be partially modified.

MAPI_W_PARTIAL_COMPLETION
: The call succeeded, but not all entries were successfully copied. To test for this warning, use the **HR_FAILED** macro. When this warning is returned, the call should be handled as successful.

Remarks

Message store providers implement the **IMAPIFolder::CopyFolder** method to copy or move folders from one location to another. The folder being copied or moved is added to the destination folder as a subfolder. Only one folder can be copied or moved at a time.

CopyFolder allows simultaneous renaming and moving of folders and the copying or moving of subfolders of the affected folder. To copy or move all subfolders nested within the copied or moved folder, a client passes the COPY_SUBFOLDERS flag in *ulFlags*.

In copy or move operations involving more than one folder, even if one or more folders specified do not exist or have already been moved elsewhere, a message store provider should complete the operation as best it can for each folder specified. The provider should stop the operation without completing it only in the case of failures it cannot control, such as running out of memory or disk space, message store corruption, and so on.

If **CopyFolder** successfully completes the copy or move operation for every folder requested by the client, it returns S_OK. If one or more folders cannot be copied or moved, **CopyFolder** returns MAPI_W_PARTIAL_COMPLETION. If **CopyFolder** returns a different value, such as MAPI_E_NOT_ENOUGH_MEMORY, that indicates the call did not complete, it might already have copied or moved one or more folders without being able to continue. The calling client cannot proceed on the assumption that an error return implies no work was done.

If an entry identifier for a folder that doesn't exist is passed in *lpEntryID*, **CopyFolder** returns MAPI_W_PARTIAL_COMPLETION or MAPI_E_NOT_FOUND, depending on the message store's implementation.

For more information on using the **HR_FAILED** macro, see Using Macros for Error Handling.

IMAPIFolder::CopyMessages

The **IMAPIFolder::CopyMessages** method copies or moves one or more of the folder's messages.

HRESULT CopyMessages(
  LPENTRYLIST *lpMsgList*,
  LPCIID *lpInterface*,
  LPVOID *lpDestFolder*,
  ULONG *ulUIParam*,
  LPMAPIPROGRESS *lpProgress*,
  ULONG *ulFlags*
)

Parameters

lpMsgList
  Input parameter pointing to an array of **ENTRYLIST** structures that identify the message or messages to copy or move.

lpInterface
  Input parameter pointing to the interface identifier (IID) for the destination folder object indicated in the *lpDestFolder* parameter. Passing NULL indicates the destination folder object is cast to the standard interface for a folder object. A client application must pass NULL. A message store provider can also set the *lpInterface* parameter to an identifier for an appropriate interface for the destination folder object. For example, a message can be copied with *lpInterface* set to IID_IUnknown or IID_IMAPIProp.

lpDestFolder
  Input parameter pointing to the open destination folder where the message or messages identified in the *lpMsgList* parameter are copied or moved.

ulUIParam
  Input parameter containing the handle of the parent window for any dialog boxes or windows this method displays. The *ulUIParam* parameter is ignored unless the client sets the MESSAGE_DIALOG flag in the *ulFlags* parameter and passes NULL in the *lpProgress* parameter.

lpProgress
  Input parameter pointing to a progress object that contains client-supplied progress information. If NULL is passed in *lpProgress*, MAPI provides the progress information. The *lpProgress* parameter is ignored unless MESSAGE_DIALOG is set in *ulFlags*.

ulFlags
  Input parameter containing a bitmask of flags that controls how the copy or move operation is accomplished. The following flags can be set:

  MAPI_DECLINE_OK
    Informs the provider that if it does not implement the **IMAPIFolder::CopyMessage** method, it can immediately return MAPI_E_DECLINE_COPY.

  MESSAGE_DIALOG
    Displays a progress indicator as the operation proceeds.

  MESSAGE_MOVE
    Moves messages. If this flag is not set, **CopyMessage** copies messages.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

MAPI_E_DECLINE_COPY
  The provider has not implemented this operation.

MAPI_W_PARTIAL_COMPLETION

The call succeeded, but not all entries were successfully copied. To test for this warning, use the **HR_FAILED** macro. When this warning is returned, the call should be handled as successful.

Remarks

Message store providers implement the **IMAPIFolder::CopyMessages** method to copy or move messages from one folder to another. A provider can move or copy the messages in any order. This functionality is especially useful in the case of search-results folders, for which the provider can group messages by parent folder.

Messages that do not exist, that have already been moved elsewhere, or that are currently submitted cannot be copied or moved. However, in copy or move operations involving more than one message, even if one or more messages specified cannot be copied or moved, a message store provider should complete the operation as best it can for each message specified. The provider should stop the operation without completing it only in the case of failures it cannot control, such as running out of memory or disk space, message store corruption, and so on.

Messages that are open with read/write access can be moved or copied. Store providers should, if possible, maintain entry identifiers across move or copy operations so as not to invalidate handles to those messages that may be held by client applications. If it is not possible to satisfy a client application's request to save changes on an object because the object has moved or changed since the client application opened it, the message store provider should return MAPI_E_OBJECT_CHANGED. Message store providers should also send notifications when moving or copying messages so that client applications are forewarned that their **IMAPIProp::SaveChanges** calls may fail.

If **CopyMessages** successfully completes the copy or move operation for every message requested by the client, it returns S_OK. If one or more messages cannot be copied or moved, **CopyMessages** returns MAPI_W_PARTIAL_COMPLETION. If **CopyMessages** returns a different value, such as MAPI_E_NOT_ENOUGH_MEMORY, that indicates the call did not complete, it might already have copied or moved one or more messages without being able to continue. The calling client cannot proceed on the assumption that an error return implies no work was done.

For more information on using the **HR_FAILED** macro, see Using Macros for Error Handling.

IMAPIFolder::CreateFolder

The **IMAPIFolder::CreateFolder** method creates a new folder as a subfolder within the message store.

HRESULT CreateFolder(
  ULONG *ulFolderType***,**
  LPTSTR *lpszFolderName***,**
  LPTSTR *lpszFolderComment***,**
  LPCIID *lpInterface***,**
  ULONG *ulFlags***,**
  LPMAPIFOLDER FAR * *lppFolder*
)

Parameters

ulFolderType
Input parameter containing a value indicating the type of folder to create. One of the following values can be passed:

FOLDER_GENERIC
Indicates a generic folder should be created.

FOLDER_SEARCH
Indicates a search-results folder should be created.

lpszFolderName
Input parameter pointing to a string containing the name for the new folder.

lpszFolderComment
Input parameter pointing to a string containing a comment associated with the new folder. This string becomes the value of the folder's PR_COMMENT property. If NULL is passed, the folder has no initial comment.

lpInterface
Input parameter indicating the interface identifier (IID) for the folder returned in the *lppFolder* parameter. Passing NULL indicates the folder object is cast to the standard interface for a folder object. A client application must pass NULL. A message store provider can also set the *lpInterface* parameter to an identifier for an appropriate interface for the folder object, for example IID_IUnknown, IID_IMAPIProp, or IID_IMAPIContainer.

ulFlags
Input parameter containing a bitmask of flags that controls how the folder is created. The following flags can be set:

MAPI_DEFERRED_ERRORS
Indicates the call is allowed to succeed even if the underlying object is not accessible to the calling client. If the object is not accessible, some subsequent call to the object might return an error.

MAPI_UNICODE
Indicates the passed-in strings are in Unicode format. If the MAPI_UNICODE flag is not set, the strings are in ANSI format.

OPEN_IF_EXISTS
Allows the method to succeed, even if the folder named in the *lpszFolderName* parameter already exists, by opening the existing folder with that name. Note that message store providers that allow sibling folders to have the same name might fail to open an existing folder if more than one exists with the supplied name.

lppFolder
Output parameter pointing to a variable where a pointer to the newly created folder object is stored.

Return Values

S_OK
    The call succeeded and has returned the expected value or values.

MAPI_E_BAD_CHARWIDTH
    Either the MAPI_UNICODE flag was set and the implementation does not support Unicode, or MAPI_UNICODE was not set and the implementation only supports Unicode.

MAPI_E_COLLISION
    A folder with the name given in *lpszFolderName* already exists. Folder names must be unique.

Remarks

Message store providers implement the **IMAPIFolder::CreateFolder** method to create new generic and search-results folders in a message store. Root folders cannot be created. Most message store providers require the name of the new folder to be unique with respect to the names of its sibling folders. Clients should be able to handle MAPI_E_COLLISION, which is returned if this rule is not followed.

To determine the entry identifier of the newly created folder, the implementation calls the **IMAPIProp::GetProps** method to read the new folder's PR_ENTRYID property while the folder is still open.

See Also

IMAPIProp::GetProps method

IMAPIFolder::CreateMessage

The **IMAPIFolder::CreateMessage** method creates a new message or new associated information within a folder.

HRESULT CreateMessage(
  LPCIID *lpInterface***,**
  ULONG *ulFlags***,**
  LPMESSAGE FAR * *lppMessage*
)

Parameters

lpInterface
: Input parameter pointing to the interface identifier (IID) for the message returned in the *lppMessage* parameter. Passing NULL indicates the message object is cast to the standard interface for a message object. A client application must pass NULL. A message store provider can also set the *lpInterface* parameter to an identifier for an appropriate interface for the message object, for example IID_IUnknown or IID_IMAPIProp.

ulFlags
: Input parameter containing a bitmask of flags that controls how the message is created. The following flags can be set:

  MAPI_ASSOCIATED
  : Indicates that an associated item, such as a view, should be created.

  MAPI_DEFERRED_ERRORS
  : Indicates the call is allowed to succeed even if the underlying object is not accessible to the calling client. If the object is not accessible, some subsequent call to the object might return an error.

lppMessage
: Output parameter pointing to a variable where a pointer to the newly created message object is stored.

Return Values

S_OK
: The call succeeded and has returned the expected value or values.

Remarks

Message store providers implement the **IMAPIFolder::CreateMessage** method to create a new message or new associated information in a folder.

To create objects associated with a folder that are listed in its associated contents table, set the MAPI_ASSOCIATED flag in the *ulFlags* parameter. Associated information entries connect invisible data, such as views, with a folder. For more information on working with associated information, see About Contents Tables.

Although a new message has a unique PR_RECORD_KEY property, its PR_ENTRYID property might not be available until a client saves the message by using the **IMAPIProp::SaveChanges** method. This potential unavailability stems from the fact that some message store providers generate the entry identifier when the message is created and others wait to do so until the message has been saved. The latter functionality is typical; that is, a new message usually does not appear in a folder contents table until a client calls **SaveChanges**.

If a folder is deleted before a new message within it is saved, the results of a call to **CreateMessage** are undefined.

See Also

IMAPIProp::SaveChanges method

IMAPIFolder::DeleteFolder

The **IMAPIFolder::DeleteFolder** method deletes a subfolder from a folder.

HRESULT DeleteFolder(
  ULONG *cbEntryID*,
  LPENTRYID *lpEntryID*,
  ULONG *ulUIParam*,
  LPMAPIPROGRESS *lpProgress*,
  ULONG *ulFlags*
)

Parameters

cbEntryID
  Input parameter containing the size, in bytes, of the entry identifier pointed to by the *lpEntryID* parameter.

lpEntryID
  Input parameter pointing to the entry identifier of the subfolder to delete.

ulUIParam
  Input parameter containing the handle of the parent window for any dialog boxes or windows this method displays. The *ulUIParam* parameter is ignored unless the FOLDER_DIALOG flag is set in the *ulFlags* parameter and NULL is passed in the *lpProgress* parameter.

lpProgress
  Input parameter pointing to a progress object that contains client-supplied progress information. If NULL is passed in the *lpProgress* parameter, MAPI provides the progress information. The *lpProgress* parameter is ignored unless the FOLDER_DIALOG flag is set in the *ulFlags* parameter.

ulFlags
  Input parameter containing a bitmask of flags that controls the deletion of the subfolder. The following flags can be set:

  DEL_FOLDERS
    Deletes all subfolders of the subfolder indicated in *lpEntryID*.

  DEL_MESSAGES
    Deletes all messages in the subfolder indicated in *lpEntryID*.

  FOLDER_DIALOG
    Displays a progress indicator while the operation proceeds.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

MAPI_E_HAS_FOLDERS
  The subfolder being deleted contains subfolders, and the DEL_FOLDERS flag was not set. The subfolder was not deleted.

MAPI_E_HAS_MESSAGES
  The subfolder being deleted contains messages, and the DEL_MESSAGES flag was not set. The subfolder was not deleted.

MAPI_W_PARTIAL_COMPLETION
  The call succeeded, but not all of the entries were successfully deleted. To test for this warning, use the **HR_FAILED** macro. When this warning is returned, the call should be handled as successful.

Remarks

Message store providers implement the **IMAPIFolder::DeleteFolder** method to delete a subfolder of a folder. If the message store provider so indicates, **DeleteFolder** can also delete all messages or all subfolders in a subfolder, or both. To delete all messages in a subfolder, the client sets DEL_MESSAGES in *ulFlags*; to delete all subfolders in a subfolder, the client application sets DEL_FOLDERS in *ulFlags*. However, no flag need be set to delete any associated items, such as views or form definitions, from a folder.

In deletions of more than one subfolder, even if one or more subfolders marked for deletion do not exist or have been moved elsewhere, a message store provider should complete the deletion as best it can for each subfolder marked. The provider should stop the deletion without completing it only in the case of failures it cannot control, such as running out of memory or disk space, message store corruption, and so on.

If **DeleteFolder** successfully deletes every subfolder marked for deletion, it returns S_OK. If one or more subfolders cannot be deleted, **DeleteFolder** returns MAPI_W_PARTIAL_COMPLETION or MAPI_E_NOT_FOUND, depending on the message store's implementation. If **DeleteFolder** returns a different value, such as MAPI_E_NOT_ENOUGH_MEMORY, that indicates the call did not complete, it might have already deleted one or more subfolders or messages without being able to continue. The calling client cannot proceed on the assumption that an error return implies no work was done.

During a **DeleteFolder** call, messages being processed by the MAPI spooler are not deleted, nor should the message store provider call the **IMsgStore::AbortSubmit** method for such messages. A message being processed by the MAPI spooler is left in the folder in which it resides. This functionality might prevent one or more subfolders from being deleted because they still have contents.

For more information on using the **HR_FAILED** macro, see Using Macros for Error Handling.

IMAPIFolder::DeleteMessages

The **IMAPIFolder::DeleteMessages** method deletes one or more messages from a folder.

HRESULT DeleteMessages(
  LPENTRYLIST *lpMsgList***,**
  ULONG *ulUIParam***,**
  LPMAPIPROGRESS *lpProgress***,**
  ULONG *ulFlags*
)

Parameters

lpMsgList
  Input parameter pointing to an array of **ENTRYLIST** structures that identify the message or messages to delete.

ulUIParam
  Input parameter containing the handle of the parent window for any dialog boxes or windows this method displays. The *ulUIParam* parameter is ignored unless the MESSAGE_DIALOG flag is set in the *ulFlags* parameter and NULL is passed in the *lpProgress* parameter.

lpProgress
  Input parameter pointing to a progress object that contains client-supplied progress information. If NULL is passed in the *lpProgress* parameter, MAPI provides the progress information. The *lpProgress* parameter is ignored unless the MESSAGE_DIALOG flag is set in the *ulFlags* parameter.

ulFlags
  Input parameter containing a bitmask of flags that controls how the messages are deleted. The following flag can be set:

  MESSAGE_DIALOG
    Displays a progress indicator as the operation proceeds.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

MAPI_W_PARTIAL_COMPLETION
  The call succeeded, but not all of the entries were successfully deleted. To test for this warning, use the **HR_FAILED** macro. When this warning is returned, the call should be handled as successful.

Remarks

Message store providers implement the **IMAPIFolder::DeleteMessages** method to delete messages from a folder.

Messages that do not exist, that have been moved elsewhere, that are open with read/write access, or that are currently submitted cannot be deleted. However, in deletions of more than one message, even if one or more messages marked for deletion cannot be deleted, a message store provider should complete the deletion as best it can for each message marked. The provider should stop the deletion without completing it only in the case of failures it cannot control, such as running out of memory or disk space, message store corruption, and so on.

If **DeleteMessages** successfully deletes every message marked for deletion, it returns S_OK. If one or more messages cannot be deleted, **DeleteMessages** returns MAPI_W_PARTIAL_COMPLETION or MAPI_E_NOT_FOUND, depending on the message store's implementation. If **DeleteMessages** returns a different value, such as MAPI_E_NOT_ENOUGH_MEMORY, that indicates that call did not complete, it might already have deleted one or more messages without being able to continue. The calling client cannot proceed on the assumption that an error return implies no work was done.

During a **DeleteMessages** call, messages being processed by the MAPI spooler are not deleted, nor should the message store provider call the **IMsgStore::AbortSubmit** method for such messages. A message being processed by the MAPI spooler is left in the folder in which it resides.

For more information on using the **HR_FAILED** macro, see Using Macros for Error Handling.

IMAPIFolder::EmptyFolder

The **IMAPIFolder::EmptyFolder** method deletes all items from a folder without deleting the folder itself.

HRESULT EmptyFolder(
  ULONG *ulUIParam***,**
  LPMAPIPROGRESS *lpProgress***,**
  ULONG *ulFlags*
)

Parameters

ulUIParam
: Input parameter containing the handle of the parent window for any dialog boxes or windows this method displays. The *ulUIParam* parameter is ignored unless the FOLDER_DIALOG flag is set in the *ulFlags* parameter and NULL is passed in the *lpProgress* parameter.

lpProgress
: Input parameter pointing to a progress object that contains client-supplied progress information. If NULL is passed in *lpProgress*, MAPI provides the progress information. The *lpProgress* parameter is ignored unless the FOLDER_DIALOG flag is set in the *ulFlags* parameter.

ulFlags
: Input parameter containing a bitmask of flags that controls how deletion is performed for the messages and subfolders of a folder. The following flags can be set:

  DEL_ASSOCIATED
  : Deletes subfolders in their entirety, including all their subfolders and associated items. The DEL_ASSOCIATED flag only has meaning for the top-level folder the call acts on.

  FOLDER_DIALOG
  : Displays a progress indicator while the operation proceeds.

Return Values

S_OK
: The call succeeded and has returned the expected value or values.

MAPI_W_PARTIAL_COMPLETION
: The call succeeded, but some entries were not successfully deleted. To test for this warning, use the **HR_FAILED** macro. When this warning is returned, the call should be handled as successful.

Remarks

Message store providers implement the **IMAPIFolder::EmptyFolder** method to delete all of a folder's contents without deleting the folder itself.

When the calling client sets DEL_ASSOCIATED in *ulFlags*, **EmptyFolder** deletes all messages, subfolders, and associated information of a folder. A folder's associated information includes such items as views and form definitions. When the calling client does not set DEL_ASSOCIATED, **EmptyFolder** does not delete associated information. DEL_ASSOCIATED only has meaning for the top-level folder the call acts on.

If **EmptyFolder** successfully deletes all subfolders and messages, it returns S_OK. If one or more items cannot be deleted, **EmptyFolder** returns MAPI_W_PARTIAL_COMPLETION. If **EmptyFolder** returns a different value, such as MAPI_E_NOT_ENOUGH_MEMORY, that indicates the call did not complete, it might already have deleted one or more messages or subfolders without being able to continue. The calling client cannot proceed on the assumption that an error return implies no work was done.

During an **EmptyFolder** call, submitted messages are not deleted, nor should the message store provider call the **IMsgStore::AbortSubmit** method for such messages. A submitted message is left in the folder in which it resides.

For more information on using the **HR_FAILED** macro, see Using Macros for Error Handling.

IMAPIFolder::GetMessageStatus

The **IMAPIFolder::GetMessageStatus** method obtains the status associated with a message in a particular folder – for example, whether that message is marked for deletion.

HRESULT GetMessageStatus(
  ULONG *cbEntryID***,**
  LPENTRYID *lpEntryID***,**
  ULONG *ulFlags***,**
  ULONG FAR * *lpulMessageStatus*
)

Parameters

cbEntryID
  Input parameter containing the size, in bytes, of the entry identifier pointed to by the *lpEntryID* parameter.

lpEntryID
  Input parameter pointing to the entry identifier for the message whose status is obtained.

ulFlags
  Reserved; must be zero.

lpulMessageStatus
  Output parameter pointing to a variable where a bitmask of flags indicating the message's status is stored. Bits 5 through 15 are reserved and must be zero; bits 16 through 31 are available for implementation-specific use.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

Remarks

Client applications call the **IMAPIFolder::GetMessageStatus** method to find out the status of a message. **GetMessageStatus** returns status information in the *lpulMessageStatus* parameter.

How a message store provider sets, clears, and uses the message status bits in *lpulMessageStatus* depends entirely on its implementation, except that bits 5 through 15 are reserved and must be zero. In the interpersonal message (IPM) subtree, MAPI reserves bits 16 through 31 for use by the IPM client. Users can choose which IPM client they use. Message store providers that store their messages in other subtrees can use bits 16 through 31 for their own purposes.

See Also

IMAPIFolder::SetMessageStatus method

IMAPIFolder::SaveContentsSort

The **IMAPIFolder::SaveContentsSort** method sets the default sort order for a folder's contents table.

HRESULT SaveContentsSort(
  LPSSortOrderSet *lpSortCriteria*,
  ULONG *ulFlags*
)

Parameters

lpSortCriteria
  Input parameter pointing to an **SSortOrderSet** structure containing the sort criteria for the default sort order.

ulFlags
  Input parameter containing a bitmask of flags that controls how the default sort order is set. The following flag can be set:

  RECURSIVE_SORT
    Indicates the default sort order set applies to the indicated folder and all its subfolders.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

MAPI_E_NO_SUPPORT
  The operation is not supported by one or more service providers.

Remarks

Message store providers implement the **IMAPIFolder::SaveContentsSort** method to set the default sort order for a folder's contents table. The **IMAPIContainer::GetContentsTable** method uses this default sort order; before returning a folder's contents table, **GetContentsTable** sorts it by using the default sort order. Not all message store providers support **SaveContentsSort**; providers that don't return MAPI_E_NO_SUPPORT.

See Also

IMAPIContainer::GetContentsTable method, **SSortOrderSet** structure

IMAPIFolder::SetMessageStatus

The **IMAPIFolder::SetMessageStatus** method sets the status associated with a message in a particular folder – for example, whether that message is marked for deletion.

HRESULT SetMessageStatus(
  ULONG *cbEntryID***,**
  LPENTRYID *lpEntryID***,**
  ULONG *ulNewStatus***,**
  ULONG *ulNewStatusMask***,**
  ULONG FAR * *lpulOldStatus*
)

Parameters

cbEntryID
  Input parameter containing the size, in bytes, of the entry identifier pointed to by the *lpEntryID* parameter.

lpEntryID
  Input parameter pointing to the entry identifier for the message whose status is set.

ulNewStatus
  Input parameter containing a bitmask of flags used to set the status of the message. These flags are set by the bitmask in the *ulNewStatusMask* parameter.

ulNewStatusMask
  Input parameter containing a bitmask of flags used to set the bitmask in the *ulNewStatus* parameter. For each bit set in *ulNewStatusMask*, the corresponding bit in *ulNewStatus* is set or cleared for the message.

lpulOldStatus
  Output parameter pointing to a variable where the previous value of the message status flags is stored.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

Remarks

Message store providers implement the **IMAPIFolder::SetMessageStatus** method to set the status associated with a message within a folder in a message store.

Some message store providers use **SetMessageStatus** to enable clients to negotiate a message lockout operation among themselves. To do so, clients designate a bit for the lockout bit. The provider sets the lockout bit using **SetMessageStatus** and examines the previous value in the *lpulOldStatus* parameter to determine if a client caused the designated bit to be set. By using the bitmask, applications can use other bits in the *ulNewStatus* parameter to track message status without interfering with the lockout bit.

See Also

IMAPIFolder::GetMessageStatus method

IMAPIFolder::SetReadFlags

The **IMAPIFolder::SetReadFlags** method sets or clears the read flags for the messages in a folder and manages the sending of read reports.

HRESULT SetReadFlags(
  LPENTRYLIST *lpMsgList*,
  ULONG *ulUIParam*,
  LPMAPIPROGRESS *lpProgress*,
  ULONG *ulFlags*
)

Parameters

lpMsgList
  Input parameter pointing to an array of **ENTRYLIST** structures that identify the message or messages for which to set or clear read flags. If a client application passes NULL in the *lpMsgList* parameter, all messages are processed.

ulUIParam
  Input parameter containing the handle of the parent window for any dialog boxes or windows this method displays. The *ulUIParam* parameter is ignored unless the client sets the MESSAGE_DIALOG flag in the *ulFlags* parameter and passes NULL in the *lpProgress* parameter.

lpProgress
  Input parameter pointing to a progress object that contains client-supplied progress information. If NULL is passed in *lpProgress*, MAPI provides the progress information. The *lpProgress* parameter is ignored unless the FOLDER_DIALOG flag is set in *ulFlags*.

ulFlags
  Input parameter containing a bitmask of flags that controls the setting of a message's read flag (that is, a message's MSGFLAG_READ flag in its PR_MESSAGE_FLAGS property) and the processing of read reports. The following flags can be set:

  CLEAR_READ_FLAG
    Clears MSGFLAG_READ. No read report is sent.

  GENERATE_RECEIPT_ONLY
    Generates a read report if it is pending but does not change the state of MSGFLAG_READ.

  MAPI_DEFERRED_ERRORS
    Indicates the call is allowed to succeed even if the underlying object is not accessible to the calling client. If the object is not accessible, some subsequent call to the object might return an error.

  MESSAGE_DIALOG
    Displays a progress indicator while the operation proceeds.

  SUPPRESS_RECEIPT
    Directs the message store provider to cancel the generation of a read report if this call changes the state of the message from unread to read and a read report has been requested. If this call does not change the state of the message, the message store provider can ignore this flag.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

MAPI_E_NO_SUPPRESS
  The message store provider does not support the suppression of read reports.

MAPI_W_PARTIAL_COMPLETION
  The call succeeded, but not all the entries were successfully processed. To test for this warning, use the **HR_FAILED** macro. When this warning is returned, the call should be handled as successful.

Remarks

Message store providers implement the **IMAPIFolder::SetReadFlags** method when messages are being moved or copied from one message store to another to set or clear the read flags for the messages in a folder. **SetReadFlags** also manages the sending of read reports in such situations. **SetReadFlags** sets or clears the MSGFLAG_READ flag in the PR_MESSAGE_FLAGS property. If MSGFLAG_READ is set for a message, the message is marked as having been read, which does not necessarily indicate the intended recipient has read the message.

Messages that do not exist, that have been moved elsewhere, that are open with read/write access, or that are currently submitted cannot have their read flags set. However, in read-flag setting operations involving more than one message, even if one or more messages specified cannot have their read flags set, a message store provider should complete the operation as best it can for a message specified. The provider should stop the operation without completing it only in the case of failures it cannot control, such as running out of memory or disk space, message store corruption, and so on.

If none of the flags are set in the *ulFlags* parameter, the following rules apply:

- If MSGFLAG_READ is already set, no change should be made.
- If the PR_READ_RECEIPT_REQUESTED property is set, the client should send the read report and set MSGFLAG_READ.

SetReadFlags returns MAPI_E_INVALID_PARAMETER if any of the following combinations are set in *ulFlags*:

- SUPPRESS_RECEIPT | CLEAR_READ_FLAG
- SUPPRESS_RECEIPT | CLEAR_READ_FLAG | GENERATE_RECEIPT_ONLY
- CLEAR_READ_FLAG | GENERATE_RECEIPT_ONLY

If both the SUPPRESS_RECEIPT and GENERATE_RECEIPT_ONLY flags are set, PR_READ_RECEIPT_REQUESTED, if set, should be cleared, and a read report should not be sent.

Note Providers can optimize report behavior so that a client's setting a message attribute to get a read or delivery report is only a request and so that the provider can support not sending read or delivery reports. However, some message store providers do not support the suppression of read reports for some messages. If a client calls **SetReadFlags** on such a message with SUPPRESS_RECEIPT set in *ulFlags*, **SetReadFlags** returns MAPI_E_NO_SUPPRESS; in this case, MAPI does not set the read flag and does not generate a report.

If **SetReadFlags** successfully completes processing for every message specified, it returns S_OK. If one or more messages cannot be processed, **SetReadFlags** returns MAPI_W_PARTIAL_COMPLETION. If **SetReadFlags** returns a different value, such as MAPI_E_NOT_ENOUGH_MEMORY, that indicates the call did not compete, it might already have processed one or more messages without being able to continue. The calling client cannot proceed on the assumption that an error return implies no work was done.

For more information on using the **HR_FAILED** macro, see Using Macros for Error Handling.

See Also

IMessage::SetReadFlag method, PR_MESSAGE_FLAGS property

IMAPIForm : IUnknown

The **IMAPIForm** interface is implemented by form servers for the benefit of form viewers. Its methods are used to work with form view contexts and form notification, to perform form verbs, and to shut down forms.

At a Glance

Specified in header file:	MAPIFORM.H
Object that supplies this interface:	Form object
Corresponding pointer type:	LPMAPIFORM
Implemented by:	Form object
Called by:	Form viewers

Vtable Order

SetViewContext	Sets a form view context as the current view context for a form.
GetViewContext	Returns the current view context for a form.
ShutdownForm	Closes a form.
DoVerb	Requests a form object perform one of its verbs.
Advise	Registers a form viewer for notifications about changes to a form.
Unadvise	Removes a form viewer's registration for notification of form object changes previously established with a call to the **IMAPIForm::Advise** method.

IMAPIForm::Advise

The **IMAPIForm::Advise** method registers a form viewer for notifications about changes to a form.

HRESULT Advise(
  LPMAPIVIEWADVISESINK *pAdvise*,
  ULONG FAR * *pulConnection*
)

Parameters

pAdvise
  Input parameter pointing to the view advise sink object to be called when an event occurs for the form object about which notification has been requested.

pulConnection
  Output parameter pointing to a variable that upon a successful return holds the connection number for the notification registration. The connection number must be nonzero.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

Remarks

Form viewers call the **IMAPIForm::Advise** method to register for notification callbacks when changes occur to a form. A form server's implementation of **Advise** keeps a copy of the pointer to the view advise sink object passed in the *pAdvise* parameter and uses this pointer in its notification callbacks. The **Advise** implementation should call the **IUnknown::AddRef** method on the view advise sink object to retain this pointer to the advise sink until notification registration is canceled. The **Advise** implementation should also assign a nonzero connection number to the notification registration and call **AddRef** on this number before returning it in the *pulConnection* parameter.

To cancel a notification registration, a form viewer calls the **IMAPIForm::Unadvise** method. During the call to **Unadvise**, or shortly thereafter, the saved pointer to the view advise sink object is released.

For more information on the notification process, see About Notification.

See Also

IMAPIForm::Unadvise method, **IMAPIViewAdviseSink : IUnknown** interface

IMAPIForm::DoVerb

The **IMAPIForm::DoVerb** method requests a form object perform one of its verbs.

HRESULT DoVerb(
  LONG *iVerb***,**
  LPMAPIVIEWCONTEXT *lpViewContext***,**
  ULONG *hwndParent***,**
  LPCRECT *lprcPosRect*
)

Parameters

iVerb
  Input parameter containing the number of the verb to be performed.

lpViewContext
  Input parameter pointing to a view context object. The *lpViewContext* parameter can be NULL.

hwndParent
  Input parameter containing the handle of the parent window for any dialog boxes or windows this method displays. The *hwndParent* parameter should be NULL if the dialog box or window is not modal.

lprcPosRect
  Input parameter pointing to a Win32 **RECT** structure containing the current form's window size and position.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

OLEOBJ_S_CANNOT_DOVERB_NOW
  The verb is valid, but the object cannot perform the operation now.

Remarks

Form viewers call the **IMAPIForm::DoVerb** method to request an object to perform one of its verbs. Typical form server implementations of **DoVerb** contain a **switch** statement that tests the valid values for the *iVerb* parameter.

The *lpViewContext* parameter can be NULL. If a view context is passed in *lpViewContext*, the form must use this view context for the duration of the verb processing rather than the view context passed in an earlier call to the **IMAPIForm::SetViewContext** method. The view context should not be saved.

Some verbs, such as Print, should be modal with respect to the **DoVerb** call – that is, the indicated operation must be finished before the **DoVerb** call returns. Nonmodal verbs can be made to act as modal verbs by passing in a view context object in *lpViewContext* for which a call to the **IMAPIViewContext::GetViewStatus** method returns the VCSTATUS_MODAL flag.

The handle in *hwndParent* usually remains valid throughout the **DoVerb** call, but because it can be destroyed immediately upon the call's return, forms should not save the handle.

To obtain the **RECT** structure used by a form's window, call the Windows **GetWindowRect** function.

See Also

IMAPIForm::SetViewContext method, **IMAPIViewContext::GetViewStatus** method

IMAPIForm::GetViewContext

The **IMAPIForm::GetViewContext** method returns the current view context for a form.

HRESULT GetViewContext(
  LPMAPIVIEWCONTEXT FAR * *ppViewContext*
)

Parameters

ppViewContext
  Output parameter pointing to a variable where the pointer to the view context object is stored.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

S_FALSE
  The form object doesn't have a view context.

See Also

IMAPIForm::SetViewContext method, **IMAPIViewContext : IUnknown** interface

IMAPIForm::SetViewContext

The **IMAPIForm::SetViewContext** method sets a form view context as the current view context for a form.

HRESULT SetViewContext(
  LPMAPIVIEWCONTEXT *pViewContext*
)

Parameters

pViewContext
  Input parameter pointing to the view context object to set as current.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

Remarks

Form viewers call the **IMAPIForm::SetViewContext** method to set a particular form view context as current. A typical implementation of **SetViewContext** sets up a view advise sink object by calling the **IMAPIViewContext::SetAdviseSink** method, so that notifications can be received for the view context, and calls the **IMAPIViewContext::GetViewStatus** method for the view context to set as current to determine which status flags have been set.

A form can have only one view context at a time. If a previous view context exists, the implementation must call **IMAPIViewContext::SetAdviseSink**, passing in NULL in the *pmnvs* parameter, before returning from the **SetViewContext** call.

The **SetViewContext** implementation can also perform other actions based on the **GetViewStatus** flags returned. For example, if the VCSTATUS_NEXT and VCSTATUS_PREV flags are set, the implementation can enable Next and Previous buttons for the view context that it sets as current.

See Also

IMAPIViewContext::GetViewStatus method, **IMAPIViewContext::SetAdviseSink** method

IMAPIForm::ShutdownForm

The **IMAPIForm::ShutdownForm** method closes a form.

HRESULT Close(
  ULONG *ulSaveOptions*
)

Parameters

ulSaveOptions
  Input parameter containing a value that controls how or whether a form's contents are saved when the form is closed. The following mutually exclusive values can be set:
  SAVEOPTS_NOSAVE
    Indicates form data should not be saved.
  SAVEOPTS_PROMPTSAVE
    Prompts the user to save data related to the form if the form has changed.
  SAVEOPTS_SAVEIFDIRTY
    Indicates data related to the form should be saved if the form has changed. Forms can implement the SAVEOPTS_SAVEIFDIRTY value with its own or the SAVEOPTS_NOSAVE value's functionality if no user interface is currently being displayed.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

Remarks

Form viewers call the **IMAPIForm::ShutdownForm** method to close a form object. A typical form server's implementation of **ShutdownForm** does the following:

1. Calls the **IUnknown::AddRef** method for the form's **IPersistMessage** object so it isn't released before processing is finished.
2. Handles saving form data as indicated by the flag set in the *ulSaveOptions* parameter.
3. Destroys the form's window.
4. Releases the form's subobjects.
5. Calls the **IMAPIViewAdviseSink::OnShutdown** method to update notifications.
6. Calls the **IMAPIViewContext::SetAdviseSink** method, setting the advise sink pointer to NULL.
7. Calls the **MAPIFreeBuffer** function to release the form's properties.
8. Releases the **IPersistMessage** object for which it added a reference in Step 1.
9. Releases all form viewer and message site interfaces.
10. Returns S_OK.

Form viewers can ignore errors returned by **ShutdownForm** and release the form interface after making the call.

See Also

IMAPIViewAdviseSink::OnShutdown method

IMAPIForm::Unadvise

The **IMAPIForm::Unadvise** method removes a form viewer's registration for notification of form object changes previously established with a call to the **IMAPIForm::Advise** method.

HRESULT Unadvise(
  ULONG *ulConnection*
)

Parameters

ulConnection
  Input parameter containing the number of the registration connection returned by a call to **IMAPIForm::Advise**.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

Remarks

Form viewers call the **IMAPIForm::Unadvise** method to release the pointer to the view advise sink object passed in the *pAdvise* parameter in the previous call to **IMAPIForm::Advise**, thereby canceling a notification registration. As part of discarding the pointer to the view advise sink, the view advise sink's **IUnknown::Release** method is called. Generally, **Release** is called during the **Unadvise** call, but if another thread is in the process of calling one of the **IMAPIViewAdviseSink** methods for the view advise sink object, the **Release** call is delayed until the method call returns.

See Also

IMAPIForm::Advise method, **IMAPIViewAdviseSink : IUnknown** interface

IMAPIFormAdviseSink : IUnknown

The **IMAPIFormAdviseSink** interface is implemented by form objects, which receive notifications from form viewers, for notification purposes. Although part of the form object, **IMAPIFormAdviseSink** is not an interface for a form object. Hence, client applications should not attempt to use the **QueryInterface** method for a form object to query for this interface.

At a Glance

Specified in header file:	MAPIFORM.H
Object that supplies this interface:	Form advise sink object
Corresponding pointer type:	LPMAPIFORMADVISESINK
Implemented by:	Form objects
Called by:	Form viewers

Vtable Order

OnChange	Notifies a form about a change in a form viewer's status.
OnActivateNext	Identifies whether the message class of the next message to display can be handled by the current form.

IMAPIFormAdviseSink::OnActivateNext

The **IMAPIFormAdviseSink::OnActivateNext** method identifies whether the message class of the next message to display can be handled by the current form.

HRESULT OnActivateNext(
  LPCSTR *lpszMessageClass***,**
  ULONG *ulMessageStatus***,**
  ULONG *ulMessageFlags***,**
  LPPERSISTMESSAGE FAR * *ppPersistMessage*
)

Parameters

lpszMessageClass
  Input parameter pointing to a string naming the message class of the current form.

ulMessageStatus
  Input parameter containing a bitmask of client- or provider-defined flags, copied from the PR_MSG_STATUS property of the next message to display, that provides information on the state of the message.

ulMessageFlags
  Input parameter pointing to a bitmask of flags, copied from the PR_MESSAGE_FLAGS property of the next message to display, that indicates the current state of the message.

ppPersistMessage
  Output parameter pointing to a variable where the pointer to the form object used for the new form is stored, if a new form is required. A pointer to NULL can be returned if the current form object can be used to display the next message.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

S_FALSE
  The form does not handle the message class.

Remarks

Form viewers call the **IMAPIFormAdviseSink::OnActivateNext** method to find out if the message class of a message can be handled by the current form object. The form object should return S_OK and NULL in the *ppPersistMessage* parameter if it can handle the message class. If it cannot, it should return S_FALSE.

If the form object can activate message classes other than the base class or the currently loaded class, it should return S_OK and a pointer to the **IPersistMessage** object of the appropriate form object for the message class in the *ppPersistMessage* parameter. The message in question is then loaded by the form viewer using the **IPersistMessage::Load** method of that object.

See Also

IPersistMessage::Load method

IMAPIFormAdviseSink::OnChange

The **IMAPIFormAdviseSink::OnChange** method notifies a form object about a change in a form viewer's status.

HRESULT OnChange(
  ULONG *ulDir*
)

Parameters

ulDir
 Input parameter containing a bitmask of flags that controls handling changes to the form viewer's status. The following flags can be set:

 VCSTATUS_INTERACTIVE
  Indicates the form should display a user interface. If this flag is not set, the form should suppress displaying a user interface even in response to a verb that usually causes a user interface to be displayed.

 VCSTATUS_MODAL
  Indicates the form is to be modal to the form viewer.

 VCSTATUS_NEXT
  Indicates there is a next message in the form viewer.

 VCSTATUS_PREV
  Indicates there is a previous message in the form viewer.

 VCSTATUS_READONLY
  Indicates the deletion, submission, and moving operations should be disabled.

Return Values

S_OK
 The call succeeded and has returned the expected value or values.

Remarks

Form viewers call the **IMAPIFormAdviseSink::OnChange** method to notify the form object about a change in a viewer's status. Usually, the only change is setting or clearing the VCSTATUS_NEXT or VCSTATUS_PREVIOUS flag based on the presence or absence of a next or previous message in the viewer. The form object then enables or disables any next or previous actions it supports accordingly.

The settings of VCSTATUS_MODAL and VCSTATUS_INTERACTIVE cannot change in a view context once it has been created.

See Also

IMAPIViewContext::ActivateNext method

IMAPIFormContainer : IUnknown

The **IMAPIFormContainer** interface creates and manages form libraries within folders. This interface is used to create application-specific form libraries.

At a Glance

Specified in header file:	MAPIFORM.H
Object that supplies this interface:	Form container object
Corresponding pointer type:	LPMAPIFORMCONTAINER
Implemented by:	Form library providers
Called by:	Client applications

Vtable Order

InstallForm	Installs a form into a form container.
RemoveForm	Removes a particular form from a form container.
ResolveMessageClass	Resolves a message class to its form within a form container and returns a form information object for that form.
ResolveMultipleMessageClasses	Resolves a group of message classes to their forms within a form container and returns an array of form information objects for those forms.
CalcFormPropSet	Returns an array of the properties used by all forms installed within a form container.
GetDisplay	Returns the display name of a form container.

IMAPIFormContainer::CalcFormPropSet

The **IMAPIFormContainer::CalcFormPropSet** method returns an array of the properties used by all forms installed within a form container.

HRESULT CalcFormPropSet(
  ULONG *ulFlags***,**
  LPMAPIFORMPROPARRAY FAR * *ppResults*
)

Parameters

ulFlags
  Input parameter containing a bitmask of flags that controls how the property array in the *ppResults* parameter is returned. The following flags can be set:
  FORMPROPSET_INTERSECTION
    Indicates the returned array contains the intersection of the forms' properties.
  FORMPROPSET_UNION
    Indicates the returned array contains the union of the forms' properties.
  MAPI_UNICODE
    Indicates the strings returned in the array are in Unicode format. If the MAPI_UNICODE flag is not set, the strings are in ANSI format.

ppResults
  Output parameter pointing to a variable where the pointer to the returned **SMAPIFormPropArray** structure is stored. This structure contains all properties used by the installed forms.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

MAPI_E_BAD_CHARWIDTH
  Either the MAPI_UNICODE flag was set and the implementation does not support Unicode, or MAPI_UNICODE was not set and the implementation only supports Unicode.

Remarks

Client applications call the **IMAPIFormContainer::CalcFormPropSet** method to obtain an array of properties used by all forms installed within a form container. **CalcFormPropSet** either takes an intersection or a union of these forms' property sets, depending on the flag set in the *ulFlags* parameter, and it returns an **SMAPIFormPropArray** structure containing the resulting group of properties. Passing the PROPSET_UNION flag in *ulFlags* obtains the results from a union; passing the PROPSET_INTERSECTION flag obtains the results from an intersection.

If a client passes the MAPI_UNICODE flag in *ulFlags,* all strings returned are Unicode. Form library providers that do not support Unicode strings should return MAPI_E_BAD_CHARWIDTH if MAPI_UNICODE is passed.

CalcFormPropSet acts as does the **IMAPIFormMgr::CalcFormPropSet** method, except that it operates on every form registered in a particular container.

See Also

IMAPIFormMgr::CalcFormPropSet method, **SMAPIFormPropArray** structure

IMAPIFormContainer::GetDisplay

The **IMAPIFormContainer::GetDisplay** method returns the display name of a form container.

HRESULT GetDisplay(
  ULONG *ulFlags*,
  LPTSTR FAR * *pszDisplayName*
)

Parameters

ulFlags
Input parameter containing a bitmask of flags that controls the type of the returned string. The following flag can be set:

MAPI_UNICODE
Indicates the returned string is in Unicode format. If the MAPI_UNICODE flag is not set, the strings are in ANSI format.

pszDisplayName
Output parameter pointing to a string containing the display name of the form container.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

IMAPIFormContainer::InstallForm

The **IMAPIFormContainer::InstallForm** method installs a form into a form container.

HRESULT InstallForm(
  ULONG *ulUIParam***,**
  ULONG *ulFlags***,**
  LPCTSTR *szCfgPathName*
)

Parameters

ulUIParam
 Input parameter containing the handle of the parent window for any dialog boxes or windows this method displays. The *ulUIParam* parameter is ignored unless the client application sets the MAPI_DIALOG flag in the *ulFlags* parameter. The *ulUIParam* parameter can be NULL if MAPI_DIALOG is not also passed.

ulFlags
 Input parameter containing a bitmask of flags that controls the installation of the form. The following flags can be set:

 MAPI_DIALOG
  Displays a dialog box to provide status or prompt the user for additional information. If this flag is not set, no dialog box is displayed.

 MAPI_UNICODE
  Indicates the passed-in strings are in Unicode format. If the MAPI_UNICODE flag is not set, the strings are in ANSI format.

 MAPIFORM_INSTALL_OVERWRITEONCONFLICT
  Indicates a previous definition of the form is to be replaced with the definition of the form being installed. This flag is ignored when MAPI_DIALOG is present.

szCfgPathName
 Input parameter containing the path to the form configuration file that describes the form and its implementation.

Return Values

S_OK
 The call succeeded and has returned the expected value or values.

MAPI_E_EXTENDED_ERROR
 An implementation error occurred; to get the **MAPIERROR** structure associated with the error, call the **IMAPIFormContainer::GetLastError** method.

MAPI_E_USER_CANCEL
 The user canceled the installation of the form, typically by clicking the **Cancel** button in a dialog box.

Remarks

Client applications call the **IMAPIFormContainer::InstallForm** method to install a form into a specific form container. The *szCfgPathName* parameter must contain the path of a form configuration file – that is, a file with the .CFG extension – that describes the form and its implementation. The *ulFlags* parameter specifies:

- That a user interface enabling the user installing the form to specify details of installation is displayed, if the MAPI_DIALOG flag is set.
- That the previous form definition is overlaid with the form being installed, if the MAPIFORM_INSTALL_OVERWRITEONCONFLICT flag is set. Otherwise, the form installation is merged with the current form description, if one exists.

- That MAPIFORM_INSTALL_OVERWRITEONCONFLICT is ignored when MAPI_DIALOG is present.
- That the absence of MAPIFORM_INSTALL_OVERWRITEONCONFLICT in the flag set means that a merge will be done. Any new platforms in the .CFG file not currently present in the form description will be installed and no other changes will take place.
- That the path to the form configuration file is a Unicode string, if the MAPI_UNICODE flag is set.

Form library providers implementing **InstallForm** should fill in a **MAPIERROR** structure and return MAPI_E_EXTENDED_ERROR if any of the following conditions occur:

- The configuration file is not found.
- The configuration file is not readable.
- The configuration file is invalid.

Clients should call **IMAPIFormContainer::GetLastError** if **InstallForm** returns MAPI_E_EXTENDED_ERROR and should check the returned **MAPIERROR** structure to determine the condition causing the error.

IMAPIFormContainer::RemoveForm

The **IMAPIFormContainer::RemoveForm** method removes a particular form from a form container.

HRESULT RemoveForm(
  LPCSTR *szMessageClass*
)

Parameters

szMessageClass
  Input parameter containing a string naming the message class of the form to be removed from the form container. Message class names are always ANSI strings, never Unicode.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

MAPI_E_NOT_FOUND
  The message class passed in the *szMessageClass* parameter does not match the message class for any form in the form container.

IMAPIFormContainer::ResolveMessageClass

The **IMAPIFormContainer::ResolveMessageClass** method resolves a message class to its form within a form container and returns a form information object for that form.

HRESULT ResolveMessageClass(
  LPCSTR *szMessageClass*,
  ULONG *ulFlags*,
  LPMAPIFORMINFO FAR * *ppforminfo*
)

Parameters

szMessageClass
  Input parameter containing a string naming the message class being resolved.

ulFlags
  Input parameter containing a bitmask of flags that controls how the message class is resolved. The following flag can be set:

  MAPIFORM_EXACTMATCH
    Indicates only message class strings that are an exact match should be resolved.

ppforminfo
  Output parameter pointing to a variable where a pointer to the returned form information object is stored.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

MAPI_E_NOT_FOUND
  The message class passed in the *szMessageClass* parameter does not match the message class for any form in the form container.

Remarks

Client applications call the **IMAPIFormContainer::ResolveMessageClass** method to resolve a message class to a form within a form container. The form information object returned in the *ppforminfo* parameter provides further access to the properties of the form with the given message class.

To resolve a message class to a form, a client passes in the name of the message class to be resolved, for example IPM.HelpDesk.Software. To force the resolution to be exact – that is, to prevent resolution to a superclass of the message class – the MAPIFORM_EXACTMATCH flag can be passed in the *ulFlags* parameter.

Message class names are always ANSI strings, never Unicode.

The class identifier for the resolved message class is returned as part of the form information object. A client should not work on the assumption that the class identifier exists in the OLE library until after the client has called either the **IMAPIFormMgr::PrepareForm** method or the **IMAPIFormMgr::CreateForm** method.

See Also

IMAPIFormInfo : IMAPIProp interface, **IMAPIFormMgr::CreateForm** method, **IMAPIFormMgr::PrepareForm** method

IMAPIFormContainer::ResolveMultipleMessageClasses

The **IMAPIFormContainer::ResolveMultipleMessageClasses** method resolves a group of message classes to their forms within a form container and returns an array of form information objects for those forms.

HRESULT ResolveMultipleMessageClasses (
  LPSMESSAGECLASSARRAY *pMsgClassArray***,**
  ULONG *ulFlags***,**
  LPSMAPIFORMINFOARRAY FAR * *ppfrminfoarray*
)

Parameters

pMsgClassArray
  Input parameter pointing to an array containing the names of the message classes to resolve.

ulFlags
  Input parameter containing a bitmask of flags that controls how the message classes are resolved. The following flag can be set:

  MAPIFORM_EXACTMATCH
    Indicates only message class strings that are an exact match should be resolved.

ppfrminfoarray
  Output parameter pointing to a variable where the pointer to an array of form information objects is stored. If a client application passes NULL in the *pMsgClassArray* parameter, the *ppfrminfoarray* parameter contains form information objects for all forms in the container.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

Remarks

Client applications call the **IMAPIFormContainer::ResolveMultipleMessageClasses** method to resolve a group of message classes to forms within a form container. The array of form information objects returned in *ppfrminfoarray* provides further access to each of the forms' properties.

To resolve a group of message classes to forms, a client passes in an array of message class names to be resolved. To force the resolution to be exact – that is, to prevent resolution to a superclass of the message class – the MAPIFORM_EXACTMATCH flag can be passed in the *ulFlags* parameter.

Message class names are always ANSI strings, never Unicode.

If a message class cannot be resolved to a form, NULL is returned for that message class in the form information array. Therefore, even if the method returns S_OK, clients should not work on the assumption that all message classes have been successfully resolved. Instead, clients should check the values in the returned array.

See Also

IMAPIFormContainer::ResolveMessageClass method

IMAPIFormFactory : IUnknown

The **IMAPIFormFactory** interface supports the use of configurable run-time forms in distributed computing environments. This interface is based on the OLE **IClassFactory** interface, and objects implementing **IMAPIFormFactory** should also inherit from **IClassFactory**. For more information on **IClassFactory**, see the *OLE Programmer's Reference.*

At a Glance

Specified in header file:	MAPIFORM.H
Object that supplies this interface:	Form factory object
Corresponding pointer type:	LPMAPIFORMFACTORY
Implemented by:	Form servers
Called by:	Form viewers

Vtable Order

CreateClassFactory	Returns a class factory object for a form.
LockServer	Keeps an open form server in memory.

IMAPIFormFactory::CreateClassFactory

The **IMAPIFormFactory::CreateClassFactory** method returns a class factory object for a form.

```
HRESULT CreateClassFactory(
    REFCLSID clsidForm,
    ULONG ulFlags,
    LPCLASSFACTORY FAR * lppClassFactory
  )
```

Parameters

clsidForm
: Input parameter containing the message class identifier of the form for which to create the class factory.

ulFlags
: Reserved; must be zero.

lppClassFactory
: Output parameter pointing to a variable where the returned class factory object is stored.

Return Values

S_OK
: The call succeeded and has returned the expected value or values.

Remarks

Form viewers call the **IMAPIFormFactory::CreateClassFactory** method to obtain a class factory for a specific form from a form run-time package. **CreateClassFactory** can return the same object for **CreateClassFactory** on multiple calls for the same message class identifier; creating a new instance is not required. The class factory returned is used to generate a new instance of the form.

IMAPIFormFactory::LockServer

The **IMAPIFormFactory::LockServer** method keeps an open form server in memory.

HRESULT LockServer(
  ULONG *ulFlags***,**
  ULONG *fLockServer*
)

Parameters

ulFlags
  Reserved; must be zero.

fLockServer
  Input parameter containing a variable set to TRUE if the lock count is to be incremented, and FALSE otherwise.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

Remarks

Form viewers call the **IMAPIFormFactory::LockServer** method to keep an open form server application in memory. Keeping the form server in memory improves its performance when form objects are created and released frequently.

IMAPIFormInfo : IMAPIProp

The **IMAPIFormInfo** interface gives client applications access to useful properties particular to form definition. By keeping form information in a separate object, the form library provider can describe a form to a client without activating the form.

Unlike most interfaces defined in the MAPIFORM.H header file, **IMAPIFormInfo** inherits from the **IMAPIProp** interface, as it exports most form information through calls to the **IMAPIProp::GetProps** method.

At a Glance

Specified in header file:	MAPIFORM.H
Object that supplies this interface:	Form information object
Corresponding pointer type:	LPMAPIFORMINFO
Implemented by:	Form library providers
Transaction model:	Non-transacted
Called by:	Client applications

Vtable Order

CalcFormPropSet	Returns a pointer to the complete set of properties used by a form.
CalcVerbSet	Returns a pointer to the complete set of verbs used by a form.
MakeIconFromBinary	Builds an icon from an icon property of a form.
SaveForm	Saves a description of a particular form in a configuration file.
OpenFormContainer	Returns a pointer to the form container in which a particular form is installed.

IMAPIFormInfo::CalcFormPropSet

The **IMAPIFormInfo::CalcFormPropSet** method returns a pointer to the complete set of properties used by a form.

HRESULT CalcFormPropSet (
  ULONG *ulFlags*,
  LPMAPIFORMPROPARRAY FAR * *ppFormPropArray*
)

Parameters

ulFlags
  Input parameter containing a bitmask of flags that controls the type of the returned strings. The following flag can be set:

  MAPI_UNICODE
    Indicates the returned strings are in Unicode format. If the MAPI_UNICODE flag is not set, the strings are in ANSI format.

ppFormPropArray
  Output parameter pointing to a variable where the pointer to the returned **SMAPIFormPropArray** structure is stored.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

IMAPIFormInfo::CalcVerbSet

The **IMAPIFormInfo::CalcVerbSet** method returns a pointer to the complete set of verbs used by a form.

HRESULT CalcVerbSet(
  ULONG *ulFlags*,
  LPMAPIVERBARRAY FAR * *ppMAPIVerbArray*
)

Parameters

ulFlags
  Input parameter containing a bitmask of flags that controls the type of the returned strings. The following flag can be set:

  MAPI_UNICODE
    Indicates the returned strings are in Unicode format. If the MAPI_UNICODE flag is not set, the strings are in ANSI format.

ppMAPIVerbArray
  Output parameter pointing to a variable where the pointer to the returned **SMAPIVerbArray** structure is stored. The **SMAPIVerbArray** structure contains the form's verbs.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

Remarks

Client applications call the **IMAPIFormInfo::CalcVerbSet** method to obtain a pointer to the set of verbs used by a form. Within the **SMAPIVerbArray** structure returned in the *ppMAPIVerbArray* parameter, the verbs are returned in order of index number; each verb's index is found in its **lVerb** member.

See Also

SMAPIVerbArray structure

IMAPIFormInfo::MakeIconFromBinary

The **IMAPIFormInfo::MakeIconFromBinary** method builds an icon from an icon property of a form.

HRESULT MakeIconFromBinary(
  ULONG *nPropID***,**
  HICON FAR * *phicon*
)

Parameters

nPropID
  Input parameter containing the property identifier for an icon property.

phicon
  Output parameter pointing to the returned icon.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

Remarks

Client applications call the **IMAPIFormInfo::MakeIconFromBinary** method to build an icon from an icon property of a form. In the *nPropID* parameter, **MakeIconFromBinary** takes the property identifier of any of the icon properties of a form. Using this property identifier, it builds an icon that can be displayed in table views that include property columns for icons.

IMAPIFormInfo::OpenFormContainer

The **IMAPIFormInfo::OpenFormContainer** method returns a pointer to the form container in which a particular form is installed.

HRESULT OpenFormContainer (
  LPMAPIFORMCONTAINER FAR * *ppformcontainer*
)

Parameters

ppformcontainer
  Output parameter pointing to a variable where the pointer to the returned form container object is stored.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

IMAPIFormInfo::SaveForm

The **IMAPIFormInfo::SaveForm** method saves a description of a particular form in a configuration file.

HRESULT SaveForm(
  LPCTSTR *szFileName*
)

Parameters

szFileName
 Input parameter containing a string naming the form descriptor message file where the form description is saved. This filename must have the .FDM extension.

Return Values

S_OK
 The call succeeded and has returned the expected value or values.

MAPI_E_EXTENDED_ERROR
 The configuration file could not be written. To get the **MAPIERROR** structure associated with the error, call the **IMAPIProp::GetLastError** method.

MAPI_E_NO_SUPPORT
 SaveForm was probably called to save a form in the local form container. **SaveForm** is not supported on the local form container.

Remarks

Client applications call the **IMAPIFormInfo::SaveForm** method to save a description of the current form in the file with the given filename. **SaveForm** creates a configuration file; the filename passed in the *szFileName* parameter must have the extension .FDM.

Forms can be reinstalled by being selected from a list of form descriptor messages in a dialog box displayed by form library providers. The recommended extension for form descriptor messages is .FDM.

Clients should call **IMAPIProp::GetLastError** if **SaveForm** returns MAPI_E_EXTENDED_ERROR and should check the returned **MAPIERROR** structure to determine the condition causing the error.

IMAPIFormMgr : IUnknown

The **IMAPIFormMgr** interface is used by form viewers to get information about and activate form servers.

At a Glance

Specified in header file:	MAPIFORM.H
Object that supplies this interface:	Form manager object
Corresponding pointer type:	LPMAPIFORMMGR
Implemented by:	Form library providers
Called by:	Form viewers

Vtable Order

LoadForm	Launches a form to open an existing message.
ResolveMessageClass	Resolves a message class to its form within a form container and returns a form information object for that form.
ResolveMultipleMessageClasses	Resolves a group of message classes to their forms within a form container and returns an array of form information objects for those forms.
CalcFormPropSet	Returns an array of the properties used by a group of forms.
CreateForm	Launches a form to create a new message based on that form.
SelectForm	Presents a dialog box that enables the user to select a form and returns a form information object describing that form.
SelectMultipleForms	Presents a dialog box that enables the user to select multiple forms and returns an array of form information objects describing those forms.
SelectFormContainer	Presents a dialog box that enables the user to select a form container and returns an interface for the container object the user selected.
OpenFormContainer	Opens an **IMAPIFormContainer** interface for a specific form container.
PrepareForm	Downloads a form for launching.
IsInConflict	Determines whether a form can handle its own message conflicts.

IMAPIFormMgr::CalcFormPropSet

The **IMAPIFormMgr::CalcFormPropSet** method returns an array of the properties used by a group of forms.

HRESULT CalcFormPropSet (
  LPSMAPIFORMINFOARRAY *pfrminfoarray*,
  ULONG *ulFlags*,
  LPMAPIFORMPROPARRAY FAR * *ppResults*
)

Parameters

pfrminfoarray
Input parameter pointing to an array of form information objects identifying the forms for which to return properties.

ulFlags
Input parameter containing a bitmask of flags that controls how the property array in the *ppResults* parameter is returned. The following flags can be set:

FORMPROPSET_INTERSECTION
Indicates the returned array contains the intersection of the form's properties.

FORMPROPSET_UNION
Indicates the returned array contains the union of the form's properties.

MAPI_UNICODE
Indicates the strings returned in the array are in Unicode format. If the MAPI_UNICODE flag is not set, the strings are in ANSI format.

ppResults
Output parameter pointing to a variable where the pointer to the returned **SMAPIFormPropArray** structure is stored. This structure contains the properties used by the forms.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

MAPI_E_BAD_CHARWIDTH
Either the MAPI_UNICODE flag was set and the implementation does not support Unicode, or MAPI_UNICODE was not set and the implementation only supports Unicode.

Remarks

Form viewers call the **IMAPIFormContainer::CalcFormPropSet** method to obtain an array of the properties used by a group of forms. **CalcFormPropSet** either takes an intersection or a union of these forms' property sets, depending on the flag set in the *ulFlags* parameter, and it returns an **SMAPIFormPropArray** structure containing the resulting group of properties. Passing the PROPSET_UNION flag in *ulFlags* obtains the results from a union; passing the PROPSET_INTERSECTION flag obtains the results from an intersection.

If a form viewer passes the MAPI_UNICODE flag in *ulFlags,* all strings returned are Unicode. Form library providers that do not support Unicode strings should return MAPI_E_BAD_CHARWIDTH if MAPI_UNICODE is passed.

See Also

SMAPIFormPropArray structure

IMAPIFormMgr::CreateForm

The **IMAPIFormMgr::CreateForm** method launches a form to create a new message based on that form.

HRESULT CreateForm(
  ULONG *ulUIParam*,
  ULONG *ulFlags*,
  IMAPIFormInfo *pfrminfoToActivate*,
  REFIID *refiidToAsk*,
  LPVOID FAR * *ppvObj*
)

Parameters

ulUIParam
: Input parameter containing the handle to the parent window for the progress indicator displayed while the form is launched. The *ulUIParam* parameter is ignored unless the MAPI_DIALOG flag is set in the *ulFlags* parameter.

ulFlags
: Input parameter containing a bitmask of flags that controls how the form is launched. The following flag can be set:

  MAPI_DIALOG
  : Displays a user interface to provide status or prompt the user for additional information. If this flag is not set, no user interface is displayed.

pfrminfoToActivate
: Input parameter pointing to the form information object used to launch the form.

refiidToAsk
: Input parameter pointing to the interface identifier (IID) for the interface to be returned for the form object created. The *refiidToAsk* parameter must not be NULL.

ppvObj
: Output parameter pointing to a variable where the pointer to the returned interface is stored.

Return Values

S_OK
: The call succeeded and has returned the expected value or values.

MAPI_E_NO_INTERFACE
: The requested interface is not supported by the form object.

Remarks

Form viewers call the **IMAPIFormMgr::CreateForm** method to launch a form to create a new message based on the form. **CreateForm** launches the form by creating an instance of the form server for that form as described in the given form information object. If required, **CreateForm** calls **IMAPIFormMgr::PrepareForm** to download the form server code to the user's disk.

The *pfrminfoToActivate* parameter must point to a form information object that has been correctly resolved.

After the form has been launched, the calling form viewer must set up a message using the **IPersistMessage** interface and can optionally set up a view context for the form.

See Also

IMAPIViewContext : IUnknown interface, **IPersistMessage : IUnknown** interface

IMAPIFormMgr::IsInConflict

The **IMAPIFormMgr::IsInConflict** method determines whether a form can handle its own message conflicts.

HRESULT IsInConflict(
  ULONG *ulMessageFlags***,**
  ULONG *ulMessageStatus***,**
  LPCSTR *szMessageClass*
  LPMAPIFOLDER *pFolderFocus*
)

Parameters

ulMessageFlags
  Input parameter pointing to a bitmask of flags, copied from the PR_MESSAGE_FLAGS property of a message, that indicates the current state of the message.

ulMessageStatus
  Input parameter containing a bitmask of client-defined or provider-defined flags, copied from the PR_MSG_STATUS property of a message, that provides further information on the state of the message.

szMessageClass
  Input parameter containing a string naming the message's message class.

pFolderFocus
  Input parameter pointing to the folder that contains the message. The *pFolderFocus* parameter can be NULL if such a folder doesn't exist, for example, in a case where the message is embedded in another message.

Return Values

S_OK
  The form does not handle its own message conflicts.

S_FALSE
  The form handles its own message conflicts, or the message for which information was passed is not in conflict.

Remarks

Form viewers call the **IMAPIFormMgr::IsInConflict** method to discover if a particular form does not handle its own message conflicts. **IsInConflict** checks the bitmasks in the *ulMessageFlags* and *ulMessageStatus* parameters for the presence of a conflict flag. If a conflict flag is set, **IsInConflict** resolves the message class passed in the *szMessageClass* parameter and returns S_OK if the form does not handle its own conflicts. **IsInConflict** returns S_FALSE if the form handles its own conflicts.

A form that does not handle its own conflicts must be launched using the **IMAPIFormMgr::LoadForm** method and cannot reuse an existing form object.

See Also

IMAPIFormAdviseSink::OnActivateNext method, PR_MESSAGE_FLAGS property, PR_MSG_STATUS property

IMAPIFormMgr::LoadForm

The **IMAPIFormMgr::LoadForm** method launches a form to open an existing message.

HRESULT LoadForm(
  ULONG *ulUIParam*,
  ULONG *ulFlags*,
  LPCSTR *lpszMessageClass*,
  ULONG *ulMessageStatus*,
  ULONG *ulMessageFlags*,
  LPMAPIFOLDER *pFolderFocus*,
  LPMAPIMESSAGESITE *pMessageSite*,
  LPMESSAGE *pmsg*,
  LPMAPIVIEWCONTEXT *pViewContext*,
  REFIID *riid*,
  LPVOID FAR * *ppvObj*
)

Parameters

ulUIParam
  Input parameter containing the handle of the parent window for the progress indicator displayed while the form is launched. The *ulUIParam* parameter is ignored unless the MAPI_DIALOG flag is set in the *ulFlags* parameter.

ulFlags
  Input parameter containing a bitmask of flags that controls how the form is launched. The following flags can be set:

  MAPI_DIALOG
    Displays a user interface to provide status or prompt the user for additional information. If this flag is not set, no user interface is displayed.

  MAPIFORM_EXACTMATCH
    Indicates only message class strings that are an exact match should be resolved.

lpszMessageClass
  Input parameter pointing to a string naming the message class of the message to be loaded. If NULL is passed in the *lpszMessageClass* parameter, the message class is determined from the message pointed to by the *pmsg* parameter.

ulMessageStatus
  Input parameter containing a bitmask of client- or provider-defined flags, copied from the PR_MSG_STATUS property of the message, that provides information on the state of the message. The *ulMessageStatus* parameter must be set if *lpszMessageClass* is non-null; otherwise *ulMessageStatus* is ignored.

ulMessageFlags
  Input parameter pointing to a bitmask of flags, copied from the PR_MESSAGE_FLAGS property of the message, that indicates the current state of the message. The *ulMessageFlags* parameter must be set if *lpszMessageClass* is non-null; otherwise *ulMessageFlags* is ignored.

pFolderFocus
  Input parameter pointing to the folder that directly contains the message. The *pFolderFocus* parameter can be NULL if such a folder doesn't exist, for example if the message is embedded in another message.

pMessageSite
  Input parameter pointing to the message site of the message.

pmsg
  Input parameter pointing to the message.

pViewContext
: Input parameter pointing to the view context for the message. The *pViewContext* parameter can be NULL.

riid
: Input parameter containing the interface identifier (IID) of the interface to be used for the returned form object. The *refiidToAsk* parameter must not be NULL.

ppvObj
: Output parameter pointing to a variable where the pointer to the returned interface is stored.

Return Values

S_OK
: The call succeeded and has returned the expected value or values.

MAPI_E_NO_INTERFACE
: The form does not support the requested interface.

MAPI_E_NOT_FOUND
: The message class passed in *lpszMessageClass* does not match the message class for any form in the form library.

Remarks

Form viewers call the **IMAPIFormMgr::LoadForm** method to launch a form for an existing message. **LoadForm** launches the form object, loads the message into the form object, sets up the appropriate view context if necessary, and returns the requested interface for the form object.

The *pFolderFocus* parameter points to the folder containing the message. If the message is embedded within another message, *pFolderFocus* should be NULL. If NULL is passed in *lpszMessageClass*, then the implementation obtains the message's message class and its PR_MSG_STATUS and PR_MESSAGE_FLAGS properties. If a message class string is provided in *lpszMessageClass*, then the implementation must use the values as *ulMessageStatus* and *ulMessageFlags*.

See Also

PR_MESSAGE_FLAGS property, PR_MSG_STATUS property

IMAPIFormMgr::OpenFormContainer

The **IMAPIFormMgr::OpenFormContainer** method opens an **IMAPIFormContainer** interface for a specific form container.

HRESULT OpenFormContainer(
  HFRMREG *hfrmreg***,**
  LPUNKNOWN *lpunk***,**
  LPMAPIFORMCONTAINER FAR * *lppfcnt*
)

Parameters

hfrmreg
  Input parameter containing an **HFRMREG** enumeration indicating the form library to open – that is, the form container to open. A **HFRMREG** enumeration is an enumeration specific to a form library provider. Possible **HFRMREG** values include:

  HFRMREG_DEFAULT
    Indicates a convenient form container.

  HFRMREG_FOLDER
    Indicates a folder container.

  HFRMREG_PERSONAL
    Indicates the container for the default message store.

  HFRMREG_LOCAL
    Indicates the local form container.

lpunk
  Input parameter pointing to the object for which the interface is opened. The *lpunk* parameter must be NULL unless the value for the *hfrmreg* parameter requires an object pointer.

lppfcnt
  Output parameter pointing to a variable where the pointer to the returned form container object is stored.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

MAPI_E_NO_INTERFACE
  The object pointed to by *lpUnk* does not support the required interface.

Remarks

Form viewers call the **IMAPIFormMgr::OpenFormContainer** method to open an **IMAPIFormContainer** interface for a specific form container. This interface can then be used for installing forms into and removing forms from a form container.

If the value in *hfrmreg* is HFRMREG_FOLDER, which indicates a folder container, the interface identifier used in *lpunk* must be non-null and must allow **QueryInterface** calls to an **IMAPIFolder** interface.

To open the local form container, a call to **OpenFormContainer** or the **MAPIOpenLocalFormContainer** function must be used; the **IMAPIFormMgr::SelectFormContainer** method cannot be used to enable the user to select the local form container.

See Also

IMAPIFormContainer::InstallForm method, **IMAPIFormMgr::SelectFormContainer** method,

MAPIOpenLocalFormContainer function

IMAPIFormMgr::PrepareForm

The **IMAPIFormMgr::PrepareForm** method downloads a form for launching.

HRESULT PrepareForm(
  ULONG *ulUIParam***,**
  ULONG *ulFlags***,**
  LPMAPIFORMINFO *pfrmiInfo*
)

Parameters

ulUIParam
  Input parameter containing the handle of the parent window for the progress indicator displayed while the form is downloaded. The *ulUIParam* parameter is ignored unless the MAPI_DIALOG flag is set in the *ulFlags* parameter.

ulFlags
  Input parameter containing a bitmask of flags that controls how the form is downloaded. The following flag can be set:

  MAPI_DIALOG
    Displays a user interface to provide status or prompt the user for additional information. If this flag is not set, no user interface is displayed.

pfrmiInfo
  Input parameter pointing to a form information object for the form to be downloaded.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

Remarks

Form viewers call the **IMAPIFormMgr::PrepareForm** method to download a form from a form container for launching. Most form viewers do not need to call **PrepareForm** as both the **IMAPIFormMgr::CreateForm** and **IMAPIFormMgr::LoadForm** methods call **PrepareForm** if necessary.

PrepareForm can be used to obtain the dynamic-link libraries (DLLs) and other files associated with a form to modify them. If the modified form is loaded back into its form container, it must be reinstalled.

See Also

IMAPIFormMgr::CreateForm method, **IMAPIFormMgr::LoadForm** method

IMAPIFormMgr::ResolveMessageClass

The **IMAPIFormMgr::ResolveMessageClass** method resolves a message class to its form within a form container and returns a form information object for that form.

HRESULT ResolveMessageClass(
  LPCSTR *szMsgClass*,
  ULONG *ulFlags*,
  LPMAPIFOLDER *pFolderFocus*,
  LPMAPIFORMINFO FAR * *ppResult*
)

Parameters

szMsgClass
  Input parameter containing a string naming the message class being resolved.

ulFlags
  Input parameter containing a bitmask of flags that controls how the message class is resolved. The following flag can be set:

  MAPIFORM_EXACTMATCH
    Indicates only message class strings that are an exact match should be resolved.

pFolderFocus
  Input parameter pointing to the folder containing the message being resolved. The *pFolderFocus* parameter can be NULL.

ppResult
  Output parameter pointing to a variable where the pointer to a returned form information object is stored.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

MAPI_E_NOT_FOUND
  The message class passed in the *szMsgClass* parameter does not match the message class for any form in the form library.

Remarks

Form viewers call the **IMAPIFormMgr::ResolveMessageClass** method to resolve a message class to its form within a form container. The form information object returned in the *ppResult* parameter provides further access to the properties of the form with the given message class.

To resolve a message class to a form, a form viewer passes in the name of the message class to be resolved, for example IPM.HelpDesk.Software. To force the resolution to be exact − that is, to prevent resolution to a superclass of the message class − the MAPIFORM_EXACTMATCH flag can be passed in the *ulFlags* parameter. The *pFolderFocus* parameter should point to the folder containing the message whose class is being resolved; it can, however, be NULL, in which case the message-class resolution process does not search a folder container.

The order of the containers searched depends on the implementation of the form library provider. The default library searches first the local container, then the folder container for the passed-in folder, the personal form container, and finally the organization container.

Message class names are always ANSI strings, never Unicode.

The class identifier for the resolved message class is returned as part of the form information object. A form viewer should not work on the assumption that the class identifier exists in the OLE library until after the form viewer has called either the **IMAPIFormMgr::PrepareForm** method or the **IMAPIFormMgr::CreateForm** method.

See Also

IMAPIFormInfo : IMAPIProp interface, **IMAPIFormMgr::CreateForm** method, **IMAPIFormMgr::PrepareForm** method

IMAPIFormMgr::ResolveMultipleMessageClasses

The **IMAPIFormMgr::ResolveMultipleMessageClasses** method resolves a group of message classes to their forms within a form container and returns an array of form information objects for those forms.

HRESULT ResolveMultipleMessageClasses(
  LPSMESSAGECLASSARRAY *pMsgClasses*,
  ULONG *ulFlags*,
  LPMAPIFOLDER *pFolderFocus*,
  LPSMAPIFORMINFOARRAY FAR * *ppfrminfoarray*
)

Parameters

pMsgClasses
Input parameter pointing to an array containing the names of the message classes to resolve.

ulFlags
Input parameter containing a bitmask of flags that controls how the message classes are resolved. The following flag can be set:

MAPIFORM_EXACTMATCH
Indicates only message class strings that are an exact match should be resolved.

pFolderFocus
Input parameter pointing to the folder containing the form whose message class is being resolved. The *pFolderFocus* parameter can be NULL.

ppfrminfoarray
Output parameter pointing to a variable where the pointer to an array of form information objects is stored. If a form viewer passes NULL in the *pMsgClasses* parameter, the *ppfrminfoarray* parameter contains form information objects for all forms in the container.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

Remarks

Form viewers call the **IMAPIFormMgr::ResolveMultipleMessageClasses** method to resolve a group of message classes to forms within a form container. The array of form information objects returned in *ppfrminfoarray* provides further access to each of the forms' properties.

To resolve a group of message classes to forms, a form viewer passes in an array of message class names to be resolved. To force the resolution to be exact – that is, to prevent resolution to a superclass of the message class – the MAPIFORM_EXACTMATCH flag can be passed in the *ulFlags* parameter.

Message class names are always ANSI strings, never Unicode.

If a message class cannot be resolved to a form, NULL is returned for that message class in the form information array. Therefore, even if the method returns S_OK, form viewers should not work on the assumption that all message classes have been successfully resolved. Instead, form viewers should check the values in the returned array.

See Also

IMAPIFormMgr::ResolveMessageClass method

IMAPIFormMgr::SelectForm

The **IMAPIFormMgr::SelectForm** method presents a dialog box that enables the user to select a form and returns a form information object describing that form.

HRESULT SelectForm(
  ULONG *ulUIParam*,
  ULONG *ulFlags*,
  LPCTSTR *pszTitle*,
  LPMAPIFOLDER *pfld*,
  LPMAPIFORMINFO FAR * *ppfrminfoReturned*
)

Parameters

ulUIParam
  Input parameter containing the handle of the parent window for the dialog box displayed.

ulFlags
  Input parameter containing a bitmask of flags that controls the type of the passed-in strings. The following flag can be set:

  MAPI_UNICODE
    Indicates the passed-in strings are in Unicode format. If the MAPI_UNICODE flag is not set, the strings are in ANSI format.

pszTitle
  Input parameter pointing to a string containing the caption of the dialog box. If the *pszTitle* parameter is NULL, the form library provider supplies a default caption.

pfld
  Input parameter pointing to the folder from which to select the form. If the *pfld* parameter is NULL, the form can be selected from the local, personal, or organization form container.

ppfrminfoReturned
  Output parameter pointing to a pointer to the returned form information object.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

MAPI_E_BAD_CHARWIDTH
  Either the MAPI_UNICODE flag was set and the implementation does not support Unicode, or MAPI_UNICODE was not set and the implementation only supports Unicode.

MAPI_E_USER_CANCEL
  The user canceled the operation, typically by clicking the **Cancel** button in the dialog box.

Remarks

Form viewers call the **IMAPIFormMgr::SelectForm** method to first present a dialog box that enables the user to select a form and then to retrieve a form information object describing the selected form. The dialog box constrains the user to select a single form.

The **SelectForm** dialog box only displays forms that are not hidden, that is, that have their hidden properties clear. If a form viewer passes the MAPI_UNICODE flag in the *ulFlags* parameter, all strings are Unicode. Form library providers that do not support Unicode strings should return MAPI_E_BAD_CHARWIDTH if MAPI_UNICODE is passed.

IMAPIFormMgr::SelectMultipleForms

The **IMAPIFormMgr::SelectMultipleForms** method presents a dialog box that enables the user to select multiple forms and returns an array of form information objects describing those forms.

HRESULT SelectMultipleForms(
  ULONG *ulUIParam*,
  ULONG *ulFlags*,
  LPCTSTR *pszTitle*,
  LPMAPIFOLDER *pfld*,
  LPMAPIFORMINFOARRAY *pfrminfoarray*,
  LPMAPIFORMINFOARRAY FAR * *ppfrminfoarray*
)

Parameters

ulUIParam
  Input parameter containing the handle of the parent window for the dialog box displayed.

ulFlags
  Input parameter containing a bitmask of flags that controls the type of the passed-in strings. The following flag can be set:

  MAPI_UNICODE
    Indicates the passed-in strings are in Unicode format. If the MAPI_UNICODE flag is not set, the strings are in ANSI format.

pszTitle
  Input parameter pointing to a string containing the caption of the dialog box. If the *pszTitle* parameter is NULL, the form library provider that provides the forms supplies a default caption.

pfld
  Input parameter pointing to the folder from which to select the forms. If the *pfld* parameter is NULL, the forms are selected from the local, personal, or organization form container.

pfrminfoarray
  Input parameter pointing to an array of form information objects that are preselected for the user.

ppfrminfoarray
  Output parameter pointing to a variable where the pointer to the returned array of form information objects is stored.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

MAPI_E_BAD_CHARWIDTH
  Either the MAPI_UNICODE flag was set and the implementation does not support Unicode, or MAPI_UNICODE was not set and the implementation only supports Unicode.

MAPI_E_USER_CANCEL
  The user canceled the operation, typically by clicking the **Cancel** button in the dialog box.

Remarks

Form viewers call the **IMAPIFormMgr::SelectMultipleForms** method to first present a dialog box that enables the user to select multiple forms and then to retrieve an array of form information objects describing the selected forms.

The **SelectMultipleForms** dialog box displays all forms, whether or not they are hidden − that is, whether or not their hidden properties are clear. If a form viewer passes the MAPI_UNICODE flag in the *ulFlags* parameter, all strings are Unicode. Form library providers that do not support Unicode strings should return MAPI_E_BAD_CHARWIDTH if MAPI_UNICODE is passed.

IMAPIFormMgr::SelectFormContainer

The **IMAPIFormMgr::SelectFormContainer** method presents a dialog box that enables the user to select a form container and returns an interface for the container object the user selected.

HRESULT SelectFormContainer(
  ULONG *ulUIParam***,**
  ULONG *ulFlags***,**
  LPMAPIFORMCONTAINER FAR * *lppfcnt*
)

Parameters

ulUIParam
  Input parameter containing the handle of the parent window for the dialog box displayed.

ulFlags
  Input parameter containing a bitmask of flags that controls how the form library is selected − that is, how the form container is selected. The following flags can be set:

  MAPIFORM_SELECT_ALL_REGISTRIES
    Indicates selection can be made from all containers. This type of selection is the default.

  MAPIFORM_SELECT_FOLDER_REGISTRY_ONLY
    Indicates selection can be made only from folder containers.

  MAPIFORM_SELECT_NON_FOLDER_REGISTRY_ONLY
    Indicates selection can be made only from containers not associated with folders.

lppfcnt
  Output parameter pointing to a variable where the pointer to the returned interface is stored. This interface is for the container object selected by the user.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

Remarks

Form viewers commonly call the **IMAPIFormMgr::SelectFormContainer** method to select a form container into which a form is installed. **SelectFormContainer** cannot be used to select the local form container, which has the value HFRMREG_LOCAL.

IMAPIMessageSite : IUnknown

The **IMAPIMessageSite** interface manipulates messages and is implemented by the form viewer code that responds to such manipulation.

At a Glance

Specified in header file:	MAPIFORM.H
Object that supplies this interface:	Message site object
Corresponding pointer type:	LPMAPIMESSAGESITE
Implemented by:	Form viewers
Called by:	Form objects

Vtable Order

GetSession	Returns the MAPI session in which the current message was created or opened.
GetStore	Returns the message store containing the current message, if such a store exists.
GetFolder	Returns the folder in which the current message was created or opened, if such a folder exists.
GetMessage	Returns the current message.
GetFormManager	Returns a form manager interface, which a form server can use to launch another form server.
NewMessage	Creates a new message.
CopyMessage	Copies the current message to a folder.
MoveMessage	Moves the current message to a folder.
DeleteMessage	Deletes the current message.
SaveMessage	Requests that the current message be saved.
SubmitMessage	Requests that the current message be submitted to the MAPI spooler.
GetSiteStatus	Returns information from a message site object about the message site's capabilities for the current message.

IMAPIMessageSite::CopyMessage

The **IMAPIMessageSite::CopyMessage** method copies the current message to a folder.

HRESULT CopyMessage(
   LPMAPIFOLDER *pFolderDestination*
)

Parameters

pFolderDestination
   Input parameter pointing to the folder where the message is copied.

Return Values

S_OK
   The call succeeded and has returned the expected value or values.

MAPI_E_NO_SUPPORT
   The operation is not supported by this message site.

Remarks

Form servers call the **IMAPIMessageSite::CopyMessage** method to copy the current message to a new folder. **CopyMessage** does not change the message currently being displayed to the user, and no interface for the newly created message is returned to the form.

A typical implementation of **CopyMessage** does the following:

1. Creates the new message the current message is copied to.
2. Calls the **IPersistMessage::Save** method with a pointer to the new message in the *pMessage* parameter and FALSE in the *fSameAsLoad* parameter.
3. Calls the **IPersistMessage::SaveCompleted** method passing NULL for its *pMessage* parameter.
4. Calls the **IMAPIProp::SaveChanges** method on the new message.

See Also

IMAPIProp::SaveChanges method, **IPersistMessage::Save** method, **IPersistMessage::SaveCompleted** method

IMAPIMessageSite::DeleteMessage

The **IMAPIMessageSite::DeleteMessage** method deletes the current message.

HRESULT DeleteMessage(
  LPMAPIVIEWCONTEXT *pViewContext***,**
  LPCRECT *prcPosRect*
)

Parameters

pViewContext
  Input parameter pointing to a view context object.

prcPosRect
  Input parameter pointing to a **RECT** structure containing the current form's window size and position. The next form displayed also uses this window rectangle.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

MAPI_E_NO_SUPPORT
  The operation is not supported by this message site.

Remarks

Form servers call the **IMAPIMessageSite::DeleteMessage** method to delete the message the form is currently attached to. If a form viewer's implementation of **DeleteMessage** moves to the next message after deleting a message, it should call the **IMAPIViewContext::ActivateNext** method passing the VCDIR_DELETE flag prior to performing the actual deletion. If a form viewer's implementation of **DeleteMessage** moves the deleted message, for example to a Deleted Items folder, it must save changes to the message if the message has been modified.

Following the return of **DeleteMessage**, form objects must check for a new message and then dismiss themselves if none exists. To determine whether a message **DeleteMessage** acted on is deleted or moved to a Deleted Items folder, a form server can call the **IMAPIMessageSite::GetSiteStatus** method and check if the DELETE_IS_MOVE flag was returned.

A typical implementation of **DeleteMessage** does the following:

1. If moving the message, it calls the **IPersistMessage::Save** method passing NULL for its *pMessage* parameter and TRUE for its *fSameAsLoad* parameter.
2. It calls the **IMAPIViewContext::ActivateNext** method passing the VCDIR_DELETE flag in its *ulDir* parameter.
3. If the **ActivateNext** call failed, it returns. If **ActivateNext** returned S_FALSE, it calls the **IPersistMessage::HandsOffMessage** method.
4. It deletes or moves the message.

To obtain the **RECT** structure used by a form's window, call the Windows **GetWindowRect** function.

See Also

IMAPIMessageSite::GetSiteStatus method, **IMAPIViewContext::ActivateNext** method, **IPersistMessage::HandsOffMessage** method, **IPersistMessage::Save** method

IMAPIMessageSite::GetFolder

The **IMAPIMessageSite::GetFolder** method returns the folder in which the current message was created or opened, if such a folder exists.

HRESULT GetFolder(
  LPMAPIFOLDER FAR * *ppFolder*
)

Parameters

ppFolder
  Output parameter pointing to a pointer to the returned folder.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

S_FALSE
  No folder exists for the message.

IMAPIMessageSite::GetFormManager

The **IMAPIMessageSite::GetFormManager** method returns a form manager interface, which a form server can use to launch another form server.

HRESULT GetFormManager(
  LPMAPIFORMMGR FAR * *ppFormMgr*
)

Parameters

ppFormMgr
  Output parameter pointing to a pointer to the returned form manager interface.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

IMAPIMessageSite::GetMessage

The **IMAPIMessageSite::GetMessage** method returns the current message.

HRESULT GetMessage(
  LPMESSAGE FAR * *ppmsg*
)

Parameters

ppmsg
  Output parameter pointing to a pointer to the returned interface for the message.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

S_FALSE
  No message currently exists for the calling form.

Remarks

Forms call the **IMAPIMessageSite::GetMessage** method to obtain a message interface for the current message. The current message is the same message as was previously passed in the **IPersistMessage::InitNew**, **IPersistMessage::Load,** or **IPersistMessage::SaveCompleted** method.

GetMessage returns S_FALSE if no message currently exists. This state can occur after calls to the **IPersistMessage::HandsOffMessage** method or before the next call to **IPersistMessage::Load** or **IPersistMessage::SaveCompleted** is made.

See Also

IPersistMessage : IUnknown interface

IMAPIMessageSite::GetSession

The **IMAPIMessageSite::GetSession** method returns the MAPI session in which the current message was created or opened.

```
HRESULT GetSession(
  LPMAPISESSION FAR * ppSession
 )
```

Parameters

ppSession
Output parameter pointing to a variable where the pointer to the returned session object is stored.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

S_FALSE
No session exists for the current message.

IMAPIMessageSite::GetSiteStatus

The **IMAPIMessageSite::GetSiteStatus** method returns information from a message site object about the message site's capabilities for the current message.

HRESULT GetSiteStatus(
  ULONG FAR * *lpulStatus*
)

Parameters

lpulStatus
: Output parameter pointing to a variable in which a bitmask of flags giving information on message status is returned. The following flags can be set:

VCSTATUS_COPY	Indicates the message can be copied.
VCSTATUS_DELETE	Indicates the message can be deleted.
VCSTATUS_DELETE_IS_MOVE	Indicates the message is not deleted but moved to a Deleted Items folder.
VCSTATUS_MOVE	Indicates the message can be moved.
VCSTATUS_NEW_MESSAGE	Indicates a new message can be created.
VCSTATUS_SAVE	Indicates the message can be saved.
VCSTATUS_SUBMIT	Indicates the message can be submitted.

Return Values

S_OK
: The call succeeded and has returned the expected value or values.

Remarks

Forms call the **IMAPIMessageSite::GetSiteStatus** method to obtain the message site's capabilities for the current message. The flags returned in the *lpulStatus* parameter provide information on the message site; typically, a form enables or disables menu commands based on information the flags provide on the capabilities of the message site implementation. If a new message is loaded into a form by the **IPersistMessage::SaveCompleted** method or the **IPersistMessage::Load** method, the status flags must be checked. Some message sites, particularly read-only sites, do not permit messages to be saved or deleted.

See Also

IPersistMessage::Load method, **IPersistMessage::SaveCompleted** method

IMAPIMessageSite::GetStore

The **IMAPIMessageSite::GetStore** method returns the message store containing the current message, if such a store exists.

HRESULT GetStore(
  LPMDB FAR * *ppStore*
)

Parameters

ppStore
 Output parameter pointing to a variable where the pointer to the message store is stored.

Return Values

S_OK
 The call succeeded and has returned the expected value or values.

S_FALSE
 There is no store containing the message.

IMAPIMessageSite::MoveMessage

The **IMAPIMessageSite::MoveMessage** method moves the current message to a folder.

HRESULT MoveMessage(
  LPFOLDER *pFolderDestination*,
  LPMAPIVIEWCONTEXT *pViewContext*,
  LPCRECT *prcPosRect*
)

Parameters

pFolderDestination
  Input parameter pointing to the folder where the message is moved.

pViewContext
  Input parameter pointing to a view context object.

prcPosRect
  Input parameter pointing to a **RECT** structure containing the current form's window size and position. The next form displayed also uses this window rectangle.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

MAPI_E_NO_SUPPORT
  The operation is not supported by this message site.

Remarks

Forms call the **IMAPIMessageSite::MoveMessage** method to move the current message to a new folder. A form viewer's implementation of **MoveMessage** must call the **IMAPIViewContext::ActivateNext** method, passing the VCDIR_MOVE flag, prior to actually moving the message to a new folder. Following the return of **MoveMessage**, forms must check for a current message and then dismiss themselves if none exists.

To obtain the **RECT** structure used by a form's window, call the Windows **GetWindowRect** function.

See Also

IMAPIViewContext::ActivateNext method

IMAPIMessageSite::NewMessage

The **IMAPIMessageSite::NewMessage** method creates a new message.

HRESULT NewMessage(
  ULONG *fComposeInFolder***,**
  LPMAPIFOLDER *pFolderFocus***,**
  LPPERSISTMESSAGE *pPersistMessage***,**
  LPMESSAGE FAR * *ppMessage***,**
  LPMAPIMESSAGESITE FAR * *ppMessageSite***,**
  LPMAPIVIEWCONTEXT FAR * *ppViewContext*
)

Parameters

fComposeInFolder
  Input parameter containing a variable indicating in which folder the message should be composed. If the variable is FALSE, the *pFolderFocus* parameter is ignored and the form viewer can compose the message in any folder. If the variable is TRUE and NULL is passed in *pFolderFocus*, the message is composed in the current folder. If the variable is TRUE and a non-null value is passed in *pFolderFocus*, the message is composed in the folder pointed to by *pFolderFocus*.

pFolderFocus
  Input parameter pointing to the folder where the new message is created.

pPersistMessage
  Input parameter pointing to the form object for the new form.

ppMessage
  Output parameter pointing to a pointer to the new message.

ppMessageSite
  Output parameter pointing to a pointer to a message site object for the new message.

ppViewContext
  Output parameter pointing to a pointer to a view context appropriate for passing to a new form with the new message. If the form implements its own view context, NULL can be passed in the *ppViewContext* parameter.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

Remarks

Forms call the **IMAPIMessageSite::NewMessage** method to create a new message. The form uses **NewMessage** to get a new message and the associated message site from its view. It can then modify the new message and either aggregate the message site or use it message site directly.

An associated view context can also be obtained by passing in a non-null value in the *ppViewContext* parameter. This view context can be used directly, or it can be aggregated and passed to the new message. If a complete implementation is required, NULL should be passed in *ppViewContext*.

See Also

IMAPIViewContext : IUnknown interface

IMAPIMessageSite::SaveMessage

The **IMAPIMessageSite::SaveMessage** method requests that the current message be saved.

HRESULT SaveMessage()

Parameters

None.

Return Values

S_OK
: The call succeeded and has returned the expected value or values.

Remarks

Forms call the **IMAPIMessageSite::SaveMessage** method to request that a message be saved. After being saved, the message generates the usual form object behavior.

IMAPIMessageSite::SubmitMessage

The **IMAPIMessageSite::SubmitMessage** method requests that the current message be submitted to the MAPI spooler.

```
HRESULT SubmitMessage(
    ULONG ulFlags
  )
```

Parameters

ulFlags
Input parameter containing a bitmask of flags that controls how a message is submitted. The following flag can be set:

FORCE_SUBMIT
Indicates MAPI should submit the message even if it might not be sent right away.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

Remarks

Forms call the **IMAPIMessageSite::SubmitMessage** method to request that a message be submitted to the MAPI spooler. The message site should call the **IPersistMessage::HandsOffMessage** method prior to submitting the message. The message need not have previously been saved, as **SubmitMessage** should cause the message to be saved if it has been modified. Following the return of **SubmitMessage**, the form must check for a current message and then dismiss itself if none exists.

See Also

IPersistMessage::HandsOffMessage method

IMAPIProgress : IUnknown

The **IMAPIProgress** interface is used by service providers to provide client applications with information to update progress indicators, which indicate the progress toward completion of a lengthy operation, such as the copying of folders between message stores. MAPI and client applications implement progress objects and then pass those objects to provider-implemented methods, such as **IMAPIFolder::CopyFolder**, which take progress objects as input parameters.

At a Glance

Specified in header file:	MAPIDEFS.H
Object that supplies this interface:	Progress object
Corresponding pointer type:	LPMAPIPROGRESS
Implemented by:	MAPI and client applications
Called by:	Service providers

Vtable Order

Progress	Updates the progress indicator with status on the progress made toward completion of the operation in question.
GetFlags	Returns flag settings from the progress object for the level of operation on which progress information is calculated.
GetMax	Returns the maximum number of items in the operation for which progress information is displayed.
GetMin	Returns the minimum number of items in the operation for which progress information is displayed.
SetLimits	Sets the minimum number of items operated on, the maximum number of items operated on, and the flags that control how progress information is calculated for the operation in question.

IMAPIProgress::GetFlags

The **IMAPIProgress::GetFlags** method returns flag settings from the progress object for the level of operation on which progress information is calculated.

HRESULT GetFlags(
  ULONG FAR * *lpulFlags*
)

Parameters

lpulFlags
  Output parameter containing a bitmask of flags that controls the level of operation on which progress information is calculated. The following flag can be returned:

  MAPI_TOP_LEVEL
    Uses the values in the **IMAPIProgress::Progress** method's *ulCount* and *ulTotal* parameters, which indicate the item being operated on and the total items, respectively, to increment progress on the operation. When this flag is set, the values of the global lower and upper limits have to be set.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

See Also

IMAPIProgress::SetLimits method

IMAPIProgress::GetMax

The **IMAPIProgress::GetMax** method returns the value stored in **IMAPIProgress::SetLimits** showing the maximum number of items in the operation for which progress information is displayed.

HRESULT GetMax(
    ULONG FAR * *lpulMax*
)

Parameters

lpulMax
    Output parameter pointing to a variable containing the maximum number of items in the operation.

Return Values

S_OK
    The call succeeded and has returned the expected value or values.

See Also

IMAPIProgress::GetMin method, **IMAPIProgress::Progress** method, **IMAPIProgress::SetLimits** method

IMAPIProgress::GetMin

The **IMAPIProgress::GetMin** method returns the minimum value in **IMAPIProgress::SetLimits** for which progress information is displayed.

```
HRESULT GetMin(
    ULONG FAR * lpulMin
  )
```

Parameters

lpulMin
  Output parameter pointing to a variable containing the minimum number of items in the operation.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

See Also

IMAPIProgress::GetMax method, **IMAPIProgress::Progress** method, **IMAPIProgress::SetLimits** method

IMAPIProgress::Progress

The **IMAPIProgress::Progress** method updates the progress indicator with a display of the progress as it is made toward completion of the operation.

HRESULT Progress(
  ULONG *ulValue***,**
  ULONG *ulCount***,**
  ULONG *ulTotal*
)

Parameters

ulValue
 Input parameter that contains a number showing progress, calculated from the *ulCount* and *ulTotal* parameters or from the *lpulMin* and *lpulMax* parameters of the **IMAPIProgress::SetLimits** method between the global lower limit and the global upper limit.

ulCount
 Input parameter containing the current count of the items being operated on.

ulTotal
 Input parameter containing the total count of items.

Return Values

S_OK
 The call succeeded and has returned the expected value or values.

Remarks

Service providers call the **IMAPIProgress::Progress** method to update the progress indicator for the progress object passed by a client application.

If an implementation is copying all messages within a single folder, then the value in *ulTotal* should indicate the total number of messages being copied. If it is copying a folder, then the value in *ulTotal* should be the number of subfolders within the folder. If the folder to be copied contains no subfolders and only messages, the value in *ulTotal* should be set to 1.

See Also

IMAPIProgress::SetLimits method

IMAPIProgress::SetLimits

The **IMAPIProgress::SetLimits** method sets the lower limit for the number of items operated on, the maximum number of items operated on, and the flags that control how progress information is calculated for the operation in question.

HRESULT SetLimits(
  LPULONG *lpulMin*,
  LPULONG *lpulMax*,
  LPULONG *lpulFlags*
)

Parameters

lpulMin
  Input parameter pointing to a variable containing the lower limit of items in the operation.

lpulMax
  Input parameter pointing to a variable containing the upper limit of items in the operation.

lpulFlags
  Input parameter containing a bitmask of flags that controls the level of operation on which progress information is calculated. The following flag can be set:

  MAPI_TOP_LEVEL
    Uses the values in the **IMAPIProgress::Progress** method's *ulCount* and *ulTotal* parameters, which indicate the item being operated on and the total items, respectively, to increment progresson the operation. When this flag is set, the values of the global lower and upper limits have to be set.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

Remarks

Client applications implement the **IMAPIProgress::SetLimits** method to establish the values that control how operation progress is calculated for the progress indicator.

See Also

IMAPIProgress::Progress method

IMAPIProp : IUnknown

The **IMAPIProp** interface is the core interface for working with properties for all MAPI objects.

At a Glance

Specified in header file:	MAPIDEFS.H
Object that supplies this interface:	No object supplies this interface directly.
Corresponding pointer type:	LPMAPIPROP
Implemented by:	Service providers
Called by:	All applications

Vtable Order

GetLastError	Returns a **MAPIERROR** structure containing information about the last error that occurred for an object.
SaveChanges	Makes permanent any changes made to an object since the last save operation.
GetProps	Retrieves the property values of one or more properties of an object.
GetPropList	Returns a list of all properties of an object.
OpenProperty	Opens an interface for a property of an object.
SetProps	Sets the property value of one or more properties of an object.
DeleteProps	Deletes all properties in the given list.
CopyTo	Copies or moves all properties from a source object to a destination object, except for a given set of excluded properties.
CopyProps	Copies or moves a selected set of properties from a source object to a destination object.
GetNamesFromIDs	Provides property names, given a list of property identifiers.
GetIDsFromNames	Provides property identifiers, given a list of property names.

IMAPIProp::CopyProps

The **IMAPIProp::CopyProps** method copies or moves a selected set of properties from a source object to a destination object. The source object is the object on which the call to **CopyProps** is made.

HRESULT CopyProps(
  LPSPropTagArray *lpIncludeProps*,
  ULONG *ulUIParam*,
  LPMAPIPROGRESS *lpProgress*,
  LPCIID *lpInterface*,
  LPVOID *lpDestObj*,
  ULONG *ulFlags*,
  LPSPropProblemArray FAR * *lppProblems*
)

Parameters

lpIncludeProps
Input parameter pointing to an **SPropTagArray** structure holding a counted array of property tags indicating the properties to copy or move. The *lpIncludeProps* parameter cannot be NULL.

ulUIParam
Input parameter containing the handle of the parent window for the progress indicator displayed.

lpProgress
Input parameter pointing to a progress object that contains client- or provider-supplied progress information. If NULL is passed in the *lpProgress* parameter, MAPI provides the progress information. The *lpProgress* parameter is ignored unless the MAPI_DIALOG flag is set in the *ulFlags* parameter.

lpInterface
Input parameter pointing to the interface identifier (IID)for the destination object indicated in the *lpDestObj* parameter. The *lpInterface* parameter must not be NULL.

lpDestObj
Input parameter pointing to the open destination object.

ulFlags
Input parameter containing a bitmask of flags that controls how the copy or move operation is performed. The following flags can be set:

MAPI_DECLINE_OK
Informs the service provider that if it does not implement **IMAPIProp::CopyProps**, it should immediately return MAPI_E_DECLINE_COPY. MAPI uses this flag to limit recursion within MAPI's **CopyProps** implementation.

MAPI_DIALOG
Displays a user interface to provide progress information for the copy or move operation.

MAPI_MOVE
Indicates a move operation. The default operation is copying.

MAPI_NOREPLACE
Indicates that existing properties in the destination object should not be overwritten. The default action is to overwrite existing properties.

lppProblems
Output parameter pointing to a variable where the pointer to an **SPropProblemArray** structure is stored. If NULL is passed in the *lppProblems* parameter, no property problem array is returned.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

MAPI_E_COLLISION
A sibling folder in the destination object already has the name in the PR_DISPLAY_NAME property being copied from the source object.

MAPI_E_DECLINE_COPY
Indicates the provider does not implement the copy operation.

MAPI_E_FOLDER_CYCLE
The source object directly or indirectly contains the destination object. Significant work might have been performed before this condition was discovered, so the source and destination objects might be partially modified.

MAPI_E_INTERFACE_NOT_SUPPORTED
An appropriate interface cannot be obtained.

MAPI_E_NO_ACCESS
An attempt was made to modify a read-only object or an attempt to access an object for which the user has insufficient permissions.

The following values can be returned in the **SPropProblemArray** structure, but not as return values for **CopyProps**:

MAPI_E_BAD_CHARWIDTH
Either the MAPI_UNICODE flag was set and the implementation does not support Unicode, or MAPI_UNICODE was not set and the implementation only supports Unicode.

MAPI_E_COMPUTED
The property can't be written because it is computed by the destination object's provider. This error is not severe; the implementation should allow the process to continue.

MAPI_E_INVALID_TYPE
The property type is invalid.

MAPI_E_UNEXPECTED_TYPE
The property type is not the type expected by the calling client application.

Remarks

Use the **IMAPIProp::CopyProps** method to copy or move to the destination object those properties designated in *lpIncludeProps* that are present in the source object. When copying properties between like objects, for example between two message objects, the interface identifiers and object types must be the same for both the source and destination objects. If any of the copied or moved properties already exist in the destination object, the existing properties are overwritten by the new, unless the MAPI_NOREPLACE flag is set in the *ulFlags* parameter. Existing information in the destination object that is not overwritten is not deleted or modified. Note that the PR_NULL property should not be included in the *lpIncludeProps* array.

For message objects, the PR_MESSAGE_RECIPIENTS and PR_MESSAGE_ATTACHMENTS properties can be included in the **SPropTagArray** structure passed in *lpIncludeProps* to permit copying or moving of message recipients and attachments. For folder objects, the PR_CONTAINER_HIERARCHY, PR_CONTAINER_CONTENTS, and PR_FOLDER_ASSOCIATED_CONTENTS properties can be included in the **SPropTagArray** to permit copying or moving of subfolders, messages, or associated objects. If subfolders are copied or moved, they are copied or moved in their entirety, regardless of the use of properties indicated by the **SPropTagArray**.

If the source object directly or indirectly contains the destination object, the overall call fails and returns MAPI_E_FOLDER_CYCLE. Some implementations, however, perform significant work before discovering this error and leave the source and destination objects partially modified, so implementations should avoid such calls. If the same pointer is used for both the source and destination objects, the call returns MAPI_E_NO_ACCESS.

The interface of the destination object, indicated in *lpInterface*, is usually the same interface as for the source object. If *lpInterface* is set to NULL, then **CopyProps** returns MAPI_E_INVALID_PARAMETER. If an acceptable interface is passed in *lpInterface* but an invalid pointer is passed in the *lpDestObj* parameter, the results are unpredictable; the most likely result is that the calling client stops.

If the MAPI_DIALOG flag is not set in *ulFlags*, **CopyProps** ignores the *ulUIParam* and *lpProgress* parameters and no progress indicator is provided. If a client sets MAPI_DIALOG in *ulFlags* and passes NULL in *lpProgress*, the provider is responsible for generating a progress indicator. If a client sets MAPI_DIALOG in *ulFlags* and passes a progress object in *lpProgress*, the provider uses the information supplied by the progress object to display progress information.

If the call succeeds overall but there are problems with copying or moving some properties, **CopyProps** returns S_OK and an **SPropProblemArray** structure in the *lppProblems* parameter. The **SPropProblemArray** structure contains details about each property problem. In some cases, a **CopyProps** call can successfully set some of the requested properties, but not others; in these cases, which properties were not successfully copied or moved can be determined from the **SPropProblemArray**. If message recipients or attachments cannot be copied or moved, PR_MESSAGE_RECIPIENTS or PR_MESSAGE_ATTACHMENTS is returned in the **SPropProblemArray**.

The value returned in *lppProblems* is only valid if the call returns S_OK. When S_OK is returned, check the values returned in the **SPropProblemArray**. If an error occurs on the call, then the **SPropProblemArray** is not filled in; call the **IMAPIProp::GetLastError** method to get the **MAPIERROR** structure describing the error.

The calling implementation must free the returned **SPropProblemArray** by calling the **MAPIFreeBuffer** function, but this should only be done if **CopyProps** returns S_OK.

See Also

IMAPIFolder::CopyMessages method, **IMAPIProp::CopyTo** method, **IMAPIProp::GetLastError** method, **SPropProblemArray** structure, **SPropTagArray** structure

IMAPIProp::CopyTo

The **IMAPIProp::CopyTo** method copies or moves all properties from a source object to a destination object, except for a given set of excluded properties. The source object is the object on which the call to **CopyTo** is made.

HRESULT CopyTo(
  ULONG *ciidExclude***,**
  LPCIID *rgiidExclude***,**
  LPSPropTagArray *lpExcludeProps***,**
  ULONG *ulUIParam***,**
  LPMAPIPROGRESS *lpProgress***,**
  LPCIID *lpInterface***,**
  LPVOID *lpDestObj***,**
  ULONG *ulFlags***,**
  LPSPropProblemArray FAR * *lppProblems*
)

Parameters

ciidExclude
  Input parameter containing the number of interfaces to exclude when copying or moving properties.

rgiidExclude
  Input parameter containing an array of interface identifiers (IIDs) indicating interfaces that should not be used when copying or moving supplemental information to the destination object.

lpExcludeProps
  Input parameter pointing to an **SPropTagArray** structure containing the property identifiers of the properties that should not be copied or moved to the destination object. Passing NULL in the *lpExcludeProps* parameter indicates all properties are copied or moved. Passing zero in the **cValues** member of the *lpExcludeProps* **SPropTagArray** structure results in MAPI_E_INVALID_PARAMETER being returned.

ulUIParam
  Input parameter containing the handle of the parent window for the progress indicator.

lpProgress
  Input parameter pointing to a progress object that contains client- or provider-supplied progress information. If NULL is passed in the *lpProgress* parameter, MAPI provides the progress information is provided by MAPI. The *lpProgress* parameter is ignored unless the MAPI_DIALOG flag is set in the *ulFlags* parameter.

lpInterface
  Input parameter pointing to the IID for the destination object.

lpDestObj
  Input parameter pointing to the open destination object.

ulFlags
  Input parameter containing a bitmask of flags that controls how the copy or move operation is performed. The following flags can be set:

  MAPI_DECLINE_OK
    Informs the service provider that if it does not implement **CopyTo**, it should immediately return MAPI_E_DECLINE_COPY. MAPI uses this flag to limit recursion within its **CopyTo** implementation.

  MAPI_DIALOG
    Displays a user interface to provide progress information for the copy or move operation.

  MAPI_MOVE
    Indicates a move operation. The default operation is copying.

MAPI_NOREPLACE
Indicates that existing properties in the destination object should not be overwritten. The default action is to overwrite existing properties.

lppProblems
Output parameter pointing to a variable where the pointer to an **SPropProblemArray** structure is stored. If NULL is passed in the *lppProblems* parameter, no property problem array is returned.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

MAPI_E_COLLISION
A sibling folder in the destination object already has the name in the PR_DISPLAY_NAME property being copied from the source object.

MAPI_E_DECLINE_COPY
The provider does not implement the copy operation.

MAPI_E_FOLDER_CYCLE
The source object directly or indirectly contains the destination object. Significant work might have been performed before this condition was discovered, so the source and destination objects might be partially modified.

MAPI_E_INTERFACE_NOT_SUPPORTED
An appropriate interface cannot be obtained.

MAPI_E_NO_ACCESS
An attempt was made to modify a read-only object or an attempt to access an object for which the user has insufficient permissions.

The following values can be returned in the **SPropProblemArray** structure, but not as return values for **CopyTo**:

MAPI_E_BAD_CHARWIDTH
Either the MAPI_UNICODE flag was set and the implementation does not support Unicode, or MAPI_UNICODE was not set and the implementation only supports Unicode.

MAPI_E_COMPUTED
The property can't be written because it is computed by the destination object's provider. This error is not severe; the provider should allow the process to continue.

MAPI_E_INVALID_TYPE
The property type is invalid.

MAPI_E_UNEXPECTED_TYPE
The property type is not the type expected by the calling implementation.

Remarks

Use the **IMAPIProp::CopyTo** method to copy or move all properties of the source object to the destination object, except for a given set of properties that are excluded from the copy or move operation. Any objects contained in the source object and any subobjects of the source object are included in the copy or move operation.

If any of the copied or moved properties already exist in the destination object, the existing properties are overwritten by the new, unless the MAPI_NOREPLACE flag is set in the *ulFlags* parameter. Existing information in the destination object that is not overwritten is not deleted or modified. When the MAPI_NOREPLACE flag is clear, preferred but not required behavior of the implementation is to delete the existing collection.

To exclude some properties from the copy or move operation, pass their property identifiers in the *lpExcludeProps* parameter. Passing in a specific value causes any property in the source object whose identifier matches that value to be excluded from the copy or move operation. For example, passing in PROP_TAG(PT_LONG, 0x8002) excludes both the properties PROP_TAG(PT_STRING8, 0x8002) and PROP_TAG(PT_OBJECT, 0x8002). PR_NULL should not be included in the *lpExcludeProps* array.

For message objects, the PR_MESSAGE_RECIPIENTS and PR_MESSAGE_ATTACHMENTS properties can be included in the **SPropTagArray** structure passed in *lpExcludeProps* to prevent copying or moving message recipients and attachments. For folder objects, the PR_CONTAINER_HIERARCHY, PR_CONTAINER_CONTENTS, and PR_FOLDER_ASSOCIATED_CONTENTS properties can be included in the **SPropTagArray** to prevent copying or moving of subfolders, messages, or associated objects. If subfolders are copied or moved, they are copied or moved in their entirety, regardless of the use of properties indicated by the **SPropTagArray**.

Providers implementing **CopyTo** should not attempt to set any known read-only properties in the destination object and should ignore MAPI_E_COMPUTED errors returned in the **SPropProblemArray** structure in the *lppProblems* parameter.

If the source object directly or indirectly contains the destination object, the overall call fails and returns MAPI_E_FOLDER_CYCLE. Some implementations, however, perform significant work before discovering this error and leave the source and destination objects partially modified, so implementations should avoid such calls. If the same pointer is used for both the source and destination objects, the call returns MAPI_E_NO_ACCESS.

The interface of the destination object, indicated in *lpInterface*, is usually the same interface as for the source object. If *lpInterface* is set to NULL, then **CopyTo** returns MAPI_E_INVALID_PARAMETER. If an acceptable interface is passed in *lpInterface* but an invalid pointer is passed in the *lpDestObj* parameter, the results are unpredictable; the most likely result is that the calling implementation stops.

Some objects contain supplemental information, which can be accessed with the interface pointer in *lpInterface*. To copy or move such information, a provider's **CopyTo** implementation first calls the **IUnknown::QueryInterface** method for the destination object to see if it can accept the extra data. Conversely, if the calling implementation is aware of supplemental information and requires that **CopyTo** not copy or move it, the implementation can specify in the array passed in the *rgiidExclude* parameter IIDs for the properties that **CopyTo** should not copy or move. For example, if the client must copy messages, but not embedded objects within the messages, it can pass IID_IMessage in the *rgiidExclude* array. **CopyTo** ignores any interfaces listed in *rgiidExclude* it doesn't recognize.

Note When you use the *rgiidExclude* parameter to exclude an interface, it also excludes all interfaces derived from that interface. For example, excluding the **IMAPIProp** interface also excludes the **IMAPIFolder**, **IMessage**, and **IAttach** interfaces, and so on. If all known interfaces are excluded, **CopyTo** returns the error value MAPI_E_INTERFACE_NOT_SUPPORTED. For that reason, you should not pass IID_IUnknown or IID_IMAPIProp in *rgiidExclude*.

If the MAPI_DIALOG flag is not set in *ulFlags*, then **CopyTo** ignores the *ulUIParam* and *lpProgress* parameters and no progress indicator is provided. If a client sets MAPI_DIALOG in *ulFlags* and passes NULL in *lpProgress*, the provider is responsible for generating a progress indicator. If a client sets MAPI_DIALOG in *ulFlags* and passes a progress object in *lpProgress*, the provider uses the information supplied by the progress object to display progress information.

If the call succeeds overall but there are problems with copying or moving some properties, **CopyTo** returns S_OK and an **SPropProblemArray** structure in the *lppProblems* parameter. The **SPropProblemArray** structure contains details about each property problem. In some cases, a **CopyTo** call can successfully set some of the requested properties, but not others; in these cases, which properties were not successfully copied or moved can be determined from the **SPropProblemArray**. If message recipients or attachments cannot be copied or moved, PR_MESSAGE_RECIPIENTS or PR_MESSAGE_ATTACHMENTS is returned in the **SPropProblemArray**.

The value returned in *lppProblems* is only valid if the call returns S_OK. When S_OK is returned, check the values returned in the **SPropProblemArray**. If an error occurs on the call, then the **SPropProblemArray** is not filled in; call the **IMAPIProp::GetLastError** method to get the **MAPIERROR** structure describing the error.

If an error occurs on the **CopyTo** call, do not use or free the **SPropProblemArray** structure. Implementations should ignore the **ulIndex** member in **SPropProblemArray** structures returned by **CopyTo**.

The calling implementation must free the returned **SPropProblemArray** by calling the **MAPIFreeBuffer** function, but this should only be done if **CopyTo** returns S_OK.

See Also

IMAPIFolder::CopyMessages method, **IMAPIProp::GetLastError** method, **SPropProblemArray** structure, **SPropTagArray** structure

IMAPIProp::DeleteProps

The **IMAPIProp::DeleteProps** method deletes all properties in the given list.

HRESULT DeleteProps(
  LPSPropTagArray *lpPropTagArray***,**
  LPSPropProblemArray FAR * *lppProblems*
)

Parameters

lpPropTagArray
  Input parameter pointing to an **SPropTagArray** structure containing an array of property tags indicating the properties to delete. The *lpPropTagArray* parameter must not be NULL. The property type in each property tag is ignored, and only the property identifier is used. Passing zero in the **cValues** member of the **SPropTagArray** structure results in MAPI_E_INVALID_PARAMETER being returned.

lppProblems
  Output parameter pointing to a variable where the pointer to an **SPropProblemArray** structure is stored. If NULL is passed in the *lppProblems* parameter, no property problem array is returned.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

MAPI_E_NO_ACCESS
  An attempt was made to modify a read-only object or an attempt to access an object for which the user has insufficient permissions.

Remarks

Use the **IMAPIProps::DeleteProps** method to delete properties from an object. Not all objects allow deletion of properties. If the **DeleteProps** call fails completely, its HRESULT returns with a nonzero value.

If the call succeeds overall but there are problems with deleting some properties, **DeleteProps** returns S_OK and an **SPropProblemArray** structure in *lppProblems*. The **SPropProblemArray** structure contains details about each property problem. In some cases, a **DeleteProps** call can successfully set some of the requested properties, but not others; in these cases, which properties were not successfully deleted can be determined from the **SPropProblemArray**. If message recipients or attachments cannot be deleted, the PR_MESSAGE_RECIPIENTS or PR_MESSAGE_ATTACHMENTS property is returned in the **SPropProblemArray** structure.

The value returned in *lppProblems* is only valid if the call returns S_OK. When S_OK is returned, check the values returned in **SPropProblemArray**. If an error occurs on the call, then **SPropProblemArray** is not filled in; call the **IMAPIProp::GetLastError** method to get the **MAPIERROR** structure describing the error.

The calling implementation must free the returned **SPropProblemArray** by calling the **MAPIFreeBuffer** function, but this should only be done if **DeleteProps** returns S_OK.

See Also

IMAPIProp::GetProps method, **IMAPIProp::SaveChanges** method, **MAPIFreeBuffer** function, **SPropTagArray** structure

IMAPIProp::GetIDsFromNames

The **IMAPIProp::GetIDsFromNames** method provides property identifiers, given a list of property names.

HRESULT GetIDsFromNames(
  ULONG *cPropNames***,**
  LPMAPINAMEID FAR * *lppPropNames***,**
  ULONG *ulFlags***,**
  LPSPropTagArray FAR * *lppPropTags*
)

Parameters

cPropNames
  Input parameter containing the number of pointers to **MAPINAMEID** structures returned in the *lppPropNames* parameter. If *lppPropNames* is NULL, the *cPropNames* parameter must be zero.

lppPropNames
  Input parameter pointing to an array of pointers to **MAPINAMEID** structures containing names of properties. Passing NULL requests property identifiers for all property names about which the object has information. If the MAPI_CREATE flag is set in the *ulFlags* parameter, the *lppPropNames* parameter must not be NULL.

ulFlags
  Input parameter containing a bitmask of flags that controls how the property identifiers are returned. The following flag can be set:

  MAPI_CREATE
    Allocates a property identifier for each named property in *lppPropNames* not registered in the name-to-identifier mapping table and internally registers the identifier in this table.

lppPropTags
  Output parameter pointing to a variable where a pointer to the array of existing or newly assigned property identifiers is stored. The property types in this array are set to PT_UNSPECIFIED.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

MAPI_E_NO_SUPPORT
  The operation is not supported by MAPI or by one or more service providers.

MAPI_E_NOT_ENOUGH_MEMORY
  Insufficient memory was available to complete the operation.

MAPI_E_TOO_BIG
  The operation cannot be performed because it requires too many property tags be returned.

MAPI_W_ERRORS_RETURNED
  The call succeeded overall, but one or more properties could not be accessed and were returned with a property type of PT_ERROR and an identifier of zero. To test for this warning, use the **HR_FAILED** macro. When this warning is returned, the call should be handled as successful.

Remarks

Use the **IMAPIProp::GetIDsFromNames** method to get an array of property tags that holds the property identifiers for the named properties. The returned property tags are in the same order as the names passed in the **LPMAPINAMEID** array in *lppPropNames*. **GetIDsFromNames** returns the type portion of each tag as PT_UNSPECIFIED; to set these, call the **IMAPIProp::SetProps** method.

Message store providers that need to extend property sets typically implement the **GetIDsFromNames** and **IMAPIProp::GetNamesFromIDs** methods for message and folder objects. Other interfaces derived from **IMAPIProp** typically return MAPI_E_NO_SUPPORT for calls to these methods.

Only identifiers in the range of 0x8000 to 0xFFFE use name-to-identifier mapping. MAPI requires a store provider that supports named properties in folder contents tables to use the same name-to-identifier mapping for all objects in a folder. A store provider that supports named properties in search-results folders' contents tables must use the same name-to-identifier mapping for all objects in the message store. Objects that support name-to-identifier mapping often contain an additional binary property, PR_MAPPING_SIGNATURE (a **MAPIUID** structure). If two objects have mapping signatures and both signatures have the same value, those objects use the same name-to-identifier mapping.

Implementations that move or copy objects with named properties must preserve names during such operations by adjusting property identifiers to match the name-to-identifier mapping of the destination object. The exception is if the source and destination objects have the same value for PR_MAPPING_SIGNATURE, in which case the implementation can skip this step.

If a name set with **GetIDsFromNames** does not have an identifier, **GetIDsFromNames** returns MAPI_W_ERRORS_RETURNED and in the appropriate entry of the property tag array returns a property tag with a type of PT_ERROR and an identifier of zero. If a name passed with **GetIDsFromNames** does not have an identifier and MAPI_CREATE was set, the **GetIDsFromNames** call allocates an identifier for the name and internally registers the identifier in the name-to-identifier mapping table. When NULL is passed in *lppPropNames*, the provider returns the entire name to identifier mapping – that is, all names and all property sets defined by the provider. The value in the *cPropNames* parameter must also be zero if NULL is passed in *lppPropNames*. If the number of defined identifiers in the store exceeds the implementation limit, **GetIDsFromNames** returns MAPI_E_TOO_BIG and clients should query by identifier.

For more information on using the **HR_Failed** macro, see Using Macros for Error Handling.

See Also

IMAPIProp::GetNamesFromIDs method, **MAPINAMEID** structure, **MAPIUID** structure

IMAPIProp::GetLastError

The **IMAPIProp::GetLastError** method returns a **MAPIERROR** structure containing information about the last error that occurred for an object.

HRESULT GetLastError(
  HRESULT *hResult***,**
  ULONG *ulFlags***,**
  LPMAPIERROR FAR * *lppMAPIError*
)

Parameters

hResult
  Input parameter containing the result returned for the last call for the object that returned an error.

ulFlags
  Input parameter containing a bitmask of flags that controls the type of the returned strings. The following flag can be set:

  MAPI_UNICODE
    Indicates the strings in the **MAPIERROR** structure returned in the *lppMAPIError* parameter are in Unicode format. If the MAPI_UNICODE flag is not set, the strings are in ANSI format.

lppMAPIError
  Output parameter pointing to a pointer to the returned **MAPIERROR** structure containing version, component, and context information for the error. The *lppMAPIError* parameter can be set to NULL if there is no **MAPIERROR** structure to return.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

MAPI_E_BAD_CHARWIDTH
  Either the MAPI_UNICODE flag was set and the implementation does not support Unicode, or MAPI_UNICODE was not set and the implementation only supports Unicode.

Remarks

Use the **IMAPIProp::GetLastError** method to retrieve information to display in a message to the user regarding the last error returned from a method call for the object.

To release all the memory allocated by MAPI, clients need only call the **MAPIFreeBuffer** function for the returned **MAPIERROR** structure.

The return value from **GetLastError** must be S_OK for the implementation to make use of the **MAPIERROR** structure. Even if the return value is S_OK, a **MAPIERROR** structure might not be returned. If the implementation cannot determine what the last error was, or if a **MAPIERROR** structure is not available for that error, **GetLastError** returns a pointer to NULL in *lppMAPIError* instead.

See Also

MAPIERROR structure, **MAPIFreeBuffer** function

IMAPIProp::GetNamesFromIDs

The **IMAPIProp::GetNamesFromIDs** method provides property names, given a list of property identifiers.

HRESULT GetNamesFromIDs(
  LPSPropTagArray FAR * *lppPropTags***,**
  LPGUID *lpPropSetGuid***,**
  ULONG *ulFlags***,**
  ULONG FAR * *lpcPropNames***,**
  LPMAPINAMEID FAR * FAR * *lpppPropNames*
)

Parameters

lppPropTags
  Input or output parameter pointing to a variable where the pointer to an **SPropTagArray** structure of property tags is stored.

lpPropSetGuid
  Optional input parameter pointing to a globally unique identifier (GUID) for the property set. This parameter is zero if the *lppPropTags* parameter is zero.

ulFlags
  Input parameter containing a bitmask of flags that controls how names are mapped. The following flags can be used:

  MAPI_NO_IDS
    Indicates integer names, for which *ulKind* is MNID_ID, should not be returned.

  MAPI_NO_STRINGS
    Indicates string names, for which *ulKind* is MNID_STRING, should not be returned.

  If both flags are set, no names will be returned.

lpcPropNames
  Output parameter pointing to a variable containing the number of strings in *lpppPropNames*.

lpppPropNames
  Output parameter pointing to a variable pointer to an array of pointers to **MAPINAMEID** structures containing names of properties.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

MAPI_E_NO_SUPPORT
  The operation is not supported by MAPI or by one or more service providers.

MAPI_W_ERRORS_RETURNED
  The call succeeded overall, but one or more properties could not be accessed and were returned with a property type of PT_ERROR. To test for the warning, use the **HR_FAILED** macro. When this warning is returned, the call should be handled as successful.

Remarks

Use the **IMAPIProp::GetNamesFromIDs** method to get an array of pointers to Unicode strings that are the names of the indicated properties. While access to a property is mainly by property identifier, some implementations also need to associate names in Unicode string format with some properties in their objects.

GetNamesFromIDs ignores the property type in each property tag passed in *lppPropTags*;   only the property identifier is significant. If a specified identifier does not have a Unicode name,

GetNamesFromIDs returns NULL in that identifier's place in the structure returned in *lpppPropNames* and also returns MAPI_W_ERRORS_RETURNED. If *lppPropTags* is NULL, then the method allocates a new property tag array and returns all names and identifiers mapped for the object. The returned buffer for these strings is freed by calling the **MAPIFreeBuffer** function.

Two GUIDs identify particularly useful property sets: PS_MAPI and PS_PUBLIC_STRINGS. To use a MAPI folder as a container for string properties, use PS_PUBLIC_STRINGS as the property set for storing the ad hoc document property strings. Implementations that need to get the public names within the container call **GetNamesFromIDs** with a pointer to a null property-tag array, a GUID of PS_PUBLIC_STRINGS, and the MAPI_NO_IDS flag set in the *ulFlags* parameter.

Implementations attempting to retrieve all the registrations should set *ulFlags* to zero and pass in a null property tag array and a null GUID. If NULL is passed in an **SPropTagArray** structure, it must be freed by calling the **MAPIFreeBuffer** function.

This structure holds the identifiers for which names are needed. Passing zero in the **cValues** member of the **SPropTagArray** structure results in MAPI_E_INVALID_PARAMETER being returned. The *lppPropTags* parameter can be a pointer to NULL, in which case all the names are returned. If *lppPropTags* is zero, the caller is asking for the identifiers and names of all properties in a given property set. If it is nonzero, the caller is asking for the names of a given list of identifiers. If *lppPropTags* and *lppropSetGuid* are both zero, the implementation should return all names from all property sets. If both the *lppPropTag* and *lppropSetGuid* parameters are non-null, clients cannot depend on the results and MAPI will not protect providers with parameter validation. However, it is possible to ignore the GUID and get the names for the identifiers in the property tag array. Applications should call **MAPIFreeBuffer** on *lppPropTags* and *lpppPropnames* as usual when success is returned, and should check *lpcPropNames* to see if any names were found.

If there are no properties in the requested property set, or all of the properties are of a type excluded by the flags, then a request for all properties will result in no properties being found. In this case, the implementation should return S_OK, allocate and return an **SPropTagArray** structure with **cValues**==0 in *lppPropTags*, return 0 in *lpcPropNames*, and return NULL in *lpppPropNames*. Applications should call **MAPIFreeBuffer** on *lppPropTags* and *lpppPropNames* as usual when success is returned, and should check *lpcPropNames* to see if any names were found.

For more information on using the **HR_FAILED** macro, see Using Macros for Error Handling.

See Also

IMAPIProp::GetIDsFromNames method, **MAPIFreeBuffer** function, **MAPINAMEID** structure, **SPropTagArray** structure

IMAPIProp::GetPropList

The **IMAPIProp::GetPropList** method returns a list of all properties of an object.

HRESULT GetPropList(
  ULONG *ulFlags*,
  LPSPropTagArray FAR * *lppPropTagArray*
)

Parameters

ulFlags
  Input parameter containing a bitmask of flags that controls the type of the strings in the returned property tags. The following flag can be set:

  MAPI_UNICODE
    Indicates the returned strings are in Unicode format. If the MAPI_UNICODE flag is not set, the strings are in ANSI format.

lppPropTagArray
  Output parameter pointing to a variable where the pointer to the returned **SPropTagArray** structure is stored.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

MAPI_E_BAD_CHARWIDTH
  Either the MAPI_UNICODE flag was set and the implementation does not support Unicode, or MAPI_UNICODE was not set and the implementation only supports Unicode.

Remarks

Use the **IMAPIProp::GetPropList** method to get the property tag for each property of an object. If no properties exist for the object, **GetPropList** returns a property tag array containing just a count, and that count must have a value of zero. Properties of type PT_OBJECT are returned by **GetPropList**.

Some service providers exclude those properties for which the calling implementation does not have access from the list of returned properties.

The calling implementation must free the property tag structure pointed to by the *lppPropTagArray* parameter by calling the **MAPIFreeBuffer** function.

If the object for which properties are retrieved supports Unicode, string properties are returned with the preferred character width, either PT_UNICODE or PTSTRING8. If the object does not support Unicode, **GetPropList** returns MAPI_E_BAD_CHARWIDTH, even if there are no string properties defined for the object.

See Also

MAPIFreeBuffer function

IMAPIProp::GetProps

The **IMAPIProp::GetProps** method retrieves the property values of one or more properties of an object.

HRESULT GetProps(
  LPSPropTagArray *lpPropTagArray***,**
  ULONG *ulFlags***,**
  ULONG FAR * *lpcValues***,**
  LPSPropValue FAR * *lppPropArray*
)

Parameters

lpPropTagArray
: Input parameter pointing to an **SPropTagArray** structure containing an array of property tags of the properties whose values are to be retrieved. If NULL is passed in the *lpPropTagArray* parameter, then values for all properties of the object are returned. If zero is passed in the **cValues** member of the **SPropTagArray** structure, MAPI_E_INVALID_PARAMETER is returned.

ulFlags
: Input parameter containing a bitmask of flags that controls the type of the text for a returned property value if an implementation passes the PT_UNSPECIFIED property type in the property tag array. The following flag can be set:

  MAPI_UNICODE
  : Indicates the returned strings are in Unicode format. If the MAPI_UNICODE flag is not set, the strings are in ANSI format.

lpcValues
: Output parameter pointing to a variable that holds the number of properties for which tags are returned in the *lppPropArray* parameter. The *lpcValues* parameter is always equal to the size of the property value array, unless *lppPropArray* is NULL.

lppPropArray
: Output parameter pointing to a pointer to the returned **SPropValue** array of property values.

Return Values

S_OK
: The call succeeded and has returned the expected value or values.

MAPI_W_ERRORS_RETURNED
: The call succeeded overall, but one or more properties could not be accessed and were returned with a property type of PT_ERROR and an identifier of zero. To test for this warning, use the **HR_FAILED** macro. When this warning is returned, the call should be handled as successful.

Remarks

Use the **IMAPIProp::GetProps** method to get the property values of one or more properties from an object. An implementation should first set up a counted array of property tags in an **SPropTagArray** structure and then call the **IMAPIProp::GetProps** method on the object for whose properties it requires property values.

The order of the properties in the **SPropValue** structure returned in *lppPropArray* exactly matches the order in which the **SPropTagArray** in *lpPropTagArray* requested the properties. If property types are specified in the **SPropTagArray** in *lpPropTagArray*, then the property values in the **SPropValue** returned in *lppPropArray* have types that exactly match the requested types, unless an error value is returned instead.

If the calling implementation has the property identifier of a property for which it requires information but not the property type, it can pass in a property tag formed from the identifier and the property type PT_UNSPECIFIED. The actual type of the returned property is indicated in the returned **SPropValue** structure.

When an implementation requires access to the contents of properties with property type PT_OBJECT, it should use the **IMAPIProp::OpenProperty** method. It is not an error to request the value of a property of type PT_OBJECT by using **GetProps**, but the data in the returned **SPropValue** contains no useful information. To request values for secure properties, an implementation must explicitly request the properties by identifier; secure properties' values are not returned when an implementation passes NULL in *lppPropTagArray*.

Service providers must allocate memory for the **SPropValue** structure in *lpPropTagArray* using the **MAPIAllocateBuffer** function; any additional memory needed for the structure's members is allocated using the **MAPIAllocateMore** function. The calling implementation must free the returned **SPropValue** structure by calling the **MAPIFreeBuffer** function, but it should only do so if **GetProps** returns S_OK or MAPI_W_ERRORS_RETURNED.

When access to one or more properties fails, for example when the specified properties do not exist, **GetProps** returns MAPI_W_ERRORS_RETURNED. The calling implementation should check the property tags of the returned properties to determine for which properties access failed. Those for which access failed have their type set to PT_ERROR, and their values indicate which error occurred. For example, when a non-existent property is requested, the corresponding entry in the property tag array is updated with the property type member set to PT_ERROR and the provider value member set to MAPI_E_NOT_FOUND.

If no properties exist, and the calling implementation has requested values for all properties by passing NULL in *lpPropTagArray*, **GetProps** returns S_OK, sets the count value in the **cValues** member of the **SPropTagArray** structure to zero, and returns a zero length in *lpcValues* and NULL in *lppPropArray*. **GetProps** must not return multivalued properties with **cValues** set to zero.

If **GetProps** encounters an individual string or binary property that is too large to conveniently be returned, typically 4K or 8K, **GetProps** marks that property with PT_ERROR, sets its value to MAPI_E_NOT_ENOUGH_MEMORY, and returns MAPI_W_ERRORS_RETURNED. To get the data within the property, the implementation should call **IMAPIProp::OpenProperty**.

If the MAPI_UNICODE flag is set in the *ulFlags* parameter, then any string properties the implementation did not specify be returned in ANSI format are returned in Unicode format. If MAPI_UNICODE is not set, string properties with unspecified types are returned in ANSI format. String properties with unspecified types occur when **GetProps** is called with null for *lpPropTagArray* and when PT_UNSPECIFIED is passed for a property type in the *lpPropTagArray* parameter's **SPropTagArray** structure. Providers that do not support string format conversion should return PT_ERROR for the property type and MAPI_E_BAD_CHARWIDTH for the property value.

For more information on using the **HR_FAILED** macro, see Using Macros for Error Handling.

See Also

IMAPIProp::OpenProperty method, **MAPIAllocateBuffer** function, **MAPIAllocateMore** function, **MAPIFreeBuffer** function, **SPropTagArray** structure, **SPropValue** structure

IMAPIProp::OpenProperty

The **IMAPIProp::OpenProperty** method opens an interface for a property of an object.

HRESULT OpenProperty(
  ULONG *ulPropTag***,**
  LPCIID *lpiid***,**
  ULONG *ulInterfaceOptions***,**
  ULONG *ulFlags***,**
  LPUNKNOWN FAR * *lppUnk*
)

Parameters

ulPropTag
  Input parameter containing the property tag for the property for which an interface is required. A complete property tag should be passed.

lpiid
  Input parameter pointing to the interface identifier (IID) to be used. The *lpiid* parameter must not be NULL.

ulInterfaceOptions
  Input parameter indicating interface-specific behavior.

ulFlags
  Input parameter containing a bitmask of flags that controls how the property is opened. The following flags can be set:

  MAPI_CREATE
    If the property does not exist, it should be created. If the property does exist, the current data in the property should be discarded. When an implementation sets the MAPI_CREATE flag, it should also set the MAPI_MODIFY flag.

  MAPI_DEFERRED_ERRORS
    Indicates the call is allowed to succeed even if the underlying object is not accessible to the calling implementation. If the object is not accessible, some subsequent call to the object might return an error.

  MAPI_MODIFY
    Requests read/write access. The default interface is read-only. MAPI_MODIFY must be set when MAPI_CREATE is set.

lppUnk
  Output parameter pointing to a variable where the pointer to the newly created interface pointer is stored.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

MAPI_E_INTERFACE_NOT_SUPPORTED
  The requested interface is not supported for this property.

MAPI_E_NO_ACCESS
  An attempt was made to modify a read-only object or an attempt to access an object for which the user has insufficient permissions.

MAPI_E_NO_SUPPORT
  The operation is not supported by MAPI or by one or more service providers.

MAPI_E_NOT_FOUND
  The requested object does not exist and MAPI_CREATE was not set in the *ulFlags* parameter.

Remarks

Use the **IMAPIProp::OpenProperty** method to open an interface to access a specified property. Using this method is the only way to access a property of type PT_OBJECT, this method can also be used for other properties, typically large string and binary properties, depending on the implementation.

With OLE 2 attachments, call **IMAPIProp::OpenProperty** to open the PR_ATTACH_DATA_OBJ property with the IStreamDocfile interface. IStreamDocfile is a derivative of IStream that is based on an OLE2. IStreamDocfile is the best choice for accessing OLE 2 attachments because it involves the least amount of overhead. For more information, see PR_ATTACH_DATA_OBJ and Opening an Attachment.

When opening an interface for an object with **OpenProperty**, an implementation uses an IID to identify the interface. IID_IStream is used for string and binary properties. IID_IMessage is used when opening a PR_ATTACH_DATA_OBJ property for a message; PR_ATTACH_DATA_OBJ contains an embedded OLE object or embedded message data. IID_IStreamDocFile can be used for those properties that contain data stored in structured storage accessible through the **IStorage** interface.

To identify the property to open, a complete property tag should be passed in the *ulPropTag* parameter. Using a property type of PT_UNSPECIFIED in a property tag passed with *ulPropTag* returns MAPI_E_INVALID_PARAMETER.

Note Implementations using an **IStream** interface to access double-byte character set (DBCS) text should not seek or call the **IStream::SetSize** method on the stream other than with a zero position or size variable. Neither should implementations rely upon the value of the *plibNewPosition* output parameter. In addition, implementations should not call the **IMAPIProp::GetProps**, **IMAPIProp::SetProps**, or **IMAPIProp::DeleteProps** method or perform other calls that affect a property opened with an **IStream** interface. Violation of these rules can cause implementations to behave poorly, stop, or lose data.

The implementation of multiple openings of the same property is undefined and provider-dependent.

When a client implementation requires the ability to write to a stream it is opening, the client should set MAPI_MODIFY in *ulFlags*, whether or not MAPI_CREATE is set in *ulFlags*. Some service providers return MAPI_E_INVALID_PARAMETER when an implementation attempts to create a stream without setting MAPI_MODIFY. If MAPI_CREATE is set, MAPI_MODIFY must also be set.

The calling implementation is responsible for recasting the interface pointer returned in the *lppUnk* parameter to one appropriate for the interface specified in the *lpiid* parameter. The calling implementation should release the opened interface when done using it.

Message store providers can limit access to certain properties and property types and to specific interfaces for those properties. If a requested interface is not available for a given property, **OpenProperty** returns MAPI_E_INTERFACE_NOT_SUPPORTED.

The caller should set both the *ulFlags* and the *ulInterfaceOptions* parameters according to the kind of access needed to the underlying interface. When opening a stream for writing, for example, whenever possible set both STGM_WRITE in *ulInterfaceOptions* and MAPI_MODIFY in *ulFlags*, although sometimes this is not possible. Among the many interfaces accessible through **OpenProperty**, some of the interface-specific options do not map to any of the bits defined for the *ulFlags* parameters. But, if there is an interface option that maps to MAPI_MODIFY or MAPI_CREATE, that bit should be set in *ulFlags*.

When opening a table using IID_IMAPITable, MAPI_UNICODE may be passed in the *ulInterfaceOptions* parameter. This controls whether the table's default column set includes Unicode or ANSI string columns.

See Also

HrIStorageFromStream function, **IMAPIProp::DeleteProps** method, **IMAPIProp::GetProps** method, **IMAPIProp::SetProps** method, **IMAPISupport::IStorageFromStream** method

IMAPIProp::SaveChanges

The **IMAPIProp::SaveChanges** method makes permanent any changes made to an object since the last save operation.

HRESULT SaveChanges(
  ULONG *ulFlags*
)

Parameters

ulFlags

Input parameter containing a bitmask of flags that controls what happens to the object when the **IMAPIProp::SaveChanges** method is called.

If no flags are set for the *ulFlags* parameter, the client application requests changes be made permanent for the object but will make no further calls to the object except for calling **Release**. If a service provider returns an error in this case, it is because the provider couldn't save the changes. If neither the KEEP_OPEN_READWRITE nor KEEP_OPEN_READONLY flag are set, the results are implementation-specific; some providers treat this state as equivalent to passing KEEP_OPEN_READWRITE; others interpret it as KEEP_OPEN_READONLY; others shut down the object. The following flags can be set:

FORCE_SAVE

Writes changes to the object and closes it. Read/write access must have been set for the operation to succeed. The FORCE_SAVE flag only forces saving if MAPI_E_OBJECT_CHANGED was returned from a preceding **SaveChanges** call. FORCE_SAVE overrides the previous changes made to the object.

KEEP_OPEN_READONLY

Indicates the client requests changes be committed and the object be kept open for reading. The KEEP_OPEN_READONLY flag informs the provider that the object should not be modified and that the calling implementation will not call **SaveChanges** again. If the provider cannot keep the object open for read-only access, then the entire call fails, changes are not saved, and MAPI_E_NO_ACCESS is returned.

The provider can leave the object open for read/write access. However, the provider cannot prevent all access to the object when KEEP_OPEN_READONLY is passed.

If a client passes KEEP_OPEN_READONLY, then calls the **IMAPIProp::SetProps** method and then **SaveChanges** again, the same implementation might stop.

KEEP_OPEN_READWRITE

Indicates the client requests changes be committed and the object be kept open for read/write access. This flag is usually set when the object was initially opened for read/write access. After calling **SaveChanges**, the client can make further changes to the object if this flag is passed. If the provider cannot keep the object open for read/write access, then the entire call fails, changes are not saved, and MAPI_E_NO_ACCESS is returned.

After receiving such an error value, the client continues to have read/write access and might pass KEEP_OPEN_READONLY or no flags with the KEEP_OPEN_suffix. Whether a provider supports the KEEP_OPEN_READWRITE flad depends on the provider's implementation. However, in no case can a provider leave an object in a read-only state when the KEEP_OPEN_READWRITE flag is passed.

MAPI_DEFERRED_ERRORS

Indicates the call is allowed to succeed even if the underlying object is not accessible to the calling implementation. If the object is not accessible, some subsequent call to the object might return an error.

Return Values

S_OK
    The call succeeded and has returned the expected value or values.
MAPI_E_NO_ACCESS
    An attempt was made to modify a read-only object or an attempt to access an object for which the user has insufficient permissions.
MAPI_E_OBJECT_CHANGED
    The object has changed since it was opened.
MAPI_E_OBJECT_DELETED
    The object has been deleted since it was opened.

Remarks

Use the **IMAPIProp::SaveChanges** method to make changes permanent for some objects, specifically messages, attachments, address book containers, and messaging user objects. Other objects, such as folders, message stores, and profile sections, make changes permanent immediately without calling **SaveChanges**.

Some message store implementations do not show newly created messages in a folder until a client saves the message changes using **SaveChanges** and releases the message objects using the **IUnknown::Release** method. In addition, some object implementations cannot generate a PR_ENTRYID property for a newly created object until after **SaveChanges** has been called, and some can only do so after **SaveChanges** has been called with KEEP_OPEN_READONLY set in *ulFlags*.

Furthermore, changes to properties, such as the message subject, that are cached in a folder's summary table cannot be processed until the implementation has called **SaveChanges** and in some situations also **Release**.

When making bulk changes, such as saving attachments to multiple messages, an implementation should defer error processing by setting the MAPI_DEFERRED_ERRORS flag in *ulFlags*. In the case of saving attachments to multiple messages, MAPI_DEFERRED_ERRORS should be set for each attachment modification and for all but the last message modification; the flag should be omitted from the **SaveChanges** call for the last message so that errors can be returned. (Providers can in fact return errors before the last modification is made in this case, or they can ignore this flag altogether.) If KEEP_OPEN_READWRITE and KEEP_OPEN_READONLY are set with the same call as MAPI_DEFERRED_ERRORS, providers usually ignore MAPI_DEFERRED_ERRORS. If MAPI_DEFERRED_ERRORS is not set in *ulFlags*, one of the previously deferred errors is returned for the **SaveChanges** call.

Standard client behavior when saving changes to an object for which a preceding call to **SaveChanges** has returned MAPI_E_OBJECT_CHANGED is a normal save operation, rather than a forced save operation, and examination of any returned error value. If the original object has been modified, the implementation usually warns the user, who can either request the changes be saved or save the message somewhere else. If the original message has been deleted, the implementation also warns the user, who can save the message somewhere else.

If the user deletes an open object, the client is unable to force a save operation, even when FORCE_SAVE has been set in *ulFlags*.

If **SaveChanges** returns an error, then the object whose changes were to be saved remains open, regardless of the flags set in the *ulFlags* parameter.

IMAPIProp::SetProps

The **IMAPIProp::SetProps** method sets the property value of one or more properties of an object.

HRESULT SetProps(
  ULONG *cValues*,
  LPSPropValue *lpPropArray*,
  LPSPropProblemArray FAR * *lppProblems*
)

Parameters

cValues
  Input parameter containing the number of property values pointed to by the *lpPropArray* parameter. The *cValues* parameter must not be zero.

lpPropArray
  Input parameter pointing to an array of **SPropValue** structures holding property values indicating the properties whose values are to be set.

lppProblems
  Output parameter pointing to a variable where the pointer to an **SPropProblemArray** structure is stored. If NULL is passed in the *lppProblems* parameter, no property problem array is returned.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

The following values can be returned in the **SPropProblemArray** structure, but not as return values for **SetProps**:

MAPI_E_BAD_CHARWIDTH
  Either the MAPI_UNICODE flag was set and the implementation does not support Unicode, or MAPI_UNICODE was not set and the implementation only supports Unicode.

MAPI_E_COMPUTED
  The property cannot be written because it is computed by the destination object's service provider.

MAPI_E_INVALID_TYPE
  The property type is invalid.

MAPI_E_NO_ACCESS
  An attempt to modify a read-only object or an attempt to access an object for which the user has insufficient permissions.

MAPI_E_NOT_ENOUGH_MEMORY
  The property cannot be written because it is larger than the RPC buffer size.

MAPI_E_UNEXPECTED_TYPE
  The property type is not the type expected by the calling implementation.

Remarks

Use the **IMAPIProp::SetProps** method to set the property values of properties. A call to **SetProps** passes a number of **SPropValue** structures. The property tag in each structure indicates which property is having its value set, and the property value in each structure indicates what should be stored as the property's value.

If a **SetProps** call passes a tag for a nonexistent property and the implementation supports the creation of new properties, **SetProps** adds the new property to the object. Any previous value stored with the property identifier used for the new property is discarded. Depending on the implementation, a calling implementation can also change the property type and value of a stored property together at one time by passing a property tag containing a different type than was previously used with a given property identifier.

Any property with a property tag of PR_NULL or a property type of PT_ERROR is ignored by **SetProps**; no value is set and no problem report is generated for the property. Properties with a property type of PT_OBJECT cannot be set using **SetProps**; attempts to pass a property value array containing properties of type PT_OBJECT result in **SetProps** returning MAPI_E_INVALID_PARAMETER. Furthermore, an implementation cannot set multivalued properties with zero values in the **cValues** member of the **SPropValue** structure. Calls to set zero-valued multivalued properties should return MAPI_E_INVALID_PARAMETER.

If the call succeeds overall but there are problems with setting some property values, **SetProps** returns S_OK and an **SPropProblemArray** structure in the *lppProblems* parameter. The **SPropProblemArray** structure contains details about each property problem. In some cases, a **SetProps** call can successfully set some of the requested properties, but not others; in these cases, which properties were not successfully set can be determined from the **SPropProblemArray**. For example, if a property type is invalid or not supported, the **SetProps** call returns MAPI_E_INVALID_TYPE for that property in the **SPropProblemArray**.

The value returned in *lppProblems* is only valid if the call returns S_OK. When S_OK is returned, check the values returned in the **SPropProblemArray** structure. If an error occurs on the call, then the **SPropProblemArray** structure is not filled in and the calling implementation should not use or free the **SPropProblemArray** structure; call the **IMAPIProp::GetLastError** method to get the **MAPIERROR** structure describing the error.

The calling implementation must free the returned **SPropProblemArray** by calling the **MAPIFreeBuffer** function, but this should only be done if **SetProps** returns S_OK.

Note that a return value that indicates success does not necessarily indicate that the property-setting operation was successful. Some providers cache **SetProps** calls on the client side of the provider until they receive a call that requires provider intervention, such as a call to the **IMAPIProp::SaveChanges** or **IMAPIProp::GetProps** method. When such a call is received, the client attempts to transmit the cached information to the provider and the client can receive error values related to the earlier calls. For information about how a specific service provider implements **SetProps**, see the provider's documentation.

When setting large properties, the **SetProps** call can fail and return the MAPI_E_NOT_ENOUGH_MEMORY value. This most often happens when setting large properties in a message store. The MAPI specification does not mandate a maximum size for properties, and different message stores can have different limits. All messaging clients that deal with potentially large properties should be prepared to call **IMAPIProp::OpenProperty** with IID_IStream as the interface identifier if **SetProps** returns this error value.

See Also

IMAPIProp::GetProps method, **IMAPIProp::SaveChanges** method, **MAPIFreeBuffer** function, **SPropProblemArray** structure, **SPropValue** structure

IMAPISession : IUnknown

The **IMAPISession** interface is used to manage objects associated with a MAPI logon session.

At a Glance

Specified in header file:	MAPIX.H
Object that supplies this interface:	Session object
Corresponding pointer type:	LPMAPISESSION
Implemented by:	MAPI
Called by:	Client applications and MAPI

Vtable Order

GetLastError	Returns a **MAPIERROR** structure containing information about the last error that occurred for a session object.
GetMsgStoresTable	Returns a table that provides information about each of the message stores configured in a session's profile.
OpenMsgStore	Opens a message store and returns a pointer that provides further access to the open store.
OpenAddressBook	Opens an address book and returns a pointer that provides further access to the open address book.
OpenProfileSection	Opens a section of the current profile and returns a pointer that provides further access to the profile object.
GetStatusTable	Gets the status table for a given MAPI session.
OpenEntry	Opens an object and returns a pointer to the object to provide further access.
CompareEntryIDs	Compares two entry identifiers to determine if they refer to the same object.
Advise	Registers a client application for notifications about changes to a session object.
Unadvise	Removes an object's registration for notification of changes previously established with a call to the **IMAPISession::Advise** method.
MessageOptions	Displays a dialog box enabling a user to change options for a particular message.
QueryDefaultMessageOpt	Returns the available message

	options and their default settings for a particular messaging address type.
EnumAdrTypes	Returns the messaging address types for which the loaded transport providers have registered support.
QueryIdentity	Returns an entry identifier for the primary identity for a user for a MAPI session.
Logoff	Ends a MAPI session.
SetDefaultStore	Sets a message store provider as the default.
AdminServices	Administers configuration changes to a profile.
ShowForm	Displays a message form for editing and sending.
PrepareForm	Creates a message instance for use by the **IMAPISession::ShowForm** method.

IMAPISession::AdminServices

The **IMAPISession::AdminServices** method administers configuration changes to a profile.

HRESULT AdminServices(
  ULONG *ulFlags*,
  LPSERVICEADMIN FAR * *lppServiceAdmin*
)

Parameters

ulFlags
  Reserved; must be zero.

lppServiceAdmin
  Output parameter pointing to a variable where a pointer to the returned message service administration object is stored.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

Remarks

Client applications call the **IMAPISession::AdminServices** method to acquire a pointer to an **IMsgServiceAdmin** interface. Once a client has the pointer to this object, it can call **IMsgServiceAdmin** methods to change the message service configuration within a profile. Changes made using the **IMsgServiceAdmin** methods do not affect the current running session. If profile configuration is the primary purpose for creating a session, a client should log on using the MAPI_NO_MAIL flag.

See Also

IMsgServiceAdmin : IUnknown interface, **IProfAdmin::AdminServices** method

IMAPISession::Advise

The **IMAPISession::Advise** method registers a client application for notifications about changes to a session object.

HRESULT Advise(
  ULONG *cbEntryID***,**
  LPENTRYID *lpEntryID***,**
  ULONG *ulEventMask***,**
  LPMAPIADVISESINK *lpAdviseSink***,**
  ULONG FAR * *lpulConnection*
)

Parameters

cbEntryID
  Input parameter containing the size, in bytes, of the entry identifier pointed to by the *lpEntryID* parameter.

lpEntryID
  Input parameter pointing to the entry identifier of the session object about which notifications should be generated. The entry identifier of a status object cannot be used. NULL can be passed to register for critical error events on the session.

ulEventMask
  Input parameter containing an event mask that describes the types of events to generate notifications. The mask filters specific cases. If the *lpEntryID* parameter is NULL, the client is by default registering for critical error notifications on the session. This is the only type of event supported by the session. If the *lpEntryID* parameter contains a valid entry identifier, MAPI forwards the **Advise** call to the service provider that owns the entry identifier. The *ulEventMask* parameter should be set to one or more types of events supported by this provider.

lpAdviseSink
  Input parameter pointing to the advise sink object to be called when an event occurs for the session object about which notification has been requested. This advise sink object must have already been allocated.

lpulConnection
  Output parameter pointing to a variable that upon a successful return holds the connection number for the notification registration. The connection number must be nonzero.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

MAPI_E_INVALID_ENTRYID
  The service provider is not able to use the entry identifier passed in *lpEntryID*.

MAPI_E_NO_SUPPORT
  The service provider either does not support changes to its objects or does not support notification of changes.

MAPI_E_UNKNOWN_ENTRYID
  A service provider that could handle the *lpEntryID* entry identifier could not be found.

Remarks

Client applications call the **IMAPISession::Advise** method to register a session object for notification callbacks. MAPI forwards this call to the service provider active for the session that is responsible for the object indicated by the entry identifier in *lpEntryID*. Whenever a change occurs to the indicated object, the provider checks to see what event mask bit was set in the *ulEventMask* parameter and thus what type of change occurred. If a bit is set, then the provider calls the **IMAPIAdviseSink::OnNotify** method for the advise sink object indicated by the *lpAdviseSink* parameter to report the event. Data passed in the notification structure to the **OnNotify** routine describes the event.

The call to **OnNotify** can occur during the call that changes the object, or at any following time. On systems that support multiple threads of execution, the call to **OnNotify** can occur on any thread. For a way to turn a call to **OnNotify** that might happen at an inopportune time into one that is safer to handle, a client should use the **HrThisThreadAdviseSink** function.

To provide notifications, the service provider implementing **Advise** needs to keep a copy of the pointer to the *lpAdviseSink* advise sink object; to do so, it calls the **IUnknown::AddRef** method for the advise sink to maintain its object pointer until notification registration is canceled with a call to the **IMAPISession::Unadvise** method. The **Advise** implementation should assign a connection number to the notification registration and call **AddRef** on this connection number before returning it in the *lpulConnection* parameter. Service providers can release the advise sink object before the registration is canceled, but they must not release the connection number until **Unadvise** has been called.

After a call to **Advise** has succeeded and before **Unadvise** has been called, clients must be prepared for the advise sink object to be released. A client should therefore release its advise sink object after **Advise** returns unless it has a specific long-term use for it.

For more information on the notification process, see About Notification.

See Also

ERROR_NOTIFICATION structure, **HrThisThreadAdviseSink** function, **IMAPIAdviseSink::OnNotify** method, **IMAPISession::Unadvise** method

IMAPISession::CompareEntryIDs

The **IMAPISession::CompareEntryIDs** method compares two entry identifiers to determine if they refer to the same object. MAPI only passes this call to a service provider if the unique identifiers (UIDs) in both entry identifiers to be compared are handled by that provider.

HRESULT CompareEntryIDs(
  ULONG *cbEntryID1***,**
  LPENTRYID *lpEntryID1***,**
  ULONG *cbEntryID2***,**
  LPENTRYID *lpEntryID2***,**
  ULONG *ulFlags***,**
  ULONG FAR * *lpulResult*
)

Parameters

cbEntryID1
  Input parameter containing the size, in bytes, of the entry identifier pointed to by the *lpEntryID1* parameter.

lpEntryID1
  Input parameter pointing to the first entry identifier to be compared.

cbEntryID2
  Input parameter containing the size, in bytes, of the entry identifier pointed to by the *lpEntryID2* parameter.

lpEntryID2
  Input parameter pointing to the second entry identifier to be compared.

ulFlags
  Reserved; must be zero.

lpulResult
  Output parameter pointing to a variable where the returned result of the comparison is stored; this variable is TRUE if the two entry identifiers refer to the same object, and FALSE otherwise.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

MAPI_E_UNKNOWN_ENTRYID
  The requested entry identifier does not exist.

Remarks

Client applications call the **IMAPISession::CompareEntryIDs** method to compare two entry identifiers for a given entry within a service provider to determine whether they refer to the same object. If the two entry identifiers refer to the same object, then **CompareEntryIDs** sets the *lpulResult* parameter to TRUE; if they refer to different objects, **CompareEntryIDs** sets *lpulResult* to FALSE.

CompareEntryIDs is useful because an object can have more than one valid entry identifier; such a situation can occur, for example, after a new version of a service provider is installed.

If **CompareEntryIDs** returns an error, the calling client or provider should not take any action based on an assumption about the comparison's result. It should instead take the most conservative approach to the action it is trying to perform.

CompareEntryIDs might fail if, for example, no provider has registered for one of the entry identifiers compared. If a client or provider compares message-store entry identifiers when one or both of the stores has not yet opened, **CompareEntryIDs** returns MAPI_E_UNKNOWN_ENTRYID.

IMAPISession::EnumAdrTypes

The **IMAPISession::EnumAdrTypes** method returns the messaging address types for which the loaded transport providers have registered support.

HRESULT EnumAdrTypes(
  ULONG *ulFlags***,**
  ULONG FAR * *lpcAdrTypes***,**
  LPTSTR FAR * FAR * *lpppszAdrTypes*
)

Parameters

ulFlags
  Input parameter containing a bitmask of flags that controls the type of the returned strings. The following flag can be set:

  MAPI_UNICODE
    Indicates the returned strings are in Unicode format. If the MAPI_UNICODE flag is not set, the strings are in ANSI format.

lpcAdrTypes
  Output parameter pointing to the number of strings indicating address types returned in the *lpppszAdrTypes* parameter.

lpppszAdrTypes
  Output parameter pointing to a variable where is stored an array of pointers to strings containing messaging address types, such as FAX, SMTP, and X500. The array contains the types for which the transport providers have registered support.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

Remarks

Client applications call the **IMAPISession::EnumAdrTypes** method to determine which address types are supported by the transport providers loaded for a given session. The list of address types returned by **EnumAddrTypes** depends on what providers are loaded at the time. If a transport provider is not open, the address types that it supports do not appear in the list.

A client should release the string array pointed to by the *lpppszAdrTypes* parameter when done with the strings by calling the **MAPIFreeBuffer** function.

See Also

MAPIFreeBuffer function

IMAPISession::GetLastError

The **IMAPISession::GetLastError** method returns a **MAPIERROR** structure containing information about the last error that occurred for a session object.

HRESULT GetLastError(
  HRESULT *hResult***,**
  ULONG *ulFlags***,**
  LPMAPIERROR FAR * *lppMAPIError*
)

Parameters

hResult
  Input parameter containing the result returned for the last call for the session object that returned an error.

ulFlags
  Input parameter containing a bitmask of flags that controls the type of the returned strings. The following flag can be set:

  MAPI_UNICODE
    Indicates the strings in the **MAPIERROR** structure returned in the *lppMAPIError* parameter are in Unicode format. If the MAPI_UNICODE flag is not set, the strings are in ANSI format.

lppMAPIError
  Output parameter pointing to a pointer to the returned **MAPIERROR** structure containing version, component, and context information for the error. The *lppMAPIError* parameter can be set to NULL if there is no **MAPIERROR** structure to return.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

MAPI_E_BAD_CHARWIDTH
  Either the MAPI_UNICODE flag was set and the implementation does not support Unicode, or MAPI_UNICODE was not set and the implementation only supports Unicode.

Remarks

Client applications call the **IMAPISession::GetLastError** method to retrieve information to display in a message to the user regarding the last error returned from a method call for the session object.

To release all the memory allocated by MAPI, clients need only call the **MAPIFreeBuffer** function for the returned **MAPIERROR** structure.

The return value from **GetLastError** must be S_OK for a client to make use of the **MAPIERROR** structure. Even if the return value is S_OK, a **MAPIERROR** structure might not be returned. If the implementation cannot determine what the last error was, or if a **MAPIERROR** structure is not available for that error, **GetLastError** returns a pointer to NULL in *lppMAPIError* instead.

See Also

MAPIERROR structure, **MAPIFreeBuffer** function

IMAPISession::GetMsgStoresTable

The **IMAPISession::GetMsgStoresTable** method returns a table that provides information about each of the message stores configured in a session's profile.

HRESULT GetMsgStoresTable(
  ULONG *ulFlags*,
  LPMAPITABLE FAR * *lppTable*
)

Parameters

ulFlags
  Input parameter containing a bitmask of flags that controls the type of the returned strings. The following flag can be set:

  MAPI_UNICODE
    Indicates the returned strings are in Unicode format. If the MAPI_UNICODE flag is not set, the strings are in ANSI format.

lppTable
  Output parameter pointing to a variable where the returned table object is stored. The table object contains message store information.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

MAPI_E_BAD_CHARWIDTH
  Either the MAPI_UNICODE flag was set and the implementation does not support Unicode, or MAPI_UNICODE was not set and the implementation only supports Unicode.

Remarks

Client applications call the **IMAPISession::GetMsgStoresTable** method to retrieve a table holding information about each open message store configured in a session's profile. The message-store information table contains the following required property columns:

PR_DEFAULT_STORE
PR_DISPLAY_NAME
PR_ENTRYID
PR_INSTANCE_KEY
PR_PROVIDER_DISPLAY
PR_RECORD_KEY
PR_RESOURCE_TYPE

The message-store information table can also contain the following optional columns:

PR_MDB_PROVIDER
PR_RESOURCE_FLAGS

The default column set includes all of the columns defined for the table.

The message-store information table is updated during the session to reflect changes to the profile. Possible changes include addition of new, permanent message stores to the table, removal of existing stores, and changes to which message store is the default. Clients should register for notification of such changes.

Setting the MAPI_UNICODE flag in the *ulFlags* parameter does the following:

- Sets the string type to Unicode for data returned for the initial active columns of the message-store information table by the **IMAPITable::QueryColumns** method. The initial active columns for a message-store information table are those columns **QueryColumns** returns before the service provider that contains the table calls the **IMAPITable::SetColumns** method.
- Sets the string type to Unicode for data returned for the initial active rows of the message-store information table by the **IMAPITable::QueryRows** method. The initial active rows for a message-store information table are those rows **QueryRows** returns before the provider that contains the table calls **SetColumns**.
- Controls the property types of the sort order returned by the **IMAPITable::QuerySortOrder** method before the provider that contains the message-store information table calls the **IMAPITable::SortTable** method.

See Also

IMAPISession::OpenMsgStore method, **IMAPITable::QueryColumns** method, **IMAPITable::QueryRows** method, **IMAPITable::QuerySortOrder** method, **IMAPITable::SetColumns** method, **IMAPITable::SortTable** method

IMAPISession::GetStatusTable

The **IMAPISession::GetStatusTable** method gets the status table for a given MAPI session.

HRESULT GetStatusTable(
  ULONG *ulFlags*,
  LPMAPITABLE FAR * *lppTable*
)

Parameters

ulFlags
  Input parameter containing a bitmask of flags that controls the type of the returned strings. The following flag can be set:

  MAPI_UNICODE
    Indicates the returned strings are in Unicode format. If the MAPI_UNICODE flag is not set, the strings are in ANSI format.

lppTable
  Output parameter pointing to a variable where the returned table object is stored. The table contains status information.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

Remarks

Client applications call the **IMAPISession::GetStatusTable** method to open the status table object that contains information about everything going on in the session. There is one status table for each MAPI session. The only way to get a status object is by calling the **IMAPISession::OpenEntry** method on an entry identifier within the status table returned from **GetStatusTable**. The status table contains at least the following property columns:

PR_DISPLAY_NAME
PR_ENTRYID
PR_IDENTITY_DISPLAY
PR_INSTANCE_KEY
PR_OBJECT_TYPE
PR_PROVIDER_DLL_NAME
PR_PROVIDER_DISPLAY
PR_RESOURCE_METHODS
PR_RESOURCE_FLAGS
PR_RESOURCE_TYPE
PR_ROWID
PR_STATUS_CODE

The following property columns are optional:

PR_IDENTITY_ENTRYID
PR_IDENTITY_SEARCH_KEY
PR_STATUS_STRING

The default column set includes all of the columns defined for the table.

Setting the MAPI_UNICODE flag in the *ulFlags* parameter does the following:

- Sets the string type to Unicode for data returned for the initial active columns of the status table by the **IMAPITable::QueryColumns** method. The initial active columns for a status table are those columns **QueryColumns** returns before the service provider that contains the table calls the **IMAPITable::SetColumns** method.
- Sets the string type to Unicode for data returned for the initial active rows of the status table by the **IMAPITable::QueryRows** method. The initial active rows for a status table are those rows **QueryRows** returns before the provider that contains the table calls **SetColumns**.
- Controls the property types of the sort order returned by the **IMAPITable::QuerySortOrder** method before the provider that contains the status table calls the **IMAPITable::SortTable** method.

See Also

IMAPITable::QueryColumns method, **IMAPITable::QueryRows** method, **IMAPITable::QuerySortOrder** method, **IMAPITable::SetColumns** method, **IMAPITable::SortTable** method

IMAPISession::Logoff

The **IMAPISession::Logoff** method ends a MAPI session.

HRESULT Logoff(
  ULONG *ulUIParam***,**
  ULONG *ulFlags***,**
  ULONG *ulReserved*
)

Parameters

ulUIParam
  Input parameter containing the handle of the parent window for any dialog boxes or windows this method displays.

ulFlags
  Input parameter containing a bitmask of flags that controls how logoff is performed. The following flags can be set:

  MAPI_LOGOFF_SHARED
    Indicates all clients logged on using the shared session are notified of the shared logoff and should log off. Any client that uses the shared session can set this flag. If the specified MAPI session is not part of the shared session, this flag is ignored.

  MAPI_LOGOFF_UI
    Indicates a logoff dialog box should be displayed – some implementations of MAPI might request confirmation of logoff if operations are pending. If this flag is not set, logoff proceeds without displaying a dialog box.

ulReserved
  Reserved; must be zero.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

Remarks

Client applications call the **IMAPISession::Logoff** method to end a MAPI session. Calling **Logoff** implicitly invalidates the session object. **IMAPISession::Logoff** is similar to the Simple MAPI function **MAPILogoff**, except that **IMAPISession::Logoff** doesn't release the session and returns a value of type HRESULT rather than ULONG.

If the MAPI_LOGOFF_SHARED flag is set in the *ulFlags* parameter, notifications are sent to all clients using the shared session indicating they should also log off.

After calling **Logoff**, a client should immediately release the session object by calling the **IUnknown::Release** method. After calling **Logoff**, a call to **Release** is the only valid call for the session object.

See Also

MAPILogoff function

IMAPISession::MessageOptions

The **IMAPISession::MessageOptions** method displays a dialog box enabling a user to change options for a particular message.

HRESULT MessageOptions(
  ULONG *ulUIParam***,**
  ULONG *ulFlags***,**
  LPTSTR *lpszAdrType***,**
  LPMESSAGE *lpMessage*
)

Parameters

ulUIParam
  Input parameter containing the handle of the parent window for the dialog box.

ulFlags
  Input parameter containing a bitmask of flags that controls the type of the passed-in strings. The following flag can be set:

  MAPI_UNICODE
    Indicates the passed-in strings are in Unicode format. If the MAPI_UNICODE flag is not set, the strings are in ANSI format.

lpszAdrType
  Input parameter pointing to a string containing the messaging address type for which the options dialog box should be displayed, such as FAX, SMTP, or X500. If all registered address types should be shown, the client should pass NULL in the *lpszAdrType* parameter.

lpMessage
  Input parameter pointing to the message for which a dialog box is displayed.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

MAPI_E_NOT_FOUND
  No options were registered for the message.

Remarks

Client applications call the **IMAPISession::MessageOptions** method to get from the user a set of preferences governing message options for the message. Message option changes made by the user only go into effect after the client calls the **IMAPIProp::SaveChanges** method for the message. Changes made to the options for a particular message do not affect the default options for an address type. These options are usually message service elements specific to a transport provider.

IMAPISession::OpenAddressBook

The **IMAPISession::OpenAddressBook** method opens an address book and returns a pointer that provides further access to the open address book.

HRESULT OpenAddressBook(
  ULONG *ulUIParam***,**
  LPCIID *lpInterface***,**
  ULONG *ulFlags***,**
  LPADRBOOK FAR * *lppAdrBook*
)

Parameters

ulUIParam
Input parameter containing the handle of the parent window for the address book window and any dialog boxes this method displays.

lpInterface
Input parameter pointing to the interface identifier (IID) for the address book object. Passing NULL indicates the returned address book is cast to the standard interface for an address book object.

ulFlags
Input parameter containing a bitmask of flags that controls how the address book is returned. The following flag can be set:

AB_NO_DIALOG
Suppresses display of dialog boxes while underlying address book providers are initialized during logon, when a call to the **IMAPISession::OpenAddressBook** method generally occurs. If the AB_NO_DIALOG flag is not set, address book providers can prompt the user to correct the logon name or password, to insert a disk, or to perform other actions necessary to establish connections.

lppAdrBook
Output parameter pointing to a variable where the pointer to the returned address book object is stored.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

MAPI_E_UNKNOWN_CPID
Indicates the server is not configured to support the client's code page.

MAPI_E_UNKNOWN_LCID
Indicates the server is not configured to support the client's locale information.

MAPI_W_ERRORS_RETURNED
The call succeeded, but one or more address book providers could not be loaded. To test for this warning, use the **HR_FAILED** macro. When this warning is returned, the call should be handled as successful.

Remarks

Client applications call the **IMAPISession::OpenAddressBook** method during the logon process to get access to an address book. The returned pointer to the address book can then be used to open address book containers, find messaging users, and display address dialog boxes.

This method can return MAPI_W_ERRORS_RETURNED if it cannot load an address book provider. This value is a warning, not an error value, and a call that returns it should be handled as successful. Even if all of the address book providers failed to load, **OpenAddressBook** succeeds and returns

MAPI_W_ERRORS_RETURNED and an address book object in the *lppAdrBook* parameter. Even when no address book providers are loaded, a client can still use the **IAddrBook** interface and must release it when done.

If one or more address book providers failed to load, a client can call the **IMAPISession::GetLastError** method on the session object to obtain a **MAPIERROR** structure containing information about the providers that did not load. If more than one provider failed to load, a single **MAPIERROR** structure is returned that contains an aggregation of the strings returned by each provider.

For more information on the **HR_FAILED** macro, see Using Macros for Error Handling.

See Also

IMAPISession::GetLastError method, **MAPIERROR** structure

IMAPISession::OpenEntry

The **IMAPISession::OpenEntry** method opens an object and returns a pointer to the object to provide further access.

HRESULT OpenEntry(
  ULONG *cbEntryID*,
  LPENTRYID *lpEntryID*,
  LPCIID *lpInterface*,
  ULONG *ulFlags*,
  ULONG FAR * *lpulObjType*,
  LPUNKNOWN FAR * *lppUnk*
)

Parameters

cbEntryID
  Input parameter containing the size, in bytes, of the entry identifier pointed to by the *lpEntryID* parameter.

lpEntryID
  Input parameter pointing to the entry identifier for the object to open.

lpInterface
  Input parameter pointing to the interface identifier (IID) for the object to open. Passing NULL indicates the object is cast to the standard interface for such an object. The *lpInterface* parameter can also be set to an identifier for an appropriate interface for the object.

ulFlags
  Input parameter containing a bitmask of flags that controls how the object is opened. The following flags can be used:

  MAPI_BEST_ACCESS
    Indicates the object should be opened with the maximum network permissions allowed for the user and the maximum client application access. For example, if the client has read/write access, the object is opened with read/write access; if the client has read-only access, the object is opened with read-only access. The client can retrieve the access level by getting the PR_ACCESS_LEVEL property.

  MAPI_DEFERRED_ERRORS
    Indicates the call is allowed to succeed even if the underlying object is not accessible to the calling client. If the object is not accessible, some subsequent call to the object might return an error.

  MAPI_MODIFY
    Requests read/write access. By default, objects are created with read-only access, and clients should not work on the assumption that read/write access has been granted.

lpulObjType
  Output parameter pointing to a variable where the type of the opened object is stored.

lppUnk
  Output parameter pointing to a variable where the pointer to the opened object is stored.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

MAPI_E_NO_ACCESS
  An attempt was made to modify a read-only object or an attempt to access an object for which the user has insufficient permissions.

MAPI_E_NOT_FOUND

The object indicated by *lpEntryID* does not exist.

MAPI_E_UNKNOWN_ENTRYID

The object indicated by the *lpEntryID* parameter is not recognized. This value is typically returned if the message store or address book provider that contains the object is not open.

Remarks

Client applications call the **IMAPISession::OpenEntry** method to open objects. Using **IMAPISession::OpenEntry** is slower than using the **IMsgStore::OpenEntry** method or the **IAddrBook::OpenEntry** method, but **IMAPISession::OpenEntry** is useful if the calling client does not have information on where to locate the object to open. The **IMAPISession::OpenEntry** call returns a pointer that provides further access to the object. Default behavior is to open the object as read-only, unless the call sets the MAPI_MODIFY or MAPI_BEST_ACCESS flag in the *ulFlags* parameter.

MAPI provides a status table with information about each of the installed service providers. In each row of the status table, a status object is available with information about a particular service provider. Calling **OpenEntry** with the entry identifier of a status object found in the status table is the only way to open a status object.

Calling **OpenEntry** and passing in the entry identifier for a message store in *lpEntryID*, as a client does when it does not have information on where that message store is located, is equivalent to calling the **IMAPISession::OpenMsgStore** method with the MDB_NO_DIALOG flag set in its *ulFlags* parameter; **OpenMsgStore** called so opens a message store and returns a pointer to it without displaying a logon dialog box. When opening a message store, flags set in **OpenEntry**'s *ulFlags* parameter map to **OpenMsgStore** flags as follows:

OpenEntry flag	**OpenMsgStore equivalent**
MAPI_BEST_ACCESS	MAPI_BEST_ACCESS
MAPI_MODIFY	MDB_WRITE
MAPI_DEFERRED_ERRORS	MAPI_DEFERRED_ERRORS

The calling client should check the value returned in the *lpulObjType* parameter to determine that the object type returned is what was expected. Commonly, after the client checks the type of the object, it then casts the pointer in the *lppUnk* parameter into a message object pointer, a folder object pointer, or another appropriate object pointer.

See Also

IAddrBook::OpenEntry method, **IMAPISession::OpenMsgStore** method, **IMsgStore::OpenEntry** method

IMAPISession::OpenMsgStore

The **IMAPISession::OpenMsgStore** method opens a message store and returns a pointer that provides further access to the open store.

HRESULT OpenMsgStore(
  ULONG *ulUIParam***,**
  ULONG *cbEntryID***,**
  LPENTRYID *lpEntryID***,**
  LPCIID *lpInterface***,**
  ULONG *ulFlags***,**
  LPMDB FAR * *lppMDB*
)

Parameters

ulUIParam
: Input parameter containing the handle of the parent window for any dialog boxes or windows this method displays.

cbEntryID
: Input parameter containing the size, in bytes, of the entry identifier pointed to by the *lpEntryID* parameter.

lpEntryID
: Input parameter pointing to the entry identifier of the message store to be opened. The *lpEntryID* parameter must be non-null.

lpInterface
: Input parameter pointing to the interface identifier for the message-store object. Passing NULL indicates the object is cast to the standard interface for a message store object.

ulFlags
: Input parameter containing a bitmask of flags that controls how the message store is accessed. The following flags can be used:

  MAPI_BEST_ACCESS
  : Indicates the object should be opened with the maximum network permissions allowed for the user and the maximum client application access. For example, if the client has read/write access, the object is opened with read/write access; if the client has read-only access, the object is opened with read-only access. The client can retrieve the access level by getting the PR_ACCESS_LEVEL property.

  MAPI_DEFERRED_ERRORS
  : Indicates the call is allowed to succeed even if the underlying object is not accessible to the calling client. If the object is not accessible, some subsequent call to the object might return an error.

  MDB_NO_DIALOG
  : Prevents display of logon dialog boxes. If this flag is set, MAPI_E_LOGON_FAILED is returned if logon is unsuccessful. If this flag is not set, the message store provider can prompt the user to correct a name or password, to insert a disk, or to perform other actions necessary to establish connection to the store.

  MDB_NO_MAIL
  : Indicates the message store should not be used for sending or receiving mail. The flag signals MAPI not to notify the MAPI spooler this message store is being opened.

  MDB_TEMPORARY
  : Instructs MAPI that the store is not to be added to the message-store information table and that the store cannot be made permanent. This flag is used to log on the store so that information can be retrieved programmatically from the profile section.

MDB_WRITE
Requests read/write access to the store.

lppMDB
Output parameter pointing to a variable where the returned pointer to the message store object is stored.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

MAPI_E_NO_ACCESS
An attempt to modify a read-only object or an attempt to access an object for which the user has insufficient permissions.

MAPI_E_NOT_FOUND
The message store indicated by *lpEntryID* does not exist.

MAPI_E_UNKNOWN_CPID
Indicates the server is not configured to support the client's code page.

MAPI_E_UNKNOWN_LCID
Indicates the server is not configured to support the client's locale information.

MAPI_W_ERRORS_RETURNED
The call succeeded, but the message store provider has error information available. To test for this warning, use the **HR_FAILED** macro. When this warning is returned, the call should be handled as successful. To get the error information from the provider, call the **IMAPISession::GetLastError** method.

Remarks

Client applications call the **IMAPISession::OpenMsgStore** method during the logon process to get access to a particular message store using its entry identifier from the message store table. Default behavior is to open the object as read-only, unless the client sets the MDB_WRITE flag in the *ulFlags* parameter.

Note the following points about the default behavior of opening a message store as read-only:

- The STORE_MODIFY_OK and the STORE_CREATE_OK flags in the PR_STORE_SUPPORT_MASK property return FALSE for the store object. This return occurs only when a message store is opened as read-only using **OpenMsgStore**, not in other read-only scenarios.
- Calls to the **IMAPISession::OpenEntry** method or the **IMAPIProp::OpenProperty** method passing the flag MAPI_MODIFY, which requests read/write access, fail.
- Calls to the following methods fail: **IMAPIFolder::CreateMessage**, **IMAPIFolder::DeleteMessages**, **IMAPIFolder::CreateFolder**, **IMAPIFolder::DeleteFolder**, **IMAPIFolder::SetMessageStatus**, **IMAPIProp::SetProps**, **IMAPIProp::DeleteProps**.
- Calls to the following methods fail if the copying destination is within the same read-only message store as the copying source: **IMAPIFolder::CopyMessages**, **IMAPIFolder::CopyFolder**, **IMAPIProp::CopyTo**.

For more information on using the **HR_FAILED** macro, see Using Macros for Error Handling.

IMAPISession::OpenProfileSection

The **IMAPISession::OpenProfileSection** method opens a section of the current profile and returns a pointer that provides further access to the profile object.

HRESULT OpenProfileSection(
  LPMAPIUID *lpUID*,
  LPCIID *lpInterface*,
  ULONG *ulFlags*,
  LPPROFSECT FAR * *lppProfSect*
)

Parameters

lpUID
: Input parameter pointing to the **MAPIUID** structure holding the MAPI unique identifier (MAPIUID) for the profile section.

lpInterface
: Input parameter pointing to the interface identifier (IID) for the profile section. Passing NULL indicates the profile section object is cast to the standard interface for a profile section. The *lpInterface* parameter can also be set to an identifier for an appropriate interface for the object. Valid interface identifiers are IID_IMAPIProp and IID_IProfSect.

ulFlags
: Input parameter containing a bitmask of flags that controls access to the profile section. The following flags can be set:

  MAPI_DEFERRED_ERRORS
  : Indicates the call is allowed to succeed even if the underlying object is not accessible to the calling client. If the object is not accessible, some subsequent call to the object might return an error.

  MAPI_MODIFY
  : Requests read/write access. By default, objects are created with read-only access, and client applications should not work on the assumption that read/write access has been granted.

lppProfSect
: Output parameter pointing to a variable where the pointer to the returned profile object is stored.

Return Values

S_OK
: The call succeeded and has returned the expected value or values.

MAPI_E_NO_ACCESS
: An attempt was made to modify a read-only profile section or an attempt to access an object for which the user has insufficient permissions.

MAPI_E_NOT_FOUND
: The requested object does not exist.

Remarks

Client applications call the **IMAPISession::OpenProfileSection** method to open a profile section for reading information from and writing information to the active profile for the session. A profile section object supporting the **IProfSect** interface is returned in the *lppProfSect* parameter. Default behavior is to open the profile section as read-only, unless a client sets the MAPI_MODIFY flag in the *ulFlags* parameter. Profile sections belonging to service providers cannot be opened by calls to **OpenProfileSection**.

More than one method call can open a profile section with read-only access at a time, but only one method call can open a profile section with read/write access at a time. If any other client has the profile section open, a read/write open operation fails and returns MAPI_E_NO_ACCESS. A read-only open operation fails if the section is open for writing.

If an **OpenProfileSection** call opens a nonexistent profile section by passing MAPI_MODIFY in *ulFlags*, the call creates the section. If an **OpenProfileSection** call attempts to open a nonexistent section with read-only access, it returns MAPI_E_NOT_FOUND.

See Also

IMAPIProp : IUnknown interface, **IProfSect : IMAPIProp** interface, **MAPIUID** structure

IMAPISession::PrepareForm

The **IMAPISession::PrepareForm** method creates a message instance for use by the **IMAPISession::ShowForm** method.

HRESULT PrepareForm(
  LPCIID *lpInterface***,**
  LPMESSAGE *lpMessage***,**
  ULONG FAR * *lpulMessageToken*
)

Parameters

lpInterface
  Input parameter pointing to the interface identifier (IID) for the message object to create. Passing NULL indicates the message object is cast to the standard interface for a message object. The *lpInterface* parameter can also be set to IID_IMessage, which is the only valid IID for the message object.

lpMessage
  Input parameter pointing to the message object.

lpulMessageToken
  Output parameter pointing to a returned message token, which must be passed in the following **ShowForm** call.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

Remarks

Client applications call the **IMAPISession::PrepareForm** method to create a message instance for use by **IMAPISession::ShowForm**. Clients should only have a single reference to the message passed in **PrepareForm**'s *lpMessage* parameter. If the call to **PrepareForm** succeeds, the client must release the message, then call **ShowForm** and pass in the *ulMessageToken* parameter the message token returned in **PrepareForm**'s *lpulMessageToken* parameter. Failure to do so causes memory leaks.

See Also

IMAPISession::ShowForm method

IMAPISession::QueryDefaultMessageOpt

The **IMAPISession::QueryDefaultMessageOpt** method returns the available message options and their default settings for a particular messaging address type.

HRESULT QueryDefaultMessageOpt(
  LPTSTR *lpszAdrType***,**
  ULONG *ulFlags***,**
  ULONG FAR * *lpcValues***,**
  LPSPropValue FAR * *lppOptions*
)

Parameters

lpszAdrType
  Input parameter pointing to a string containing the messaging address type in question, such as FAX, SMTP, or X500.

ulFlags
  Input parameter containing a bitmask of flags that controls the type of the passed-in and returned strings. The following flag can be set:

  MAPI_UNICODE
    Indicates the passed-in and returned strings are in Unicode format. If the MAPI_UNICODE flag is not set, the strings are in ANSI format.

lpcValues
  Output parameter pointing to the number of property values returned in the *lppOptions* parameter.

lppOptions
  Output parameter pointing to a variable where a pointer to the returned array of **SPropValue** structures is stored. The **SPropValue** structures contain available message options and their default values.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

Remarks

Client applications call the **IMAPISession::QueryDefaultMessageOpt** method to find out what message options are available for a particular messaging address type. The returned **SPropValue** property value array includes the property identifier for each message option and its default property value, if there is one.

Message options are properties of a message that control its behavior after it is submitted to the transport provider; they are part of the message envelope, not its content. They are not usually specific to a particular address type.

IMAPISession::QueryIdentity

The **IMAPISession::QueryIdentity** method returns an entry identifier that represents the object acting as the primary identity for the current MAPI session.

HRESULT QueryIdentity(
  ULONG FAR * *lpcbEntryID*,
  LPENTRYID FAR * *lppEntryID*
)

Parameters

lpcbEntryID
  Output parameter pointing to a variable in which is returned the size, in bytes, of the entry identifier pointed to by the *lppEntryID* parameter.

lppEntryID
  Output parameter pointing to a variable where the pointer to the newly created entry identifier is stored.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

MAPI_W_NO_SERVICE
  The call succeeded, but no provider can provide the primary identifier. To test for this warning, use the **HR_FAILED** macro. When this warning is returned, the call should be handled as successful.

Remarks

Client applications call the **IMAPSession::QueryIdentity** method to retrieve a user's primary identity for the current session.

A *primary identity* is a string that represents the user of a MAPI session. Each message service provider that MAPI has information about establishes an identity for each of its users. This identity can be established when a client logs onto the service. However, because MAPI supports connections to multiple service providers for each MAPI session, there is no firm definition of a particular user's identity for the MAPI session as a whole; a user's identity depends on which service is involved. Service providers that utilize the functionality provided by primary identities should set the STATUS_PRIMARY_IDENTITY flag in the PR_RESOURCE_FLAGS property. Clients can call the **IMsgServiceAdmin::SetPrimaryIdentity** method to designate one of the many identities established for a user by message service providers as the primary identity for that user.

The **QueryIdentity** method provides clients a convenient way to retrieve a primary identity. Once a client has a user's primary identity, it can call the session object's **IMAPISession::OpenEntry** method and query the resulting interface to obtain properties associated with that primary identity from the returned entry identifiers, such as the display name or messaging address. Sophisticated clients that are aware of the service provider, or the providers within whose context they operate, can browse the status table to determine users' identities in the context of the current service providers rather than using **QueryIdentity**. More traditional clients, which work with a single identity for a session, use **QueryIdentity** to obtain a user's primary identity.

QueryIdentity returns the best information MAPI can locate for the user's primary identity for a session. Ideally, this is the PR_IDENTITY_ENTRYID property from the status row tagged with STATUS_PRIMARY_IDENTITY, although there are other possibilities:

- If no provider can provide a primary identifier, **QueryIdentity** succeeds with the warning MAPI_W_NO_SERVICE and returns a hard-coded entry identifier in *lppEntryID*.
- If some rows make available PR_IDENTITY_ENTRYID but no row is tagged with

STATUS_PRIMARY_IDENTITY, **QueryIdentity** returns the first entry identifier found.
- If a row is tagged with STATUS_PRIMARY_IDENTITY but holds no entry identifier, **QueryIdentity** returns a custom-recipient entry identifier built with other information from that row.

When a client has finished using the entry identifier for the primary identity returned by **QueryIdentity**, it should free the memory that held it by using the **MAPIFreeBuffer** function.

For more information on using the **HR_FAILED** macro, see Using Macros for Error Handling.

See Also

IMAPISession::OpenEntry method, **IMsgServiceAdmin::SetPrimaryIdentity** method, **MAPIFreeBuffer** function

IMAPISession::SetDefaultStore

The **IMAPISession::SetDefaultStore** method sets a message store provider as the default.

HRESULT SetDefaultStore(
  ULONG *ulFlags*,
  ULONG *cbEntryID*,
  LPENTRYID *lpEntryID*
)

Parameters

ulFlags
Input parameter containing a bitmask of flags that controls the setting of the default message store. The flags used for the *ulFlags* parameter are mutually exclusive, so only one flag can be set for each call to the **IMAPISession::SetDefaultStore** method. The following flags can be set:

MAPI_DEFAULT_STORE
Sets the STATUS_DEFAULT_STORE flag in the PR_RESOURCE_FLAGS property column of the status table.

MAPI_PRIMARY_STORE
Sets the STATUS_PRIMARY_STORE flag in the PR_RESOURCE_FLAGS column of the status table. This flag indicates the store that clients should try to use when they log on. If the primary store is not the default store, clients should set it as the default.

MAPI_SECONDARY_STORE
Sets the STATUS_SECONDARY_STORE flag in the PR_RESOURCE_FLAGS column of the status table. This flag indicates the store that clients should try to use if the primary store is not available. If a client cannot open the primary store, it should open the secondary store and set it as the default store.

MAPI_SIMPLE_STORE_PERMANENT
Causes the STATUS_SIMPLE_STORE flag in the PR_RESOURCE_FLAGS column of the status table to be set and permanently saved in the profile in addition to updating the status and message store tables.

MAPI_SIMPLE_STORE_TEMPORARY
Causes modification STATUS_SIMPLE_STORE flag in the PR_RESOURCE_FLAGS column of the status table and message store table to be set, but does not save the settings permanently in the profile.

cbEntryID
Input parameter containing the size, in bytes, of the entry identifier pointed to by the *lpEntryID* parameter.

lpEntryID
Input parameter pointing to the entry identifier of the message store object intended as the default. If a client passes NULL in *lpEntryID*, no message store is selected as the default.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

Remarks

Client applications call the **IMAPISession::SetDefaultStore** method to make a single call to reset the default store that message store operations take place in.

Information on which message store is the default is kept in the message-store information table. Information in this table is consistent with information in the status table; the message-store information table has a PR_RESOURCE_FLAGS property column with the same values as the PR_RESOURCE_FLAGS property column in the status table.

When either the MAPI_DEFAULT_STORE or the MAPI_SIMPLE_STORE_PERMANENT flag is set changing the default store, the profile is updated along with any message-store information table or status table open for the profile. Whenever a change is made to the message store default setting, the following notifications are generated:

- An fnevTableModified event notification is issued for each of the affected rows in both the message-store table and the status table.
- An internal notification is issued to the MAPI spooler. Operations already in progress are completed without change; new operations involving the default message store, such as message downloading, are processed for the new default store.

No special message store capabilities are required for the simple store. However, setting the default store fails if the target store does not set the STORE_SUBMIT_OK, STORE_CREATE_OK, and STORE_MODIFY_OK flags in the PR_STORE_SUPPORT_MASK property.

See Also

PR_RESOURCE_FLAGS property, PR_STORE_SUPPORT_MASK property, **TABLE_NOTIFICATION** structure

IMAPISession::ShowForm

The **IMAPISession::ShowForm** method displays a message form for editing and sending.

HRESULT ShowForm(
  ULONG *ulUIParam***,**
  LPMDB *lpMsgStore***,**
  LPMAPIFOLDER *lpParentFolder***,**
  LPCIID *lpInterface***,**
  ULONG *ulMessageToken***,**
  LPMESSAGE *lpMessageSent***,**
  ULONG *ulFlags***,**
  ULONG *ulMessageStatus***,**
  ULONG *ulMessageFlags***,**
  ULONG *ulAccess***,**
  LPSTR *lpszMessageClass*
)

Parameters

ulUIParam
: Input parameter containing the handle of the parent window for the window in which the form is displayed or NULL.

lpMsgStore
: Input parameter pointing to the message store containing the folder pointed to by the *lpParentFolder* parameters.

lpParentFolder
: Input parameter pointing to the folder in which the message was created or NULL.

lpInterface
: Reserved; must be NULL.

ulMessageToken
: Input parameter containing the message token returned by the preceding call to the **IMAPISession::PrepareForm** method. The same token returned by the **PrepareForm** call must be passed in **ShowForm**, or the message object is not released.

lpMessageSent
: Reserved; must be NULL.

ulFlags
: Input parameter containing a bitmask of flags that controls how and whether the message is saved. The following flags can be set:

  MAPI_NEW_MESSAGE
  : Indicates the message has not previously been saved (that is, the **IMAPIProp::SaveChanges** method has not been called on the message).

  MAPI_POST_MESSAGE
  : Indicates the message should be saved to its parent folder. The message is not processed for sending but posted to the folder instead. If this flag is not set, the message is copied to the Outbox and processed for sending.

ulMessageStatus
: Input parameter containing a bitmask of client- or provider-defined flags, copied from the PR_MSG_STATUS property of the message referenced to the message in the *ulMessageToken* parameter. The flags provide information on the state of the message.

ulMessageFlags
: Input parameter pointing to a bitmask of flags, copied from the PR_MESSAGE_FLAGS property of the message referenced to the message in *ulMessageToken*. The flags provide further information

on the state of the message.

ulAccess

Input parameter containing flags copied from the PR_ACCESS_LEVEL property of the message referenced to the message in *ulMessageToken*. The flags indicate whether the message has read/write or read-only access.

lpszMessageClass

Input parameter pointing to a string naming the message class of the message referenced to the message in *ulMessageToken*. This string is copied from the PR_MESSAGE_CLASS property of the message.

Return Values

S_OK

The call succeeded and has returned the expected value or values.

MAPI_E_USER_CANCEL

The user canceled the operation, typically by clicking the **Cancel** button in a dialog box.

Remarks

Client applications call the **IMAPISession::ShowForm** method together with **IMAPISession::PrepareForm** to display a message form for editing and sending. Clients should only have a single reference to the message passed in **PrepareForm**'s *lpMessage* parameter. If the call to **PrepareForm** succeeds, the client must release the message, then call **ShowForm** and pass in *ulMessageToken* the message token returned in **PrepareForm**'s *lpulMessageToken* parameter. Failure to do so causes memory leaks.

Form implementations can return error codes other than the ones documented by MAPI. If the client application is able to use these error codes to make a more accurate determination of the error condition, it can do so; otherwise, it should treat these errors as if the return code was MAPI_E_CALL_FAILED.

See Also

IMAPISession::PrepareForm method

IMAPISession::Unadvise

The **IMAPISession::Unadvise** method removes an object's registration for notification of changes previously established with a call to the **IMAPISession::Advise** method.

HRESULT Unadvise(
  ULONG *ulConnection*
)

Parameters

ulConnection
  Input parameter containing the number of the registration connection previously returned by a call to **IMAPISession::Advise**.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

Remarks

Client applications call the **IMAPISession::Unadvise** method to release the pointer to the advise sink object passed in the *lpAdviseSink* parameter in the previous call to **IMAPISession::Advise**, thereby canceling a notification registration. As part of discarding the pointer to the advise sink object, the object's **IUnknown::Release** method is called. Generally, **Release** is called during the **Unadvise** call, but if another thread is in the process of calling the **IMAPIAdviseSink::OnNotify** method for the advise sink object, the **Release** call is delayed until the **OnNotify** method returns.

See Also

IMAPIAdviseSink::OnNotify method, **IMAPISession::Advise** method

IMAPIStatus : IMAPIProp

The **IMAPIStatus** interface is used by service providers to support client application requests for information that is not maintained in the status table. Clients can call one of the logon object's **OpenStatusEntry** methods to retrieve a pointer to a provider status object. With a status object pointer, clients can call one of the four methods in the **IMAPIStatus** interface: **ValidateState**, **SettingsDialog**, **ChangePassword**, or **FlushQueues**. However, providers are only required to support **ValidateState**; they can return MAPI_E_NO_SUPPORT from their implementations of the other three methods.

At a Glance

Specified in header file:	MAPIDEFS.H
Object that supplies this interface:	Status object
Corresponding pointer type:	LPMAPISTATUS
Implemented by:	Service providers
Transaction model:	Non-transacted
Called by:	Client applications

Vtable Order

ValidateState	Confirms the external status information available for a transport provider by checking with the transport provider itself.
SettingsDialog	Displays a dialog box enabling the user to change the configuration of the active service provider.
ChangePassword	Changes a password specific to a service provider, without displaying a user interface.
FlushQueues	Forces all messages waiting to be sent or received by a particular transport provider to be uploaded or downloaded synchronously.

Required Properties

PR_DISPLAY_NAME	Read/write
PR_ENTRYID	Read-only
PR_PROVIDER_DLL_NAME	Read-only
PR_RESOURCE_FLAGS	Read-only
PR_RESOURCE_METHODS	Read-only
PR_RESOURCE_TYPE	Read-only
PR_STATUS_CODE	Read-only

IMAPIStatus::ChangePassword

The **IMAPIStatus::ChangePassword** method changes a password specific to a service provider, without displaying a user interface.

HRESULT ChangePassword(
  LPTSTR *lpOldPass*,
  LPTSTR *lpNewPass*,
  ULONG *ulFlags*
)

Parameters

lpOldPass
 Input parameter pointing to a string containing the old password.

lpNewPass
 Input parameter pointing to a string containing the new password.

ulFlags
 Input parameter containing a bitmask of flags that controls the type of the password strings. The following flag can be set:

 MAPI_UNICODE
  Indicates the strings are in Unicode format. If the MAPI_UNICODE flag is not set, the strings are in ANSI format.

Return Values

S_OK
 The call succeeded and has returned the expected value or values.

MAPI_E_NO_ACCESS
 The password is wrong or invalid.

MAPI_E_NO_SUPPORT
 The STATUS_CHANGE_PASSWORD flag is not set in the PR_RESOURCE_METHODS property.

See Also

PR_RESOURCE_METHODS property

IMAPIStatus::FlushQueues

The **IMAPIStatus::FlushQueues** method forces all messages waiting to be sent or received by a particular transport provider to be uploaded or downloaded synchronously.

HRESULT FlushQueues(
  ULONG *ulUIParam***,**
  ULONG *cbTargetTransport***,**
  LPENTRYID *lpTargetTransport***,**
  ULONG *ulFlags*
)

Parameters

ulUIParam
  Input parameter containing the handle of the parent window for any dialog boxes or windows this method displays.

cbTargetTransport
  Input parameter containing the size, in bytes, of the entry identifier pointed to by the *lpTargetTransport* parameter. The *cbTargetTransport* parameter is only valid when called by the MAPI spooler's status object. To flush all queues, pass zero in *cbTargetTransport*.

lpTargetTransport
  Input parameter pointing to the entry identifier of the transport provider for which message queues are to be flushed – that is, emptied by uploading or downloading. The *lpTargetTransport* parameter is only valid when called from the MAPI spooler's status object. To flush all queues, pass NULL in *lpTargetTransport*.

ulFlags
  Input parameter containing a bitmask of flags that controls how message queue flushing is done. The following flags can be set:

  FLUSH_ASYNC_OK
    Notifies the MAPI spooler that the queues can be flushed asynchronously and return before the operation is complete. This flag only applies to the MAPI spooler's status object. Client applications can register for notifications on the MAPI spooler's status row to receive notifications when the flushing operation is complete.

  FLUSH_DOWNLOAD
    Indicates the inbound message queue or queues should be flushed.

  FLUSH_FORCE
    Indicates the transport provider should process this flush request if possible, even if doing so is time-consuming. Asynchronous transport providers cannot respond to unforced **IMAPIStatus::FlushQueues** and **IXPLogon::FlushQueues** method calls.

  FLUSH_NO_UI
    Indicates the transport provider should not display a user interface. This flag is used only by the MAPI spooler; providers ignore this flag.

  FLUSH_UPLOAD
    Indicates the outbound message queue or queues should be flushed.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

MAPI_E_BUSY
  Another operation is in progress; it should be allowed to complete, or it should be stopped, before this operation is attempted.

MAPI_E_NO_SUPPORT

The STATUS_FLUSH_QUEUES flag is not set in the PR_RESOURCE_METHODS property.

Remarks

Use the **IMAPIStatus::FlushQueues** method to force all messages in a particular transport provider's inbound or outbound message queue to be uploaded or downloaded. **FlushQueues** is only available from the status object. Queues are flushed synchronously when **FlushQueues** is called to the MAPI spooler and asynchronously when called to a specific transport.

Unless the FLUSH_NO_UI flag is set in the *ulFlags* parameter, the MAPI spooler displays a user interface indicating progress during the flushing operation. Transport providers ignore requests for a user interface. **FlushQueues** processing can take a long time; MAPI_E_BUSY should be returned for asynchronous requests so that clients can continue work. Transport providers can ignore **FlushQueues** requests and return MAPI_E_NO_SUPPORT if they cannot handle them.

See Also

PR_RESOURCE_METHODS property

IMAPIStatus::SettingsDialog

The **IMAPIStatus::SettingsDialog** method displays a dialog box enabling the user to change the configuration of the active service provider.

HRESULT SettingsDialog(
  ULONG *ulUIParam***,**
  ULONG *ulFlags*
)

Parameters

ulUIParam
  Input parameter containing the handle of the parent window for the dialog box.

ulFlags
  Input parameter containing a bitmask of flags that controls how the configuration dialog box is displayed. The following flag can be set:

  UI_READONLY
    Suggests the provider not enable users to change provider settings. This flag can be ignored.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

MAPI_E_NO_SUPPORT
  The STATUS_SETTINGS_DIALOG flag is not set in the PR_RESOURCE_METHODS property.

Remarks

Use the **IMAPIStatus::SettingsDialog** method to have a service provider display a user interface for configuration, usually for the user to make changes to property sheets. A call to **SettingsDialog** commonly displays a user interface, so **SettingsDialog** should only be used by interactive applications.

IMAPIStatus::ValidateState

The **IMAPIStatus::ValidateState** method confirms the external status information available for a transport provider by checking with the transport provider itself.

HRESULT ValidateState(
  ULONG *ulUIParam***,**
  ULONG *ulFlags*
)

Parameters

ulUIParam
  Input parameter containing the handle of the parent window for any dialog boxes or windows this method displays.

ulFlags
  Input parameter containing a bitmask of flags that controls how the status check is done and its results. The following flags can be set:

  ABORT_XP_HEADER_OPERATION
    Indicates the user canceled the operation, typically by clicking the **Cancel** button in a dialog box. The transport provider has the option to continue working on the operation, or it can abort the operation and return MAPI_E_USER_CANCELED.

  CONFIG_CHANGED
    Validates the state of the currently loaded transport providers by calling their **IXPLogon::AddressTypes** and **IMAPISession::MessageOptions** methods at the next convenient time, when set for calls to the MAPI spooler. This flag also can be set to allow the MAPI spooler to correct critical transport provider failures without forcing client applications to log off and then log on again. When a specific transport provider is called with this flag set, the transport provider calls the **IMAPISupport::SpoolerNotify** method to notify the MAPI spooler if its profile section has been updated.

  FORCE_XP_CONNECT
    Indicates the user selected a connect operation. When this flag is used with the REFRESH_XP_HEADER_CACHE or PROCESS_XP_HEADER_CACHE flag, the connect action occurs without caching.

  FORCE_XP_DISCONNECT
    Indicates the user selected a disconnect operation. When this flag is used with the REFRESH_XP_HEADER_CACHE or PROCESS_XP_HEADER_CACHE flag, the disconnect action occurs without caching.

  PROCESS_XP_HEADER_CACHE
    Indicates that entries in the header cache table should be processed, that all messages marked with the MSGSTATUS_REMOTE_DOWNLOAD flag should be downloaded, and that all messages marked with the MSGSTATUS_REMOTE_DELETE flag should be deleted. Messages that have both MSGSTATUS_REMOTE_DOWNLOAD and MSGSTATUS_REMOTE_DELETE set should be moved.

  REFRESH_XP_HEADER_CACHE
    Indicates that a new list of message headers should be downloaded and that all message status marking flags should be cleared.

  SUPPRESS_UI
    Prevents the transport provider from displaying a user interface.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

MAPI_E_BUSY
: Another operation is in progress; it should be allowed to complete, or it should be stopped, before this operation is attempted.

MAPI_E_NO_SUPPORT
: The STATUS_VALIDATE_STATE flag is not set in the PR_RESOURCE_METHODS property.

MAPI_E_USER_CANCEL
: The user canceled the operation, typically by clicking the **Cancel** button in a dialog box. This value is only returned by remote transport providers.

Remarks

Use the **IMAPIStatus::ValidateState** method to have a transport provider check its internal state and ensure that this internal state is consistent with its status table row settings. **ValidateState** calls can take a long time.

IMAPISupport : IUnknown

MAPI provides four different support objects. There is one support object for each type of service provider and a configuration support object that all providers receive when their message service entry point function is called. There are a few **IMAPISupport** methods that are used in all of the support objects, such as the **GetLastError** method. Other **IMAPISupport** methods are specific to one type of support object. For example, the **CopyMessages** method is only used by message store providers. Methods that are not implemented for a particular type of support object return MAPI_E_NO_SUPPORT when called.

The following table indicates the methods that are implemented for the different types of support objects.

Method	Type of support object
Address	Address book
CompareEntryIDs	Address book, message store, transport
CompleteMsg	Message store
CopyFolder	Message store
CopyMessages	Message store
CreateOneOff	Address book, message store, and transport providers
Details	Address book
DoConfigPropsheet	Address book, message store, transport, configuration support
DoCopyProps	Message store
DoCopyTo	Message store
DoProgressDialog	Address book and message store
DoSentMail	Message store
ExpandRecips	Message store
GetLastError	Address book, message store, transport, configuration support
GetMemAllocRoutines	Address book, message store, transport, configuration support
GetOneOffTable	Address book
GetSvcConfigSupportObj	Address book, message store, transport, configuration support
IStorageFromStream	Address book, message store, transport
MakeInvalid	Address book, message store, transport, configuration support
ModifyProfile	Message store
ModifyStatusRow	Address book, message store, transport
NewEntry	Address book
NewUID	Address book, message store, transport, configuration support
Notify	Address book, message store,

	transport
OpenAddressBook	Address book, message store, transport
OpenEntry	Address book, message store, transport
OpenProfileSection	Address book, message store, transport, configuration support
OpenTemplateID	Address book
PrepareSubmit	Message store
ReadReceipt	Message store
RegisterPreprocessor	Transport
SetProviderUID	Address book and message store
SpoolerNotify	Message store and transport
SpoolerYield	Transport
StatusRecips	Transport
StoreLogoffTransports	Message store
Subscribe	Address book, message store, transport
Unsubscribe	Address book, message store, transport
WrapStoreEntryID	Address book, message store, transport

At a Glance

Specified in header file:	MAPISPI.H
Object that supplies this interface:	Support object
Corresponding pointer type:	LPMAPISUP
Implemented by:	MAPI
Called by:	Service providers

Vtable Order

GetLastError	Returns a **MAPIERROR** structure containing information about the last error that occurred for a support object.
GetMemAllocRoutines	Retrieves the addresses of the MAPI memory allocation and deallocation functions, **MAPIAllocateBuffer**, **MAPIAllocateMore**, and **MAPIFreeBuffer**.
Subscribe	Sets up a subscription for notification events with the MAPI.
Unsubscribe	Removes an object's subscription for notification of changes previously established with a call to the **IMAPISupport::Subscribe** method.
Notify	Notifies registered applications about changes to an object that the service provider owns.
ModifyStatusRow	Creates or modifies a service provider's status

	table row.
OpenProfileSection	Opens a section of the current profile and returns a pointer that provides further access.
RegisterPreprocessor	Registers a preprocessor function for a transport provider.
NewUID	Returns a new MAPI unique identifier (MAPIUID) for an item.
MakeInvalid	Invalidates an object derived from the **IUnknown** interface.
SpoolerYield	Allows the transport provider to permit the MAPI spooler to give processing time to Windows.
SpoolerNotify	Informs the MAPI spooler the transport provider requires service.
CreateOneOff	Creates an entry identifier for a custom recipient.
SetProviderUID	Informs MAPI of the MAPI unique identifier (MAPIUID) assigned to the service provider using the current support object.
CompareEntryIDs	Compares two entry identifiers to determine if they refer to the same object.
OpenTemplateID	Allows run-time binding of one address book provider's code to data for an entry in another address book provider, so the entry's properties can later be updated. This method is called only for interaction between address book providers.
OpenEntry	Opens an object given its entry identifier.
GetOneOffTable	Returns a table of templates for custom recipient addresses that can be used to create recipients for a message.
Address	Updates the list of recipients for a message by displaying MAPI's default address dialog box.
Details	Displays MAPI's default dialog box showing details about an address book entry.
NewEntry	Displays MAPI's default dialog box for creating new entries within a container or custom recipient addresses within a message.
DoConfigPropsheet	Enables a service provider to display a property sheet, usually for configuration, using MAPI's default user interface.
CopyMessages	Copies or moves messages from one folder to another.
CopyFolder	Copies or moves a subfolder from its current parent folder to another folder.
DoCopyTo	Copies or moves all properties from a source object to a destination object, except for a given set of excluded properties.
DoCopyProps	Copies or moves a selected set of properties from a source object to a destination object.
DoProgressDialog	Provides a default implementation of a progress indicator for use by service providers.
ReadReceipt	Generates a read or nonread report for a

	message.
PrepareSubmit	Prepares a message for submission to the MAPI spooler.
ExpandRecips	Completes all recipient lists and expands certain distribution lists for the transmission of a message.
DoSentMail	Generates for an open message a corresponding message in the SentItems folder.
OpenAddressBook	Opens an address book and returns a pointer that provides further access to the open address book.
CompleteMsg	Calls the MAPI spooler to complete message delivery processing. This method is called only by message store providers that are tightly coupled with service providers.
StoreLogoffTransports	Specifies the orderly release of a message store to the MAPI spooler.
StatusRecips	Generates delivery and nondelivery reports on behalf of transport providers.
WrapStoreEntryID	Maps the private entry identifier of a message store object to an entry identifier more useful to the messaging system.
ModifyProfile	Makes the profile section for a message store provider permanent.
IStorageFromStream	Implements a storage object to access a stream.
GetSvcConfigSupportObj	Creates a new support object for use by calls to a message service's entry point function.

IMAPISupport::Address

The **IMAPISupport::Address** method updates the list of recipients for a message by displaying MAPI's default address dialog box.

HRESULT Address(
  ULONG FAR * *lpulUIParam***,**
  LPADRPARM *lpAdrParms***,**
  LPADRLIST FAR * *lppAdrList*
)

Parameters

lpulUIParam
  Input-output parameter containing the handle of the parent window of the dialog box. On input, a window handle must always be passed. On output, if the DIALOG_SDI flag is set in the **ADRPARM** structure pointed to by the *lpAdrParms* parameter, then the window handle of the modeless dialog box is returned.

lpAdrParms
  Input-output parameter pointing to an **ADRPARM** structure that controls the presentation and behavior of the address dialog box.

lppAdrList
  Input-output parameter pointing to a variable where the pointer to an **ADRLIST** structure holding the recipient list is stored. On input, this list is the current list of recipients in a message; on output, the list is an updated recipient list. The *lppAdrList* parameter can be NULL on input.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

Remarks

When an address book provider calls the **IMAPISupport::Address** method it passes the current list of recipients for a message, possibly empty, in the *lppAdrList* parameter, and **Address** returns in *lppAdrList* an **ADRLIST** structure holding an updated list of recipients. Client applications can use the updated list to set the recipients of a message by using the **IMessage::ModifyRecipients** method.

The recipient entries in the passed and returned **ADRLIST** structures consist of **ADRENTRY** structures, with each holding an individual recipient. The **ADRENTRY** structures are organized in the **ADRLIST** by which type of recipient they hold, as indicated by the recipient's PR_RECIPIENT_TYPE property. The possible types are MAPI_TO (that is, a primary recipient), MAPI_CC (a recipient that receives a copy of a message), and MAPI_BCC (a recipient that receives a blind carbon copy of a message).

ADRLIST structures can hold both resolved and unresolved recipient entries. An *unresolved entry* does not have a PR_ENTRYID property. Client applications that enable users to type recipient names directly into a message, in addition to choosing names from a list, create unresolved entries by creating an **ADRENTRY** with just the PR_DISPLAY_NAME and PR_RECIPIENT_TYPE properties. A *resolved entry* contains at least the following properties:

  PR_ENTRYID
  PR_RECIPIENT_TYPE
  PR_DISPLAY_NAME
  PR_ADDRTYPE
  PR_DISPLAY_TYPE

An address book provider should use the pointers to the MAPI memory allocation functions passed in during provider initialization to allocate memory. Memory for the **ADRLIST** structure passed by **Address** on output and each property value structure held within that **ADRLIST** must be allocated with the **MAPIAllocateBuffer** function. If, on output, **Address** needs to pass a larger **ADRLIST** structure than passed in by the calling provider, or if NULL is passed in *lppAdrList* on input, then **Address** allocates a larger buffer for the **ADRLIST** structure it returns using **MAPIAllocateBuffer** and returns this buffer's address in *lppAdrList*. **Address** frees the old buffer by calling the **MAPIFreeBuffer** function. The **Address** method also allocates additional property value structures in the **ADRLIST** and frees old ones as appropriate.

Address returns immediately if the DIALOG_SDI flag was set in the **ADRPARM** structure in the *lpAdrParms* parameter.

See Also

ADRENTRY structure, **ADRLIST** structure, **ADRPARM** structure, **FreePadrlist** function, **FreeProws** function, **IMAPISupport::GetMemAllocRoutines** method, **IMAPITable::QueryRows** method, **IMessage::ModifyRecipients** method, **MAPIAllocateBuffer** function, **MAPIAllocateMore** function, **MAPIFreeBuffer** function, **SPropValue** structure, **SRowSet** structure

IMAPISupport::CompareEntryIDs

The **IMAPISupport::CompareEntryIDs** method compares two entry identifiers to determine if they refer to the same object.

HRESULT CompareEntryIDs(
  ULONG *cbEntryID1***,**
  LPENTRYID *lpEntryID1***,**
  ULONG *cbEntryID2***,**
  LPENTRYID *lpEntryID2***,**
  ULONG *ulFlags***,**
  ULONG FAR * *lpulResult*
)

Parameters

cbEntryID1
  Input parameter containing the size, in bytes, of the entry identifier pointed to by the *lpEntryID1* parameter.

lpEntryID1
  Input parameter pointing to the first entry identifier to be compared.

cbEntryID2
  Input parameter containing the size, in bytes, of the entry identifier pointed to by the *lpEntryID2* parameter.

lpEntryID2
  Input parameter pointing to the second entry identifier to be compared.

ulFlags
  Reserved; must be zero.

lpulResult
  Output parameter pointing to a variable where the returned result of the comparison is stored; this variable is TRUE if the two entry identifiers refer to the same object, and FALSE otherwise.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

MAPI_E_UNKNOWN_ENTRYID
  The requested entry identifier does not exist.

Remarks

Service providers call the **IMAPISupport::CompareEntryIDs** method to compare two entry identifiers for a given entry within a service provider to determine whether they refer to the same object. If the two entry identifiers refer to the same object, then **CompareEntryIDs** sets the *lpulResult* parameter to TRUE; if they refer to different objects, **CompareEntryIDs** sets *lpulResult* to FALSE.

CompareEntryIDs is useful because an object can have more than one valid entry identifier; such a situation can occur, for example, after a new version of a service provider is installed.

If **CompareEntryIDs** returns an error, the calling provider should not take any action based on an assumption about the comparison's results. It should instead take the most conservative approach to the action it is trying to perform.

CompareEntryIDs might fail if, for example, no provider has registered for one of the entry identifiers compared. If a provider compares message store entry identifiers when one or both of the stores has not yet opened, **CompareEntryIDs** returns MAPI_E_UNKNOWN_ENTRYID.

IMAPISupport::CompleteMsg

The **IMAPISupport::CompleteMsg** method calls the MAPI spooler to complete message delivery processing. This method is called only by message store providers that are tightly coupled with transport providers.

```
HRESULT CompleteMsg(
  ULONG ulFlags,
  ULONG cbEntryID,
  LPENTRYID lpEntryID
 )
```

Parameters

ulFlags
Reserved; must be zero.

cbEntryID
Input parameter containing the size, in bytes, of the entry identifier pointed to by the *lpEntryID* parameter.

lpEntryID
Input parameter pointing to the entry identifier of the message for which to finish processing.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

Remarks

Message store providers call the **IMAPISupport::CompleteMsg** method to instruct the MAPI spooler to complete postprocessing for a message. **CompleteMsg** is available to all message store providers but should only be called by those store providers that are tightly coupled with transport providers. Those store providers should only call **CompleteMsg** in the following circumstances:

- When the message in question required preprocessing.
- When the store provider can handle all recipients of the message, the transport provider the message store is coupled with can handle all message recipients without involving the MAPI spooler, and the MAPI spooler's transport provider order indicates the transport provider can handle all the recipients.

IMAPISupport::CopyFolder

The **IMAPISupport::CopyFolder** method copies or moves a folder from its current parent folder to another.

HRESULT CopyFolder(
  LPCIID *lpSrcInterface*,
  LPVOID *lpSrcFolder*,
  ULONG *cbEntryID*,
  LPENTRYID *lpEntryID*,
  LPCIID *lpInterface*,
  LPVOID *lpDestFolder*,
  LPSTR *lpszNewFolderName*,
  ULONG *ulUIParam*,
  LPMAPIPROGRESS *lpProgress*,
  ULONG *ulFlags*
)

Parameters

lpSrcInterface
  Input parameter pointing to the interface identifier (IID) of the source folder indicated in the *lpSrcFolder* parameter.

lpSrcFolder
  Input parameter pointing to the source folder for the folder whose entry identifier is passed in the *lpEntryID* parameter.

cbEntryID
  Input parameter containing the size, in bytes, of the entry identifier pointed to by *lpEntryID*.

lpEntryID
  Input parameter pointing to the entry identifier of the folder to copy or move.

lpInterface
  Reserved; must be NULL.

lpDestFolder
  Input parameter pointing to the open destination folder where the folder identified in the *lpEntryID* parameter is copied or moved.

lpszNewFolderName
  Input parameter pointing to a string naming the newly created or moved folder. If a message store provider passes NULL in the *lpszNewFolderName* parameter, the name of the newly created or moved folder is the same as the name of the original.

ulUIParam
  Input parameter containing the handle of the window for any dialog boxes or windows this method displays. The *ulUIParam* parameter is ignored unless the provider sets the FOLDER_DIALOG flag in the *ulFlags* parameter and passes NULL in the *lpProgress* parameter.

lpProgress
  Input parameter pointing to a progress object that contains client-supplied progress information. If NULL is passed in *lpProgress*, MAPI provides the progress information. The *lpProgress* parameter is ignored unless the FOLDER_DIALOG flag is set in *ulFlags*.

ulFlags
  Input parameter containing a bitmask of flags that controls how the copy or move operation is accomplished. The following flags can be set:

  COPY_SUBFOLDERS
    Indicates all subfolders are included in the copy operation. This functionality is optional for copy operations and is implied for move operations.

FOLDER_DIALOG
: Displays a progress indicator while the operation proceeds.

FOLDER_MOVE
: Indicates that the folder is moved. If this flag is not set, the folder is copied.

MAPI_UNICODE
: Indicates the passed-in strings are in Unicode format. If the MAPI_UNICODE flag is not set, the strings are in ANSI format.

Return Values

S_OK
: The call succeeded and has returned the expected value or values.

MAPI_E_COLLISION
: The first folder being created as a result of the copy or move operation has the same name as the source folder. The operation stops without completing.

MAPI_W_PARTIAL_COMPLETION
: The call succeeded, but not everything was copied or moved. To test for this warning, use the **HR_FAILED** macro. When this warning is returned, the call should be handled as successful.

Remarks

Message store providers call the **IMAPISupport::CopyFolder** method to copy or move folders from one location to another. The folder being copied or moved is added to the destination folder as a subfolder; the destination folder can be in a message store other than that where the copied or moved folder currently resides. Only one folder can be copied or moved at a time; messages contained in the copied or moved folder are not copied or moved at the same time as the folder.

CopyFolder allows simultaneous renaming and moving of folders and the copying or moving of subfolders of the affected folder. To copy or move all subfolders nested within the copied or moved folder, a provider passes the COPY_SUBFOLDERS flag in *ulFlags*.

In copy or move operations involving more than one folder, even if one or more folders specified do not exist or have already been moved elsewhere, a message store provider should complete the operation as best it can for each folder specified. The provider should stop the operation without completing it only in the case of failures it cannot control, such as running out of memory or disk space, message store corruption, and so on.

If **CopyFolder** successfully completes the copy or move operation for every folder, it returns S_OK. If one or more folders cannot be copied or moved, **CopyFolder** returns MAPI_W_PARTIAL_COMPLETION. If **CopyFolder** returns a different value, such as MAPI_E_NOT_ENOUGH_MEMORY, that indicates the call did not complete, it might already have copied or moved one or more folders without being able to continue. The calling provider cannot proceed on the assumption that an error return implies no work was done.

If an entry identifier for a folder that doesn't exist is passed in *lpEntryID*, **CopyFolder** returns MAPI_W_PARTIAL_COMPLETION.

IMAPISupport::CopyMessages

The **IMAPISupport::CopyMessages** method copies or moves messages from one folder to another.

HRESULT CopyMessages(
  LPCIID *lpSrcInterface*,
  LPVOID *lpSrcFolder*,
  LPENTRYLIST *lpMsgList*,
  LPCIID *lpDestInterface*,
  LPVOID *lpDestFolder*,
  ULONG *ulUIParam*,
  LPMAPIPROGRESS *lpProgress*,
  ULONG *ulFlags*
)

Parameters

lpSrcInterface
: Input parameter pointing to the interface identifier (IID) of the source folder from which to copy or move messages.

lpSrcFolder
: Input parameter pointing to the source folder from which to copy or move messages.

lpMsgList
: Input parameter pointing to an array of **ENTRYLIST** structures that identify the message or messages to be copied or moved.

lpDestInterface
: Input parameter pointing to the IID for the destination folder to which messages are copied or moved.

lpDestFolder
: Input parameter pointing to the open destination folder where the message or messages identified in the *lpMsgList* parameter are copied or moved.

ulUIParam
: Input parameter containing the handle of the parent window for any dialog boxes or windows this method displays. The *ulUIParam* parameter is ignored unless the provider sets the MESSAGE_DIALOG flag in the *ulFlags* parameter and passes NULL in the *lpProgress* parameter.

lpProgress
: Input parameter pointing to a progress object that contains client- or provider-supplied progress information. If NULL is passed in *lpProgress*, MAPI provides the progress information. The *lpProgress* parameter is ignored unless MESSAGE_DIALOG is set in *ulFlags*.

ulFlags
: Input parameter containing a bitmask of flags that controls how the copy or move operation is accomplished. The following flags can be set:

  MESSAGE_DIALOG
  : Displays a progress indicator as the operation proceeds.

  MESSAGE_MOVE
  : Moves messages. If this flag is not set, the method copies messages.

Return Values

S_OK
: The call succeeded and has returned the expected value or values.

MAPI_E_USER_CANCEL
: The user canceled the operation, typically by choosing the Cancel button in a dialog box.

Remarks

Message store providers call the **IMAPISupport::CopyMessages** method to call MAPI to move or copy messages from one folder to another as specified by a client application. As part of the **CopyMessages** call, the message store provider can specify that MAPI display a user interface showing progress information.

IMAPISupport::CreateOneOff

The **IMAPISupport::CreateOneOff** method creates an entry identifier for a custom recipient.

HRESULT CreateOneOff(
  LPTSTR *lpszName*,
  LPTSTR *lpszAdrType*,
  LPTSTR *lpszAddress*,
  ULONG *ulFlags*,
  ULONG FAR * *lpcbEntryID*,
  LPENTRYID FAR * *lppEntryID*
)

Parameters

lpszName
  Input parameter pointing to a string containing the display name of the recipient. The *lpszName* parameter can be NULL.

lpszAdrType
  Input parameter pointing to a string containing the messaging address type of the recipient, such as FAX, SMTP, or X500. The *lpszAdrType* parameter cannot be NULL.

lpszAddress
  Input parameter pointing to a string containing the messaging address of the recipient. The *lpszAddress* parameter cannot be NULL.

ulFlags
  Input parameter containing a bitmask of flags that controls the type of the passed-in strings. The following flags can be set:

  MAPI_SEND_NO_RICH_INFO
    If a service provider sets this flag, MAPI sets the custom recipient's PR_SEND_RICH_INFO property to FALSE. If this flag is not set, in most cases MAPI sets this property to TRUE. The one exception is when the custom recipient's address is interpreted to be an Internet address, MAPI sets PR_SEND_RICH_INFO to FALSE.

  MAPI_UNICODE
    Indicates the passed-in strings are in Unicode format. If the MAPI_UNICODE flag is not set, the strings are in ANSI format.

lpcbEntryID
  Output parameter pointing to a variable in which is returned the size, in bytes, of the entry identifier pointed to by the *lppEntryID* parameter.

lppEntryID
  Output parameter pointing to a variable where the pointer to the newly created entry identifier is stored.

Return Values

S_OK
  The custom recipient entry identifier was successfully created.

Remarks

Service providers call the **IMAPISupport::CreateOneOff** method to create an entry identifier for a custom recipient. This entry identifier can be used to represent a recipient on a message.

Although most transport providers by default send messages with TNEF (Transport Neutral Encapsulation Format), some do not regardless of how the recipient sets its PR_SEND_RICH_INFO property. This is not an issue for messaging clients that work with IPM messages, but because TNEF is typically used to send custom properties for custom message classes, not supporting it can be a problem for form-based clients or clients that require custom MAPI properties.

When the provider is done using the entry identifier returned by **CreateOneOff**, it should free the memory allocated for the entry identifier by using the **MAPIFreeBuffer** function.

See Also

MAPIFreeBuffer function

IMAPISupport::Details

The **IMAPISupport::Details** method displays a modal dialog box showing details about an address book entry.

HRESULT Details(
  ULONG FAR * *lpulUIParam*,
  LPFNDISMISS *lpfnDismiss*,
  LPVOID *lpvDismissContext*,
  ULONG *cbEntryID*,
  LPENTRYID *lpEntryID*,
  LPFNBUTTON *lpfButtonCallback*,
  LPVOID *lpvButtonContext*,
  LPTSTR *lpszButtonText*,
  ULONG *ulFlags*
)

Parameters

lpulUIParam
  Output parameter containing the handle of the parent window for the returned dialog box.

lpfnDismiss
  Input parameter pointing to the address of a function based on the **DISMISSMODELESS** function prototype. This function is called when the modeless variety of the details dialog box is dismissed. However, because MAPI does not support a modeless details dialog box, this parameter is ignored.

lpvDismissContext
  Input parameter containing data that is passed to the function specified by the *lpfnDismiss* parameter. However, because MAPI does not support a modeless details dialog box, this parameter is ignored.

cbEntryID
  Input parameter containing the size, in bytes, of the entry identifier pointed to by the *lpEntryID* parameter.

lpEntryID
  Input parameter pointing to the entry identifier for the object for which details are displayed.

lpfButtonCallback
  Input parameter pointing to a pointer to a button callback function that adds a button to the dialog box. The callback function is based on the **LPFNBUTTON** function prototype.

lpvButtonContext
  Input parameter pointing to data used as a parameter for the button callback function.

lpszButtonText
  Input parameter pointing to a string containing text to be applied to the added button if that button is extensible. The *lpszButtonText* parameter should be NULL if an extensible button is not needed.

ulFlags
  Input parameter containing a bitmask of flags that controls the type of the text for *lpszButtonText*. The following flag can be set:

  MAPI_UNICODE
    Indicates the passed-in strings are in Unicode format. If the MAPI_UNICODE flag is not set, the strings are in ANSI format.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

Remarks

Address book providers call the **IMAPISupport::Details** method to display a modal dialog box giving details on a particular entry in an address book. The *lpfButtonCallback*, *lpvButtonContext*, and *lpButtonText* parameters can be used to add a button the provider has defined to the dialog box. When the button is chosen, MAPI calls the callback function pointed to by *lpfButtonCallback*, passing both the entry identifier of the button and the data in *lpvButtonContext*. If an extensible button is not needed, *lpszButtonText* should be NULL. The callback function pointed to by *lpfButtonCallback* is based on the **LPFNBUTTON** function prototype.

See Also

IMAPISupport::Address method, **LPFNBUTTON** function prototype

IMAPISupport::DoConfigPropsheet

The **IMAPISupport::DoConfigPropsheet** method enables a service provider to display a property sheet, usually for configuration, using MAPI's default user interface.

HRESULT DoConfigPropsheet(
  ULONG *ulUIParam*,
  ULONG *ulFlags*,
  LPTSTR *lpszTitle*,
  LPMAPITABLE *lpDisplayTable*,
  LPMAPIPROP *lpConfigData*,
  ULONG *ulTopPage*
)

Parameters

ulUIParam
  Input parameter containing the handle of the parent window for the property sheet displayed.

ulFlags
  Reserved; must be zero.

lpszTitle
  Input parameter pointing to the string that contains the title of the property sheet.

lpDisplayTable
  Input parameter pointing to the display table that contains information about the controls in the property sheet.

lpConfigData
  Input parameter pointing to the property object containing the default values for the properties used to build the display table pointed to by the *lpDisplayTable* parameter.

ulTopPage
  Input parameter containing a zero-based index to the default top page of the property sheet.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

Remarks

Service providers call the **IMAPISupport::DoConfigPropSheet** method to display a configuration property sheet. **DoConfigPropSheet** can be called as part of the implementation of the **IMAPIStatus::SettingsDialog** method or from a button used to display details on properties. It can also be called from the entry point for a message service.

The display table passed in the *lpDisplayTable* parameter can be built using the **BuildDisplayTable** function or by any other means convenient for the provider.

Microsoft strongly recommends that service providers use property sheets for their configuration user interfaces so that users benefit from a consistent Windows interface.

See Also

BuildDisplayTable function, **CreateIProp** function, **IABProvider::Logon** method, **IMAPIProp : IUnknown** interface, **IMAPIStatus::SettingsDialog** method, **IMsgServiceAdmin : IUnknown** interface, **IMSProvider::Logon** method, **IXPProvider::TransportLogon** method

IMAPISupport::DoCopyProps

The **IMAPISupport::DoCopyProps** method copies or moves a selected set of properties from a source object to a destination object. The source object is the object on which the call to **DoCopyProps** is made.

HRESULT DoCopyProps(
  LPCIID *lpSrcInterface*,
  LPVOID *lpSrcObj*,
  LPSPropTagArray *lpIncludeProps*,
  ULONG *ulUIParam*,
  LPMAPIPROGRESS *lpProgress*,
  LPCIID *lpDestInterface*,
  LPVOID *lpDestObj*,
  ULONG *ulFlags*,
  LPSPropProblemArray FAR * *lppProblems*
)

Parameters

lpSrcInterface
  Input parameter pointing to the interface identifier (IID) of the source object from which the properties are copied or moved.

lpSrcObj
  Input parameter pointing to the source object from which the properties are copied or moved.

lpIncludeProps
  Input parameter pointing to an **SPropTagArray** structure holding a counted array of property tags indicating the properties to copy or move. The *lpIncludeProps* parameter cannot be NULL.

ulUIParam
  Input parameter containing the handle of the parent window for the progress indicator.

lpProgress
  Input parameter pointing to a progress object that contains client- or provider-supplied progress information. If NULL is passed in the *lpProgress* parameter, MAPI provides the progress information. The *lpProgress* parameter is ignored unless the MAPI_DIALOG flag is set in the *ulFlags* parameter.

lpDestInterface
  Input parameter pointing to the IID for the destination object to which properties are copied or moved.

lpDestObj
  Input parameter pointing to the open destination object.

ulFlags
  Input parameter containing a bitmask of flags that controls how the copy or move operation is performed. The following flags can be set:

  MAPI_DIALOG
    Displays a user interface to provide progress information for the copy operation.

  MAPI_MOVE
    Indicates a move operation. The default operation is copying.

  MAPI_NOREPLACE
    Indicates that existing properties in the destination object should not be overwritten. The default action is to overwrite existing properties.

lppProblems
  Output parameter pointing to a variable where the pointer to an **SPropProblemArray** structure is stored. If NULL is passed in the *lppProblems* parameter, no property problem array is returned.

Return Values

S_OK
: The call succeeded and has returned the expected value or values.

MAPI_E_COLLISION
: A sibling folder in the destination object already has the name in the PR_DISPLAY_NAME property being copied from the source object.

MAPI_E_FOLDER_CYCLE
: The source object directly or indirectly contains the destination object. Significant work might have been performed before this condition was discovered, so the source and destination objects might be partially modified.

MAPI_E_INTERFACE_NOT_SUPPORTED
: An appropriate interface cannot be obtained.

MAPI_E_NO_ACCESS
: An attempt was made to modify a read-only object or an attempt to access an object for which the user has insufficient permissions.

The following values can be returned in the **SPropProblemArray** structure, but not as return values for **DoCopyProps**:

MAPI_E_BAD_CHARWIDTH
: Either the MAPI_UNICODE flag was set and the implementation does not support Unicode, or MAPI_UNICODE was not set and the implementation only supports Unicode.

MAPI_E_COMPUTED
: The property can't be written because it is computed by the destination object's provider. This error is not severe; the implementation should allow the process to continue.

MAPI_E_INVALID_TYPE
: The property type is invalid.

MAPI_E_UNEXPECTED_TYPE
: The property type is not the type expected by the calling provider.

Remarks

Message store providers call the **IMAPISupport::DoCopyProps** method to copy or move to the destination object those properties designated in *lpIncludeProps* that are present in the source object. When copying properties between like objects, for example between two message objects, the interface identifiers and object types must be the same for both the source and destination objects. If any of the copied or moved properties already exist in the destination object, the existing properties are overwritten by the new, unless the MAPI_NOREPLACE flag is set in the *ulFlags* parameter. Existing information in the destination object that is not overwritten is not deleted or modified.

For message objects, the PR_MESSAGE_RECIPIENTS and PR_MESSAGE_ATTACHMENTS properties can be included in the **SPropTagArray** structure passed in *lpIncludeProps* to permit copying or moving of message recipients and attachments. For folder objects, the PR_CONTAINER_HIERARCHY, PR_CONTAINER_CONTENTS, and PR_FOLDER_ASSOCIATED_CONTENTS properties can be included in the **SPropTagArray** to permit copying or moving of subfolders, messages, or associated objects. If subfolders are copied or moved, their contents are copied or moved in their entirety, regardless of the use of properties indicated by the **SPropTagArray**.

If the source object directly or indirectly contains the destination object, the overall call fails and returns MAPI_E_FOLDER_CYCLE. Some implementations, however, perform significant work before discovering this error and leave the source and destination objects partially modified, so providers should try to avoid such calls. If the same pointer is used for both the source and destination objects, the call returns MAPI_E_NO_ACCESS.

The interface of the destination object, indicated in *lpInterface*, is usually the same interface as for the source object. If *lpInterface* is set to NULL, then **DoCopyProps** returns MAPI_E_INVALID_PARAMETER. If an acceptable interface is passed in *lpInterface* but an invalid pointer is passed in the *lpDestObj* parameter, the results are unpredictable; the most likely result is that the calling provider stops.

If the MAPI_DIALOG flag is not set in *ulFlags*, **DoCopyProps** ignores the *ulUIParam* and *lpProgress* parameters and no progress indicator is provided. If the calling implementation sets MAPI_DIALOG in *ulFlags* and passes NULL in *lpProgress*, then the provider is responsible for generating a progress indicator. If the calling implementation sets MAPI_DIALOG in *ulFlags* and passes a progress object in *lpProgress*, the information supplied by the progress object is used to display progress information.

If the call succeeds overall but there are problems with copying or moving some properties, **DoCopyProps** returns S_OK and an **SPropProblemArray** structure in the *lppProblems* parameter. The **SPropProblemArray** structure contains details about each property problem. In some cases, a **DoCopyProps** call can successfully set some of the requested properties, but not others; in these cases, which properties were not successfully copied or moved can be determined from the **SPropProblemArray** structure. If message recipients or attachments cannot be copied or moved, PR_MESSAGE_RECIPIENTS or PR_MESSAGE_ATTACHMENTS is returned in the **SPropProblemArray** structure.

The value returned in *lppProblems* is only valid if the call returns S_OK. When S_OK is returned, check the values returned in the **SPropProblemArray** structure. If an error occurs on the call, then the **SPropProblemArray** structure is not filled in; call the **IMAPISupport::GetLastError** method to get the **MAPIERROR** structure describing the error.

The calling provider must free the returned **SPropProblemArray** structure by calling the **MAPIFreeBuffer** function, but this should only be done if **DoCopyProps** returns S_OK.

See Also

IMAPISupport::CopyMessages method, **IMAPISupport::DoCopyTo** method, **SPropProblemArray** structure, **SPropTagArray** structure

IMAPISupport::DoCopyTo

The **IMAPISupport::DoCopyTo** method copies or moves all properties from a source object to a destination object, except for a given set of excluded properties. The source object is the object on which the call to **DoCopyTo** is made.

HRESULT DoCopyTo(
  LPCIID *lpSrcInterface*,
  LPVOID *lpSrcObj*,
  ULONG *ciidExclude*,
  LPCIID *rgiidExclude*,
  LPSPropTagArray *lpExcludeProps*,
  ULONG *ulUIParam*,
  LPMAPIPROGRESS *lpProgress*,
  LPCIID *lpDestInterface*,
  LPVOID *lpDestObj*,
  ULONG *ulFlags*,
  LPSPropProblemArray FAR * *lppProblems*
)

Parameters

lpSrcInterface
  Input parameter pointing to the interface identifier (IID) of the source object from which properties and objects are copied or moved.

lpSrcObj
  Input parameter pointing to the source object from which the properties and objects are copied or moved.

ciidExclude
  Input parameter containing the number of interfaces to exclude when copying or moving properties.

rgiidExclude
  Input parameter containing an array of IIDs indicating interfaces that should not be used when copying or moving supplemental information to the destination object.

lpExcludeProps
  Input parameter pointing to an **SPropTagArray** structure containing the property identifiers of the properties that should not be copied or moved to the destination object. Passing NULL in the *lpExcludeProps* parameter indicates all properties are copied or moved. Passing zero in the **cValues** member of the *lpExcludeProps* **SPropTagArray** structure results in MAPI_E_INVALID_PARAMETER being returned.

ulUIParam
  Input parameter containing the handle of the parent window of the progress indicator displayed.

lpProgress
  Input parameter pointing to a progress object that contains client- or provider-supplied progress information. If NULL is passed in the *lpProgress* parameter, MAPI provides the progress information. The *lpProgress* parameter is ignored unless the MAPI_DIALOG flag is set in the *ulFlags* parameter.

lpDestInterface
  Input parameter pointing to the IID for the destination object.

lpDestObj
  Input parameter pointing to the open destination object.

ulFlags
  Input parameter containing a bitmask of flags that controls how the copy or move operation is performed. The following flags can be set:

  MAPI_DIALOG

Displays a user interface to provide progress information for the copy or move operation.

MAPI_MOVE
Indicates a move operation. The default operation is copying.

MAPI_NOREPLACE
Indicates that existing properties in the destination object should not be overwritten. The default action is to overwrite existing properties.

lppProblems
Output parameter pointing to a variable where the pointer to an **SPropProblemArray** structure is stored. If NULL is passed in the *lppProblems* parameter, no property problem array is returned.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

MAPI_E_COLLISION
A sibling folder in the destination object already has the name in the PR_DISPLAY_NAME property being copied from the source object.

MAPI_E_FOLDER_CYCLE
The source object directly or indirectly contains the destination object. Significant work might have been performed before this condition was discovered, so the source and destination objects might be partially modified.

MAPI_E_INTERFACE_NOT_SUPPORTED
An appropriate interface cannot be obtained.

MAPI_E_NO_ACCESS
An attempt was made to modify a read-only object or an attempt to access an object for which the user has insufficient permissions.

The following values can be returned in the **SPropProblemArray** structure, but not as return values for **DoCopyTo**:

MAPI_E_BAD_CHARWIDTH
Either the MAPI_UNICODE flag was set and the implementation does not support Unicode, or MAPI_UNICODE was not set and the implementation only supports Unicode.

MAPI_E_COMPUTED
The property can't be written because it is computed by the destination object's provider. This error is not severe; the implementation should allow the process to continue.

MAPI_E_INVALID_TYPE
The property type is invalid.

MAPI_E_UNEXPECTED_TYPE
The property type is not the type expected by the calling provider.

Remarks

Message store providers call the **IMAPISupport::DoCopyTo** method to copy or move all properties of the source object to the destination object, except for a given set of properties that are excluded from the copy or move operation. Any objects contained in the source object and any subobjects of the source object are included in the copy or move operation.

If any of the copied or moved properties already exist in the destination object, the existing properties are overwritten by the new, unless the MAPI_NOREPLACE flag is set in the *ulFlags* parameter. Existing information in the destination object that is not overwritten is not deleted or modified.

To exclude some properties from the copy or move operation, pass their property identifiers in the *lpExcludeProps* parameter. Passing in a specific value causes any property in the source object whose identifier matches that value to be excluded from the copy or move operation. For example, passing in

`PROP_TAG(PT_LONG, 0x8002)` excludes both the properties `PROP_TAG(PT_STRING8, 0x8002)` and `PROP_TAG(PT_OBJECT, 0x8002)`.

For message objects, the PR_MESSAGE_RECIPIENTS and PR_MESSAGE_ATTACHMENTS properties can be included in the **SPropTagArray** structure passed in *lpExcludeProps* to prevent copying or moving message recipients and attachments. For folder objects, the PR_CONTAINER_HIERARCHY, PR_CONTAINER_CONTENTS, and PR_FOLDER_ASSOCIATED_CONTENTS properties can be included in the **SPropTagArray** structure to prevent copying or moving of subfolders, messages, or associated objects. If subfolders are copied or moved, their contents are copied or moved in their entirety, regardless of the use of properties indicated by the **SPropTagArray** structure.

Implementations of **DoCopyTo** should not attempt to set any known read-only properties in the destination object and should ignore MAPI_E_COMPUTED errors returned in the **SPropProblemArray** structure in the *lppProblems* parameter.

If the source object directly or indirectly contains the destination object, the overall call fails and returns MAPI_E_FOLDER_CYCLE. Some implementations, however, perform significant work before discovering this error and leave the source and destination objects partially modified, so providers should avoid such calls. If the same pointer is used for both the source and destination objects, the call returns MAPI_E_NO_ACCESS.

The interface of the destination object, indicated in *lpInterface*, is usually the same interface as for the source object. If *lpInterface* is set to NULL, then **DoCopyTo** returns MAPI_E_INVALID_PARAMETER. If an acceptable interface is passed in *lpInterface* but an invalid pointer is passed in the *lpDestObj* parameter, the results are unpredictable; the most likely result is that the calling provider stops.

Some objects contain supplemental information, which can be accessed with the interface pointer in *lpInterface*. To copy or move such information, a **DoCopyTo** implementation first calls the **IUnknown::QueryInterface** method for the destination object to see if it can accept the extra data. Conversely, if the calling provider is aware of supplemental information and requires that **DoCopyTo** not copy or move it, the provider can specify in the array passed in the *rgiidExclude* parameter IIDs for the properties that **DoCopyTo** should not copy or move. For example, if the provider must copy messages, but not embedded objects within the messages, it can pass IID_IMessage in the *rgiidExclude* array. **DoCopyTo** ignores any interfaces listed in *rgiidExclude* it doesn't recognize.

Note When you use the *rgiidExclude* parameter to exclude an interface, you also exclude all interfaces derived from that interface. For example, excluding the **IMAPIProp** interface also excludes the **IMAPIFolder**, **IMessage**, and **IAttach** interfaces, and so on. If all known interfaces are excluded, **DoCopyTo** returns the error value MAPI_E_INTERFACE_NOT_SUPPORTED. For that reason, you should not pass IID_IUnknown or IID_IMAPIProp in *rgiidExclude*.

If the MAPI_DIALOG flag is not set in *ulFlags*, then **DoCopyTo** ignores the *ulUIParam* and *lpProgress* parameters and no progress indicator is provided. If the calling implementation sets MAPI_DIALOG in *ulFlags* and passes NULL in *lpProgress*, then the provider is responsible for generating a progress indicator. If the calling implementation sets MAPI_DIALOG in *ulFlags* and passes a progress object in *lpProgress*, the information supplied by the progress object is used to display progress information.

If the call succeeds overall but there are problems with copying or moving some properties, **DoCopyTo** returns S_OK and an **SPropProblemArray** structure in the *lppProblems* parameter. The **SPropProblemArray** structure contains details about each property problem. In some cases, a **DoCopyTo** call can successfully set some of the requested properties, but not others; in these cases, which properties were not successfully copied or moved can be determined from the **SPropProblemArray** structure. If message recipients or attachments cannot be copied or moved, PR_MESSAGE_RECIPIENTS or PR_MESSAGE_ATTACHMENTS is returned in the **SPropProblemArray**.

The value returned in *lppProblems* is only valid if the call returns S_OK. When S_OK is returned, check the values returned in the **SPropProblemArray** structure. If an error occurs on the call, then the **SPropProblemArray** structure is not filled in; call the **IMAPISupport::GetLastError** method to get the **MAPIERROR** structure describing the error.

If an error occurs on the **DoCopyTo** call, do not use or free the **SPropProblemArray** structure. Providers should ignore the **ulIndex** member in **SPropProblemArray** structures returned by **DoCopyTo**.

The calling provider must free the returned **SPropProblemArray** by calling the **MAPIFreeBuffer** function, but this should only be done if **DoCopyTo** returns S_OK.

See Also

IMAPISupport::CopyFolder method, **IMAPISupport::CopyMessages** method, **SPropProblemArray** structure, **SPropTagArray** structure

IMAPISupport::DoProgressDialog

The **IMAPISupport::DoProgressDialog** method provides a default implementation of a progress indicator for use by service providers.

HRESULT DoProgressDialog(
  ULONG *ulUIParam***,**
  ULONG *ulFlags***,**
  LPMAPIPROGRESS FAR * *lppProgress*
)

Parameters

ulUIParam
  Input parameter containing the handle of the parent window for the user interface.

ulFlags
  Input parameter containing a bitmask of flags that controls how progress information is calculated. The following flag can be set:

  MAPI_TOP_LEVEL
    Uses the values in the **IMAPIProgress::Progress** method's *ulCount* and *ulTotal* parameters, which indicate the item being operated on and the total items to operate on respectively, to increment progress made on the operation. For example, if a folder is being copied that contains 10 subfolders and the MAPI_TOP_LEVEL flag is set in the *ulFlags* parameter, progress increments for each subfolder copied: 1 of 10, 2 of 10, 3 of 10, and so on.

lppProgress
  Output parameter pointing to a variable where the pointer to the progress object is stored.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

Remarks

Address book and message store providers call the **IMAPISupport::DoProgressDialog** method in concert with the methods of the **IMAPIProgress** interface to calculate progress information and display the result in a user interface when requested to do so by a client application. A pointer to a progress object supporting the **IMAPIProgress** interface is returned in the *lppProgress* parameter.

See Also

IMAPIProgress : IUnknown interface

IMAPISupport::DoSentMail

The **IMAPISupport::DoSentMail** method generates for an open message a corresponding message in another folder. This method is only valid when called from within the MAPI spooler's process.

HRESULT DoSentMail(
  ULONG *ulFlags***,**
  LPMESSAGE *lpMessage*
)

Parameters

ulFlags
  Reserved; must be zero.

lpMessage
  Input parameter pointing to the open message for which a message should be generated in the folder designated to hold sent items.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

Remarks

When the MAPI spooler has finished processing a message, it calls a message store provider's **IMsgStore::FinishedMsg** method. Message store providers call the **IMAPISupport::DoSentMail** method from within their **FinishedMsg** implementation. **FinishedMsg** should unlock the message and ensure that the reference count is only one when this **DoSentMail** call is made. After **DoSentMail** returns, the message has been fully released.

The MAPI spooler's implementation of **DoSentMail** does the following:

- Determines whether the message should be deleted after sending by checking the message for the PR_DELETE_AFTER_SUBMIT property.
- Determines the location of the SentItems folder.
- Initiates message hook processing for any hooks set on the SentItems folder.
- Moves the message to the SentItems folder, DeletedItems folder, or to another folder as specified by any message hooks.
- Releases the message.

See Also

IMsgStore::FinishedMsg method

IMAPISupport::ExpandRecips

The **IMAPISupport::ExpandRecips** method completes all recipient lists and expands certain distribution lists for the transmission of a message.

HRESULT ExpandRecips(
  LPMESSAGE *lpMessage***,**
  ULONG FAR * *lpulFlags*
)

Parameters

lpMessage
  Input parameter pointing to the open message object that has read/write access.

lpulFlags
  Output parameter pointing to a variable where a bitmask of flags that controls what occurs after recipient entries are resolved is stored. The following flags can be set:

  NEEDS_PREPROCESSING
    Indicates the message needs preprocessing before sending.

  NEEDS_SPOOLER
    Indicates the message store provider should have the MAPI spooler send the message rather than the transport provider to which this store provider is tightly coupled.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

Remarks

Message store providers call the **IMAPISupport::ExpandRecips** method to prompt MAPI to complete all recipient entries and distribution lists in the recipient list for a particular message. The process of completing recipient entries and distribution lists involves expanding certain personal distribution lists to their component recipients, replacing all display names that have been altered with the original names, marking any duplicate entries, and resolving all custom recipient addresses. **ExpandRecips** expands any distribution lists that have the messaging address type of MAPIPDL.

After MAPI reads and updates the recipient list for the message indicated in the *lpMessage* parameter, MAPI checks to see if the message needs preprocessing. If it does, MAPI sets and returns the NEEDS_PREPROCESSING flag in the *lpulFlags* parameter.

Message store providers should always call **ExpandRecips** as part of processing a message, and it should be one of the first calls made as part of a provider's **IMessage::SubmitMessage** implementation.

See Also

IMessage::SubmitMessage method

IMAPISupport::GetLastError

The **IMAPISupport::GetLastError** method returns a **MAPIERROR** structure containing information about the last error that occurred for a support object.

HRESULT GetLastError(
  HRESULT *hResult***,**
  ULONG *ulFlags***,**
  LPMAPIERROR FAR * *lppMAPIError*
)

Parameters

hResult
: Input parameter containing the result returned for the last call for the support object that returned an error.

ulFlags
: Input parameter containing a bitmask of flags that controls the type of the returned strings. The following flag can be set:

  MAPI_UNICODE
  : Indicates the strings in the **MAPIERROR** structure returned in the *lppMAPIError* parameter are in Unicode format. If the MAPI_UNICODE flag is not set, the strings are in ANSI format.

lppMAPIError
: Output parameter pointing to a pointer to the returned **MAPIERROR** structure containing version, component, and context information for the error. The *lppMAPIError* parameter can be set to NULL if there is no **MAPIERROR** structure to return.

Return Values

S_OK
: The call succeeded and has returned the expected value or values.

MAPI_E_BAD_CHARWIDTH
: Either the MAPI_UNICODE flag was set and the implementation does not support Unicode, or MAPI_UNICODE was not set and the implementation only supports Unicode.

Remarks

Service providers call the **IMAPISupport::GetLastError** method to retrieve information to display in a message to the user regarding the last error returned from a method call for the support object.

To release all the memory allocated by MAPI, providers need only call the **MAPIFreeBuffer** function for the returned **MAPIERROR** structure.

The return value from **GetLastError** must be S_OK for a provider to make use of the **MAPIERROR** structure. Even if the return value is S_OK, a **MAPIERROR** structure might not be returned. If the implementation cannot determine what the last error was, or if a **MAPIERROR** structure is not available for that error, **GetLastError** returns a pointer to NULL in *lppMAPIError* instead.

See Also

MAPIERROR structure, **MAPIFreeBuffer** function

IMAPISupport::GetMemAllocRoutines

The **IMAPISupport::GetMemAllocRoutines** method retrieves the addresses of the MAPI memory allocation and deallocation functions, **MAPIAllocateBuffer**, **MAPIAllocateMore**, and **MAPIFreeBuffer**.

```
HRESULT GetMemAllocRoutines(
    LPALLOCATEBUFFER FAR * lppAllocateBuffer,
    LPALLOCATEMORE FAR * lppAllocateMore,
    LPFREEBUFFER FAR * lppFreeBuffer
  )
```

Parameters

lppAllocateBuffer
Output parameter pointing to a variable where a pointer to the **MAPIAllocateBuffer** function is stored. The **MAPIAllocateBuffer** function allocates memory.

lppAllocateMore
Output parameter pointing to a variable where a pointer to the **MAPIAllocateMore** function is stored. The **MAPIAllocateMore** function allocates additional memory where required

lppFreeBuffer
Output parameter pointing to a variable where a pointer to the **MAPIFreeBuffer** function is stored. The **MAPIFreeBuffer** function frees memory.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

Remarks

A service provider calls the **IMAPISupport::GetMemAllocRoutines** method to get the addresses of the three memory allocation functions passed in its initialization call. For the syntax of each function, see its reference entry.

See Also

MAPIAllocateBuffer function, **MAPIAllocateMore** function, **MAPIFreeBuffer** function

IMAPISupport::GetOneOffTable

The **IMAPISupport::GetOneOffTable** method returns a table of templates for custom recipient addresses that can be used to create recipients for a message.

HRESULT GetOneOffTable(
  ULONG *ulFlags***,**
  LPMAPITABLE FAR * *lppTable*
)

Parameters

ulFlags
  Input parameter containing a bitmask of flags that controls the type of the returned strings. The following flag can be set:

  MAPI_UNICODE
    Indicates the returned strings are in Unicode format. If the MAPI_UNICODE flag is not set, the strings are in ANSI format.

lppTable
  Output parameter pointing to a variable where the pointer to the returned table object is stored. The table contains a complete list of all custom recipient addresses supported within the current MAPI session.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

Remarks

Address book providers call the **IMAPISupport::GetOneOffTable** method to retrieve the list of available, new custom recipient addresses (also called one-off addresses) relevant to the current MAPI session. Providers use the returned custom recipient table to identify the kinds of entries that can be added to their address book container.

The five property columns required in a table of custom recipients are as follows:

PR_DISPLAY_NAME
PR_DISPLAY_TYPE
PR_ENTRYID
PR_DEPTH
PR_SELECTABLE
PR_ADDRTYPE
PR_INSTANCE_KEY

Setting the MAPI_UNICODE flag in the *ulFlags* parameter does the following:

- Sets the string type to Unicode for data returned for the initial active columns of the custom recipient table by the **IMAPITable::QueryColumns** method. The initial active columns for a custom recipient table are those columns **QueryColumns** returns before the provider that contains the table calls the **IMAPITable::SetColumns** method.
- Sets the string type to Unicode for data returned for the initial active rows of the custom recipient table by the **IMAPITable::QueryRows** method. The initial active rows for a custom recipient table are those rows **QueryRows** returns before the provider that contains the table calls **SetColumns**.
- Controls the property types of the sort order returned by the **IMAPITable::QuerySortOrder** method before the provider that contains the custom recipient table calls the **IMAPITable::SortTable**

method.

If a provider registers for notification of changes to its custom recipient table, it also receives notifications of changes made to other providers' custom recipient tables. A provider can, based on these notifications, support new address types that are added during the current session.

See Also

IABContainer::CreateEntry method, **IMAPISupport::NewEntry** method, PR_CREATE_TEMPLATES property

IMAPISupport::GetSvcConfigSupportObj

The **IMAPISupport::GetSvcConfigSupportObj** method creates a new support object for use by calls to a message service's entry point function.

HRESULT GetSvcConfigSupportObj(
  ULONG *ulFlags*,
  LPMAPISUP FAR * *lppSvcSupport*
)

Parameters

ulFlags
  Reserved; must be zero.

lppSvcSupport
  Output parameter pointing to a pointer to the newly created support object.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

Remarks

Service providers call the **IMAPISupport::GetSvcConfigSupportObj** method to acquire a pointer to a support object that can be used for a message service's implementation of the **MSGSERVICEENTRY** function prototype. An entry point function based on **MSGSERVICEENTRY** is called by methods of the **IMsgServiceAdmin** interface and allows message services to configure themselves or perform other actions when an implementation changes the profile. The primary action of such a function is to furnish a dialog box in which the user can change message service settings. A function based on **MSGSERVICEENTRY** can also support configuration by a client through a property value array passed through the **IMsgServiceAdmin::ConfigureMsgService** method.

See Also

IMsgServiceAdmin::CreateMsgService method, **IProfAdmin : IUnknown** interface, **MSGSERVICEENTRY** function prototype

IMAPISupport::IStorageFromStream

The **IMAPISupport::IStorageFromStream** method implements a storage object to access a stream.

HRESULT IStorageFromStream(
  LPUNKNOWN *lpUnkIn***,**
  LPCIID *lpInterface***,**
  ULONG *ulFlags***,**
  LPSTORAGE FAR * *lppStorageOut*
)

Parameters

lpUnkIn
  Input parameter pointing to the stream interface implementation.

lpInterface
  Input parameter pointing to the identifier (IID) for the interface implementation pointed to by the *lpUnkIn* parameter. Any of the following values can be passed in the *lpInterface* parameter: NULL, IID_IStream, or IID_ILockBytes. Passing NULL in *lpInterface* is the same as passing IID_IStream.

ulFlags
  Input parameter containing a bitmask of flags that controls how the storage is to be created relative to the stream. The default setting is that the storage has read-only access and occurs within the stream starting at position zero. The following flags can be set:

  STGSTRM_CREATE
    Creates a new storage object for the stream object.

  STGSTRM_CURRENT
    Starts storage at the current position of the stream.

  STGSTRM_MODIFY
    Allows the calling service provider to write to the returned storage.

  STGSTRM_RESET
    Starts storage at position zero.

lppStorageOut
  Output parameter pointing to a variable where the pointer to the returned storage object is stored.

Return Values

S_OK
  The storage object was successfully created.

Remarks

Service providers that don't implement the OLE **IStorage** interface themselves call the **IMAPISupport::IStorageFromStream** method to implement **IStorage**. **IStorage** must be implemented by any message store that supports streams to hold binary properties, because MAPI guarantees client applications that such message stores support **IStorage** to store these binary properties.

When handling a call from the **IMAPIProp::OpenProperty** method to open an **IStorage** interface on a property, as indicated by an interface identifier of IID_IStorage having been passed, a provider first opens an OLE stream object with read/write access for the property, then internally marks the property stream as an storage object, generates an **IStorage** interface from the support function, and returns the MAPI-generated **IStorage** interface to the calling implementation.

A newly created storage object returned by **IStorageFromStream** calls the **IUnknown::AddRef** method to add a reference for the stream to the stream reference count, then releases the reference when the storage is released. Providers that support interfaces in addition to **IStorage** – for instance, an administrative interface for all object properties – must wrap the storage object returned by

IStorageFromStream using their own **IUnknown::QueryInterface** method.

Message store providers should not allow a property to be opened as a stream with **IMAPIProp::OpenProperty** calls if it was created using **IStorage**. With one exception, message stores can use IID_IStreamDocfile to stream an storage object from one container to another, but they must pass IID_IStreamDocfile in the **OpenProperty** method's *lpInterface* parameter. Message store providers should also not allow a property to be opened as a storage object if it was created using the OLE **IStream** interface.

For more information on OLE programming, see *Inside OLE, Second Edition*, by Kraig Brockschmidt, and the *OLE Programmer's Reference.*

See Also

IMAPIProp::OpenProperty method

IMAPISupport::MakeInvalid

The **IMAPISupport::MakeInvalid** method invalidates an object derived from the **IUnknown** interface.

HRESULT MakeInvalid(
  ULONG *ulFlags*,
  LPVOID *lpObject*,
  ULONG *ulRefCount*,
  ULONG *cMethods*
)

Parameters

ulFlags
  Reserved; must be zero.

lpObject
  Input parameter pointing to the object to be invalidated. The object's interface must be derived from **IUnknown**.

ulRefCount
  Input parameter containing the object's present reference count.

cMethods
  Input parameter containing the number of methods in the object's vtable.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

Remarks

Service providers call the **IMAPISupport::MakeInvalid** method to invalidate an object, typically when a provider is shutting down. However, a provider can call **MakeInvalid** at any time; for instance, if a client application releases an object without releasing some of its subobjects, a provider can call **MakeInvalid** immediately to release those subobjects. The object to be invalidated must be derived from the **IUnknown** interface or from an interface derived from **IUnknown**.

Note An implementation must own any object it attempts to invalidate with **MakeInvalid**. In addition, it must update the *cMethods* parameter if it changes the number of methods in the vtable for the object to be invalidated, as usually results from a **MakeInvalid** call.

The object passed to **MakeInvalid** must have been allocated by using the **MAPIAllocateBuffer** function and must be at least 16 bytes long. The number of methods for the object, as indicated in *cMethods*, must be at least 3 and should not be more than 2000.

MAPI handles calls to **MakeInvalid** by replacing the object's vtable with a stub vtable of similar size, in which the **IUnknown::AddRef** and **IUnknown::Release** methods perform as expected but any other methods fail, returning the value MAPI_E_INVALID_OBJECT.

A provider can call **MakeInvalid** and then perform any shutdown work, such as discarding associated data structures, that it usually does if it releases an object. Code to support the object need not be kept in memory because MAPI frees the memory by calling **MAPIFreeBuffer** and then releases the object. Providers can release their resources, call **MakeInvalid**, and then ignore the invalidated object. Client applications are required to release any memory associated with the object.

See Also

MAPIAllocateBuffer function

IMAPISupport::ModifyProfile

The **IMAPISupport::ModifyProfile** method makes the profile section for a message store provider permanent.

HRESULT ModifyProfile(
  ULONG *ulFlags*
)

Parameters

ulFlags
  Input parameter containing a bitmask of flags that controls how a store logs onto the current profile. The following flags can be set:

  MDB_TEMPORARY
    Advises MAPI the store is temporary and should not be added to the message store information table. The method returns S_OK immediately.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

Remarks

Message store providers call the **IMAPISupport::ModifyProfile** method to prompt MAPI to modify their profile information. When a message store provider calls **IMAPISupport::ModifyProfile**, MAPI adds the profile section associated with that provider resource to the list of installed message store provider resources. MAPI's doing so causes the message store to be listed in the table returned by a call to the **IMAPISession::GetMsgStoresTable** method and makes it possible to open the message store without displaying a dialog box. If the MDB_TEMPORARY flag is set, MAPI does nothing and the method returns with S_OK immediately.

See Also

IMAPISession::GetMsgStoresTable method

IMAPISupport::ModifyStatusRow

The **IMAPISupport::ModifyStatusRow** method creates or modifies a service provider's status table row.

HRESULT ModifyStatusRow(
  ULONG *cValues*,
  LPSPropValue *lpColumnVals*,
  ULONG *ulFlags*
)

Parameters

cValues
  Input parameter containing the number of property columns in the status table row to pass to MAPI.

lpColumnVals
  Input parameter pointing to an **SPropValue** structure containing the property values used to define the columns in the status table row.

ulFlags
  Input parameter containing a bitmask of flags that controls how information defining the status table row is processed. The following flag can be set:

  STATUSROW_UPDATE
    Directs MAPI to merge new status information with an existing status table row.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

Remarks

Service providers call the **IMAPISupport::ModifyStatusRow** method to enter a row in the status table at logon. A provider should call **ModifyStatusRow** during its logon process. For any subsequent **ModifyStatusRow** call, the STATUSROW_UPDATE flag should be set. Doing so informs MAPI that only the columns being changed are passed in the *lpColumnVals* parameter.

ModifyStatusRow provides MAPI with the information necessary to build the initial status table. During **ModifyStatusRow** processing, MAPI copies the data in the *lpColumnVals* parameter defining status table columns and from this data creates the status table. MAPI requires service providers to pass the following property column values in the *lpColumnVals* parameter:

  PR_PROVIDER_DISPLAY
  PR_RESOURCE_METHODS
  PR_STATUS_CODE

The following property columns are required if a service provider lists its user identity in the status table:

  PR_IDENTITY_ENTRYID
  PR_IDENTITY_DISPLAY
  PR_IDENTITY_SEARCH_KEY

The following property columns should not be passed by a provider in *lpColumnVals*; they are provided by the messaging subsystem:

  PR_ENTRYID
  PR_OBJECT_TYPE

PR_PROVIDER_DLL_NAME
PR_RESOURCE_FLAGS
PR_RESOURCE_TYPE
PR_ROWID

The following optional property columns can be passed by a provider in *lpColumnVals*:

PR_DISPLAY_NAME
PR_STATUS_STRING

Although not required, the values for PR_DISPLAY_NAME and PR_STATUS_STRING are computed by MAPI if they are not provided by service providers.

MAPI displays all these properties in the status table, and implementations can retrieve the settings for a provider by opening its status object.

IMAPISupport::NewEntry

The **IMAPISupport::NewEntry** displays MAPI's default dialog box for creating new entries within a container or custom recipient addresses within a message.

HRESULT NewEntry(
  ULONG *ulUIParam*,
  ULONG *ulFlags*,
  ULONG *cbEIDContainer*,
  LPENTRYID *lpEIDContainer*,
  ULONG *cbEIDNewEntryTpl*,
  LPENTRYID *lpEIDNewEntryTpl*,
  ULONG FAR * *lpcbEIDNewEntry*,
  LPENTRYID FAR * *lppEIDNewEntry*
)

Parameters

ulUIParam
  Input parameter containing the handle of the parent window for the dialog box.

ulFlags
  Reserved; must be zero.

cbEIDContainer
  Input parameter containing the size, in bytes, of the entry identifier pointed to by the *lpEIDContainer* parameter.

lpEIDContainer
  Input parameter pointing to the entry identifier of the container where the new custom recipient address is added. If the *cbEIDContainer* method is zero, **NewEntry** returns a recipient entry identifier and a list of templates as if the **IMAPISupport::CreateOneOff** method was called.

cbEIDNewEntryTpl
  Input parameter containing the size, in bytes, of the entry identifier pointed to by the *lpEIDNewEntryTpl* parameter.

lpEIDNewEntryTpl
  Input parameter pointing to a template to be used to create the new entry or custom recipient address. If the *cbEIDNewEntryTpl* parameter is zero, passing NULL in the *lpEIDNewEntryTpl* parameter displays a dialog box enabling the user to select an address-creation template.

lpcbEIDNewEntry
  Output parameter pointing to a variable where the size, in bytes, of the entry identifier pointed to by the *lppEIDNewEntry* parameter is returned.

lppEIDNewEntry
  Output parameter pointing to a variable where the returned pointer to the entry identifier for the new entry or custom recipient address is stored.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

Remarks

Address book providers call the **IMAPISupport::NewEntry** method to display a dialog box to create a new container entry or custom recipient address. **NewEntry** can be used in several different modes. To add a custom recipient address directly to the open message and not to a modifiable container, a provider passes zero in *cbEIDContainer* and NULL in *lpEIDContainer*. To display a dialog box enabling the user to select a template for adding custom recipients to a modifiable container, the provider passes zero in *cbEIDNewEntryTpl* and NULL in *lpEIDNewEntryTpl*.

To create an entry in a modifiable address book and not get its entry identifier back, a provider passes the container's entry identifier in the *lpEIDContainer*, zero in *cbEIDContainer*, and NULL for the rest of the parameters.

To open a specific custom recipient dialog box directly, so that users enter custom recipients in their personal address books using a predetermined template, a provider uses the following series of calls. First, it calls the **IMAPISupport::OpenEntry** method and passes in either the entry identifier of a modifiable container or zero; passing zero opens the root folder of the address book container. Next, the provider calls the **IABContainer::OpenProperty** method and passes the PR_CREATE_TEMPLATES property in the *ulPropTag* parameter so it can open PR_CREATE_TEMPLATES. Doing so returns a table object that lists the types of objects that can be created in the address book container. The provider finds in this table the entry identifier for the template with which new entries should be created. Then, the provider calls **NewEntry** and passes NULL in *lpEIDContainer* and the entry identifier for the entry-creation template to use in *lpEIDNewEntryTpl*.

Calls made to **NewEntry** return the entry identifier of the new custom recipient address in *lppEIDNewEntry*, unless the provider passed NULL in *lppEIDNewEntry*. The calling provider is responsible for freeing the returned entry identifier by calling the **MAPIFreeBuffer** function.

See Also

IMAPIProp::OpenProperty method, **IMAPISupport::OpenEntry** method, PR_CREATE_TEMPLATES property

IMAPISupport::NewUID

The **IMAPISupport::NewUID** method returns a new MAPI unique identifier (MAPIUID) for an item.

HRESULT NewUID(
  LPMAPIUID *lpMuid*
)

Parameters

lpMuid
  Points to the 16 bytes of memory where the new **MAPIUID** structure holding the MAPIUID is placed.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

Remarks

Service providers call the **IMAPISupport::NewUID** method in a wide variety of situations where they need to generate a long-term unique tag for an item. A message store provider, for instance, might call **NewUID** to obtain a unique identifier to place in the PR_SEARCH_KEY property of a newly created message to enable searching within tables.

Each service provider has its own MAPIUID, which is used to distinguish entry identifiers created by that provider. When you create a service provider, you should choose a MAPIUID for your provider and hard-code it, as opposed to obtaining it from **NewUID** during provider initialization. By doing so, you ensure your provider will have the same MAPIUID on all systems. To generate the MAPIUID, use the UUIDGEN.EXE utility program.

See Also

MAPIUID structure

IMAPISupport::Notify

The **IMAPISupport::Notify** method notifies registered client applications and service providers about changes to an object that the calling provider owns.

HRESULT Notify(
  LPNOTIFKEY *lpKey*,
  ULONG *cNotification*,
  LPNOTIFICATION *lpNotifications*,
  ULONG FAR * *lpulFlags*
)

Parameters

lpKey
  Input parameter pointing to a key for the object whose notification status is reported. The *lpKey* parameter provides a unique key to the object and cannot be NULL.

cNotification
  Input parameter containing the number of notifications in the *lpNotifications* parameter.

lpNotifications
  Input parameter pointing to an array of **NOTIFICATION** structures holding notifications to be broadcast.

lpulFlags
  Input-output parameter containing a bitmask of flags that controls how the notification is performed. On input, the following flag can be set by the service provider:

  MAPI_UNICODE
    Indicates the passed-in strings are in Unicode format. If the MAPI_UNICODE flag is not set, the strings are in ANSI format.

On output, the following flag can be set by MAPI:

  NOTIFY_CANCELED
    Indicates a callback function canceled a synchronous notification.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

Remarks

Service providers call the **IMAPISupport::Notify** method to request that MAPI generate a notification for an advise sink that has previously registered for the notification using **IMAPISupport::Subscribe**. **Notify** copies the structures indicated by the *lpNotifications* parameter into memory that is accessible to the advise sink's process. It is also responsible for calling the callback function that broadcasts the notifications, and for releasing the memory involved. The provider calling **Notify** does not need to allocate memory; MAPI performs all necessary memory allocation.

Providers can use only the notification events and event structures defined by MAPI; to support a custom notification event that does not fit easily into the other structures defined by MAPI, a provider can use an event of the `fnevExtended` type. For more information on the event types that can be used to trigger notifications, see **NOTIFICATION**.

The notification key passed in the **Notify** method's *lpKey* parameter for the object whose notification status is reported should be the same as the key formerly passed for that object in the *lpKey* parameter of the **IMAPISupport::Subscribe** method that set up notification registration. This key is similar to the PR_RECORD_KEY property in that it is binary-comparable and can be used to ensure the correct object is used. MAPI uses this key to find the objects that are registered for notifications about the identified object.

Many notification structures include the entry identifier of the object receiving notifications; providers should be careful to pass the long-term entry identifier for the receiving object to provide an entry identifier useful to all implementations registered for notifications.

When a provider instructs MAPI to broadcast a notification held in an **ERROR_NOTIFICATION** or **NEWMAIL_NOTIFICATION** structure and the error or new message string held in the structure is in Unicode format, the provider must set the MAPI_UNICODE flag in the **ulFlags** member of the **NOTIFICATION** structure. If in such a case the error or new message string is in ANSI format, the provider should not set MAPI_UNICODE. MAPI_UNICODE only applies to the strings within the notification structures.

For asynchronous notifications, **Notify** returns before callbacks are made to the implementation registered for notifications. For synchronous notifications, **Notify** makes the callbacks before returning so the calling implementation can determine whether a process is active. It does so by checking whether the NOTIFY_CANCELED flag is set; if any callback function has set the CALLBACK_DISCONTINUE flag, MAPI stops sending notifications and returns NOTIFY_CANCELED in the **Notify** method's *lpulFlags* parameter. Providers can then stop making notifications for that process. If zero is returned in *lpulFlags*, the process is still active and the provider should continue to send notifications as appropriate.

Providers using synchronous notifications must be careful to avoid deadlock situations.

For more information on the notification process, see About Notification.

See Also

IMAPISupport::Subscribe method, **IMAPISupport::Unsubscribe** method, **NOTIFCALLBACK** function prototype **NOTIFICATION** structure, **NOTIFKEY** structure, PR_RECORD_KEY property

IMAPISupport::OpenAddressBook

The **IMAPISupport::OpenAddressBook** method opens an address book and returns a pointer that provides further access to the open address book.

HRESULT OpenAddressBook(
  LPCIID *lpInterface***,**
  ULONG *ulFlags***,**
  LPADRBOOK FAR * *lppAdrBook*
)

Parameters

lpInterface
  Input parameter pointing to the interface identifier (IID) for the address book object. Passing NULL indicates the returned address book is cast to the standard interface for an address book. The *lpInterface* parameter can also be set to IID_IAddrBook.

ulFlags
  Reserved; must be zero.

lppAdrBook
  Output parameter pointing to a pointer to the returned address book object.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

MAPI_W_ERRORS_RETURNED
  The call succeeded, but one or more address book providers could not be loaded. To test this warning, use the **HR_FAILED** macro. When this warning is returned, the call should be handled as successful.

Remarks

Tightly coupled message store and transport providers call the **IMAPISupport::OpenAddressBook** method to get access to an address book. The returned pointer to the address book can then be used to open address book containers, find messaging users, and display address dialog boxes.

This method can return MAPI_W_ERRORS_RETURNED if it cannot load an address book provider. This value is a warning, not an error value, and a call that returns it should be handled as successful. Even if all of the address book providers failed to load, **OpenAddressBook** succeeds and returns MAPI_W_ERRORS_RETURNED and an address book object in the *lppAdrBook* parameter. Even when no address book providers are loaded, a provider can still use the **IAddrBook** interface and must release it when done.

If one or more address book providers failed to load, a provider can call the **IMAPISupport::GetLastError** method on the support object to obtain a **MAPIERROR** structure containing information about the providers that did not load. If more than one provider failed to load, a single **MAPIERROR** structure is returned that contains an aggregation of the strings returned by each provider.

For more information on using the **HR_FAILED** macro, see Using Macros for Error Handling.

See Also

IAddrBook : IUnknown interface, **IMAPISession::OpenAddressBook** method

IMAPISupport::OpenEntry

The **IMAPISupport::OpenEntry** method opens an object given its entry identifier.

HRESULT OpenEntry(
  ULONG *cbEntryID*,
  LPENTRYID *lpEntryID*,
  LPCIID *lpInterface*,
  ULONG *ulOpenFlags*,
  ULONG FAR * *lpulObjType*,
  LPUNKNOWN FAR * *lppUnk*
)

Parameters

cbEntryID
  Input parameter containing the size, in bytes, of the entry identifier pointed to by the *lpEntryID* parameter.

lpEntryID
  Input parameter pointing to the entry identifier of the object to open.

lpInterface
  Input parameter pointing to the interface identifier (IID) for the object to open. Passing NULL indicates the object is cast to the standard interface for such an object.

ulOpenFlags
  Input parameter containing a bitmask of flags that controls how the object is opened. The following flags can be set:

  MAPI_BEST_ACCESS
    Indicates the object should be opened with the maximum network permissions allowed for the user and the maximum client application access. For example, if the client has read/write access, the object is opened with read/write access; if the client has read-only access, the object is opened with read-only access. The client can retrieve the access level by getting the PR_ACCESS_LEVEL property.

  MAPI_DEFERRED_ERRORS
    Indicates the call is allowed to succeed even if the underlying object is not accessible to the calling implementation. If the object is not accessible, some subsequent call to the object might return an error.

  MAPI_MODIFY
    Requests write access. By default, objects are created with read-only access, and clients should not work on the assumption that read/write access has been granted.

lpulObjType
  Output parameter pointing to a variable where the type of the opened object is stored.

lppUnk
  Output parameter pointing to a variable where the pointer to the opened object is stored.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

MAPI_E_NO_ACCESS
  An attempt was made to modify a read-only object or an attempt was made to access an object for which the user has insufficient permissions.

MAPI_E_NOT_FOUND
  The object indicated by *lpEntryID* does not exist.

MAPI_E_UNKNOWN_ENTRYID
The object indicated by the *lpEntryID* parameter is not recognized. This value is typically returned if the message store or address book provider that contains the object is not open.

Remarks

Service providers call the **IMAPISupport::OpenEntry** method to open objects. Default behavior is to open an object as read-only, unless the call sets the MAPI_MODIFY or MAPI_BEST_ACCESS flag in the *ulFlags* parameter. Unlike when calling the **IMAPISession::OpenEntry** method, a provider cannot open the root folder by passing NULL in *lpEntryID* for **IMAPISupport::OpenEntry**.

The calling provider should check the value returned in the *lpulObjType* parameter to determine whether the object type returned is what was expected. Commonly, after the provider checks the type of the object, it then casts the pointer in the *lppUnk* parameter into a message object pointer, a folder object pointer, or another appropriate object pointer. In order to open address book objects, a provider must open the address book first.

If a provider passes an entry identifier in the *lpEntryID* parameter that belongs to a container, such as a message store or address book that is not currently open, the call might fail and return MAPI_E_UNKNOWN_ENTRYID.

IMAPISupport::OpenProfileSection

The **IMAPISupport::OpenProfileSection** method opens a section of the current profile and returns a pointer that provides further access.

HRESULT OpenProfileSection(
  LPMAPIUID *lpUid*,
  ULONG *ulFlags*,
  LPPROFSECT FAR * *lppProfileObj*
)

Parameters

lpUid
  Input parameter pointing to the **MAPIUID** structure holding the MAPI unique identifier (MAPIUID) for the profile section. Passing NULL for the *lpUid* parameter opens a profile section containing service provider information and credentials for the current provider session.

ulFlags
  Input parameter containing a bitmask of flags that controls how the profile section is opened. The following flags can be set:

  MAPI_DEFERRED_ERRORS
    Indicates the call is allowed to succeed even if the underlying object is not accessible to the calling provider. If the object is not accessible, some subsequent call to the object might return an error.

  MAPI_MODIFY
    Requests read/write access. By default, objects are created with read-only access, and clients should not work on the assumption that read/write access has been granted.

lppProfileObj
  Output parameter pointing to a variable where the pointer to the returned profile section object is stored.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

MAPI_E_NO_ACCESS
  An attempt was made to modify a read-only profile section or an attempt was made to access an object for which the user has insufficient permissions.

MAPI_E_NOT_FOUND
  The requested object does not exist.

MAPI_E_UNKNOWN_FLAGS
  Reserved or unsupported flags were used, and therefore the operation did not complete.

Remarks

Service providers call the **IMAPISupport::OpenProfileSection** method to open a property interface on a section of the profile. A profile section object supporting the **IProfSect** interface is returned in the *lppProfSect* parameter. Providers can use an **OpenProfileSection** call to make their configuration information available to other implementations. To do so, a provider passes in *lpUid* a MAPIUID for the profile section that is known to other client applications or providers.

Default behavior for **OpenProfileSection** is to open the profile section object as read-only, unless a provider sets the MAPI_MODIFY flag in the *ulFlags* parameter. If an **OpenProfileSection** call attempts to open a nonexistent section with read-only access, MAPI_E_NOT_FOUND is returned. If an **OpenProfileSection** call opens a nonexistent profile section with read/write access by passing

MAPI_MODIFY in *ulFlags*, the call creates the section.

See Also

IMAPIProp : IUnknown interface, **IProfSect : IMAPIProp** interface, **MAPIUID** structure

IMAPISupport::OpenTemplateID

The **IMAPISupport::OpenTemplateID** method allows run-time binding of a foreign address book provider's interface implementation to an entry in a container in the calling provider. This method is called only for interaction between address book providers.

HRESULT OpenTemplateID(
  ULONG *cbTemplateID***,**
  LPENTRYID *lpTemplateID***,**
  ULONG *ulTemplateFlags***,**
  LPMAPIPROP *lpMAPIPropData***,**
  LPCIID *lpInterface***,**
  LPMAPIPROP FAR * *lppMAPIPropNew***,**
  LPMAPIPROP *lpMAPIPropSibling*
)

Parameters

cbTemplateID
  Input parameter containing the size, in bytes, of the *lpTemplateID* parameter.

lpTemplateID
  Input parameter pointing to the template identifier representing an entry in a foreign provider's container. This value is obtained from the entry's PR_TEMPLATEID property.

ulTemplateFlags
  Input parameter containing a bitmask of flags used to describe to the foreign provider how the entry is to be opened. The following flag can be set:

  FILL_ENTRY
    Should be used when calling **OpenTemplateID** to bind code for a new entry in the calling provider's address book container. This flag indicates to MAPI and to the foreign provider that receives a subsequent **IABLogon::OpenTemplateID** call, that a new entry is being created. The foreign provider can control how the entry is created by modifying properties in the object pointed to by the *lpMAPIPropData* parameter or by returning a specific interface implementation in *lppMAPIPropNew* to control how properties for the new entry are set.

lpMAPIPropData
  Input parameter pointing to the property object with the data for the entry in the calling provider's container. This is the object that is to be bound to the foreign provider's property object implementation returned in the lppMAPIPropNew parameter.This object must support the interface being requested in the *lpInterface* parameter and it must provide read/write access..

lpInterface
  Input parameter pointing to the interface identifier (IID) specifying the desired interface for the object returned in *lppMAPIPropNew*. Passing NULL for this parameter is the same as passing the standard interface for a messaging user, IID_IMailUser.

lppMAPIPropNew
  Output parameter pointing to the property object implementation supplied by the foreign provider.

lpMAPIPropSibling
  Reserved; must be NULL.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

MAPI_E_UNKNOWN_ENTRYID
  The foreign address book provider doesn't exist.

Remarks

Address book providers call the **IMAPISupport**::**OpenTemplateID** method to bind, at run time, interface implementation code that exists in a foreign address book provider to data for an entry in the calling provider.

An address book provider only needs to call **OpenTemplateID** if it supports the storage of entries with template identifiers from foreign address book providers. Such support places additional requirements on the calling provider's **CreateEntry** and **OpenEntry** implementations.

The calling provider should ensure that the object it returns from **CreateEntry** or **OpenEntry** uses the bound interface, the interface returned by **OpenTemplateID** in the *lppMAPIPropNew* parameter, to manipulate data in the property object.

If the **OpenTemplateID** call returns as the bound interface the same property object implementation as the calling provider passed in, the calling provider can **Release** its reference to this property object. This is because the foreign provider has called the object's **AddRef** method to keep its own reference. If the foreign provider does not need to keep a reference to the property object, then **OpenTemplateID** will return the unbound property object back to the calling provider.

If **OpenTemplateID** fails with MAPI_E_UNKNOWN_ENTRYID, the calling provider should try to continue by treating the entry as read-only.

See Also

IABLogon::OpenTemplateID method, **IPropData : IMAPIProp** interface, PR_TEMPLATEID property

IMAPISupport::PrepareSubmit

The **IMAPISupport::PrepareSubmit** method prepares a message for submission to the MAPI spooler.

HRESULT PrepareSubmit(
  LPMESSAGE *lpMessage***,**
  ULONG FAR * *lpulFlags*
)

Parameters

lpMessage
  Input parameter pointing to the message to prepare.

lpulFlags
  On input, the *lpulFlags* parameter is reserved and must be zero. On output, *lpulFlags* must be NULL.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

Remarks

Message store providers call the **IMAPISupport::PrepareSubmit** method to prepare a message for submission to the MAPI spooler. Message store providers call **PrepareSubmit** as one of the first steps in their **IMessage::SubmitMessage** implementation. Clients call **IMessage::SubmitMessage** to initiate the transmission of a message.

Message store providers must call the **IMAPISupport::SpoolerNotify** method, passing the NOTIFY_READYTOSEND flag in the *ulFlags* parameter, to synchronize the MAPI spooler and to ensure that all needed transport providers are logged on and their address types registered. The **SpoolerNotify** call must be made once per session before the call to **PrepareSubmit**.

Message store providers call the **IMAPIFolder::GetMessageStatus** method to check the status of a message. If a message has its status flag MSGSTATUS_RESEND set, its recipient list is checked and the PR_RESPONSIBILITY property is set to TRUE for all recipients that don't have the MAPI_SUBMITTED flag set. The **IMessage::ModifyRecipients** method is then called to remove the recipients from the list that have already received the message.

See Also

IMAPIFolder::GetMessageStatus method, **IMessage::SubmitMessage** method

IMAPISupport::ReadReceipt

The **IMAPISupport::ReadReceipt** method generates a read or nonread report for a message.

HRESULT ReadReceipt(
  ULONG *ulFlags***,**
  LPMESSAGE *lpReadMessage***,**
  LPMESSAGE FAR * *lppEmptyMessage*
)

Parameters

ulFlags
  Input parameter containing a bitmask of flags that controls how the read or nonread report is generated. The following flag can be set:
  MAPI_NON_READ
    Indicates a nonread report is generated.

lpReadMessage
  Input parameter pointing to the newly read message.

lppEmptyMessage
  Input-output parameter pointing to a variable where the pointer to the newly created message to use as the report is stored. On input, the message store provider passes in the newly created message. On output, MAPI returns the changed message; only the contents of the message are changed, not the parameter's value.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

Remarks

Message store providers call the **IMAPISupport::ReadReceipt** method to instruct MAPI to generate a read or nonread report for the message indicated by the *lpReadMessage* parameter and to place the pointer to the resulting report in the *lppEmptyMessage* parameter. Setting the MAPI_NON_READ flag in the *ulFlags* parameter generates a nonread report.

Providers typically call **ReadReceipt** in response to a message being read, but also when a message is moved or copied. Providers do not call **ReadReceipt** when a message is deleted. Providers can call **ReadReceipt** in response to calls to the **IMessage::SetReadFlag** method, but a read or nonread report should only be sent once for a message. Providers need to keep track of messages' read status and should not send multiple reports for a single message.

If *lppEmptyMessage* references a valid message when MAPI returns from **ReadReceipt**, the store provider should call **IMessage::SubmitMessage** followed by the **IUnknown::Release** method for the message.

Providers can either hide or display read and nonread reports generated by stores in their folders. Storing read and nonread reports in hidden folders allows providers to implement tighter security.

If **ReadReceipt** fails, the message should be released without being submitted. Providers can also be implemented to store the message's status and again attemp to generate the read or nonread report later.

See Also

IMAPIFolder::DeleteMessages method, **IMessage::SubmitMessage** method, PR_READ_RECEIPT_REQUESTED property

IMAPISupport::RegisterPreprocessor

The **IMAPISupport::RegisterPreprocessor** method registers a preprocessor function for a transport provider.

HRESULT RegisterPreprocessor(
  LPMAPIUID *lpMuid*,
  LPTSTR *lpszAdrType*,
  LPTSTR *lpszDLLName*,
  LPSTR *lpszPreprocess*,
  LPSTR *lpszRemovePreprocessInfo*,
  ULONG *ulFlags*
)

Parameters

lpMuid
  Input parameter pointing to the **MAPIUID** structure holding the MAPI unique identifier (MAPIUID) for the entry identifier that the preprocessor function handles. The *lpMuid* parameter can be NULL.

lpszAdrType
  Input parameter pointing to a string containing the messaging address type for the messages the function operates on, such as FAX, SMTP, or X500. The *lpszAdrType* parameter can be NULL or empty.

lpszDLLName
  Input parameter pointing to a string naming the dynamic-link library (DLL) containing the entry point for the preprocessor function.

lpszPreprocess
  Input parameter pointing to a string naming the function. The *lpszPreprocess* parameter can be NULL.

lpszRemovePreprocessInfo
  Input parameter pointing to a string naming the function that removes preprocessor information. The *lpszRemovePreprocessInfo* parameter can be NULL.

ulFlags
  Reserved; must be zero.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

Remarks

Transport providers call the **IMAPISupport::RegisterPreprocessor** method to register their preprocessor function. Such a function must be registered before they can be called by the MAPI spooler to process messages.

The *lpMuid* and *lpszAdrType* parameters are used to match the preprocessor function with specific address types. If both *lpMuid* and *lpszAdrType* are non-null, matching occurs if either MAPIUID or the address type match. If *lpMuid* is NULL and *lpszAdrType* is non-null, then matching only occurs on the address type. If *lpMuid* is non-null and *lpszAdrType* is NULL, then matching only occurs on the specific MAPIUID and for any address type. If both are NULL, then the message is processed.

The *lpszPreprocess*, *lpszRemovePreprocessInfo* and *lpszDLLName* parameters should all point to strings that can be used in conjunction with calls to the Win32 **GetProcAddress** function so the preprocessor's DLL entry point is called correctly.

Calls to preprocessors are specific to transport provider order. This functionality means that if a transport provider ahead of your transport provider in MAPI's transport order is able to handle a message, your provider's preprocessor won't get called for that message.

See Also

MAPIUID structure, **PreprocessMessage** function, **RemovePreprocessInfo** function

IMAPISupport::SetProviderUID

The **IMAPISupport::SetProviderUID** method informs MAPI of the MAPI unique identifier (MAPIUID) assigned to the service provider using the current support object.

HRESULT SetProviderUID(
  LPMAPIUID *lpProviderID***,**
  ULONG *ulFlags*
)

Parameters

lpProviderID
  Input parameter pointing to the **MAPIUID** structure holding the MAPIUID identifying the service provider object.

ulFlags
  Reserved; must be zero.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

Remarks

Address book and message store providers call the **IMAPISupport::SetProviderUID** method to inform MAPI that they handle all globally unique identifiers (GUIDs) as indicated by the **MAPIUID** structure passed in the *lpProvider* parameter. MAPI uses the provider MAPIUID when sending outbound messages to the MAPI spooler or when routing instructions from client applications to the appropriate providers, for example a call to the **IMAPISupport::OpenEntry** method to open another object belonging to a particular service provider.

A call to **SetProviderUID** should be made at logon to provide a MAPIUID for the provider for the logon session, although **SetProviderUID** can also be called later.

If a provider supports access to its entries with a variety of provider identifiers, it can make multiple calls to **SetProviderUID**. A call to **SetProviderUID** is additive in that it always adds a MAPIUID, even if one already exists. It never subtracts a MAPIUID, and once one is created, it cannot be removed.

See Also

MAPIUID structure

IMAPISupport::SpoolerNotify

The **IMAPISupport::SpoolerNotify** method informs the MAPI spooler the provider requires service.

HRESULT SpoolerNotify(
  ULONG *ulFlags***,**
  LPVOID *lpvData*
)

Parameters

ulFlags

Input parameter containing a bitmask of flags that controls how the transport provider is serviced. The following flags can be set:

NOTIFY_CONFIG_CHANGE

Indicates to the MAPI spooler that the transport provider's configuration has changed. This flag causes the MAPI spooler to call the transport provider's **IXPLogon::AddressTypes** and **IMAPISession::MessageOptions** methods at the next convenient time.

NOTIFY_CRITICAL_ERROR

Indicates that the transport provider has encountered an unrecoverable error situation and that the MAPI spooler should stop message processing and deinitialize the transport provider. If the provider calls the **IMAPISupport::SpoolerNotify** method with this flag set during an **IXPLogon::StartMessage** or **IXPLogon::SubmitMessage** method call, it should return from the **StartMessage** or **SubmitMessage** call immediately after the **SpoolerNotify** call with an appropriate error value.

NOTIFY_CRITSEC

Indicates the transport provider cannot allow the MAPI spooler to automatically yield to Windows during processing of calls from the provider to the MAPI spooler. The provider will either yield when it can or begin automatically yielding again later. The provider's **IXPLogon::Idle** and **IXPLogon::Poll** methods should not be called. The *lpvData* parameter is undefined and should be NULL.

NOTIFY_NEWMAIL

Indicates the transport provider has new incoming messages the MAPI spooler should request. Message downloading occurs at the next available time. The *lpvData* parameter is undefined and should be set to NULL.

NOTIFY_NEWMAIL_RECEIVED

Indicates *lpvData* points to a **NEWMAIL_NOTIFICATION** structure. This flag is used for transport providers that are tightly coupled with message store providers and is ignored if the store provider logged on with the MAPI_NO_MAIL flag set.

NOTIFY_NONCRIT

Indicates the transport provider needs to end the critical section it declared earlier. The *lpvData* parameter is undefined and should be set to NULL.

NOTIFY_READYTOSEND

Indicates the transport provider has recovered from a condition that caused it to fail earlier and is again ready to accept messages. For more information on failure conditions, see the documentation for the possible return values in **IXPLogon::SubmitMessage**. When set by a message store provider, this flag indicates the MAPI spooler should synchronize with the client application's session. If this flag is set, *lpvData* is undefined and should be set to NULL.

NOTIFY_SENTDEFERRED

Indicates the transport provider has completed some part of the process of sending a deferred message and requests to be notified at the next **IXPLogon::SubmitMessage** call. The entry identifier of the deferred message is contained in an **SBinary** structure pointed to by *lpvData*.

lpvData

Input parameter pointing to a notification structure holding data associated with the notification. The meaning of the data in *lpvData* depends on what flags are set in the *ulFlags* parameter; for more information on what *lpvData* can hold, see the preceding description for *ulFlags*.

Return Values

S_OK
: The call succeeded and has returned the expected value or values.

Remarks

Transport providers call the **IMAPISupport::SpoolerNotify** method to notify the MAPI spooler to process a new message. A transport provider can call **IMAPISupport::SpoolerNotify** at any time.

When a message store provider tightly coupled with a transport provider calls **SpoolerNotify**, the MAPI spooler opens the message and begins processing the hook function that handles new messages. This process culminates in the MAPI spooler calling the **IMsgStore::NotifyNewMail** method to inform the message store about its own new message.

The notification structure pointed to by *lpvData* is sent to the MAPI spooler by MAPI. Note that only the NOTIFY_SENT_DEFERRED and NOTIFY_CRITICAL_ERROR flags for *ulFlags* have data associated with them. These two flags must not be set with the same **SpoolerNotify** call, although any other combination of flags can be set.

If a transport provider places one of its processes within a critical section, it should set the NOTIFY_CRITSEC flag in *ulFlags*. For example, a remote transport provider uploading messages might need to display a dialog box so the user can select a telephone number to dial to establish the remote connection. If the transport provider displays the dialog box using its own function, it should declare a critical section before looping through the dialog box function. After the user closes the dialog box and the dialog box function terminates, the transport provider should end the critical section.

In response to receiving NOTIFY_CRITSEC with a **SpoolerNotify** call, the MAPI spooler stops making calls to the transport provider until the provider notifies it that the critical section has ended. Furthermore, a transport provider can return the value MAPI_E_BUSY for any calls made to update its status for an open status object while it has a critical section open.

If a transport provider has changed its configuration, it should set the NOTIFY_CONFIG_CHANGED flag in *ulFlags* so that the MAPI spooler can reconfigure the transport on **IXPLogon::AddressTypes** and **IMAPISession::MessageOptions** calls.

Message store providers that need to ensure the MAPI spooler is synchronized with the client application as part of their **IMAPISupport::PrepareSubmit** method implementation should call **SpoolerNotify**, passing the NOTIFY_READYTOSEND flag in *ulFlags*.

See Also

IMsgStore::NotifyNewMail method, **IXPLogon::StartMessage** method, **IXPLogon::SubmitMessage** method

IMAPISupport::SpoolerYield

The **IMAPISupport::SpoolerYield** method allows the transport provider to permit the MAPI spooler to give processing time to Windows.

HRESULT SpoolerYield(
  ULONG *ulFlags*
)

Parameters

ulFlags
  Reserved; must be zero.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

MAPI_W_CANCEL_MESSAGE
  Indicates the user wants to stop transfer for an incoming or outgoing message regardless of how many recipients might already have received it.

Remarks

Transport providers call the **IMAPISupport::SpoolerYield** method to allow the MAPI spooler to cede code processing to Windows. Transport providers typically make this call when they are performing lengthy operations that can be paused. This functionality allows foreground applications to run during long transport provider operations.

During the processing of a **SpoolerYield** call, the MAPI spooler might detect a number of conditions important to the transport provider, which it signals to the provider in the **SpoolerYield** return value. If **SpoolerYield** returns with MAPI_W_CANCEL_MESSAGE, the MAPI spooler has determined that the message should no longer be sent. If it receives this value a provider should return MAPI_E_USER_CANCEL to the calling process and exit if possible.

IMAPISupport::StatusRecips

The **IMAPISupport::StatusRecips** method generates delivery and nondelivery reports on behalf of transport providers.

HRESULT StatusRecips(
  LPMESSAGE *lpMessage***,**
  LPADRLIST *lpRecipList*
)

Parameters

lpMessage
  Input parameter pointing to the message for which a report is generated.

lpRecipList
  Input parameter pointing to an **ADRLIST** structure holding a set of recipients for which delivery or nondelivery information is stored.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

MAPI_W_ERRORS_RETURNED
  The call succeeded overall, but there are no recipient options for this type of recipient. To test for this warning, use the **HR_FAILED** macro. When this warning is returned, the call should be handled as successful.

Remarks

Transport providers call the **IMAPISupport::StatusRecips** method to notify MAPI that delivery or nondelivery reports should be sent for a set of one or more recipients. Transport providers can call **StatusRecips** multiple times during the processing of a message. However, transport providers that call **StatusRecips** for an open message should do their best to collect all delivery and nondelivery information for the message recipients and call **StatusRecips** for that recipient list. A single point of collection is important because each time a transport provider calls **StatusRecips**, MAPI can generate delivery and nondelivery reports, so multiple **StatusRecips** calls for one recipient can result in multiple identical reports being sent.

A provider should store properties relating to message delivery or nondelivery in the **ADRLIST** structure indicated by the *lpRecipList* parameter. The following properties are required for both delivery reports and nondelivery reports:

  PR_MESSAGE_CLASS
  PR_REPORT_TEXT
  PR_REPORT_TIME
  PR_ROWID
  PR_SEARCH_KEY

The following additional property is required for delivery reports:

  PR_MESSAGE_DELIVERY_TIME

The following additional property is required for nondelivery reports:

  PR_NDR_DIAG_CODE

MAPI adds additional properties to the final delivery or nondelivery report.

A provider should allocate memory for the **ADRLIST** structure in *lpRecipList* using the **MAPIAllocateBuffer** and **MAPIAllocateMore** functions. MAPI releases the memory by calling the **MAPIFreeBuffer** function only if **IMAPISupport::StatusRecips** succeeds.

See Also

ADRLIST structure, **IMAPISupport::Address** method, **IMAPISupport::SpoolerNotify** method, **IXPLogon::EndMessage** method, **MAPIAllocateBuffer** function, **MAPIAllocateMore** function, **MAPIFreeBuffer** function

IMAPISupport::StoreLogoffTransports

The **IMAPISupport::StoreLogoffTransports** method specifies the orderly release of a message store to the MAPI spooler.

HRESULT StoreLogoffTransports(
  ULONG FAR * *lpulFlags*
)

Parameters

lpulFlags
  Input-output parameter containing a bitmask of flags that controls how message store logoff occurs.

  On input, all flags for this parameter are mutually exclusive; a message store provider can set only one per call. Which flag is set is determined by which flag is returned from a previous client application call to the **IMsgStore::StoreLogoff** method. The following flags can be set on input for the *lpulFlags* parameter:

  LOGOFF_ABORT
    Indicates any transport provider activity for this store should be stopped before logoff. Control is returned to the client after the activity is stopped and the MAPI spooler has logged off the store. If any transport activity is taking place, the logoff does not occur and no change in MAPI spooler or transport provider behavior occurs. If transport activity is quiet, the MAPI spooler releases the store.

  LOGOFF_NO_WAIT
    Indicates the message store should not wait for messages from transport providers before closing. All outbound mail that is ready to be sent, is sent; if this store has the default Inbox, any in-process message is received, and then further reception is disabled. When all activity is completed, the MAPI spooler releases the store and control is returned to the client immediately.

  LOGOFF_ORDERLY
    Indicates the message store should not wait for information from transport providers before closing. Any message being processed by the store is completed, and no new messages are processed. When all activity is completed, the MAPI spooler releases the store and control is returned to the store provider immediately.

  LOGOFF_PURGE
    Works the same as the LOGOFF_NO_WAIT flag. The LOGOFF_PURGE flag returns control to the client after completion.

  LOGOFF_QUIET
    Indicates that if any transport provider activity is taking place, the logoff does not occur and the type of activity taking place is returned as a flag on output.

On output, the MAPI spooler can return more than one flag. The following flags can be set on output for *lpulFlags*:

  LOGOFF_COMPLETE
    Indicates the logoff can complete. All resources associated with the store have been released, and the object has been invalidated. The MAPI spooler has performed or will perform all requests. Only the **IUnknown::Release** method should be called at this point.

  LOGOFF_INBOUND
    Indicates a message is currently coming into the store from one or more transport providers.

  LOGOFF_OUTBOUND
    Indicates a message is currently being sent from the store by one or more transport providers.

  LOGOFF_OUTBOUND_QUEUE
    Indicates there are currently messages in the outbound queue for the store.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

Remarks

Message store providers call the **IMAPISupport::StoreLogoffTransports** method to give client applications some control over how MAPI handles transport provider activity as the message store is closing.

Before a **StoreLogoffTransports** call occurs, a client calls the **IMsgStore::StoreLogoff** method and sets flags in its *lpulFlags* parameter to indicate how the client requires the message store to be shut down. A message store provider should pass these client-set **IMsgStore::StoreLogoff** flags as input values for the **StoreLogoffTransports** method's *lpulFlags*. The status returned from a client call to **IMsgStore::StoreLogoff** is usually the same status that the message store provider then passes on the call to **StoreLogoffTransports**.

If the client calls **Release** on a message store object without calling **IMsgStore::StoreLogoff** first, the message store provider involved should set the LOGOFF_ABORT flag in *lpulFlags* for **StoreLogoffTransports**.

If another process has the store to be logged off open for the same profile, MAPI ignores a call to **StoreLogoffTransports** and returns the flag LOGOFF_COMPLETE in *lpulFlags*.

The behavior of the store provider following the return from **StoreLogoffTransports** should be based on the value of *lpulFlags*, which indicates system status and conveys client instructions on logoff behavior.

See Also

IMsgStore::StoreLogoff method, **IXPLogon::FlushQueues** method

IMAPISupport::Subscribe

The **IMAPISupport::Subscribe** method sets up a subscription for notification events with MAPI.

HRESULT Subscribe(
  LPNOTIFKEY *lpKey*,
  ULONG *ulEventMask*,
  ULONG *ulFlags*,
  LPMAPIADVISESINK *lpAdviseSink*,
  ULONG FAR * *lpulConnection*
)

Parameters

lpKey
: Input parameter pointing to the **NOTIFKEY** structure for the object whose changes should generate notifications. The *lpKey* parameter provides a unique key to the object and cannot be NULL.

ulEventMask
: Input parameter containing an event mask of the types of notification events occurring for the object for which MAPI will generate notifications. The event mask filters specific cases. Each event type has a structure associated with it that holds additional information about the event. The following table lists the possible event types along with their corresponding structures.

Notification event type	Corresponding structure
`fnevCriticalError`	**ERROR_NOTIFICATION**
`fnevNewMail`	**NEWMAIL_NOTIFICATION**
`fnevObjectCreated`	**OBJECT_NOTIFICATION**
`fnevObjectDeleted`	**OBJECT_NOTIFICATION**
`fnevObjectModified`	**OBJECT_NOTIFICATION**
`fnevObjectCopied`	**OBJECT_NOTIFICATION**
`fnevSearchComplete`	**OBJECT_NOTIFICATION**
`fnevTableModified`	**TABLE_NOTIFICATION**
`fnevStatusObjectModified`	**STATUS_OBJECT_NOTIFICATION**
`fnevExtended`	**EXTENDED_NOTIFICATION**

ulFlags
: Input parameter containing a bitmask of flags that controls how notification occurs. The following flag can be set:

NOTIFY_SYNC
: Indicates all notification callbacks should be made before the **Notify** method returns to the calling service provider. If this flag is not set, notification callbacks are queued to the processes that have subscribed and started when those processes gain control of the CPU.

lpAdviseSink
: Input parameter pointing to an advise sink object created by the calling provider. This advise sink is called when an event occurs for the object about which notification has been requested.

lpulConnection
: Output parameter pointing to a variable that upon a successful return holds the connection number for the notification subscription.

Return Values

S_OK
: The call succeeded and has returned the expected value or values.

Remarks

Service providers call the **IMAPISupport::Subscribe** method to allow MAPI to handle the registration of notifications for their objects. A provider can make one call to **Subscribe** for every call to its **HrAllocAdviseSink** function or to an **Advise** method. Three **IMAPISupport** methods, **Subscribe**, **Unsubscribe** and **Notify**, help provider developers implement notification. These methods are merely a convenience; providers can use them as the basis for their implementation of notification support, or they can supply their own implementation.

To use the MAPI support methods for notification, providers must create a key for the object for which notifications should be generated and place that key in the *lpKey* parameter. MAPI uses this key after notification subscription is set up to search for any callbacks registered for the corresponding object. The value of the key must be unique to the item being subscribed; any value that uniquely identifies the object and is easily regenerated each time the object changes is acceptable. However, a key must be provided; NULL cannot be passed in *lpKey*. A **NOTIFKEY** structure is used in *lpKey* so a provider can map different entry identifiers to the same object using the same key.

Once notification subscription is set up, a provider, when calling **IMAPISupport::Notify** to signal that a notification event has taken place, should provide the notification key in **Notify**'s *lpKey* parameter.

The NOTIFY_SYNC flag in the **Subscribe** *ulFlags* parameter indicates whether the service provider requested synchronous or asynchronous notifications. MAPI records this information and issues the appropriate type of callback when the provider calls **Notify**. Synchronous notifications can never be requested on behalf of client applications; they are only for internal service provider use, for situations where careful sequencing of access to provider data structures is necessary.

There are limitations on what a synchronous callback function can do:

- It cannot cause another synchronous notification to be generated.
- It cannot display a user interface.

Although a synchronous callback function is called with the same parameters as an asynchronous callback function, its possible return values are predetermined. A synchronous callback function can return CALLBACK_DISCONTINUE, indicating that MAPI should immediately stop processing the callbacks for this notification, something an asynchronous callback function cannot do. If a callback function returns CALLBACK_DISCONTINUE, MAPI does not queue additional eligible notifications but calls **Notify** and sets the NOTIFY_CANCELED flag in its *lpulFlags* parameter.

A synchronous callback function can also stop callback processing by returning the CALLBACK_DISCONTINUE flag in the **HrAllocAdviseSink** function's *lppAdviseSink* parameter.

See Also

HrAllocAdviseSink function, **IMAPIAdviseSink::OnNotify** method, **IMAPISupport::Notify** method, **NOTIFICATION** structure, **NOTIFKEY** structure

IMAPISupport::Unsubscribe

The **IMAPISupport::Unsubscribe** method removes an object's subscription for notification of changes previously established with a call to the **IMAPISupport::Subscribe** method.

```
HRESULT Unsubscribe(
  ULONG ulConnection
 )
```

Parameters

ulConnection
Input parameter containing the number of the registration connection returned by a call to **IMAPISupport::Subscribe**.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

MAPI_E_NOT_FOUND
The connection number passed in does not exist.

Remarks

Providers call the **IMAPISupport::Unsubscribe** method to cancel a notification subscription. **Unsubscribe** does so by releasing the pointer to the advise sink object passed in the *lpAdviseSink* parameter in the previous call to **Subscribe**. As part of discarding the pointer to the advise sink object, the object's **IUnknown::Release** method is called. Generally, **Release** is called during the **Unsubscribe** call, but if another thread is in the process of calling the **IMAPIAdviseSink::OnNotify** method for the advise sink object, the **Release** call is delayed until the **OnNotify** method returns.

See Also

IMAPIAdviseSink::OnNotify method, **IMAPISupport::Subscribe** method

IMAPISupport::WrapStoreEntryID

The **IMAPISupport::WrapStoreEntryID** method maps the private entry identifier of a message store object to an entry identifier more useful to the messaging system.

HRESULT WrapStoreEntryID(
  ULONG *cbOrigEntry*,
  LPENTRYID *lpOrigEntry*,
  ULONG FAR * *lpcbWrappedEntry*,
  LPENTRYID FAR * *lppWrappedEntry*
)

Parameters

cbOrigEntry
 Input parameter containing the size, in bytes, of the entry identifier pointed to by the *lpOrigEntry* parameter.

lpOrigEntry
 Input parameter pointing to the private entry identifier for the message store provider.

lpcbWrappedEntry
 Output parameter pointing to a variable that contains the size, in bytes, of the entry identifier pointed to by the *lppWrappedEntry* parameter.

lppWrappedEntry
 Output parameter pointing to a pointer to the new entry identifier returned, to which MAPI has mapped the message store provider's private entry identifier.

Return Values

S_OK
 The call succeeded and has returned the expected value or values.

Remarks

Service providers use the **IMAPISupport::WrapStoreEntryID** method to have MAPI generate a wrapped entry identifier for the store provider object. To allow storage of a message store's PR_STORE_ENTRYID property by the messaging system, MAPI must map the store's private entry identifier to an identifier useful to the messaging system. MAPI returns the wrapped version in response to any requests for PR_STORE_ENTRYID. In other words, when a client application calls the **IMAPIProp::GetProps** method to retrieve PR_STORE_ENTRYID, a store provider should call **WrapStoreEntryID** instead of returning its entry identifier directly.

One use of **WrapStoreEntryID** is to allow an entry in a message store to be opened if the store itself is not open. In such cases, a valid entry identifier for the entry to open must be passed in the **OpenEntry** call.

Calls to the **IMSProvider::Logon** and **IMSLogon::CompareEntryIDs** methods always obtain the store's private entry identifier; the wrapped version is used only between client applications and MAPI.

The memory for the entry identifier returned in the *lppWrappedEntry* parameter must be freed using the **MAPIFreeBuffer** function after the provider is done with the identifier.

See Also

IMAPIProp::GetProps method, **IMAPISupport::CompareEntryIDs** method, **IMSLogon::CompareEntryIDs** method, **IMSProvider::Logon** method, **MAPIFreeBuffer** function

IMAPITable : IUnknown

The **IMAPITable** interface is used for working with MAPI table objects.

At a Glance

Specified in header file:	MAPIDEFS.H
Object that supplies this interface:	Table object
Corresponding pointer type:	LPMAPITABLE
Implemented by:	Service providers
Called by:	Client applications, service providers

Vtable Order

GetLastError	Returns a **MAPIERROR** structure containing information about the last error that occurred for a table object.
Advise	Registers an implementation for notification on changes to a table object.
Unadvise	Removes a table's registration for notification of changes previously established with a call to the **IMAPITable::Advise** method.
GetStatus	Returns the status and type of a table.
SetColumns	Sets the order of columns for table rows returned by the **IMAPITable::QueryRows** method.
QueryColumns	Returns either the current list of columns for a table view or the full list of columns available for the table.
GetRowCount	Returns the total number of rows in a table view.
SeekRow	Moves the cursor to a specific position in a table.
SeekRowApprox	Moves the cursor to an approximate fractional position in a table.
QueryPosition	Retrieves the current table row position of the cursor, based on a fractional value.
FindRow	Finds the next row in a table that contains a property matching the specified criteria.
Restrict	Applies a restriction to a table, reducing the rows visible to only those matching the restriction criteria.
CreateBookmark	Marks the current position of the table cursor so an implementation can return to that position even when the table is updated.
FreeBookmark	Releases a bookmark from memory.
SortTable	Sorts table rows based on the sort criteria provided.
QuerySortOrder	Retrieves the current sort order for a table.
QueryRows	Returns one or more rows from a table, beginning at the current cursor position.
Abort	Stops any asynchronous operations currently in progress for a table.
ExpandRow	Expands a collapsed table category and adds the rows

	of that category to the table view.
CollapseRow	Collapses a table category and removes it from the table view.
WaitForCompletion	Suspends the calling implementation while asynchronous operations occur on a table.
GetCollapseState	Returns the data necessary to rebuild the current table view.
SetCollapseState	Reestablishes the expanded or collapsed state of the table view that was saved by a call to the **IMAPITable::GetCollapseState** method.

IMAPITable::Abort

The **IMAPITable::Abort** method stops any asynchronous operations currently in progress for a table.

HRESULT Abort()

Parameters

None

Return Values

S_OK
   The call succeeded and has returned the expected value or values.

MAPI_E_UNABLE_TO_ABORT
   The operation is in progress and could not be canceled.

Remarks

Use the **IMAPITable::Abort** method to stop any asynchronous operation that is in progress on a table. To detect asynchronous operations currently in progress, a client or provider can call the **IMAPITable::GetStatus** method.

If a call to the **IMAPITable::Restrict** method is interrupted by a call to **Abort**, whatever data is in the table when the **Abort** call is processed remains in the table. If a call to the **IMAPITable::SortTable** method is interrupted by a call to **Abort**, the order of rows in the table is not changed from their order before the sort operation.

If the asynchronous operation cannot be stopped or if it completes before **Abort** returns, **Abort** returns MAPI_E_UNABLE_TO_ABORT.

See Also

IMAPITable::GetStatus method, **IMAPITable::Restrict** method, **IMAPITable::SortTable** method

IMAPITable::Advise

The **IMAPITable::Advise** method registers an implementation for notification on changes to a table object.

HRESULT Advise(
  ULONG *ulEventMask***,**
  LPMAPIADVISESINK *lpAdviseSink***,**
  ULONG FAR * *lpulConnection*
)

Parameters

ulEventMask
  Input parameter containing an event mask of the types of notification events occurring for the object about which MAPI will generate notifications. The mask filters specific cases. Each event type has a structure associated with it that holds additional information about the event. The only possible event type is fnevTableModified; the corresponding data structure is **TABLE_NOTIFICATION**.

lpAdviseSink
  Input parameter pointing to an advise sink object to be called when an event occurs for the table object about which notification has been requested. This advise sink object must have already been allocated.

lpulConnection
  Output parameter pointing to a variable that upon a successful return holds the connection number for the notification registration. The connection number must be nonzero.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

MAPI_E_NO_SUPPORT
  The service provider either does not support changes to its objects or does not support notification of changes.

Remarks

Use the **IMAPITable::Advise** method to register a table object implemented in the provider for notification callbacks. Whenever a change occurs to the table object, the provider checks to see what event mask bit was set in the *ulEventMask* parameter and thus what type of change occurred. If a bit is set, then the provider calls the **IMAPIAdviseSink::OnNotify** method for the advise sink object indicated by the *lpAdviseSink* parameter to report the event. Data passed in the notification structure to the **OnNotify** routine describes the event.

The call to **OnNotify** can occur during the call that changes the object, or at any following time. On systems that support multiple threads of execution, the call to **OnNotify** can occur on any thread. For a way to turn a call to **OnNotify** that might happen at an inopportune time into one that is safer to handle, a provider should use the **HrThisThreadAdviseSink** function.

To provide notifications, the provider implementing **Advise** needs to keep a copy of the pointer to the *lpAdviseSink* advise sink object; to do so, it calls the **IUnknown::AddRef** method for the advise sink to maintain its object pointer until notification registration is canceled with a call to the **IMAPITable::Unadvise** method. The **Advise** implementation should assign a connection number to the notification registration and call **AddRef** on this connection number before returning it in the *lpulConnection* parameter. Service providers can release the advise sink object before the registration is canceled, but they must not release the connection number until **Unadvise** has been called.

After a call to **Advise** has succeeded and before **Unadvise** has been called, clients must be prepared for the advise sink object to be released. A clients should therefore release its advise sink object after **Advise** returns unless it has a specific long-term use for it.

Because of the asynchronous behavior of notification, implementations that change table column settings can receive notifications with information organized in a previous column order. For instance, a table row might be returned for a message that has just been deleted from the container. Such a notification is sent when the column setting change has been made and information about it sent but the notification table view has not been updated with that information yet.

For more information on the notification process, see About Notification.

See Also

HrThisThreadAdviseSink function, **IMAPIAdviseSink::OnNotify** method, **IMAPITable::Unadvise** method, **TABLE_NOTIFICATION** structure

IMAPITable::CollapseRow

The **IMAPITable::CollapseRow** method collapses a table category and removes it from the table view.

HRESULT CollapseRow(
  ULONG *cbInstanceKey*,
  LPBYTE *pbInstanceKey*,
  ULONG *ulFlags*,
  ULONG FAR * *lpulRowCount*
)

Parameters

cbInstanceKey
  Input parameter containing the size, in bytes, of the instance key pointed to by the *pbInstanceKey* parameter.

pbInstanceKey
  Input parameter pointing to a variable containing the instance key – that is, the PR_INSTANCE_KEY property – for the categorization row.

ulFlags
  Reserved; must be zero.

lpulRowCount
  Output parameter pointing to a variable containing the total number of rows, including heading and data rows, that are being removed from the table view.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

MAPI_E_NOT_FOUND
  The categorization row does not exist.

MAPI_E_INVALID_ENTRYID
  The categorization row does not exist. MAPI service providers that implement IMAPITable are allowed to return this error instead of MAPI_E_NOT_FOUND.

Remarks

Use the **IMAPITable::CollapseRow** method to collapse a table category and remove it from the table view. The rows are collapsed at the row containing the PR_INSTANCE_KEY property passed in the *pbInstanceKey* parameter. The number of rows that are removed from the view is returned in the *lpulRowCount* parameter.

Providers must not generate notifications on rows that are collapsed out of a table view.

See Also

IMAPITable::ExpandRow method, **IMAPITable::GetCollapseState** method, **IMAPITable::QuerySortOrder** method, **IMAPITable::SetCollapseState** method, **IMAPITable::SortTable** method, **SSortOrderSet** structure

IMAPITable::CreateBookmark

The **IMAPITable::CreateBookmark** method marks the current position of the table cursor so an implementation can return to that position even when the table is updated.

HRESULT CreateBookmark(
  BOOKMARK FAR * *lpbkPosition*
)

Parameters

lpbkPosition
  Output parameter pointing to a variable where the returned 32-bit bookmark value is stored. This bookmark can later be passed in a call to the **IMAPITable::SeekRow** method.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

MAPI_E_UNABLE_TO_COMPLETE
  The requested operation could not be completed.

Remarks

Use the **IMAPITable::CreateBookmark** method to create a bookmark, which is used to retain information about a position in a table so as to return to that position. The bookmarked position is associated with the object at that row in the table.

Because of the memory expense of maintaining cursor positions in this way, an implementation can limit the number of bookmarks permitted. If an implementation has done so and an attempt is made to create a bookmark that would surpass the number allowed, the call to **CreateBookmark** returns MAPI_E_UNABLE_TO_COMPLETE.

A bookmark pointing to a row that is no longer in the table view can still be used. If a client or provider attempts to move the cursor to such a bookmark, the cursor moves to the next visible row and stops there. A call using a bookmark pointing to a collapsed row returns MAPI_W_POSITION_CHANGED. Table implementations can move bookmarks for positions collapsed out of view either at the time of use or at the time the row is collapsed. If a bookmark is moved at the time the row is collapsed, a bit must be retained in the bookmark that indicates whether the bookmark has moved since its last use or, if it has never been used, since its creation.

CreateBookmark can allocate memory for the bookmark it creates. An implementation must release the resources for the bookmark by calling the **IMAPITable::FreeBookmark** method.

Bookmarks are not supported for a message's attachment table, and **CreateBookmark** will return MAPI_E_NO_SUPPORT.

See Also

IMAPITable::FreeBookmark method, **IMAPITable::SeekRow** method

IMAPITable::ExpandRow

The **IMAPITable::ExpandRow** method expands a collapsed table category and adds the rows of that category to the table view.

HRESULT ExpandRow(
  ULONG *cbInstanceKey*,
  LPBYTE *pbInstanceKey*,
  ULONG *ulRowCount*,
  ULONG *ulFlags*,
  LPSRowSet FAR * *lppRows*,
  ULONG FAR * *lpulMoreRows*
)

Parameters

cbInstanceKey
  Input parameter containing the size, in bytes, of the instance key pointed to by the *pbInstanceKey* parameter.

pbInstanceKey
  Input parameter pointing to a variable containing the instance key – that is, the PR_INSTANCE_KEY property – for the categorization row.

ulRowCount
  Input parameter containing the maximum number of rows to return in the *lppRows* parameter.

ulFlags
  Reserved; must be zero.

lppRows
  Output parameter pointing to a variable where a returned **SRowSet** structure is stored. The **SRowSet** holds the set of new rows inserted into the table view after the row indicated by the *pbInstanceKey* parameter. The *lppRows* parameter can be NULL if the *ulRowCount* parameter is zero.

lpulMoreRows
  Output parameter pointing to a variable holding the total number of rows added to the table.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

MAPI_E_NOT_FOUND
  The categorization row doesn't exist.

Remarks

Use the **IMAPITable::ExpandRow** method to expand a collapsed table category, adding its rows to the table view. The position of the bookmark BOOKMARK_CURRENT is moved to the row immediately following the last row in the **SRowSet** returned in *lppRows*. If zero rows were requested, or zero rows were returned, the position of BOOKMARK_CURRENT is set to the row following the row specified in *pbInstanceKey*.

Providers must not generate notifications on rows that are collapsed out of a table view.

See Also

IMAPITable::CollapseRow method

IMAPITable::FindRow

The **IMAPITable::FindRow** method finds the next row in a table that contains a property matching the specified criteria.

HRESULT FindRow(
  LPSRestriction *lpRestriction***,**
  BOOKMARK *BkOrigin***,**
  ULONG *ulFlags*
)

Parameters

lpRestriction
  Input parameter pointing to an **SRestriction** structure containing the property to search for.

BkOrigin
  Input parameter indicating the bookmark from which the search originates. A bookmark can be created using the **IMAPITable::CreateBookmark** method, or one of the following predefined values can be passed:

  BOOKMARK_BEGINNING
    Searches from the beginning of the table.

  BOOKMARK_CURRENT
    Searches from the row in the table where the cursor is located.

  BOOKMARK_END
    Searches from the end of the table.

ulFlags
  Input parameter containing a bitmask of flags that controls the direction of the search. The following flag can be set:

  DIR_BACKWARD
    Searches backward from the bookmark.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

MAPI_E_INVALID_BOOKMARK
  The bookmark is invalid because it has been removed or because it is beyond the last row requested.

MAPI_E_NOT_FOUND
  No rows were found that matched the restriction.

MAPI_W_POSITION_CHANGED
  The call succeeded, but the bookmark used in the operation is no longer set at the same row as when it was last used; if the bookmark has not been used, it is no longer in the same position as when it was created. To test for this warning, use the **HR_FAILED** macro. When this warning is returned, the call should be handled as successful.

Remarks

Service providers implement the **IMAPITable::FindRow** method to support scrolling based on strings typed by the user, especially in list boxes within addressing dialog boxes. In this type of scrolling, users enter progressively longer prefixes of a desired string value, and the client application periodically issues a **FindRow** call to jump to the first row that matches the prefix. Which direction the cursor jumps depends on which direction the search is set to run.

To use **FindRow**, a bookmark must be set. The string search can originate from any bookmark, including from the preset bookmarks indicating the current position and the beginning and end of the table. If there is a large number of rows in the table, the search operation can be slow.

Clients use a restriction to find a string prefix for scrolling as follows: For forward searching on a column sorted in ascending order, and for backward searching on a column sorted in descending order, a client passes in the *lpRestriction* parameter an **SPropertyRestriction** structure with relation **RELOP_GE** and the appropriate property tag and prefix, using the format *tag* **GE** *prefix*.

Note The type of prefix searching performed by **FindRow** is only useful when the search follows the same direction as the table organization. Implementers of the **IMAPITable** interface should note that in order to achieve the required behavior, the comparison function implied by the **RELOP_GE** passed in the property restriction structure should be the same comparison function on which the table sort order is based.

Usually, **FindRow** searches forward from the specified bookmark. The calling implementation can set the search to move backward from the bookmark by setting the DIR_BACKWARD flag in the *ulFlags* parameter. Searching forward starts from the current bookmark; searching backward starts from the row prior to the bookmark. The end position of the search is just before the first row found that satisfied the restriction. **FindRow** returns the data of the found row.

If the row pointed to by the bookmark in the *BkOrigin* parameter no longer exists in the table and the table cannot establish a new position for the bookmark, **FindRow** returns MAPI_E_INVALID_BOOKMARK. If the row pointed to by *BkOrigin* no longer exists and the table is able to establish a new position for the bookmark, **FindRow** returns MAPI_W_POSITION_CHANGED.

If the bookmark passed in *BkOrigin* is either BOOKMARK_BEGINNING or BOOKMARK_END, **FindRow** returns MAPI_E_NOT_FOUND if no matching row is found. If the bookmark used in *BkOrigin* is BOOKMARK_CURRENT, **FindRow** can return MAPI_W_POSITION_CHANGED but not MAPI_E_INVALID_BOOKMARK, because there is always a current cursor position.

The PR_INSTANCE_KEY property column is required for all tables, and all implementations of **FindRow** are required to support calls seeking a row based on PR_INSTANCE_KEY.

For more information on using the **HR_FAILED** macro, see Using Macros for Error Handling.

See Also

IMAPITable::CreateBookmark method, **SPropertyRestriction** structure, **SRestriction** structure

IMAPITable::FreeBookmark

The **IMAPITable::FreeBookmark** method releases a bookmark from memory.

HRESULT FreeBookmark(
  BOOKMARK *bkPosition*
)

Parameters

bkPosition
  Input parameter containing a token representing the bookmark to be freed, as obtained from a call to the **IMAPITable::CreateBookmark** method.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

MAPI_E_INVALID_BOOKMARK
  The bookmark does not exist.

Remarks

Use the **IMAPITable::FreeBookmark** method to release a bookmark that is no longer needed. The bookmark is no longer valid after this call. When MAPI frees a table from memory, it also frees all associated bookmarks.

If the calling implementation passes one of the three predefined bookmarks to **FreeBookmark** to be freed, **FreeBookmark** ignores it and returns no error.

See Also

IMAPITable::CreateBookmark method

IMAPITable::GetCollapseState

The **IMAPITable::GetCollapseState** method returns the data necessary to rebuild the current table view.

HRESULT GetCollapseState(
  ULONG *ulFlags*,
  ULONG *cbInstanceKey*,
  LPBYTE *lpbInstanceKey*,
  ULONG FAR * *lpcbCollapseState*,
  LPBYTE FAR * *lppbCollapseState*
)

Parameters

ulFlags
  Reserved; must be zero.

cbInstanceKey
  Input parameter containing the size, in bytes, of the instance key pointed to by the *lpbInstanceKey* parameter.

lpbInstanceKey
  Input parameter pointing to the instance key – that is, the PR_INSTANCE_KEY property – identifying the table row at which the current collapsed or expanded state should be rebuilt. The *lpbInstanceKey* parameter cannot be NULL; a row's instance key must be passed in.

lpcbCollapseState
  Output parameter pointing to a variable containing the size, in bytes, of the structures pointed to by the *lppbCollapseState* parameter.

lppbCollapseState
  Output parameter pointing to a variable where the pointer to structures containing data describing the current table view is stored.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

MAPI_E_BUSY
  Another operation is in progress; it should be allowed to complete, or it should be stopped, before this operation is attempted.

MAPI_E_NO_SUPPORT
  The operation is not supported by one or more service providers.

Remarks

Use the **IMAPITable::GetCollapseState** method to get the data necessary to rebuild a table view to its current collapsed or expanded state. Use **GetCollapseState** and the **IMAPITable::SetCollapseState** method together to present to the user, upon opening a table in your implementation, a table that in its expanded or collapsed state can be recognized as the table the user formerly viewed. Implementations of **GetCollapseState** commonly store the entire current state of all nodes of a table in the *lppbCollapseState* parameter.

To restore the expanded or collapsed state retrieved by **GetCollapseState**, an implementation calls **SetCollapseState** and rebuilds the stored state using the information in the structures saved in the *lppbCollapseState* parameter.

See Also

IMAPITable::SetCollapseState method

IMAPITable::GetLastError

The **IMAPITable::GetLastError** method returns a **MAPIERROR** structure containing information about the last error that occurred for a table object.

HRESULT GetLastError(
  HRESULT *hResult*,
  ULONG *ulFlags*,
  LPMAPIERROR FAR * *lppMAPIError*
)

Parameters

hResult
  Input parameter containing the result returned for the last call for the table object that returned an error.

ulFlags
  Input parameter containing a bitmask of flags that controls the type of the returned strings. The following flag can be set:

  MAPI_UNICODE
    Indicates the strings in the **MAPIERROR** structure returned in the *lppMAPIError* parameter are in Unicode format. If the MAPI_UNICODE flag is not set, the strings are in ANSI format.

lppMAPIError
  Output parameter pointing to a pointer to the returned **MAPIERROR** structure containing version, component, and context information for the error. The *lppMAPIError* parameter can be set to NULL if there is no **MAPIERROR** structure to return.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

MAPI_E_BAD_CHARWIDTH
  Either the MAPI_UNICODE flag was set and the implementation does not support Unicode, or MAPI_UNICODE was not set and the implementation only supports Unicode.

Remarks

Use the **IMAPITable::GetLastError** method to retrieve information to display in a message to the user regarding the last error returned from a method call for the table object.

To release all the memory allocated by MAPI for the **MAPIERROR** structure, implementations need only call the **MAPIFreeBuffer** function.

The return value from **GetLastError** must be S_OK for an implementation to make use of the **MAPIERROR** structure. Even if the return value is S_OK, a **MAPIERROR** structure might not be returned. If the implementation cannot determine what the last error was, or if a **MAPIERROR** structure is not available for that error, **GetLastError** returns a pointer to NULL in *lppMAPIError* instead.

See Also

MAPIERROR structure, **MAPIFreeBuffer** function

IMAPITable::GetRowCount

The **IMAPITable::GetRowCount** method returns the total number of rows in a table view.

HRESULT GetRowCount(
  ULONG *ulFlags*,
  ULONG FAR * *lpulCount*
)

Parameters

ulFlags
  Reserved; must be zero.

lpulCount
  Output parameter pointing to a variable where the returned total number of rows in the table view is stored.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

MAPI_E_BUSY
  Another operation is in progress; it should be allowed to complete, or it should be stopped, before this operation is attempted.

MAPI_E_NO_SUPPORT
  The operation is not supported by one or more service providers.

MAPI_W_APPROX_COUNT
  The call succeeded, but an approximate row count was returned because the exact row count could not be determined. To test for this warning, use the **HR_FAILED** macro. When this warning is returned, the call should be handled as successful.

Remarks

Use the **IMAPITable::GetRowCount** method to find out how many rows a table view holds before making a call to the **IMAPITable::QueryRows** method to return rows of data from that table. If the row count returned in the *lpulCount* parameter is less than 20, **QueryRows** can be called for the whole table. If the row count is greater than 20, the calling implementation might limit the rows **QueryRows** returns.

Some providers do not support **GetRowCount** and return MAPI_E_NO_SUPPORT. If **GetRowCount** is not supported, the implementation should call the **IMAPITable::QueryPosition** method to determine the current cursor position in the table. **QueryPosition** retrieves the fractional position of the cursor within the table.

If **GetRowCount** cannot determine a table view's exact row count – that is, if the value returned in *lpulCount* is approximate and not exact – it returns MAPI_W_APPROX_COUNT. This result can occur for implementations where an exact count expends too much memory to perform, for example for implementations that use large tables that are sparsely populated.

When **GetRowCount** is temporarily unable to return the number of rows in a table, it returns MAPI_E_BUSY. This result can occur because of asynchronous operations in progress. If the calling client or provider deduces from a return value of MAPI_E_BUSY that asynchronous operations are in progress, it can call the **IMAPITable::WaitForCompletion** method, then once the asynchronous operations are complete retry the call to **GetRowCount**. Another way to detect whether asynchronous operations are in progress is to call the **IMAPITable::GetStatus** method, which returns the status and type of a table.

For more information on using the **HR_FAILED** macro, see Using Macros for Error Handling.

See Also

IMAPITable::GetStatus method, **IMAPITable::QueryPosition** method, **IMAPITable::QueryRows** method, **IMAPITable::WaitForCompletion** method

IMAPITable::GetStatus

The **IMAPITable::GetStatus** method returns the status and type of a table.

HRESULT GetStatus(
  ULONG FAR * *lpulTableStatus*,
  ULONG FAR * *lpulTableType*
)

Parameters

lpulTableStatus
: Output parameter pointing to a variable where the status of the table is stored. One of the following values can be returned:

  TBLSTAT_COMPLETE
  : No operations are in progress.

  TBLSTAT_QCHANGED
  : The contents of the table have changed. This status is not returned for changes that result from sort or restriction operations but rather indicates unexpected changes.

  TBLSTAT_RESTRICT_ERROR
  : An error occurred during restriction.

  TBLSTAT_RESTRICTING
  : A restriction operation is in progress.

  TBLSTAT_SETCOL_ERROR
  : An error occurred while columns were being set.

  TBLSTAT_SETTING_COLS
  : A column-setting operation is in progress.

  TBLSTAT_SORT_ERROR
  : An error occurred during sorting.

  TBLSTAT_SORTING
  : A sort operation is in progress.

lpulTableType
: Output parameter pointing to a variable where the type of the table is stored. One of the following three table types can be returned:

  TBLTYPE_DYNAMIC
  : The table's contents are dynamic and can change as the underlying data changes.

  TBLTYPE_KEYSET
  : The rows within the table are fixed, but the values within these rows are dynamic and can change as the underlying data changes.

  TBLTYPE_SNAPSHOT
  : The table is static, and its contents do not change when the underlying data changes.

Return Values

S_OK
: The call succeeded and has returned the expected value or values.

Remarks

Use the **IMAPTable::GetStatus** method to gather information on the type and current status of a table. **GetStatus** can be used in conjunction with the **IMAPITable::Restrict** method to poll the status of updates to the table occurring as the result of a new restriction. **GetStatus** can also be used with the **IMAPITable::SortTable** method to monitor the status of a sort operation and with the **IMAPITable::SetColumns** method to see whether a column-setting operation has completed.

See Also

IMAPITable::Restrict method, **IMAPITable::SetColumns** method, **IMAPITable::SortTable** method

IMAPITable::QueryColumns

The **IMAPITable::QueryColumns** method returns either the current list of columns for a table view or the full list of columns available for the table.

HRESULT QueryColumns(
  ULONG *ulFlags***,**
  LPSPropTagArray FAR * *lpPropTagArray*
)

Parameters

ulFlags
: Input parameter containing a bitmask of flags that controls what table column information is returned. The following flag can be set:

  TBL_ALL_COLUMNS
  : Returns all available columns.

lpPropTagArray
: Output parameter pointing to a variable where the returned **SPropTagArray** structure is stored. This structure contains a counted array of property tags in which each tag identifies a particular table column.

Return Values

S_OK
: The call succeeded and has returned the expected value or values.

MAPI_E_BUSY
: Another operation is in progress; it should be allowed to complete, or it should be stopped, before this operation is attempted.

Remarks

Use the **IMAPITable::QueryColumns** method to get any one of the following:

- The current list of table columns for a table view as set when the table was created.
- The current list of columns for a table view as set by a call to the **IMAPITable::SetColumns** method.
- The full list of columns available for the table.

To retrieve the entire set of columns available for a table, the calling implementation should set the TBL_ALL_COLUMNS flag in the *ulFlags* parameter. Otherwise, **QueryColumns** returns the columns in the provider's current table columns, typically those columns that are cached. To retrieve additional columns, a client or provider calls the **IMAPITable::SetColumns** method.

To free the memory holding the structure returned in the *lpPropTagArray* parameter, an implementation uses the **MAPIFreeBuffer** function.

See Also

IMAPITable::SetColumns method, **MAPIFreeBuffer** function, **SPropTagArray** structure

IMAPITable::QueryPosition

The **IMAPITable::QueryPosition** method retrieves the current table row position of the cursor, based on a fractional value.

HRESULT QueryPosition(
  ULONG FAR * *lpulRow***,**
  ULONG FAR * *lpulNumerator***,**
  ULONG FAR * *lpulDenominator*
)

Parameters

lpulRow
  Output parameter pointing to the variable where the current row number is stored. The row number is zero-based, with the first row in the table being zero.

lpulNumerator
  Output parameter pointing to the variable where the numerator of the fraction representing the table position is stored.

lpulDenominator
  Output parameter pointing to the variable where the denominator of the fraction representing the table position is stored. The *lpulDenominator* parameter cannot be zero.

Return Values

S_OK
  The method returned valid values in *lpulRow*, *lpulNumerator*, and *lpulDenominator*.

Remarks

Use the **IMAPITable::QueryPosition** method to determine the current row based on a fractional value that approximates the position of the scroll box in the scroll bar compared to the number of rows in the table. For example, in a table containing 100 rows, if the scroll box indicates a cursor position 3/4 into the table, **QueryPosition** returns a value of 75 in the *lpulNumerator* parameter, 100 in the *lpulDenominator* parameter, and 75/100 in the *lpulRow* parameter. The value in *lpulDenominator* is not guaranteed to be the number of rows in the table, and **QueryPosition** cannot identify the exact row that the cursor is positioned in. MAPI defines the current row as the next row to be read.

Note that calculation of the values for the **QueryPosition** parameters can expend large amounts of memory in cases where the implementation must provide a useful cursor position value for a large categorized table. If **QueryPosition** cannot determine the current row, it returns a value of 0xFFFFFFFF in *lpulRow*; this result can occur for implementations where the **QueryPosition** calculation requires too much memory to perform. Clients and providers that receive such a return should call the **IMAPITable::SeekRowApprox** method to retrieve an approximate fractional value for the cursor position.

Calling **SeekRowApprox** with the same fraction as returned by **QueryPosition** does not necessarily reposition the cursor to the same row.

See Also

IMAPITable::SeekRowApprox method

IMAPITable::QueryRows

The **IMAPITable::QueryRows** method returns one or more rows from a table, beginning at the current cursor position.

HRESULT QueryRows(
  LONG *lRowCount*,
  ULONG *ulFlags*,
  LPSRowSet FAR * *lppRows*
)

Parameters

lRowCount
  Input parameter containing the number of rows requested.

ulFlags
  Input parameter containing a bitmask of flags that controls how rows are returned. The following flag can be set:

  TBL_NOADVANCE
    Prevents the cursor from advancing, so that the position value HRESULT is returned, indicating the current cursor position.

lppRows
  Output parameter pointing to a variable where the pointer to the returned **SRowSet** structure is stored. The **SRowSet** holds the set of table rows returned.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

MAPI_E_BUSY
  Another operation is in progress; it should be allowed to complete, or it should be stopped, before this operation is attempted.

Remarks

Use the **IMAPITable::QueryRows** method to get rows of data from a table.

If the value in the *lRowCount* parameter is positive, rows are read starting at the current position and reading forward. If the value in *lRowCount* is negative, the cursor position moves backward the indicated number of rows, and then rows are read in forward order.

The **cRows** member in the **SRowSet** structure returned in the *lppRows* parameter indicates the number of rows returned. If zero rows are returned, the cursor was already positioned at the beginning of the table and the value of *lRowCount* is negative or the cursor was already positioned at the end of the table and the value of *lRowCount* is positive. Fewer rows might be returned than are requested if memory or implementation limits are reached and in situations when **QueryRows** reaches the beginning or end of the table before returning all requested rows. Service providers cannot return zero rows unless the current position is at the beginning or end of the table. If fewer rows than requested are returned, the implementation acts as if a smaller value was passed in the *lRowCount* parameter.

The **IMAPITable::SetColumns** method supports the insertion of PR_NULL placeholders in the column set to reserve empty property-value slots in the **SPropValue** arrays within the *lppRows* **SRowSet**. This functionality enables calling implementations that must later add properties to the **SRowSet** to avoid having to copy a new **SPropValue** to the **SRowSet** before adding a new property.

Upon completion of a **QueryRows** call, the table cursor is positioned by default at the row following the last row returned. However, if the TBL_NOADVANCE flag is set in the *ulFlags* parameter, the table cursor is positioned at the first of the returned rows.

The columns returned in each row contain properties as previously specified by a call to the **IMAPITable::SetColumns** method. If no **SetColumns** call has been made, the columns returned reflect the default column set for that particular type of table. The number of properties and their ordering is the same for each table row. If a property does not exist in a row, the property value column returned holds a property type of PT_ERROR and a property value of MAPI_E_NOT_FOUND. Other errors in reading individual properties are indicated in a similar manner following the model used by the **IMAPIProp::GetProps** method, except that **IMAPITable::QueryRows** does not return warnings in its HRESULT. Specifically, **IMAPITable::Query Rows** does not return MAPI_W_ERRORS_RETURNED.

If the calling implementation requests zero rows, **QueryRows** returns MAPI_E_INVALID_PARAMETER. If an asynchronous operation is in progress, a call to **QueryRows** can return MAPI_E_BUSY. If an implementation receives MAPI_E_BUSY in such a situation, it can call the **IMAPITable::WaitForCompletion** method, then once the asynchronous operation is complete retry the call to **QueryRows**.

For most complex MAPI structures, implementations use the **MAPIFreeBuffer** function to free the structure by freeing the top level pointer. The **SRowSet** structure returned by **QueryRows** is an exception to this general behavior. Memory used for the properties held in the **SRow** structures that make up the **SRowSet** in the *lppRows* parameter and for the **rgPropVals** members of the **ADRENTRY** structures is separately allocated for each row, and memory for each row must be separately freed by a call to **MAPIFreeBuffer**. The **SRowSet** structure itself must also be freed. Thus, to free all the memory returned for 10 rows requires 11 calls to **MAPIFreeBuffer**. This process might seem complex, but freeing memory for each row separately enables implementations to free different rows at different times. When a call to **QueryRows** returns zero, however, indicating the beginning or end of the table, only the **SRowSet** structure itself needs to be freed.

See Also

ADRENTRY structure, **FreeProws** function, **HrQueryAllRows** function, **IMAPIProp::GetProps** method, **IMAPITable::SetColumns** method, **IMAPITable::WaitForCompletion** method, **MAPIFreeBuffer** function, **SRow** structure, **SRowSet** structure

IMAPITable::QuerySortOrder

The **IMAPITable::QuerySortOrder** method retrieves the current sort order for a table.

HRESULT QuerySortOrder(
  LPSSortOrderSet FAR * *lppSortCriteria*
)

Parameters

lppSortCriteria
 Output parameter pointing to a variable where the pointer to the returned **SSortOrderSet** structure holding the current sort order is stored.

Return Values

S_OK
 The call succeeded and has returned the expected value or values.
MAPI_E_BUSY
 Another operation is in progress; it should be allowed to complete, or it should be stopped, before this operation is attempted.

Remarks

Use the **IMAPITable::QuerySortOrder** method to retrieve the current sort order for a table. The sort order is returned in an **SSortOrderSet** structure.

QuerySortOrder returns an **SSortOrderSet** holding zero columns when:

- The table is unsorted.
- The provider does not have information on how the table is sorted.
- The provider cannot express the sort order within the capabilities of an **SSortOrderSet** structure.

In cases where an implementation calls **IMAPITable::SortTable** with an **SSortOrderSet** containing zero columns, the sort order is removed and the default sort order is applied. Subsequent calls to **QuerySortOrder** can return zero or more columns as being sorted, depending on the implementation of the service provider, and can return more columns than are in the present view.

To free the returned **SSortOrderSet** structure, an implementation uses the **MAPIFreeBuffer** function.

See Also

IMAPITable::SortTable method, **MAPIFreeBuffer** function, **SSortOrderSet** structure

IMAPITable::Restrict

The **IMAPITable::Restrict** method applies a restriction to a table, reducing the rows visible to only those matching the restriction criteria.

HRESULT Restrict(
  LPSRestriction *lpRestriction*,
  ULONG *ulFlags*
)

Parameters

lpRestriction
  Input parameter pointing to an **SRestriction** structure defining the conditions of the restriction. Passing NULL in the *lpRestriction* parameter removes the current restriction criteria.

ulFlags
  Input parameter containing a bitmask of flags that controls how the call returns when asynchronous operations are in progress. The following flags can be set:

  TBL_ASYNC
    Starts the operation asynchronously and returns before the operation completes.

  TBL_BATCH
    Defers evaluation until the results of the operation are required.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

MAPI_E_BUSY
  Another operation is in progress; it should be allowed to complete, or it should be stopped, before this operation is attempted.

MAPI_E_TOO_COMPLEX
  Although the calling client or provider has passed valid parameters, the specified sort operation, typically a subrestriction, is too complex for the implementation and could not be performed.

Remarks

Use the **IMAPITable::Restrict** method to set a restriction on a table view. MAPI discards the previous restriction, if any. A restriction should be thought of as a filter; it does not remove the underlying data. Rather, it simply suppresses some data so only certain rows are seen. To discard the current restriction without creating a new one, pass NULL in *lpRestriction*.

A restriction on a multivalued property works just as does a restriction on a single-valued property. To reference columns of multivalued properties on which a restriction is to operate, the restriction must have the MVI_FLAG flag set. This means that the following two members of an **SPropertyRestriction** structure can have different values:

```
SPropertyRestriction.ulPropTag
SPropertyRestriction.lpProp->ulPropTag
```

Restrictions on a column of multivalued properties without the MVI_FLAG being set treat the column's values as a totally ordered tuple. A comparison of two multivalued columns compares the column elements in order, reporting the relation of the columns at the first inequality, and returns equality only if the columns compared contain the same values in the same order. If one column has fewer values than the other, the reported relation is that of a null value to the other value.

A call to **Restrict** completes its operation before returning, unless different treatment of asynchronous operations is indicated by the flag set in the *ulFlags* parameter. If the TBL_BATCH flag is set in *ulFlags*, the implementation can defer the restriction operation until it requires the results. If the TBL_ASYNC flag is set in *ulFlags*, the **Restrict** operation can begin asynchronously and return before the operation completes.

Asynchronous calls in progress when a **Restrict** call is required can be stopped by calling the **IMAPITable::Abort** method. Most implementations of **Restrict** return MAPI_E_BUSY if a call is made to start an asynchronous restriction operation while a previous asynchronous call is still running.

All bookmarks for a table are discarded when a call to **Restrict** is made, and the BOOKMARK_CURRENT bookmark, indicating the current cursor position, is set to the beginning of the table.

The result of a property value restriction is undefined when the property does not exist. When a client requires well-defined behavior for such a restriction and is not sure whether the property exists – for example, it is not a required column of a table – it should combine the property restriction with an **SExistRestriction** in an **SAndRestriction**.

Service providers must not generate notifications for table rows that are hidden from view by calls to **Restrict**.

See Also

IMAPITable::Abort method, **IMAPITable::FindRow** method, **IMAPITable::GetRowCount** method, **IMAPITable::QueryRows** method, **SPropertyRestriction** structure

IMAPITable::SeekRow

The **IMAPITable::SeekRow** method moves the cursor to a specific position in a table.

HRESULT SeekRow(
  BOOKMARK *bkOrigin*,
  LONG *lRowCount*,
  LONG FAR * *lplRowsSought*
)

Parameters

bkOrigin
  Input parameter indicating the bookmark from which the operation searching for the table position starts. A bookmark can be created using the **IMAPITable::CreateBookmark** method, or one of the following predefined values can be passed:

  BOOKMARK_BEGINNING
    Searches from the beginning of the table.

  BOOKMARK_CURRENT
    Searches from the row in the table where the cursor is located.

  BOOKMARK_END
    Searches from the end of the table.

lRowCount
  Input parameter indicating the signed number of rows to move, starting from the bookmark in the *bkOrigin* parameter.

lplRowsSought
  Output parameter pointing to a variable where the returned number of rows actually searched through is stored. Passing NULL in the *lplRowsSought* parameter indicates that a number of rows searched need not be returned.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

MAPI_E_BUSY
  Another operation is in progress; it should be allowed to complete, or it should be stopped, before this operation is attempted.

MAPI_E_INVALID_BOOKMARK
  The bookmark is invalid because it has been removed or because it is beyond the last row requested.

MAPI_W_POSITION_CHANGED
  The call succeeded, but the bookmark used in the operation is no longer set at the same row as when it was last used; if the bookmark has not been used, it is no longer in the same position as when it was created. To test for this warning, use the **HR_FAILED** macro. When this warning is returned, the call should be handled as successful.

Remarks

Use the **IMAPITable::SeekRow** method to establish a new BOOKMARK_CURRENT position for the cursor. The *lRowCount* parameter indicates the number of rows the cursor moves. The number of rows in *lRowCount* should be less than 50; to search through a larger number of rows, use the **IMAPITable::SeekRowApprox** method.

If the resulting position is beyond the last row of the table, the cursor is positioned after the last row; if the resulting position is before the first row of the table, the cursor is placed at the beginning of the first row. To indicate a backward move for **SeekRow**, pass a negative value in *lRowCount*. To search to the beginning of the table, pass zero in *lRowCount* and the value BOOKMARK_BEGINNING in *bkOrigin*. If there are large numbers of rows in the table, the **SeekRow** operation can be slow.

SeekRow returns the number of rows actually searched through, positive or negative, in the variable pointed to by *lplRowsSought*. In normal operation, it should return the same value for *lplRowsSought* as passed in for *lRowCount*, unless the search reached the beginning or end of the table.

The operation of **SeekRow** can be slower than otherwise if the calling implementation requires a number of rows to be returned in *lplRowsSought*. If the calling implementation does not require a return count, it should pass NULL for *lplRowsSought*.

If the row pointed to by *bkOrigin* no longer exists in the table and the provider cannot establish a new position for the bookmark, **SeekRow** returns MAPI_E_INVALID_BOOKMARK. If the row pointed to by *bkOrigin* no longer exists and the provider is able to establish a new position for the bookmark, **SeekRow** returns MAPI_W_POSITION_CHANGED.

A bookmark pointing to a row that is collapsed out of the table view can still be used. If an implementation attempts to move the cursor to such a bookmark, the cursor moves to the next visible row and stops there. A call using a bookmark pointing to a collapsed row returns MAPI_W_POSITION_CHANGED. Providers can move bookmarks for positions collapsed out of view either at the time of use or at the time the row is collapsed. If a bookmark is moved at the time the row is collapsed, a bit must be retained in the bookmark that indicates whether the bookmark has moved since its last use or, if it has never been used, since its creation.

For more information on using the **HR_FAILED** macro, see Using Macros for Error Handling.

See Also

IMAPITable::CreateBookmark method, **IMAPITable::FindRow** method, **IMAPITable::QueryRows** method, **IMAPITable::SeekRowApprox** method

IMAPITable::SeekRowApprox

The **IMAPITable::SeekRowApprox** method moves the cursor to an approximate fractional position in a table.

HRESULT SeekRowApprox(
 ULONG *ulNumerator***,**
 ULONG *ulDenominator*
)

Parameters

ulNumerator
: Input parameter pointing to the variable containing the numerator of the fraction representing the table position. If the *ulNumerator* parameter is zero, it points to the beginning of the table regardless of the denominator value. If *ulNumerator* is equal to the *ulDenominator* parameter, the cursor is positioned after the last table row.

ulDenominator
: Input parameter pointing to the variable containing the denominator of the fraction representing the table position. The *ulDenominator* parameter cannot be zero.

Return Values

S_OK
: The call succeeded and has returned the expected value or values.

MAPI_E_BUSY
: Another operation is in progress; it should be allowed to complete, or it should be stopped, before this operation is attempted.

Remarks

Use the **IMAPITable::SeekRowApprox** method to provide the data used to support scroll bar implementations. For example, if the user positions the scroll box 2/3 down the scroll bar, the calling implementation can model that action by calling **SeekRowApprox** and passing in an equivalent fractional value using *ulNumerator* and *ulDenominator*. The **SeekRowApprox** search is always absolute from the beginning of the table. To move to the end of the table, the values in *ulNumerator* and *ulDenominator* must be the same. A **SeekRowApprox** implementation can use any numbering scheme, based on what is convenient: 9/10, 90/100, or 900/1000.

The cursor position in a table after a call to **SeekRowApprox** is heuristically the fraction and might not be exact. For example, certain providers might implement a table on top of a binary tree, treating the table's halfway point as the top of the tree for performance reasons. If the tree is not balanced, then the halfway point used might not be exactly halfway through the table.

IMAPITable::SetCollapseState

The **IMAPITable::SetCollapseState** method reestablishes the expanded or collapsed state of the table view that was saved by a call to the **IMAPITable::GetCollapseState** method.

HRESULT SetCollapseState(
  ULONG *ulFlags*,
  ULONG *cbCollapseState*,
  LPBYTE *pbCollapseState*,
  BOOKMARK FAR * *lpbkLocation*
)

Parameters

ulFlags
  Reserved; must be zero.

cbCollapseState
  Input parameter containing the size, in bytes, of the structure pointed to by the *pbCollapseState* parameter.

pbCollapseState
  Input parameter pointing to the structures containing the saved table view.

lpbkLocation
  Output parameter pointing to a bookmark identifying the row location within the table at which the indicated table state should be rebuilt. This bookmark identifies the same row as pointed to by the instance key in the *lpbInstanceKey* parameter on the call to **IMAPITable::GetCollapseState**.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

MAPI_E_BUSY
  Another operation is in progress; it should be allowed to complete, or it should be stopped, before this operation is attempted.

MAPI_E_UNABLE_TO_COMPLETE
  The requested operation could not be completed.

Remarks

Use the **IMAPITable::SetCollapseState** method to reestablish the expanded or collapsed state of the table view that was saved by a call to **IMAPITable::GetCollapseState**. Use **SetCollapseState** and **GetCollapseState** together to present to the user, upon opening a table in your implementation, a table that in its expanded or collapsed state can be recognized as the table the user formerly viewed.

To restore an entire table state, **SetCollapseState** uses the structures pointed to in the *pbCollapseState* parameter, which hold a restriction or sort order that defines the table view.

To call **SetCollapseState**, an implementation must have previously used **GetCollapseState**. The sort order passed with **SetCollapseState** should be the same as was saved in the **GetCollapseState** call; if the client application fails to reset the columns, the results of the operation are unpredictable.

Providers must not generate notifications for table rows that are hidden from view by calls to **SetCollapseState**.

See Also

IMAPITable::CreateBookmark method, **IMAPITable::FreeBookmark** method, **IMAPITable::GetCollapseState** method

IMAPITable::SetColumns

The **IMAPITable::SetColumns** method sets the order of columns for table rows returned by the **IMAPITable::QueryRows** method.

HRESULT SetColumns(
  LPSPropTagArray *lpPropTagArray*,
  ULONG *ulFlags*
)

Parameters

lpPropTagArray
: Input parameter pointing to an **SPropTagArray** structure containing a counted array of property tags. Each property tag identifies a particular table column. Passing zero properties in the *lpPropTagArray* parameter results in the **IMAPITable::SetColumns** method returning MAPI_E_INVALID_PARAMETER.

ulFlags
: Input parameter containing a bitmask of flags that controls the return of an asynchronous call to **SetColumns**, for example when **SetColumns** is used in notification. The following flags can be set:

  TBL_ASYNC
  : Starts the operation asynchronously and returns before the operation completes.

  TBL_BATCH
  : Defers evaluation until the results of the operation are required.

Return Values

S_OK
: The call succeeded and has returned the expected value or values.

MAPI_E_BUSY
: Another operation is in progress; it should be allowed to complete, or it should be stopped, before this operation is attempted.

Remarks

Use the **IMAPITable::SetColumns** method to control the order of the property values returned for each table row by an **IMAPITable::QueryRows** call. Some providers allow a **SetColumns** call to order only table columns that are part of the available columns for a table view. Other providers allow a **SetColumns** call to order all table columns, including those containing properties not in the original column set.

SetColumns supports inclusion of one or more PR_NULL property types in the property tag array in *lpPropTagArray*. Such property tags are used by **QueryRows** to reserve empty slots in **SPropValue** arrays in the **SRowSet** structure that represents the table rows it returns. This functionality enables calling implementations that must later add properties to this **SRowSet** to avoid having to copy a new **SPropValue** array before adding a new property.

To provide multiple row instances for a multivalued property with **SetColumns**, an implementation applies the MVI_FLAG flag to a table column's property type. To do so, it specifies **MVI_PROP(ulPropTag)** for that column in the **SPropTagArray** structure in *lpPropTagArray* instead of a single-valued property tag. MVI_FLAG is only meaningful for multivalued properties; it sets both bits, including the MVI_INSTANCE flag. MAPI ignores MVI_FLAG if it is applied to a single-valued property column.

If a call to **SetColumns** changes the order of table columns that contain multivalued properties, the call can change the number of rows in the table. This situation is the only one in which a **SetColumns** call can change the number of table rows. If a **SetColumns** call changes the number of table rows, all bookmarks for the table are discarded. If a call to **SetColumns** adds or removes one or more columns, the call can also change a table's column set.

A call to **SetColumns** completes its operation before returning, unless different treatment of asynchronous operations is indicated by the flag set in the *ulFlags* parameter. If the TBL_BATCH flag is set in *ulFlags*, the implementation can defer setting table columns until it requires the results of the operation. If the TBL_ASYNC flag is set in *ulFlags*, the **SetColumns** operation can begin synchronously and returns before completing. When TBL_BATCH is set for asynchronous operations, providers should return a property type of PT_ERROR and a property value of NULL for columns that are not supported.

When a **SetColumns** call is required, asynchronous calls in progress can be stopped by using the **IMAPITable::Abort** method. Most applications of **SetColumns** return MAPI_E_BUSY if a call is made to start an asynchronous operation while a previous asynchronous call is still running.

Note that service providers must not generate notifications for table rows that are hidden from view by calls to **Restrict**. When sending table notifications, a provider must order the properties in its table column set the same as in the column set that existed when the notification request was sent.

See Also

HrQueryAllRows function, **IMAPITable::Abort** method, **IMAPITable::GetRowCount** method, **IMAPITable::QueryColumns** method, **IMAPITable::QueryRows** method, **IMAPITable::Restrict** method, **IMAPITable::SortTable** method, **SPropTagArray** structure, **SPropValue** structure, **SRowSet** structure, **TABLE_NOTIFICATION** structure

IMAPITable::SortTable

The **IMAPITable::SortTable** method sorts table rows based on the sort criteria provided.

HRESULT SortTable(
  LPSSortOrderSet *lpSortCriteria*,
  ULONG *ulFlags*
)

Parameters

lpSortCriteria
  Input parameter pointing to an **SSortOrderSet** structure containing the sort criteria to apply. Passing an **SSortOrderSet** containing zero columns indicates the calling client application or service provider doesn't require the table be sorted or doesn't require any specific sort order.

ulFlags
  Input parameter containing a bitmask of flags that controls the return of an asynchronous call to the **IMAPITable::SortTable** method. The following flags can be set:

  TBL_ASYNC
    Starts the operation asynchronously and returns before the operation completes.

  TBL_BATCH
    Defers evaluation until the results of the operation are required.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

MAPI_E_BUSY
  Another operation is in progress; it should be allowed to complete, or it should be stopped, before this operation is attempted.

MAPI_E_NO_SUPPORT
  The provider either does not support changes to its objects or does not support notification of changes.

MAPI_E_TOO_COMPLEX
  Although the calling client or provider has passed valid parameters, the specified sort operation, typically a subrestriction, is too complex for the implementation and could not be performed.

Remarks

Use the **IMAPITable::SortTable** method to reorder the rows in a table view.

MAPI does not require providers to be able to sort tables. To indicate table sorting is unavailable, providers should return MAPI_E_NO_SUPPORT for a **SortTable** call.

Address book providers commonly do not support table sorting. Message store providers commonly support sorting to the extent that they retain the sort order of folders that results when a full table − that is, a table with no restrictions − is insorted.

Some implementations allow sorting to be done on any table column. Some require that sorting only affect the current list of columns for a table view; in such an implementation, columns not set in the table view are not affected by a **SortOrder** call. Some implementations require that only active columns be sorted. The active columns for a table are those columns a call to the **IMAPITable::QueryColumns** method returns.

If an implementation calls **SortTable** with zero columns in the **SSortOrderSet** structure in the *lpSortCriteria* parameter, it indicates it doesn't require the table affected to be sorted or doesn't require information on the table's sort order. The set of currently active columns is returned in this case. When an implementation passes zero columns in *lpSortCriteria* for a particular table, it can still call **IMAPITable::QuerySortOrder** to get the current sort order for that table.

A sort operation performed on a column of multivalued properties without the flag MVI_FLAG being set treats the column's values as a totally ordered tuple. A comparison of two multivalued columns compares the column elements in order, reporting the relation of the columns at the first inequality, and returns equality only if the columns compared contain the same values in the same order. If one column has fewer values than the other, the reported relation is that of a null value to the other value.

A call to **SortTable** completes its operation before returning, unless different treatment of asynchronous operations is indicated by the flag set in the *ulFlags* parameter. If the TBL_BATCH flag is set in *ulFlags*, the implementation can defer the sort operation until it requires the results. If the TBL_ASYNC flag is set in *ulFlags*, the **SortTable** operation can begin synchronously and return before the operation completes.

Asynchronous calls in progress when a **SortTable** call is required can be stopped by using the **IMAPITable::Abort** method. Most implementations of **SortTable** return MAPI_E_BUSY if a call is made to start an asynchronous sort operation while a previous asynchronous call is still running.

All bookmarks for a table are invalidated and should be deleted when a call to **SortTable** is made, and the BOOKMARK_CURRENT bookmark, indicating the current cursor position, should be set to the beginning of the table.

SortTable can return MAPI_E_TOO_COMPLEX under any of the following conditions:

- A sort operation is requested for a property column that the implementation cannot sort.
- The implementation does not support the sort order requested in the **ulOrder** member of the **SSortOrderSet** structure.
- The number of columns to be sorted, as specified in the **cSorts** member in **SSortOrderSet**, is larger than the implementation can handle.
- A sort operation is requested, as indicated by a property tag in **SSortOrderSet**, based on a property that is not in the available or active set and the implementation does not support sorting on properties not in the available set.
- One property is specified multiple times in a sort order set, as indicated by multiple instances of the same property tag, and the implementation cannot perform such a sort operation.
- A sort operation based on multivalued property columns is requested using MVI_FLAG and the implementation does not support sorting on multivalued properties.
- A property tag for a property in **SSortOrderSet** specifies a property or type that the implementation does not support.
- A sort operation other than one that proceeds through the table from the PR_RENDERING_POSITION property forward is specified only for an attachment table that supports this type of sorting.

For best performance, implementations of **SortTable** should establish a table column set with **SetColumns** and then any restrictions with **Restrict** before sorting a table.

See Also

IMAPITable::Abort method, **IMAPITable::GetRowCount** method, **IMAPITable::QueryColumns** method, **IMAPITable::QuerySortOrder** method, **IMAPITable::SetColumns** method, **SSortOrderSet** structure

IMAPITable::Unadvise

The **IMAPITable::Unadvise** method removes a table's registration for notification of changes previously established with a call to the **IMAPITable::Advise** method.

HRESULT Unadvise(
  ULONG *ulConnection*
)

Parameters

ulConnection
  Input parameter containing the number of the registration connection returned by a call to **IMAPITable::Advise**.

Return Value

S_OK
  The call succeeded and has returned the expected value or values.

Remarks

Use the **IMAPITable::Unadvise** method to release the pointer to the advise sink object passed in the *lpAdviseSink* parameter in the previous call to **IMAPITable::Advise**, thereby canceling a notification registration. As part of discarding the pointer to the advise sink object, the object's **IUnknown::Release** method is called. Generally, **Release** is called during the **Unadvise** call, but if another thread is in the process of calling the **IMAPIAdviseSink::OnNotify** method for the advise sink, the **Release** call is delayed until the **OnNotify** method returns.

See Also

IMAPIAdviseSink::OnNotify method, **IMAPITable::Advise** method

IMAPITable::WaitForCompletion

The **IMAPITable::WaitForCompletion** method suspends the calling implementation while asynchronous operations occur on a table.

HRESULT WaitForCompletion(
  ULONG *ulFlags***,**
  ULONG *ulTimeout***,**
  ULONG FAR * *lpulTableStatus*
)

Parameters

ulFlags
  Reserved; must be zero.

ulTimeout
  Input parameter indicating the maximum number of milliseconds to wait for asynchronous operations to complete. If the operations do not complete in the time specified, the **IMAPITable::WaitForCompletion** method should return MAPI_E_TIMEOUT. If 0xFFFFFFFF is sent in the *ulTimeout* parameter, the calling implementation pauses until the operation completes, however long that takes.

lpulTableStatus
  Output parameter pointing to a variable where a returned value indicating the most recent status of the table in question is stored. If NULL is passed in the *lpulTableStatus* parameter, no status information is returned. If **WaitForCompletion** returns any nonzero HRESULT, including MAPI_E_TIMEOUT, the contents of *lpulTableStatus* are undefined.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

MAPI_E_NO_SUPPORT
  The operation is not supported by MAPI or by one or more service providers.

MAPI_E_TIMEOUT
  The operation did not complete in the specified time.

Remarks

Use the **IMAPITable::WaitForCompletion** method to suspend processing untilany asynchronous operations currently under way for a table have completed. **WaitForCompletion** can allow the asynchronous operations either to complete or to run for a certain number of milliseconds, as indicated by *ulTimeout*, before being interrupted. To detect asynchronous operations in progress, use the **IMAPITable::GetStatus** method.

See Also

IMAPITable::GetRowCount method, **IMAPITable::GetStatus** method, **IMAPITable::Restrict** method, **IMAPITable::SetColumns** method, **IMAPITable::SortTable** method

IMAPIViewAdviseSink : IUnknown

The **IMAPIViewAdviseSink** interface is implemented by form viewers. Its methods are called by a form to notify a viewer that some event has occurred in the form.

At a Glance

Specified in header file:	MAPIFORM.H
Object that supplies this interface:	View advise sink object
Corresponding pointer type:	LPMAPIVIEWADVISESINK
Implemented by:	Form viewers
Called by:	Forms

Vtable Order

OnShutdown	Notifies a form viewer that a form is being closed.
OnNewMessage	Notifies a form viewer that either a new or an existing message has been loaded in a form.
OnPrint	Notifies a form viewer of the printing status of a form.
OnSubmitted	Notifies a form viewer that the current message has been submitted to the MAPI spooler.
OnSaved	Notifies a form viewer that the current message in a form has been saved.

IMAPIViewAdviseSink::OnShutdown

The **IMAPIViewAdviseSink::OnShutdown** method notifies a form viewer that a form is being closed.

HRESULT OnShutdown()

Parameters

None

Return Values

S_OK
: The call succeeded.

IMAPIViewAdviseSink::OnNewMessage

The **IMAPIViewAdviseSink::OnNewMessage** method notifies a form viewer that either a new or an existing message has been loaded in a form.

HRESULT OnNewMessage()

Parameters

None

Return Values

S_OK
    The call succeeded.

Remarks

Forms call the **IMAPIViewAdviseSink::OnNewMessage** method whenever a message is loaded in a form using either the **IPersistMessage::InitNew** or **IPersistMessage::Load** method. A viewer often releases any **IMAPIForm** interfaces it has open at this point because the existing form object no longer points to the message the viewer was formerly viewing.

See Also

IMAPIForm : IUnknown interface, **IPersistMessage::InitNew** method, **IPersistMessage::Load** method

IMAPIViewAdviseSink::OnPrint

The **IMAPIViewAdviseSink::OnPrint** method notifies a form viewer of the printing status of a form.

HRESULT OnPrint(
  ULONG *dwPageNumber*,
  HRESULT *hrStatus*
)

Parameters

dwPageNumber
  Input parameter holding the number of the last page printed.

hrStatus
  Input parameter holding an HRESULT variable whose value shows the status of the print job. The following values can be used to indicate status:

  S_FALSE
    The printing job has finished successfully.

  S_OK
    The printing job is in progress.

  FAILED
    The printing job was terminated due to a failure.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

MAPI_E_USER_CANCEL
  The user canceled the operation, typically by clicking the **Cancel** button in a dialog box.

Remarks

Forms call the **IMAPIViewAdviseSink::OnPrint** method while printing to inform the current view of printing progress. If the printing job involves multiple pages, **OnPrint** can be called after each page is printed with the page number for the last page printed in the *dwPageNumber* parameter and S_OK in the *hrStatus* parameter to indicate that the printing job is proceeding. When the printing job is complete, **OnPrint** should be called with the page number of the last page printed in *dwPageNumber* and S_FALSE in *hrStatus*.

IMAPIViewAdviseSink::OnSaved

The **IMAPIViewAdviseSink::OnSaved** method notifies a form viewer that the current message in a form has been saved.

HRESULT OnSaved()

Parameters

None

Return Value

S_OK
    The call succeeded and has returned the expected value or values.

Remarks

Forms call the **IMAPIViewAdviseSink::OnSaved** method after the current message in a form has been successfully saved. Its doing so permits viewers to update their windows to reflect changes to the message.

IMAPIViewAdviseSink::OnSubmitted

The **IMAPIViewAdviseSink::OnSubmitted** method notifies a form viewer that the current message has been submitted to the MAPI spooler.

HRESULT OnSubmitted()

Parameters

None

Return Value

S_OK
: The call succeeded and has returned the expected value or values.

Remarks

Forms call the **IMAPIViewAdviseSink::OnSubmitted** method after a call to the **IMAPIMessageSite::SubmitMessage** method has returned successfully. After **OnSubmitted** is called, form viewers can continue on the assumption the message has been updated and can update their windows.

See Also

IMAPIMessageSite::SubmitMessage method

IMAPIViewContext : IUnknown

The **IMAPIViewContext** interface is implemented by form viewers to support form commands that activate the next or previous message, or that print or save the current message.

At a Glance

Specified in header file:	MAPIFORM.H
Object that supplies this interface:	View context object
Corresponding pointer type:	LPMAPIVIEWCONTEXT
Implemented by:	Form viewers
Called by:	Forms

Vtable Order

SetAdviseSink	Registers a form object for notifications about changes to a view's status.
ActivateNext	Activates the next or previous message in a view.
GetPrintSetup	Retrieves the current print setup so as to print a message.
GetSaveStream	Retrieves a stream to place in it a version of the current message converted to text format.
GetViewStatus	Retrieves the current viewer status.

IMAPIViewContext::ActivateNext

The **IMAPIViewContext::ActivateNext** method activates the next or previous message in a view.

HRESULT ActivateNext(
  ULONG *ulDir*,
  LPCRECT *prcPosRect*
)

Parameters

ulDir
: Input parameter containing a status value indicating which message to activate. The value can be one of the following flags:

  VCDIR_DELETE
  : Activates the next message because the current message has been deleted.

  VCDIR_MOVE
  : Activates the next message because the current message has been moved.

  VCDIR_NEXT
  : Activates the next message in the view order.

  VCDIR_PREV
  : Activates the previous message in the view order.

prcPosRect
: Input parameter pointing to a **RECT** structure containing the size and position of the window used to display the message to activate.

Return Values

S_OK
: The call succeeded and has returned the expected value or values.

S_FALSE
: The call succeeded but a different type of form was opened; call the **IMAPIForm::ShutdownForm** method for your form object.

Remarks

Forms call the **IMAPIViewContext::ActivateNext** method to change what message is displayed to the user. The value passed in the *ulDir* parameter informs the form viewer why and how the current message status has been changed. The VCDIR_NEXT and VCDIR_PREVIOUS flags correspond to users choosing the Next and Previous command in a view, respectively. These operations usually correspond to moving up or down one message in the form viewer's list of messages.

The VCDIR_DELETE and VCDIR_MOVE flags are set by the **IMAPIMessageSite::DeleteMessage** and **IMAPIMessageSite::MoveMessage** methods, respectively. Implementations of these methods call **ActivateNext** with the appropriate direction and then perform the requested operation on the message, if the **ActivateNext** call did not fail. Form viewers typically enable users to specify the direction to move in the message list.

Upon return from **ActivateNext**, form objects must check for a current message and go through normal shutdown if a message is not present. If a next or previous message is displayed, the form uses the window rectangle passed in the *prcPosRect* parameter to display it.

See Also

IMAPIViewContext::GetViewStatus method

IMAPIViewContext::GetPrintSetup

The **IMAPIViewContext::GetPrintSetup** method retrieves the current print setup so as to print a message.

HRESULT GetPrintSetup(
  ULONG *ulFlags***,**
  LPFORMPRINTSETUP FAR * *lppFormPrintSetup*
)

Parameters

ulFlags
  Input parameter containing a bitmask of flags that controls the type of the returned strings. The following flag can be set:

  MAPI_UNICODE
    Indicates the returned strings are in Unicode format. If the MAPI_UNICODE flag is not set, the strings are in ANSI format.

lppFormPrintSetup
  Output parameter pointing to where the pointer to the returned **FORMPRINTSETUP** structure is stored. The **FORMPRINTSETUP** structure holds information on the print setup.

Return Value

S_OK
  The call succeeded and has returned the expected value or values.

Remarks

Forms call the **IMAPIViewContext::GetPrintSetup** method to retrieve the current print setup to print the current message. A form should pass the MAPI_UNICODE flag in the *ulFlags* parameter if Unicode strings are required for the **hDevMode** and **hDevName** members of the **FORMPRINTSETUP** structure returned in the *lppFormPrintSetup* parameter. Otherwise, **GetPrintSetup** returns the contents of these members as ANSI strings.

The returned **FORMPRINTSETUP** must be freed by the calling form using the **MAPIFreeBuffer** function. The **hDevMode** and **hDevNames** members must be allocated using the Win32 function **GlobalAlloc** and must be freed using the Win32 function **GlobalFree.**

See Also

FORMPRINTSETUP structure

IMAPIViewContext::GetSaveStream

The **IMAPIViewContext::GetSaveStream** method retrieves a stream in which a version of the current message converted to text format will be placed.

HRESULT GetSaveStream(
  ULONG FAR * *pulFlags*,
  ULONG FAR * *pulFormat*,
  LPSTREAM FAR * *ppstm*
)

Parameters

pulFlags
  Output parameter pointing to a bitmask of flags that controls the type of the saved text. The following flag can be set:

  MAPI_UNICODE
    Indicates the returned text is in Unicode format. If the MAPI_UNICODE flag is not set, the text is in ANSI format.

pulFormat
  Output parameter pointing to a bitmask of flags that controls additional formatting characteristics. The following flags can be set:

  SAVE_FORMAT_RICHTEXT
    The message is converted to Rich Text Format.

  SAVE_FORMAT_TEXT
    The message is converted to plain text format.

ppstm
  Output parameter pointing to a pointer where the converted version of the message is written.

Return Value

S_OK
  The call succeeded and has returned the expected value or values.

Remarks

Forms call the **IMAPIViewContext::GetSaveStream** method to support the Save As verb set for form viewers. A stream interface is returned to the form. Forms are not permitted to write any data before the **seek** pointer on entry, and they must leave the **seek** pointer at the end of the converted message when done. The message should be fully converted to text and placed into the stream before returning from the **IMAPIForm::DoVerb** call.

IMAPIViewContext::GetViewStatus

The **IMAPIViewContext::GetViewStatus** method retrieves the current viewer status.

```
HRESULT GetViewStatus(
  ULONG FAR * lpulStatus
 )
```

Parameter

lpulStatus
Output parameter pointing to a variable where a returned bitmask of flags giving information on view status is stored. The following flags can be set:

VCSTATUS_READONLY
Indicates the form is to be opened in read-only mode.

VCSTATUS_INTERACTIVE
Indicates the form should suppress displaying user interface even in response to a verb that usually causes user interface to be displayed.

VCSTATUS_MODAL
Indicates the form is modal to the viewer.

VCSTATUS_NEXT
Indicates there is a next form.

VCSTATUS_PREV
Indicates there is a previous form.

Return Value

S_OK
The call succeeded and has returned the expected value or values.

Remarks

Forms call the **IMAPIViewContext::GetViewStatus** method to determine whether there are more messages to be activated in a form view in either or both directions – that is, in the direction in which a Next command activates messages, in the direction in which a Previous command activates messages, or in both directions. The value in the *lpulStatus* parameter is used to determine whether the VCSTATUS_NEXT and VCSTATUS_PREVIOUS flags are valid for **IMAPIViewContext::ActivateNext**. If the VCSTATUS_DELETE flag is set, but not the VCSTATUS_READONLY flag, then the message can be deleted using the **IMAPIMessageSite::DeleteMessage** method.

Typically, forms disable menu commands and buttons if not valid for the context. The *lpulStatus* values are dynamic and can be changed if the view context calls the **IMAPIFormAdviseSink::OnChange** method.

The VCSTATUS_MODAL flag is set if the form must be modal to the window whose handle is passed in the earlier **IMAPIForm::DoVerb** call. If VCSTATUS_MODAL is set, the form can use the thread on which the **DoVerb** call was made until the form closes. If VCSTATUS_MODAL is not set, the form should not be modal to this window and must not use the thread.

See Also

IMAPIForm::DoVerb method, **IMAPIFormAdviseSink::OnChange** method, **IMAPIMessageSite::DeleteMessage** method, **IMAPIMessageSite::GetSiteStatus** method, **IMAPIViewContext::ActivateNext** method

IMAPIViewContext::SetAdviseSink

The **IMAPIViewContext::SetAdviseSink** method registers a form for notifications about changes to a view's status.

HRESULT SetAdviseSink(
  LPMAPIFORMADVISESINK *pmvns*
)

Parameters

pmvns
  Input parameter pointing to a form advise-sink object.

Return Value

S_OK
  The call succeeded.

Remarks

Forms call the **IMAPIViewContext::SetNotifySink** method to register for notification about changes to which message is next or previous within a particular view context. When called with NULL in the *pmvns* parameter, **SetNotifySink** cancels a previous registration for notification.

IMessage : IMAPIProp

The **IMessage** interface is used for managing messages, attachments, and recipients.

At a Glance

Specified in header file:	MAPIDEFS.H
Object that supplies this interface:	Message object
Corresponding pointer type:	LPMESSAGE
Implemented by:	Message store providers
Transaction model:	Transacted
Called by:	Client applications

Vtable Order

GetAttachmentTable	Returns the attachment table for a message.
OpenAttach	Opens an attachment.
CreateAttach	Creates a new attachment in a message.
DeleteAttach	Deletes an attachment from a message.
GetRecipientTable	Returns the recipient table for a message.
ModifyRecipients	Adds, deletes, or modifies recipients in a message.
SubmitMessage	Saves all changes to a message and marks the message as ready for sending.
SetReadFlag	Sets or clears the read flags for a message and manages the sending of read reports.

Required Message Properties

PR_CREATION_TIME	Read-only - new messages
PR_DISPLAY_BCC	Read-only - saved messages
PR_DISPLAY_CC	Read-only - saved messages
PR_DISPLAY_TO	Read-only - saved messages
PR_ENTRYID	Read-only - all messages
PR_LAST_MODIFICATION_TIME	Read-only - saved messages
PR_MESSAGE_ATTACHMENTS	Read-only - all messages
PR_MESSAGE_CLASS	Read/write - all messages
PR_MESSAGE_FLAGS	Read/write - all messages
PR_MESSAGE_RECIPIENTS	Read-only - all messages
PR_PARENT_DISPLAY	Read-only - all messages
PR_PARENT_ENTRYID	Read-only - all messages
PR_RECORD_KEY	Read-only - all messages
PR_SEARCH_KEY	Read-only - all messages
PR_STORE_ENTRYID	Read-only - all messages
PR_STORE_RECORD_KEY	Read-only - all messages

Required Message Report Properties

PR_BODY	Read/write
PR_MESSAGE_CLASS	Read/write
PR_MESSAGE_DELIVERY_TIME	Read-only
PR_ORIGINAL group	Read-only
PR_REPORT_TAG	Read-only
PR_REPORT_TEXT	Read-only
PR_REPORT_TIME	Read-only
PR_SEARCH_KEY	Read-only
PR_SENDER group	Read-only
PR_SUBJECT	Read/write

Required Message Recipient Properties

PR_ADDRTYPE	Read-only
PR_DISPLAY_NAME	Read/write
PR_DISPLAY_TYPE	Read/write
PR_ENTRYID	Read-only
PR_OBJECT_TYPE	Read-only

IMessage::CreateAttach

The **IMessage::CreateAttach** method creates a new attachment in a message.

HRESULT CreateAttach(
  LPCIID *lpInterface***,**
  ULONG *ulFlags***,**
  ULONG FAR * *lpulAttachmentNum***,**
  LPATTACH FAR * *lppAttach*
)

Parameters

lpInterface
  Input parameter pointing to the interface identifier (IID) for the returned attachment object. Passing NULL indicates that IID_IAttach is used. Client applications must pass NULL. Message store providers can also set the *lpInterface* parameter to IID_IUnknown, IID_IMAPIProp, or IID_IMessage.

ulFlags
  Input parameter containing a bitmask of flags that controls how the attachment is created. The following flag can be set:

  MAPI_DEFERRED_ERRORS
    Indicates the call is allowed to succeed even if the underlying object is not accessible to the calling client. If the object is not accessible, some subsequent call to the object might return an error.

lpulAttachmentNum
  Output parameter pointing to a variable where an index number identifying the newly created attachment is stored. This number is valid only within the message.

lppAttach
  Output parameter pointing to a variable where the pointer to the open attachment object is stored.

Return Value

S_OK
  The call succeeded and has returned the expected value or values.

Remarks

Message store providers implement the **IMessage::CreateAttach** method to create a new attachment within a message. An index number uniquely identifying the attachment within the message is returned, along with a pointer to the open attachment. The index number and the pointer are needed to access and refer to the attachment after it has been created.

IMessage::DeleteAttach

The **IMessage::DeleteAttach** method deletes an attachment from a message.

HRESULT DeleteAttach(
  ULONG *ulAttachmentNum*,
  ULONG *ulUIParam*,
  LPMAPIPROGRESS *lpProgress*,
  ULONG *ulFlags*
)

Parameters

ulAttachmentNum
  Input parameter containing the index number of the attachment to delete. This index number uniquely identifies the attachment within the message but is valid only within the message.

ulUIParam
  Input parameter containing the handle of the parent window for any dialog boxes or windows this method displays. The *ulUIParam* parameter is ignored unless the client application sets the ATTACH_DIALOG flag in the *ulFlags* parameter and passes NULL in the *lpProgress* parameter.

lpProgress
  Input parameter pointing to a progress object that contains client-supplied progress information. If NULL is passed in *lpProgress*, MAPI provides the progress information. The *lpProgress* parameter is ignored unless ATTACH_DIALOG is set in *ulFlags*.

ulFlags
  Input parameter containing a bitmask of flags that controls what happens when the attachment is deleted. The following flag can be set:

  ATTACH_DIALOG
    Displays a progress indicator as the operation proceeds.

Return Value

S_OK
  The call succeeded and has returned the expected value or values.

Remarks

Message store providers implement the **IMessage::DeleteAttach** method to delete an attachment from within a message. Before calling **DeleteAttach** to delete the attachment, a client should call the **IUnknown::Release** method on all pointers to the attachment and its streams. A deleted attachment is not permanently deleted until the **IMAPIProp::SaveChanges** method has been called for the message that held the attachment.

See Also

IMAPIProp::SaveChanges method

IMessage::GetAttachmentTable

The **IMessage::GetAttachmentTable** method returns the attachment table for a message.

HRESULT GetAttachmentTable(
  ULONG *ulFlags*,
  LPMAPITABLE FAR * *lppTable*
)

Parameters

ulFlags
Input parameter containing a bitmask of flags that controls the type of the text in the returned strings. The following flag can be set:

MAPI_UNICODE
Indicates the returned strings are in Unicode format. If the MAPI_UNICODE flag is not set, the strings are in ANSI format.

MAPI_DEFERRED_ERRORS
Indicates the call is allowed to succeed even if the underlying object is not accessible to the calling implementation. If the object is not accessible, some subsequent call to the object might return an error.

lppTable
Output parameter pointing to a variable where the pointer to the returned attachment table object is stored.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

Remarks

Message store providers implement the **IMessage::GetAttachmentTable** method to acquire a pointer to a message's attachment table, which lists all the attachments within the message. The attachment table for either a sent message or a message under composition contains one row for each attachment.

The attachment table must contain the following property columns:

PR_ATTACH_NUM
PR_RECORD_KEY
PR_RENDERING_POSITION

The table can contain additional property columns depending on a message store provider's implementation. Any potential for restrictions on the table is also determined by a store provider's implementation, so clients should not function as if restrictions are supported in all cases.

An attachment does not necessarily appear in the attachment table until the **IMAPIProp::SaveChanges** method is called on the message. The attachment table can change while open if the client calls the **IMessage::CreateAttach** or **IMessage:: DeleteAttach** method to create or delete an attachment, or if an attachment is modified so its properties in the attachment table change and **SaveChanges** is called for the message.

Attachment tables when initially opened are not necessarily sorted in any particular order.

Setting the MAPI_UNICODE flag in the *ulFlags* parameter does the following:

- Sets the string type to Unicode for data returned for the initial active columns of the attachment table by the **IMAPITable::QueryColumns** method. The initial active columns for an attachment table are those columns **QueryColumns** returns before the provider that contains the table calls the **IMAPITable::SetColumns** method.

- Sets the string type to Unicode for data returned for the initial active rows of the attachment table by the **IMAPITable::QueryRows** method. The initial active rows for an attachment table are those rows **QueryRows** returns before the provider that contains the table calls **SetColumns**.
- Controls the property types of the sort order returned by the **IMAPITable::QuerySortOrder** method before the provider that contains the attachment table calls the **IMAPITable::SortTable** method.

See Also

IMessage::CreateAttach method, **IMessage::DeleteAttach** method, **IMessage::OpenAttach** method

IMessage::GetRecipientTable

The **IMessage::GetRecipientTable** method returns the recipient table for a message.

HRESULT GetRecipientTable(
  ULONG *ulFlags*,
  LPMAPITABLE FAR * *lppTable*
)

Parameters

ulFlags
: Input parameter containing a bitmask of flags that controls the return of the table. The following flags can be set:

  MAPI_DEFERRED_ERRORS
  : Indicates the call is allowed to succeed even if the underlying object is not accessible to the calling client application. If the object is not accessible, some subsequent call to the object might return an error.

  MAPI_UNICODE
  : Indicates the returned strings are in Unicode format. If the MAPI_UNICODE flag is not set, the strings are in ANSI format.

lppTable
: Output parameter pointing to a variable where the pointer to the returned recipient table object is stored.

Return Value

S_OK
: The call succeeded and has returned the expected value or values.

Remarks

Message store providers implement the **IMessage::GetRecipientTable** method to get a list of the recipients for a message. Changes to the recipient table can be made by calling the **IMessage::ModifyRecipients** method.

The recipient table for either a received message or a message under composition contains one row for each recipient of the message.

Recipient tables have a different column set depending on whether the message has been submitted. The following properties make up the required column set in recipient tables:

PR_DISPLAY_NAME
PR_RECIPIENT_TYPE
PR_ROWID

The optional properties are as follows:

PR_DISPLAY_TYPE
PR_ENTRYID
PR_SPOOLER_STATUS
PR_OBJECT_TYPE

Submitted message should include these additional properties in their required column set:

PR_ADDRTYPE
PR_RESPONSIBILITY

Depending on a provider's implementation, additional columns, such as PR_SENDER_NAME and ENTRYID, might be in the table.

Most messaging systems recognize only three recipient types: MAPI_TO, MAPI_CC, and MAPI_BCC. MAPI specifies constant values for these types and updates the PR_DISPLAY_TO, PR_DISPLAY_CC, and PR_DISPLAY_BCC properties of the message appropriately when the **IMAPIProp::SaveChanges** method is called.

MAPI and clients use an additional recipient type, MAPI_SUBMITTED, for sending delivery reports and resend messages to message store recipients that are created as a result of expansion or transmission of the original message recipients. MAPI_SUBMITTED recipients can appear in a delivery report indicating that a recipient name is resolved to a group included from an outside messaging system, or if one or more members of a group of recipients generates such a report. MAPI_SUBMITTED recipients can also appear in a message when an attempt is made to resend a message to one or more recipients named in a nondelivery report.

The initial active columns for a newly opened recipient table, that is those columns that the **IMAPITable::QueryRows** method returns if a client does not call the **IMAPITable::SetColumns** method, include all available columns. Whether restrictions can be applied to the table depends on a provider's implementation, so clients should not function as if restrictions are supported in all cases.

A client can change the recipient table while it is open by calling the **IMessage::ModifyRecipients** method. When modifying the recipient table, the client must call the **IMAPITable::QueryColumns**, **IMAPITable::SetColumns**, and **IMAPITable::QueryRows** methods for each column to appear in the table – not just the modified columns – because **ModifyRecipients** deletes from the recipient list all columns not specified.

Setting the MAPI_UNICODE flag in the *ulFlags* parameter does the following:

- Sets the string type to Unicode for data returned for the initial active columns of the recipient table by **QueryColumns**.
- Sets the string type to Unicode for data returned for the initial active rows of the recipient table by **QueryRows**. The initial active rows for a recipient table are those rows **QueryRows** returns before the provider that contains the table calls **SetColumns**.
- Controls the property types of the sort order returned by the **IMAPITable::QuerySortOrder** method before the provider that contains the recipient table calls the **IMAPITable::SortTable** method.

See Also

IMAPIProp::SaveChanges method, **IMAPITable::QueryRows** method, **IMessage::ModifyRecipients** method

IMessage::ModifyRecipients

The **IMessage::ModifyRecipients** method adds, deletes, or modifies recipients in a message.

HRESULT ModifyRecipients(
  ULONG *ulFlags*,
  LPADRLIST *lpMods*
)

Parameters

ulFlags
  Input parameter containing a bitmask of flags that controls how the recipient table is modified – that is, how recipients are added, deleted, or modified. If zero is passed for the *ulFlags* parameter, the entire recipient table is replaced with the table passed in the *lpMods* parameter. The following flags can be set for *ulFlags*:

  MODRECIP_ADD
    Adds all recipients to the current table.

  MODRECIP_MODIFY
    Replaces an entire row with the rows specified.

  MODRECIP_REMOVE
    Removes recipients using the PR_ROWID property as an index.

lpMods
  Input parameter pointing to an **ADRLIST** structure containing a table of recipients to be added, deleted, or modified in the message.

Return Value

S_OK
  The call succeeded and has returned the expected value or values.

Remarks

Message store providers implement the **IMessage::ModifyRecipients** method to make changes to a message's recipient table.

The **IMAPISupport::Address** method returns **ADRLIST** structures that are the standard MAPI recipient tables; clients should send the same type of table in the *lpMods* parameter of **ModifyRecipients** to modify, add to, or delete recipients in the existing recipient table. Clients can also pass in the recipient table returned from the **IMessage::GetRecipientTable** method. When modifying the recipient table, a client must call the **IMAPITable::QueryColumns**, **IMAPITable::SetColumns**, and **IMAPITable::QueryRows** methods for each column to appear in the table – not just the modified columns – because **ModifyRecipients** deletes from the recipient list all columns not specified.

The default behavior of **ModifyRecipients** if all bits in the *ulFlags* bitmask are set to zero is to replace the entire existing recipient table with the table passed in *lpMods*. When calling **ModifyRecipients** with the MODRECIP_MODIFY flag set in *ulFlags*, **ModifyRecipients** instead replaces each entire recipient row with the associated row in the **ADRLIST** structure passed in *lpMods*.

Following are some rules for setting the properties of the recipients in the *lpMods* table:

- The presence of a property of type PT_NULL in the *lpMods* table returns an error value for the **ModifyRecipients** call.
- Any recipient property with type PT_ERROR is ignored by **ModifyRecipients**.
- The PR_ROWID property, identifying a particular recipient table row, must be provided for recipients in the *lpMods* table when the MODRECIP_REMOVE or MODRECIP_MODIFY flag is set in *ulFlags*.
- The PR_ROWID property must not be provided for *lpMods* table recipients when the

MODRECIP_ADD flag or zero is set in *ulFlags*.

- An unresolved recipient entry in *lpMods* must contain only the PR_DISPLAY_NAME and PR_RECIPIENT_TYPE properties.
- A resolved recipient entry in *lpMods* must contain the PR_DISPLAY_NAME, PR_ADDRTYPE, PR_ENTRYID, and PR_RECIPIENT_TYPE properties.
- The PR_EMAIL_ADDRESS property, containing a messaging address for a recipient, does not have to be present for resolved recipients in the *lpMods* table, but it can be.

A client can store in a message's recipient table both resolved and unresolved entries. However, if a message with unresolved entries in its recipient table is submitted to the MAPI spooler, it causes a nondelivery report to be created and sent to the user who sent the message.

If for a recipient in the *lpMods* table either PR_ADDRTYPE, giving a recipient's address type, or PR_EMAIL_ADDRESS, giving a recipient's messaging address, is not consistent with the address of the recipient as identified by PR_ENTRYID, the address where the message with the modified recipient table is delivered is undefined. The message might be delivered to the addresses specified in PR_ADDRTYPE and PR_EMAIL_ADDRESS or to the recipient identified by PR_ENTRYID, or it might be returned as undeliverable because of the ambiguity of the address information.

The **ADRLIST** structure passed in *lpMods* must be allocated as an **SRowSet** structure. **ModifyRecipients** does not free the **ADRLIST** structure nor any of its substructures. The **ADRLIST** structure and each **SPropValue** structure must be separately allocated by using the **MAPIAllocateBuffer** function such that each can be freed individually. If the method requires additional space for any **SPropValue** structure, it can replace the **SPropValue** structure with a new one that can later be freed by the calling client using the **MAPIFreeBuffer** function. The original **SPropValue** structure should also be freed using **MAPIFreeBuffer**.

An **ADRLIST** structure contains several **ADRENTRY** structures. In addition to resolved and unresolved recipient entries, **ADRENTRY** structures can be NULL, that is, the **cValues** member is zero and there are no property values. This is the case, for example, when the dialog box presented by **IAddrBook::Address** is used to remove a recipient from the list. For more information about resolved and unresolved recipient entries, see **IAddrBook::Address**.

See Also

ADRENTRY structure, **ADRLIST** structure, **IAddrBook::Address** method, **IMAPISupport::Address** method, **MAPIAllocateBuffer** function, **MAPIFreeBuffer** function, **SPropValue** structure

IMessage::OpenAttach

The **IMessage::OpenAttach** method opens an attachment.

HRESULT OpenAttach(
  ULONG *ulAttachmentNum*,
  LPCIID *lpInterface*,
  ULONG *ulFlags*,
  LPATTACH FAR * *lppAttach*
)

Parameters

ulAttachmentNum
Input parameter containing the index number of the attachment to open. This index number uniquely identifies the attachment within the message but is valid only within the message.

lpInterface
Input parameter pointing to the interface identifier (IID) for the returned attachment object. Passing NULL indicates that IID_IAttach is used. Client applications must pass NULL. Message store providers can also set the *lpInterface* parameter to IID_IUnknown, IID_IMAPIProp, or IID_IMessage.

ulFlags
Input parameter containing a bitmask of flags that controls how the attachment is opened. The following flags can be set:

MAPI_BEST_ACCESS
Indicates the object should be opened with the maximum network permissions allowed for the user and the maximum client access. For example, if the client has read/write access, the object is opened with read/write access; if the client has read-only access, the object is opened with read-only access. The client can retrieve the access level by getting the PR_ACCESS_LEVEL property.

MAPI_DEFERRED_ERRORS
Indicates the call is allowed to succeed even if the underlying object is not accessible to the calling client. If the object is not accessible, some subsequent call to the object might return an error.

MAPI_MODIFY
Requests read/write access. By default, objects are created with read-only access, and clients should not work on the assumption that read/write access has been granted.

lppAttach
Output parameter pointing to a variable where the pointer to the open attachment object is stored.

Return Value

S_OK
The call succeeded and has returned the expected value or values.

Remarks

Message store providers implement the **IMessage::OpenAttach** method to open an attachment within a message. A client can open an attachment by first locating the attachment's PR_ATTACH_NUM property (that is, the attachment's index number) in the table that is returned from the **IMessage::GetAttachmentTable** method, and then passing this PR_ATTACH_NUM value in the *ulAttachmentNum* parameter of **OpenAttach**. **OpenAttach** returns in the *lppAttach* parameter a pointer that provides further access to the open attachment.

Each attachment in a message has a different PR_ATTACH_NUM value, but this value is only unique within the message; messages in a store can have attachments with the same PR_ATTACH_NUM values as other messages. For example, an implementation can assign the value of zero to the first attachment in every message.

The expected behavior for opening multiple instances of the same attachment in the same message is undefined and specific to a particular provider's implementation.

IMessage::SetReadFlag

The **IMessage::SetReadFlag** method sets or clears the read flags for a message and manages the sending of read reports.

HRESULT SetReadFlag(
  ULONG *ulFlags*
)

Parameters

ulFlags
: Input parameter containing a bitmask of flags that controls the setting of a message's read flag (that is, the message's MSGFLAG_READ flag in its PR_MESSAGE_FLAGS property) and the processing of read reports. The following flags can be set:

  CLEAR_READ_FLAG
  : Clears MSGFLAG_READ. No read report is sent.

  GENERATE_RECEIPT_ONLY
  : Generates a read report if it is pending but does not change the state of MSGFLAG_READ.

  MAPI_DEFERRED_ERRORS
  : Indicates the call is allowed to succeed even if the underlying object is not accessible to the calling client. If the object is not accessible, some subsequent call to the object might return an error.

  SUPPRESS_RECEIPT
  : Directs the message store provider to cancel the generation of a read report if this call changes the state of the message from unread to read and a read report has been requested. If this call does not change the state of the message, the message store provider can ignore this flag.

Return Values

S_OK
: The call succeeded and has returned the expected value or values.

MAPI_E_NO_SUPPRESS
: The message store provider does not support the suppression of read reports.

Remarks

Message store providers implement the **IMessage::SetReadFlag** method to mark a message as read and, optionally, to send a read report for that message. A read report is only sent if the user who sent the message requested it. Implementations generally cannot determine if the user requested a read report. After setting the read flag (that is, setting the MSGFLAG_READ flag in the message's PR_MESSAGE_FLAGS property), **SetReadFlag** calls the **IMAPIProp::SaveChanges** method on the message. If MSGFLAG is set for a message, the message is marked as having been read, which does not necessarily indicate the intended recipient has read the message.

If none of the flags are set in the *ulFlags* parameter, the following rules apply:

- If MSGFLAG_READ is already set, no change should be made.
- If the PR_READ_RECEIPT_REQUESTED property is set, the client should send the read report and set MSGFLAG_READ.

SetReadFlag returns MAPI_E_INVALID_PARAMETER if any of the following combinations are set in *ulFlags*:

- SUPPRESS_RECEIPT | CLEAR_READ_FLAG
- SUPPRESS_RECEIPT | CLEAR_READ_FLAG | GENERATE_RECEIPT_ONLY

- CLEAR_READ_FLAG | GENERATE_RECEIPT_ONLY

If both the SUPPRESS_RECEIPT and GENERATE_RECEIPT_ONLY flags are set, the PR_READ_RECEIPT_REQUESTED bit, if set, should be cleared and a read report should not be sent.

Note Providers can optimize report behavior so that a client's setting a message attribute to get a read or delivery report is only a request and so that the provider can support not sending read or delivery reports. However, some message store providers do not support the suppression of read reports for some messages. If a client calls **SetReadFlag** on such a message with SUPPRESS_RECEIPT set in *ulFlags*, **SetReadFlag** returns MAPI_E_NO_SUPPRESS; in this case, MAPI does not set the read flag and does not generate a report.

See Also

IMAPIContainer::OpenEntry method, **IMAPIFolder::SetReadFlags** method, **IMAPIProp::GetProps** method, **IMAPIProp::SaveChanges** method, PR_MESSAGE_FLAGS property

IMessage::SubmitMessage

The **IMessage::SubmitMessage** method saves all changes to a message and marks the message as ready for sending.

HRESULT SubmitMessage(
  ULONG *ulFlags*
)

Parameters

ulFlags
  Input parameter containing a bitmask of flags used to control how a message is submitted. The following flag can be set:
  FORCE_SUBMIT
    Indicates MAPI should submit the message even if it might not be sent immediately.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.
MAPI_E_NO_RECIPIENTS
  Indicates that the message's recipient table is empty.

Remarks

Message store providers implement the **IMessage::SubmitMessage** method to indicate to transport providers that a message is ready for sending.

After a message is successfully saved and submitted, the pointers to the message and all its associated subobjects − messages, folders, attachments, streams, tables, and so on − are no longer valid. MAPI does not permit any further operations on these pointers, except for calling the objects' **IUnknown::Release** methods. In other words, MAPI is designed such that after **SubmitMessage** is called a client should release the message object and all associated subobjects. However, if **SubmitMessage** returns an error value indicating missing or invalid fields in the message, such as results when a message is sent with no recipients listed, then MAPI keeps the message open and all pointers remain valid.

To attempt to cancel a send operation, a client must get and store a pointer to the entry identifier of the message before the message is submitted, because the entry identifier is invalidated after the message has been submitted. Once the client has this entry identifier pointer, it then passes it as the *lpEntryId* parameter of the **IMsgStore::AbortSubmit** method.

MAPI passes messages to the underlying messaging system in the order in which they are marked for sending. Because of this functionality, a message might stay in a message store for some time before the underlying messaging system can take responsibility for it. The order of receipt at the destination is in the underlying messaging system's control and does not necessarily match the order in which messages were sent.

See Also

IMsgStore::AbortSubmit method

IMsgServiceAdmin : IUnknown

The **IMsgServiceAdmin** interface is used to make changes to a message service within a profile. An implementation can get a pointer to an **IMsgServiceAdmin** interface in two ways: by calling the **IMAPISession::AdminServices** method or by calling the **IProfAdmin::AdminServices** method. For clients primarily concerned with profile configuration, for example control panels, **IProfAdmin::AdminServices** is the preferred way to get the **IMsgServiceAdmin** interface because it does not log providers onto the MAPI session. If a client requires the ability to make changes to the active profile, then **IMAPISession::AdminServices** should be called to get the **IMsgServiceAdmin** pointer. Clients that are modifying the profile that is active for the session should be designed with the awareness that although MAPI does not allow a profile that is in use to be deleted, there are no safeguards to prevent a client from removing all the message services within the profile.

At a Glance

Specified in header file:	MAPIX.H
Object that supplies this interface:	Message service administration object
Corresponding pointer type:	LPSERVICEADMIN
Implemented by:	MAPI
Called by:	Client applications

Vtable Order

GetLastError	Returns a **MAPIERROR** structure containing information about the last error that occurred for a message service administration object.
GetMsgServiceTable	Returns a table listing the message services installed in a profile.
CreateMsgService	Adds a message service to the current profile.
DeleteMsgService	Deletes a message service and its associated profile sections from a profile.
CopyMsgService	Copies a message service into a profile.
RenameMsgService	Renames a message service that cannot be copied.
ConfigureMsgService	Enables a user to reconfigure a message service using the service's configuration property sheet.
OpenProfileSection	Opens a section of the current profile and returns a pointer that provides further access to the profile section object.
MsgServiceTransportOrder	Sets the order in which transport providers are called to deliver a message.
AdminProviders	Returns a pointer providing access to a provider administration object.
SetPrimaryIdentity	Designates a message service as the supplier of the primary identity for the profile.
GetProviderTable	Returns a table listing the service providers installed in a profile.

IMsgServiceAdmin::AdminProviders

The **IMsgServiceAdmin::AdminProviders** method returns a pointer providing access to a provider administration object.

HRESULT AdminProviders(
  LPMAPIUID *lpUID*,
  ULONG *ulFlags*,
  LPPROVIDERADMIN FAR * *lppProviderAdmin*
)

Parameters

lpUID
  Input parameter pointing to the **MAPIUID** structure holding the MAPI unique identifier (MAPIUID) for the message service. This MAPIUID is typically obtained from the PR_SERVICE_UID property column in the message service administration table.

ulFlags
  Input parameter containing a bitmask of flags that controls the type of the passed-in string. The following flag can be set:

  MAPI_UNICODE
    Indicates the passed-in string is in Unicode format. If the MAPI_UNICODE flag is not set, the string is in ANSI format.

lppProviderAdmin
  Output parameter pointing to a variable where the pointer to the returned provider administration object is stored.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

MAPI_E_NOT_FOUND
  The MAPIUID passed in does not exist.

Remarks

Use the **IMsgServiceAdmin::AdminProviders** method to get a pointer to a provider administration object so the methods of the **IProviderAdmin** interface can be called for that object. Clients cannot make changes to service providers; all a client can do is determine the providers within a message service and determine each provider's MAPIUID. The types of changes that can be made to a message service while the profile is in use are implementation-specific; however, most message services do not support changes such as adding and deleting providers while the profile is in use.

See Also

IProviderAdmin : IUnknown interface, **MAPIUID** structure

IMsgServiceAdmin::ConfigureMsgService

The **IMsgServiceAdmin::ConfigureMsgService** method enables a user to reconfigure a message service using the service's configuration property sheet.

HRESULT ConfigureMsgService(
  LPMAPIUID *lpUID*,
  ULONG *ulUIParam*,
  ULONG *ulFlags*,
  ULONG *cValues*,
  LPSPropValue *lpProps*
)

Parameters

lpUID
  Input parameter pointing to the **MAPIUID** structure holding the MAPI unique identifier (MAPIUID) for the message service to configure.

ulUIParam
  Input parameter containing the handle of the parent window for the configuration property sheet.

ulFlags
  Input parameter containing a bitmask of flags that controls the display of the property sheet. The following flags can be set:

  MAPI_UNICODE
    Indicates the passed-in strings are in Unicode format. If the MAPI_UNICODE flag is not set, the strings are in ANSI format.

  MSG_SERVICE_UI_READ_ONLY
    Indicates the service's configuration property sheet should display the current configuration but not enable the user to change it. Most message services ignore this flag.

  SERVICE_UI_ALLOWED
    Displays the message service's configuration property sheet only if the service is not completely configured.

  SERVICE_UI_ALWAYS
    Requires the message service display a configuration property sheet. If SERVICE_UI_ALWAYS is not set, a configuration property sheet can still be displayed if SERVICE_UI_ALLOWED is set and valid configuration information is not available from the property value array in the *lpProps* parameter. Either SERVICE_UI_ALLOWED or SERVICE_UI_ALWAYS must be set for a property sheet to be displayed.

cValues
  Input parameter containing the number of property values in the **SPropValue** structure pointed to by *lpProps*. The *cValues* parameter should be zero if there are no properties.

lpProps
  Input parameter pointing to an **SPropValue** structure containing the property values of the properties to display to the user in the property sheet. The *lpProps* parameter should be NULL if the message service is being configured without displaying a property sheet.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

MAPI_E_EXTENDED_ERROR
  An error specific to a message service. To get the **MAPIERROR** structure describing the error, the client application should call the **IMsgServiceAdmin::GetLastError** method.

MAPI_E_NOT_FOUND

The MAPIUID does not match that of an existing message service.

MAPI_E_NOT_INITIALIZED
: The message service does not have an entry point function.

MAPI_E_USER_CANCEL
: The user canceled the operation, typically by clicking the **Cancel** button in the property sheet.

Remarks

Use the **IMsgServiceAdmin::ConfigureMsgService** method to display the configuration property sheet for a message service so that users can configure the message service in their profiles. The MAPIUID used in the *lpUID* parameter is typically retrieved from the message service administration table by calling the **IMsgServiceAdmin::GetMsgServiceTable** method after the message service has been created with the **IMsgServiceAdmin::CreateMsgService** method.

Clients can also configure the message service without displaying the property sheet to the user. This type of configuration can only be performed if the client has information on the message service's property identifiers and property values in advance. If a client is configuring the service without displaying a property sheet, neither the SERVICE_UI_ALLOWED nor the SERVICE_UI_ALWAYS flag should be set in the *ulFlags* parameter. If a client receives its configuration information from the property sheet, or if the existing information is insufficient to completely configure the service, the client should set SERVICE_UI_ALLOWED in *ulFlags*. If a client uses existing property information only to establish the default settings and the user is able to change the settings, the client should set SERVICE_UI_ALWAYS in *ulFlags*.

To allow configuration without property sheet display, message services typically prepare a header file that includes constants for all the required and optional properties and their values.

See Also

MAPIUID structure, **SPropValue** structure

IMsgServiceAdmin::CopyMsgService

The **IMsgServiceAdmin::CopyMsgService** method copies a message service into a profile.

HRESULT CopyMsgService(
  LPMAPIUID *lpUID*,
  LPTSTR *lpszDisplayName*,
  LPCIID *lpInterfaceToCopy*,
  LPCIID *lpInterfaceDst*,
  LPVOID *lpObjectDst*,
  ULONG *ulUIParam*,
  ULONG *ulFlags*
)

Parameters

lpUID
: Input parameter pointing to the **MAPIUID** structure holding the MAPI unique identifier (MAPIUID) for the message service to copy.

lpszDisplayName
: Input parameter pointing to a string containing the display name for the message service to copy

lpInterfaceToCopy
: Input parameter pointing to the interface identifier (IID) for the profile section object into which to copy the message service. Passing NULL indicates the identifier for the profile section object interface, IID_IProfSect, is used. The *lpInterfaceToCopy* parameter can also be set to an identifier for an appropriate interface, for example IID_IMAPIProp or IID_IUnknown.

lpInterfaceDst
: Input parameter pointing to the IID for the session or message service administration object indicated in the *lpObjectDst* parameter. Passing NULL indicates the MAPI session interface identifier, IID_IMAPISession, is used. The *lpInterfaceDst* parameter can also be set to IID_IMsgServiceAdmin.

lpObjectDst
: Input parameter pointing to a pointer to a session or message service administration object; the type of object should correspond to the interface identifier passed in *lpInterfaceDst*. Valid object pointers are LPMAPISESSION and LPSERVICEADMIN.

ulUIParam
: Input parameter containing the handle of the parent window for any dialog boxes or windows this method displays.

ulFlags
: Input parameter containing a bitmask of flags that controls how the message service is copied. The following flags can be set:

  MAPI_UNICODE
  : Indicates the passed-in strings are in Unicode format. If the MAPI_UNICODE flag is not set, the strings are in ANSI format.

  SERVICE_UI_ALWAYS
  : Requires the message service display a configuration property sheet. If SERVICE_UI_ALWAYS is not set, a configuration property sheet can still be displayed if SERVICE_UI_ALLOWED is set and valid configuration information is not available from the property value array in the *lpProps* parameter. Either SERVICE_UI_ALLOWED or SERVICE_UI_ALWAYS must be set for a property sheet to be displayed.

Return Values

S_OK

The call succeeded and has returned the expected value or values.

MAPI_E_NO_ACCESS
: The message service is already in the profile and does not allow multiple instances of itself.

MAPI_E_NOT_FOUND
: The MAPIUID passed in does not refer to an existing message service.

Remarks

Use the **IMsgServiceAdmin::CopyMsgService** method to copy a message service into a profile section. To do so, a client must previously have obtained an interface for the target profile, but it does not have to be logged onto the profile. The target profile is not necessarily the same as the profile from which the message service is copied. Clients can copy message services within a profile or from one profile to another.

The message service's entry point function does not get called for either the source or the destination of the copy operation. After the copy operation, the configuration settings of the service remain unchanged. To configure the copied message service, a client should call the **IMsgServiceAdmin::ConfigureMsgService** method.

See Also

IMsgServiceAdmin::ConfigureMsgService method, **MAPIUID** structure

IMsgServiceAdmin::CreateMsgService

The **IMsgServiceAdmin::CreateMsgService** method adds a message service to the current profile.

HRESULT CreateMsgService(
  LPTSTR *lpszService***,**
  LPTSTR *lpszDisplayName***,**
  ULONG *ulUIParam***,**
  ULONG *ulFlags*
)

Parameters

lpszService
  Input parameter pointing to a string naming the message service to add. This message service name must appear in the [Services] section of MAPISVC.INF.

lpszDisplayName
  Input parameter pointing to a string containing the display name for the message service to add. The *lpszDisplayName* parameter is ignored because service providers set the PR_DISPLAY_NAME property in MAPISVC.INF themselves.

ulUIParam
  Input parameter containing the handle of the parent window for any dialog boxes or windows this method displays.

ulFlags
  Input parameter containing a bitmask of flags that controls how the message service is installed. The following flags can be set:

  MAPI_UNICODE
    Indicates the passed-in strings are in Unicode format. If the MAPI_UNICODE flag is not set, the strings are in ANSI format.

  SERVICE_UI_ALLOWED
    Displays the message service's configuration property sheet only if the service is not completely configured.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

MAPI_E_NOT_FOUND
  The message service name is not in the [Services] section of MAPISVC.INF.

Remarks

Use the **IMsgServiceAdmin::CreateMsgService** method to add a message service to the current profile. A client passes the name of the service to add in the *lpszService* parameter.

If the service has defined an entry point function, it is called so the service can perform any service-specific configuration tasks. If the SERVICE_UI_ALLOWED flag is set in the *ulFlags* parameter, the message service being installed can display a property sheet to enable the user to configure its settings.

The list of providers that make up a message service and the properties for each are contained within the MAPISVC.INF file. **CreateMsgService** first creates a new profile section for the message service and then copies all of the information for that service from the MAPISVC.INF file into the profile, creating new sections for each provider. After all the information has been copied from MAPISVC.INF, the message service's entry point function is called with the MSG_SERVICE_CREATE value set in the *ulContext* parameter. If the SERVICE_UI_ALLOWED flag is set in the **CreateMsgService** method's *ulFlags* parameter, then the values in the **CreateMsgService** *ulUIParam* and *ulFlags* parameters are also passed when the message service's entry point function is called. Service providers should display their configuration property sheets so users can configure the message service.

CreateMsgService does not return the MAPIUID for the message service added to the profile. To retrieve this MAPIUID, a client should call the **IMsgServiceAdmin::GetMsgServiceTable** method to get the message service administration table. It then searches for the message service using the PR_SERVICE_NAME property and retrieves the service's PR_SERVICE_UID property. It then passes the MAPIUID from PR_SERVICE_UID in the *lpUid* parameter when calling the **IMsgServiceAdmin::ConfigureMsgService** method to configure the service.

IMsgServiceAdmin::DeleteMsgService

The **IMsgServiceAdmin::DeleteMsgService** method deletes a message service and its associated profile sections from a profile.

HRESULT DeleteMsgService(
  LPMAPIUID *lpuid*
)

Parameters

lpuid
  Input parameter pointing to the **MAPIUID** structure holding the MAPI unique identifier (MAPIUID) for the message service to delete.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

MAPI_E_NOT_FOUND
  The MAPIUID passed in does not match an existing message service.

Remarks

Use the **IMsgServiceAdmin::DeleteMsgService** method to delete a message service from a profile. To retrieve the MAPIUID for the message service, a client should call the **IMsgServiceAdmin::GetMsgServiceTable** method to get the message service administration table. It then searches for the message service using the PR_SERVICE_NAME property and retrieves the service's PR_SERVICE_UID property. It then passes the MAPIUID from PR_SERVICE_UID in the *lpuid* parameter when calling **DeleteMsgService**. **DeleteMsgService** removes all profile sections related to the message service.

If the service has a defined entry point function, that function is called with the MSG_SERVICE_DELETE value set in the *ulContext* parameter before the profile sections are removed so the service can perform any service-specific tasks. Next, the service is deleted, and then the service's profile section is deleted. The message service's entry point function is not called again after the service has been deleted.

See Also

MAPIUID structure

IMsgServiceAdmin::GetLastError

The **IMsgServiceAdmin::GetLastError** method returns a **MAPIERROR** structure containing information about the last error that occurred for a message service administration object.

HRESULT GetLastError(
  HRESULT *hResult*,
  ULONG *ulFlags*,
  LPMAPIERROR FAR * *lppMAPIError*
)

Parameters

hResult
  Input parameter containing the result returned for the last call for the message service administration object that returned an error.

ulFlags
  Input parameter containing a bitmask of flags that controls the type of the returned strings. The following flag can be set:

  MAPI_UNICODE
    Indicates the strings in the **MAPIERROR** structure returned in the *lppMAPIError* parameter are in Unicode format. If the MAPI_UNICODE flag is not set, the strings are in ANSI format.

lppMAPIError
  Output parameter pointing to a pointer to the returned **MAPIERROR** structure containing version, component, and context information for the error. The *lppMAPIError* parameter can be set to NULL if there is no **MAPIERROR** structure to return.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

MAPI_E_BAD_CHARWIDTH
  Either the MAPI_UNICODE flag was set and the implementation does not support Unicode, or MAPI_UNICODE was not set and the implementation only supports Unicode.

Remarks

Use the **IMsgServiceAdmin::GetLastError** method to retrieve information to display in a message to the user regarding the last error returned from a method call for the message service administration object.

To release all the memory allocated by MAPI, client applications need only call the **MAPIFreeBuffer** function for the returned **MAPIERROR** structure.

The return value from **GetLastError** must be S_OK for a client to make use of the **MAPIERROR** structure. Even if the return value is S_OK, a **MAPIERROR** structure might not be returned. If the implementation cannot determine what the last error was, or if a **MAPIERROR** structure is not available for that error, **GetLastError** returns a pointer to NULL in *lppMAPIError* instead.

See Also

MAPIERROR structure, **MAPIFreeBuffer** function

IMsgServiceAdmin::GetMsgServiceTable

The **IMsgServiceAdmin::GetMsgServiceTable** method returns a table listing the message services installed in a profile.

HRESULT GetMsgServiceTable(
  ULONG *ulFlags*,
  LPMAPITABLE FAR * *lppTable*
)

Parameters

ulFlags
  Input parameter containing a bitmask of flags that controls the type of the strings returned in the table's default column set. The following flag can be set:

  MAPI_UNICODE
    Indicates the returned strings are in Unicode format. If the MAPI_UNICODE flag is not set, the strings are in ANSI format.

lppTable
  Output parameter pointing to a variable where the pointer to the returned table object is stored.

Return Value

S_OK
  The call succeeded and has returned the expected value or values.

Remarks

Use the **IMsgServiceAdmin::GetMsgServiceTable** method to get a pointer to a table object that lists the message administration services currently installed in a profile. The columns of this message service table contain the current information for the following properties:

PR_DISPLAY_NAME
PR_SERVICE_NAME
PR_RESOURCE_FLAGS
PR_SERVICE_DLL_NAME
PR_SERVICE_ENTRY_NAME
PR_SERVICE_UID
PR_SERVICE_SUPPORT_FILES
PR_INSTANCE_KEY

Once a message service administration table has been returned, it does not reflect changes being made to the profile, such as the addition or deletion of providers. Calls to the **IMAPITable::Advise** method for the message service administration table return S_OK, but no changes are made to the table.

If no profile exists, **GetMsgServiceTable** does not return an error but returns a table object supporting the **IMAPITable** interface. If a client calls the **IMAPITable::QueryRows** method on that table, zero rows are returned.

Clients that are configuring or deleting message services using the **IMsgServiceAdmin::ConfigureMsgService** or **IMsgServiceAdmin::DeleteMsgService** method typically call **GetMsgServiceTable** to get the MAPIUID stored in the PR_SERVICE_UID property for the service they are working with.

Setting the MAPI_UNICODE flag in the *ulFlags* parameter does the following:

- Sets the string type to Unicode for data returned for the initial active columns of the message service administration table by the **IMAPITable::QueryColumns** method. The initial active columns for a message service administration table are those columns **QueryColumns** returns before the provider that contains the table calls the **IMAPITable::SetColumns** method.
- Sets the string type to Unicode for data returned for the initial active rows of the message service administration table by the **IMAPITable::QueryRows** method. The initial active rows for a message service administration table are those rows **QueryRows** returns before the provider that contains the table calls **SetColumns**.
- Controls the property types of the sort order returned by the **IMAPITable::QuerySortOrder** method before the provider that contains the message service administration table calls the **IMAPITable::SortTable** method.

See Also

IMsgServiceAdmin::ConfigureMsgService method, **IMsgServiceAdmin::DeleteMsgService** method

IMsgServiceAdmin::GetProviderTable

The **IMsgServiceAdmin::GetProviderTable** method returns a table listing the service providers installed in a profile.

HRESULT GetProviderTable(
  ULONG *ulFlags*,
  LPMAPITABLE FAR * *lppTable*
)

Parameters

ulFlags

Input parameter containing a bitmask of flags that controls the type of the strings returned in the provider table's default column set. The following flag can be set:

MAPI_UNICODE

Indicates the returned strings are in Unicode format. If the MAPI_UNICODE flag is not set, the strings are in ANSI format.

lppTable

Output parameter pointing to a variable where the pointer to the returned provider table object is stored.

Return Value

S_OK

The call succeeded and has returned the expected value or values.

Remarks

Use the **IMsgServiceAdmin::GetProviderTable** method to get a pointer to a table object that lists all of the address book, message store, transport, and message hook providers currently installed in a profile. The columns of the provider table contain the current information for the following properties:

PR_DISPLAY_NAME
PR_INSTANCE_KEY
PR_PROVIDER_DISPLAY
PR_PROVIDER_DLL_NAME
PR_PROVIDER_ORDINAL
PR_PROVIDER_UID
PR_RESOURCE_FLAGS
PR_RESOURCE_TYPE
PR_SERVICE_NAME
PR_SERVICE_UID

Providers that have been deleted, or are in use but have been marked for deletion, are not returned in the provider table. Once a provider table has been returned, it does not reflect changes being made to the profile, such as the addition or deletion of providers. Calls to the **IMAPITable::Advise** method for the provider table return S_OK, but no changes are made to the table.

If no provider exists, **GetProviderTable** does not return an error but returns a table object supporting the **IMAPITable** interface. If the client calls the **IMAPITable::QueryRows** method on that table, zero rows are returned.

The provider table's PR_PROVIDER_ORDINAL property can be used to restrict sort operations on the table. The first transport provider in the list has PR_PROVIDER_ORDINAL set to 0, the next provider to 1, and so on; this functionality enables a client to retrieve the table with the list of providers set to the correct order. This list of providers can then be used to select the order of transport providers by calling the **IMsgServiceAdmin::MsgServiceTransportOrder** method.

To display the transport provider order, a client should restrict the table such that `PR_RESOURCE_TYPE==MAPI Transport Provider`. The client then sorts the table by PR_PROVIDER_ORDINAL and calls **QueryRows** to get the rows of the table. These calls can also all be made as a single call to the **HrQueryAllRows** function with all of the appropriate data structures passed in. However, the information used in setting the transport provider order can only be obtained by calling **GetProviderTable**.

Setting the MAPI_UNICODE flag in the *ulFlags* parameter does the following:

- Sets the string type to Unicode for data returned for the initial active columns of the provider table by the **IMAPITable::QueryColumns** method. The initial active columns for a provider table are those columns the **QueryColumns** method returns before the provider that contains the table calls the **IMAPITable::SetColumns** method.
- Sets the string type to Unicode for data returned for the initial active rows of the provider table by **QueryRows**. The initial active rows for a provider table are those rows **QueryRows** returns before the provider that contains the table calls **SetColumns**.
- Controls the property types of the sort order returned by the **IMAPITable::QuerySortOrder** method before the client that contains the provider table calls the **IMAPITable::SortTable** method.

See Also

IMsgServiceAdmin::GetMsgServiceTable method, **IMsgServiceAdmin::MsgServiceTransportOrder** method, **IProviderAdmin::GetProviderTable** method

IMsgServiceAdmin::MsgServiceTransportOrder

The **IMsgServiceAdmin::MsgServiceTransportOrder** method sets the order in which transport providers are called to deliver a message.

HRESULT MsgServiceTransportOrder(
  ULONG *cUID***,**
  LPMAPIUID *lpUIDList***,**
  ULONG *ulFlags*
)

Parameters

cUID
  Input parameter containing the number of unique identifiers in the *lpUIDList* parameter.

lpUIDList
  Input parameter pointing to a counted array of **MAPIUID** structures holding MAPI unique identifiers (MAPIUIDs). The array contains one identifier for each transport provider configured in the current profile.

ulFlags
  Reserved; must be zero.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

MAPI_E_BUSY
  The value in the *cUID* parameter differs from the number of transport providers actually in the profile.

MAPI_E_NOT_FOUND
  One or more of the MAPIUIDs passed in the *lpUIDList* parameter does not refer to a transport provider currently in the profile.

Remarks

Use the **IMsgServiceAdmin::MsgServiceTransportOrder** method to set the delivery order of transport providers within a profile. The *lpUIDList* parameter must contain a sorted list of transport-provider entry identifiers obtained from the PR_PROVIDER_UID property of the table returned from the **IMsgServiceAdmin::GetProviderTable** method. A client application must pass the complete list in *lpUIDList*.

SetTransportOrder overrides transport provider preferences such as the STATUS_XP_PREFER_LAST flag set in the PR_RESOURCE_FLAGS property.

See Also

MAPIUID structure

IMsgServiceAdmin::OpenProfileSection

The **IMsgServiceAdmin::OpenProfileSection** method opens a section of the current profile and returns a pointer that provides further access to the profile object.

HRESULT OpenProfileSection(
  LPMAPIUID *lpUID*,
  LPCIID *lpInterface*,
  ULONG *ulFlags*,
  LPPROFSECT FAR * *lppProfSect*
)

Parameters

lpUID
  Input parameter pointing to the **MAPIUID** structure holding the MAPI unique identifier (MAPIUID) for the profile section. The *lpUID* parameter must not be NULL.

lpInterface
  Input parameter pointing to the interface identifier (IID) for the profile section. Passing NULL indicates the object is cast to the standard interface for a profile section. The *lpInterface* parameter can also be set to an identifier for an appropriate interface for the object. Valid interface identifiers are IID_IMAPIProp and IID_IProfSect.

ulFlags
  Input parameter containing a bitmask of flags that controls access to the profile section. The following flags can be set:

  MAPI_DEFERRED_ERRORS
    Indicates the call is allowed to succeed even if the underlying object is not accessible to the calling client. If the object is not accessible, some subsequent call to the object might return an error.

  MAPI_MODIFY
    Requests read/write access. By default, objects are created with read-only access, and client applications should not work on the assumption that read/write access has been granted.

lppProfSect
  Output parameter pointing to a variable where the pointer to the returned profile section object is stored.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

MAPI_E_NO_ACCESS
  An attempt was made to modify a read-only profile section or an attempt to access an object for which the user has insufficient permissions.

MAPI_E_NOT_FOUND
  The requested object does not exist.

Remarks

Use the **IMsgServiceAdmin::OpenProfileSection** method to open a profile section for reading information from and writing information to the active profile for the session. A profile section object supporting the **IProfSect** interface is returned in the *lppProfSect* parameter. Default behavior is to open the profile section as read-only, unless a client sets the MAPI_MODIFY flag in the *ulFlags* parameter. Profile sections belonging to service providers cannot be opened by calls to **OpenProfileSection**.

If an **OpenProfileSection** call opens a nonexistent profile section by passing MAPI_MODIFY in *ulFlags*, the call creates the section. If an **OpenProfileSection** call attempts to open a nonexistent section with read-only access, it returns MAPI_E_NOT_FOUND.

See Also

IMAPIProp : IUnknown interface, **IMAPISession::OpenProfileSection** method, **IProfSect : IMAPIProp** interface, **MAPIUID** structure

IMsgServiceAdmin::RenameMsgService

The **IMsgServiceAdmin::RenameMsgService** method renames a message service that cannot be copied.

HRESULT RenameMsgService(
  LPMAPIUID *lpUID*,
  ULONG *ulFlags*,
  LPTSTR *lpszDisplayName*
)

Parameters

lpUID
  Input parameter pointing to the **MAPIUID** structure holding the MAPI unique identifier (MAPIUID) for the message service being renamed.

ulFlags
  Reserved; must be zero.

lpszDisplayName
  Input parameter pointing to a string containing the new display name for the message service.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

MAPI_E_NO_SUPPORT
  This method always returns this value.

See Also

MAPIUID structure

IMsgServiceAdmin::SetPrimaryIdentity

The **IMsgServiceAdmin::SetPrimaryIdentity** method designates each service provider belonging to the message service as being the supplier of the primary identifier for the profile.

HRESULT SetPrimaryIdentity(
  LPMAPIUID *lpUID*,
  ULONG *ulFlags*
)

Parameters

lpUID
  Input parameter pointing to the unique identifier for the message service whose identity is to be designated as primary.

ulFlags
  Reserved; must be zero.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

MAPI_E_NO_ACCESS
  SetPrimaryIdentity attempted to designate a primary identity for a service that has the SERVICE_NO_PRIMARY_IDENTITY flag set in the PR_RESOURCE_FLAGS property.

Remarks

Use the **IMsgServiceAdmin::SetPrimaryIdentity** method to set the primary identity for each provider in the message service indicated by the *lpUID* parameter. The *primary identity* for a message service is the identity of the user logged onto the message service; it is represented by the entry identifier returned by a call to the **IMAPISession::QueryIdentity** method.

Each message service provider that MAPI has information about establishes an identity for each of its users. This identity can be established when a client logs onto the service. However, because MAPI supports connections to multiple service providers for each MAPI session, there is no firm definition of a particular user's identity for the MAPI session as a whole; a user's identity depends on which service is involved. Clients can call **SetPrimaryIdentity** to designate one of the many identities established for a user by message services as the primary identity for that user. Service providers that utilize the functionality provided by primary identities should set the STATUS_PRIMARY_IDENTITY flag in PR_RESOURCE_FLAGS.

If the calling client passes NULL in *lpUID*, the primary identity for the indicated message service is cleared. Calls to **SetPrimaryIdentity** fail and return MAPI_E_NO_ACCESS if the service designated in *lpUID* has the SERVICE_NO_PRIMARY_IDENTITY flag set for its PR_RESOURCE_FLAGS property, meaning that service providers cannot be used to supply an identity.

See Also

IMAPISession::QueryIdentity method, **MAPIUID** structure

IMsgStore : IMAPIProp

The **IMsgStore** interface provides access to message store information.

At a Glance

Specified in header file:	MAPIDEFS.H
Object that supplies this interface:	Message store object
Corresponding pointer type:	LPMDB
Implemented by:	Message store providers
Transaction model:	Non-transacted
Called by:	Client applications, the MAPI spooler, MAPI

Vtable Order

Advise	Registers a client application for notifications about changes within a message store.
Unadvise	Removes an object's registration for notification of message store changes previously established with a call to the **IMsgStore::Advise** method.
CompareEntryIDs	Compares two entry identifiers to determine if they refer to the same entry within a message store.
OpenEntry	Opens a folder or message in a message store given its entry identifier.
SetReceiveFolder	Sets the receive folder for a particular message class.
GetReceiveFolder	Obtains the receive folder set to receive messages for a particular message class and related information about message reception behavior of that message class.
GetReceiveFolderTable	Returns a table that shows property settings for all receive folders for a message store, including which message classes the folders are associated with.
StoreLogoff	Enables the orderly logoff of a message store under client application control.
AbortSubmit	Attempts to remove a message from a message store's outgoing queue.
GetOutgoingQueue	Returns a table listing the messages in a message store's outgoing queue. This method is called only by the MAPI spooler.
SetLockState	Allows the MAPI spooler to lock or unlock a message.
FinishedMsg	Enables a message store provider to finish its processing for a sent message. This method is called only by the MAPI spooler.
NotifyNewMail	Notifies a message store that a new message has arrived. This method is called only by the MAPI

spooler.

Required Properties

PR_DISPLAY_NAME	Read/write
PR_ENTRYID	Read-only
PR_OBJECT_TYPE	Read-only
PR_RECORD_KEY	Read-only
PR_STORE_ENTRYID	Read-only
PR_STORE_RECORD_KEY	Read-only
PR_MDB_PROVIDER	Read-only
PR_STORE_SUPPORT_MASK	Read-only

Properties for IPM Message Stores

PR_IPM_OUTBOX_ENTRYID	PR_IPM_OUTBOX_SEARCH_KEY
PR_IPM_SENTMAIL_ENTRYID	PR_IPM_SENTMAIL_SEARCH_KEY
PR_IPM_SUBTREE_ENTRYID	PR_IPM_SUBTREE_SEARCH_KEY
PR_IPM_WASTEBASKET_ENTRYID	PR_IPM_WASTEBASKET_SEARCH_KEY
PR_MDB_PROVIDER	PR_STORE_SUPPORT_MASK

IMsgStore::AbortSubmit

The **IMsgStore::AbortSubmit** method attempts to remove a message from a message store's outgoing queue.

```
AbortSubmit(
  ULONG cbEntryID,
  LPENTRYID lpEntryID,
  ULONG ulFlags
 )
```

Parameters

cbEntryID
: Input parameter containing the size, in bytes, of the entry identifier pointed to by the *lpEntryID* parameter.

lpEntryID
: Input parameter pointing to the entry identifier of the message to remove from the message queue.

ulFlags
: Reserved; must be zero.

Return Values

S_OK
: The call succeeded and has returned the expected value or values.

MAPI_E_NOT_IN_QUEUE
: The message is no longer in the message store's outgoing queue, typically because it has already been sent.

MAPI_E_UNABLE_TO_ABORT
: The message is locked by the MAPI spooler, and the operation cannot be aborted.

Remarks

Message store providers implement the **IMsgStore::AbortSubmit** method to attempt to remove a submitted message from the message store's outgoing queue. Calling **AbortSubmit** for a message in the store is the only action that a client can perform on a message after it has been submitted. If possible, the **AbortSubmit** call removes the message from the submission queue. However, depending on how the underlying messaging system is implemented, it might not be possible to cancel the sending of the message.

See Also

IMessage::SubmitMessage method

IMsgStore::Advise

The **IMsgStore::Advise** method registers a client application for notifications about changes within a message store.

HRESULT Advise(
  ULONG *cbEntryID*,
  LPENTRYID *lpEntryID*,
  ULONG *ulEventMask*,
  LPMAPIADVISESINK *lpAdviseSink*,
  ULONG FAR * *lpulConnection*
)

Parameters

cbEntryID
  Input parameter containing the size, in bytes, of the entry identifier pointed to by the *lpEntryID* parameter.

lpEntryID
  Input parameter pointing to the entry identifier of the object about which notifications should be generated. This object can be a folder or a message. Alternatively, if the client application sets the *cbEntryID* parameter to zero and passes NULL for *lpEntryID*, the advise sink provides notifications about changes to the entire message store.

ulEventMask
  Input parameter containing an event mask of the types of notification events occurring for the object about which MAPI will generate notifications. The mask filters specific cases. Each event type has a structure associated with it that holds additional information about the event. The following table lists the possible event types along with their corresponding structures.

Notification event type	Corresponding structure
`fnevCriticalError`	**ERROR_NOTIFICATION**
`fnevNewMail`	**NEWMAIL_NOTIFICATION**
`fnevObjectCreated`	**OBJECT_NOTIFICATION**
`fnevObjectDeleted`	**OBJECT_NOTIFICATION**
`fnevObjectModified`	**OBJECT_NOTIFICATION**
`fnevObjectCopied`	**OBJECT_NOTIFICATION**
`fnevObjectMoved`	**OBJECT_NOTIFICATION**
`fnevSearchComplete`	**OBJECT_NOTIFICATION**

lpAdviseSink
  Input parameter pointing to the advise sink object to be called when an event occurs for the object about which notification has been requested.

lpulConnection
  Output parameter pointing to a variable that upon a successful return holds the connection number for the notification registration. The connection number must be nonzero.

Return Value

S_OK
  The call succeeded and has returned the expected value or values.

MAPI_E_NO_SUPPORT
This error value is returned if the method is called by any process other than the MAPI spooler.

Remarks

Message store providers implement the **IMsgStore::Advise** method to register an object in a message store provider for notification callbacks. Whenever a change occurs to the indicated object, the provider checks to see what event mask bit was set in the *ulEventMask* parameter and thus what type of change occurred. If a bit is set, then the provider calls the **IMAPIAdviseSink::OnNotify** method for the advise sink object indicated by the *lpAdviseSink* parameter to report the event. Data passed in the notification structure to the **OnNotify** method describes the event.

The call to **OnNotify** can occur during the call that changes the object, or at any following time. On systems that support multiple threads of execution, the call to **OnNotify** can occur on any thread. For a way to turn a call to **OnNotify** that might happen at an inopportune time into one that is safer to handle, a client should use the **HrThisThreadAdviseSink** function.

To provide notifications, the message store provider implementing **Advise** needs to keep a copy of the pointer to the *lpAdviseSink* advise sink object; to do so, it calls the **IUnknown::AddRef** method for the advise sink to maintain its object pointer until notification registration is canceled with a call to the **IMsgStore::Unadvise** method. The **Advise** implementation should assign a connection number to the notification registration and call **AddRef** on this connection number before returning it in the *lpulConnection* parameter. Service providers can release the advise sink object before the registration is canceled, but they must not release the connection number until **Unadvise** has been called.

After a call to **Advise** has succeeded and before **Unadvise** has been called, clients must be prepared for the advise sink object to be released. A client should therefore release its advise sink object after **Advise** returns unless it has a specific long-term use for it.

For more information on the notification process, see About Notification.

See Also

HrThisThreadAdviseSink function, **IMAPIAdviseSink::OnNotify** method, **IMsgStore::Unadvise** method, **NOTIFICATION** structure

IMsgStore::CompareEntryIDs

The **IMsgStore::CompareEntryIDs** method compares two entry identifiers to determine if they refer to the same entry within a message store. MAPI only passes this call to a service provider if the unique identifiers (UIDs) in both entry identifiers to be compared are handled by that provider.

```
HRESULT CompareEntryIDs(
   ULONG cbEntryID1,
   LPENTRYID lpEntryID1,
   ULONG cbEntryID2,
   LPENTRYID lpEntryID2,
   ULONG ulFlags,
   ULONG FAR * lpulResult
  )
```

Parameters

cbEntryID1
Input parameter containing the size, in bytes, of the entry identifier pointed to by the *lpEntryID1* parameter.

lpEntryID1
Input parameter pointing to the first entry identifier to be compared.

cbEntryID2
Input parameter containing the size, in bytes, of the entry identifier pointed to by the *lpEntryID2* parameter.

lpEntryID2
Input parameter pointing to the second entry identifier to be compared.

ulFlags
Reserved; must be zero.

lpulResult
Output parameter pointing to a variable where is stored the returned result of the comparison; this variable is TRUE if the two entry identifiers refer to the same object, and FALSE otherwise.

Return Value

S_OK
The call succeeded and has returned the expected value or values.

Remarks

Message store providers implement the **IMsgStore::CompareEntryIDs** method to compare two entry identifiers for a given entry within a message store to determine whether they refer to the same object. If the two entry identifiers refer to the same object, then **CompareEntryIDs** sets the *lpulResult* parameter to TRUE; if they refer to different objects, **CompareEntryIDs** sets *lpulResult* to FALSE.

CompareEntryIDs is useful because an object can have more than one valid entry identifier; such a situation can occur, for example, after a new version of a message store provider is installed.

See Also

MAPIUID structure

IMsgStore::FinishedMsg

The **IMsgStore::FinishedMsg** method enables a message store provider to finish its processing for a sent message. This method is called only by the MAPI spooler.

HRESULT FinishedMsg(
  ULONG *ulFlags***,**
  ULONG *cbEntryID***,**
  LPENTRYID *lpEntryID*
)

Parameters

ulFlags
  Reserved; must be zero.

cbEntryID
  Input parameter containing the size, in bytes, of the entry identifier pointed to by the *lpEntryID* parameter.

lpEntryID
  Input parameter pointing to the entry identifier of the message for which processing is finished.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

MAPI_E_NO_SUPPORT
  This error value is returned if the method is called by any process other than the MAPI spooler.

Remarks

The MAPI spooler calls the **IMsgStore::FinishedMsg** method when it has finished its processing for a message. When MAPI calls a message store provider's **FinishedMsg** implementation, the provider should unlock the message for which processing is complete and, if the PR_DELETE_AFTER_SUBMIT flag is set in the PR_MESSAGE_FLAGS property, delete the message from the folder in which it was last stored. The message store provider should then call the **IMAPISupport::DoSentMail** method. The MAPI spooler never passes the entry identifier for an unlocked message to **FinishedMsg**.

See Also

IMAPISupport::DoSentMail method

IMsgStore::GetOutgoingQueue

The **IMsgStore::GetOutgoingQueue** method returns a table listing the messages in a message store's outgoing queue. This method is called only by the MAPI spooler.

HRESULT GetOutgoingQueue(
  ULONG *ulFlags***,**
  LPMAPITABLE FAR * *lppTable*
)

Parameters

ulFlags
  Reserved; must be zero.

lppTable
  Output parameter pointing to a variable where the pointer to the returned outgoing queue table is stored.

Return Value

S_OK
  The call succeeded and has returned the expected value or values.

Remarks

The MAPI spooler calls the **IMsgStore::GetOutgoingQueue** method to get a pointer to the table showing the queue of outgoing messages for a message store.

Message store providers must present the following required property columns in the outgoing queue table:

PR_CLIENT_SUBMIT_TIME	PR_DISPLAY_BCC
PR_DISPLAY_CC	PR_DISPLAY_TO
PR_ENTRYID	PR_MESSAGE_FLAGS
PR_MESSAGE_SIZE	PR_PRIORITY
PR_SENDER_NAME	PR_SUBJECT
PR_SUBMIT_FLAGS	

Because messages must be preprocessed and submitted to a transport provider in the same order that the client application called the **IMessage::SubmitMessage** method for them, some messages marked for sending by clients might not appear in the outgoing queue table immediately.

The MAPI spooler is designed to accept messages from the message store in ascending order of submission time. Message stores should either allow sorting on the outgoing queue table so the MAPI spooler can sort the messages by submission time, or the default sort order should be by ascending submission time.

The message store provider must set up notifications for the outgoing message queue and ensure the notification callback function is called when the contents of the queue change.

See Also

IMessage::SubmitMessage method

IMsgStore::GetReceiveFolder

The **IMsgStore::GetReceiveFolder** method obtains the receive folder set to receive messages for a particular message class and related information about message reception behavior of that message class.

HRESULT GetReceiveFolder(
  LPTSTR *lpszMessageClass***,**
  ULONG *ulFlags***,**
  ULONG FAR * *lpcbEntryID***,**
  LPENTRYID FAR * *lppEntryID***,**
  LPTSTR FAR * *lppszExplicitClass*
)

Parameters

lpszMessageClass
: Input parameter pointing to a string naming the message class the client application requires information about, for instance IPM.Note.Phone. If the client passes NULL or an empty string in the *lpszMessageClass* parameter, the **GetReceiveFolder** method returns the default receive folder for the message store.

ulFlags
: Input parameter containing a bitmask of flags that controls the type of the passed-in and returned strings. The following flag can be set:

  MAPI_UNICODE
  : Indicates the passed-in and returned strings are in Unicode format. If the MAPI_UNICODE flag is not set, the strings are in ANSI format.

lpcbEntryID
: Output parameter pointing to a variable in which the size, in bytes, of the entry identifier in the *lppEntryID* parameter is returned.

lppEntryID
: Output parameter pointing to a variable where the pointer to the entry identifier returned for the requested receive folder is stored.

lppszExplicitClass
: Output parameter pointing to a pointer to a string naming the message class that explicitly sets as its receive folder the folder indicated by *lppEntryID*. This message class name should either be that of the class indicated by *lpszMessageClass* or of a superclass of that class. Passing NULL indicates that no message class name should be returned and that the folder identified in *lppEntryID* is the default receive folder for the message store.

Return Value

S_OK
: The call succeeded and has returned the expected value or values.

Remarks

Message store providers implement the **IMsgStore::GetReceiveFolder** method to obtain the entry identifier of the folder where the client should put messages of a specific class when they are received. If the message class indicated in *lpszMessageClass* does not explicitly set a receive folder, then **GetReceiveFolder** returns in the *lppszExplicitClass* parameter the name of the first superclass of that message class that does explicitly set a receive folder and in *lppEntryID* the entry identifier of the receive folder that superclass sets.

For example, suppose the receive folder of the message class IPM.Note has been set to the entry identifier of the Inbox and a client calls **GetReceiveFolder** on the message class IPM.Note.Phone. If

IPM.Note.Phone does not have an explicit receive folder set, **GetReceiveFolder** returns the entry identifier of the Inbox in *lppEntryID* and IPM.Note in *lppszExplicitClass*.

If the client calls **GetReceiveFolder** for a message class and has not set a receive folder for that message class, *lppszExplicitClass* is either a zero-length string, a string in Unicode format, or a string in ANSI format depending on whether the client set the MAPI_UNICODE flag in the *ulFlags* parameter.

A default receive folder, obtained by passing NULL in the *lpszMessageClass* parameter, always exists for every message store.

A client should call the **MAPIFreeBuffer** function when it is done with the entry identifier returned in *lppEntryID* to free the memory that holds that entry identifier. It should also call **MAPIFreeBuffer** when it is done with the message class string returned in *lppszExplicitClass* to free the memory that holds that string.

See Also

MAPIFreeBuffer function

IMsgStore::GetReceiveFolderTable

The **IMsgStore::GetReceiveFolderTable** method returns a table that shows property settings for all receive folders for a message store, including which message classes the folders are associated with.

HRESULT GetReceiveFolderTable(
  ULONG *ulFlags***,**
  LPMAPITABLE FAR * *lppTable*
)

Parameters

ulFlags
  Input parameter containing a bitmask of flags that controls how the table is returned. The following flags can be set:

  MAPI_DEFERRED_ERRORS
    Indicates the call is allowed to succeed even if the underlying object is not accessible to the calling implementation. If the object is not accessible, some subsequent call to the object might return an error.

  MAPI_UNICODE
    Indicates the returned strings are in Unicode format. If the MAPI_UNICODE flag is not set, the strings are in ANSI format.

lppTable
  Output parameter pointing to a variable where the pointer to the returned receive folder table is stored.

Return Value

S_OK
  The call succeeded and has returned the expected value or values.

Remarks

Message store providers implement the **IMsgStore::GetReceiveFolderTable** method to get a table showing the property settings for all receive folders in a message store, including which message classes the receive folders are associated with. The table returned contains information for one receive folder in each of its rows and contains the following property columns:

  PR_MESSAGE_CLASS
  PR_RECORD_KEY
  PR_ENTRYID

A message store provider should implement the receive folder table to support setting restrictions to filter for particular values of PR_RECORD_KEY, allowing easy access to particular receive folders.

Setting the MAPI_UNICODE flag in the *ulFlags* parameter does the following:

- Sets the string type to Unicode for data returned for the initial active columns of the receive folder table by the **IMAPITable::QueryColumns** method. The initial active columns for a receive folder table are those columns **QueryColumns** returns before the provider that contains the table calls the **IMAPITable::SetColumns** method.
- Sets the string type to Unicode for data returned for the initial active rows of the receive folder table by the **IMAPITable::QueryRows** method. The initial active rows for a receive folder table are those rows **QueryRows** returns before the provider that contains the table calls **SetColumns**.
- Controls the property types of the sort order returned by the **IMAPITable::QuerySortOrder** method before the provider that contains the receive folder table calls the **IMAPITable::SortTable** method.

See Also

IMAPITable::QueryColumns method, **IMAPITable::QueryRows** method, **IMAPITable::QuerySortOrder** method, **IMAPITable::SetColumns** method

IMsgStore::NotifyNewMail

The **IMsgStore::NotifyNewMail** method notifies a message store that a new message has arrived. This method is called only by the MAPI spooler.

HRESULT NotifyNewMail(
  LPNOTIFICATION *lpNotification*
)

Parameters

lpNotification
  Input parameter pointing to the **NOTIFICATION** structure holding the new-message notification.

Return Value

S_OK
  The call succeeded and has returned the expected value or values.

Remarks

A message store provider implements the **IMsgStore::NotifyNewMail** method so that the MAPI spooler can notify the provider that it has received a message for delivery to the store. Although the MAPI spooler handles all notifications of changes sent to the message store provider, the provider is responsible for notifying client applications. After the MAPI spooler calls a message store's **NotifyNewMail** implementation, the store provider can then inform any clients that have registered for new mail notification about the new message. To register for notification about new messages, a client includes the `fnevNewMail` event type in the *ulEventMask* parameter passed in its call to the **IMsgStore::Advise** method. Message stores are free to implement their own client notification scheme; one way is to use the **IMAPISupport::Subscribe** and **IMAPISupport::Unsubscribe** methods.

The MAPI spooler allocates memory for the **NOTIFICATION** structure that holds information about the new mail notification, and that memory should not be released, kept, or modified by the message store. If a store must use the **NOTIFICATION** structure, it must copy it; the utility function **ScCopyNotifications** is provided as one way to perform the copy operation.

See Also

IMAPISupport::Subscribe method, **IMAPISupport::Unsubscribe** method, **NOTIFICATION** structure, **ScCopyNotifications** function

IMsgStore::OpenEntry

The **IMsgStore::OpenEntry** method opens a folder or a message in a message store given its entry identifier.

HRESULT OpenEntry(
  ULONG *cbEntryID***,**
  LPENTRYID *lpEntryID***,**
  LPCIID *lpInterface***,**
  ULONG *ulFlags***,**
  ULONG FAR * *lpulObjType***,**
  LPUNKNOWN FAR * *lppUnk*
)

Parameters

cbEntryID
  Input parameter containing the size, in bytes, of the entry identifier pointed to by the *lpEntryID* parameter.

lpEntryID
  Input parameter pointing to the entry identifier of the folder or message to open. The root folder of the store can be opened either with its entry identifier, as with any other folder, or by passing NULL for *lpEntryID*.

lpInterface
  Input parameter pointing to the interface identifier (IID) for the object to open. Passing NULL indicates the object is cast to the standard interface for such an object. The *lpInterface* parameter can also be set to an identifier for an appropriate interface for the object.

ulFlags
  Input parameter containing a bitmask of flags that controls how the object is opened. The following flags can be set:

  MAPI_BEST_ACCESS
    Indicates the object should be opened with the maximum network permissions allowed for the user and the maximum client application access. For example, if a client has read/write access, the object is opened with read/write access; if the client has read-only access, the object is opened with read-only access. The client can retrieve the access level by getting the PR_ACCESS_LEVEL property.

  MAPI_DEFERRED_ERRORS
    Indicates the call is allowed to succeed even if the underlying object is not accessible to the calling client. If the object is not accessible, some subsequent call to the object might return an error.

  MAPI_MODIFY
    Requests read/write access. By default, objects are created with read-only access, and clients should not work on the assumption that read/write access has been granted.

lpulObjType
  Output parameter pointing to a variable where the type of the opened object is stored.

lppUnk
  Output parameter pointing to a variable where the pointer to the opened object is stored.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

MAPI_E_NO_ACCESS
  An attempt was made to modify a read-only object or an attempt to access an object for which the

user has insufficient permissions.

Remarks

Message store providers implement the **IMsgStore::OpenEntry** method to open a folder or a message in the message store. The calling implementation passes in the entry identifier of the object to open, and **OpenEntry** returns a pointer in the *lppUnk* parameter that provides further access to the object. **OpenEntry**'s behavior is to open the object as read-only, unless the call sets the MAPI_MODIFY or MAPI_BEST_ACCESS flag in the *ulFlags* parameter.

Although a client can use **IMsgStore::OpenEntry** to open any folder or message, it is usually faster if the calling client instead uses the **IMAPIContainer::OpenEntry** method on the folder containing the object being opened.

The calling client should check the value returned in the *lpulObjType* parameter to determine that the object type returned is what was expected. The object types, as included in PR_OBJECT_TYPE, are MAPI_MESSAGE and MAPI_FOLDER. Commonly, after the client checks the type of the object, it then casts the pointer in the *lppUnk* parameter into a message object pointer, a folder object pointer, or another appropriate object pointer.

See Also

IMAPIContainer::OpenEntry method

IMsgStore::SetLockState

The **IMsgStore::SetLockState** method allows the MAPI spooler to lock or unlock a message.

HRESULT SetLockState(
  LPMESSAGE *lpMessage***,**
  ULONG *ulLockState*
)

Parameters

lpMessage
  Input parameter pointing to the message to lock or unlock.

ulLockState
  Input parameter containing a value indicating whether the MAPI spooler is locking or unlocking the message. One of the following values can be used with the *ulLockState* parameter:

  MSG_LOCKED
    Locks the message.

  MSG_UNLOCKED
    Releases a previous lock on the message.

Return Value

S_OK
  The call succeeded and has returned the expected value or values.

Remarks

The MAPI spooler calls the **IMsgStore::SetLockState** method to lock a message while it is sending the message. The MAPI spooler locks the message so that calls to the **IMsgStore::AbortSubmit** method cannot succeed. **SetLockState** also provides the MAPI spooler with a mechanism to unlock messages it previously locked. If the MAPI spooler passes the MSG_LOCKED value in *ulLockState*, the message affected is locked; if the MAPI spooler passes the MSG_UNLOCKED value in *ulLockState*, the message affected is unlocked.

Usually, when the MAPI spooler calls **SetLockState** to lock a message, it only locks the oldest message – that is, the next message queued for the MAPI spooler to send. If the oldest message in the queue is waiting for a temporarily unavailable transport provider, and the next message in the queue uses a different transport provider, the MAPI spooler can begin processing the later message. It begins processing by locking that message using **SetLockState**.

The message store provider can call the **IMAPIProp::SaveChanges** method as a part of its response to the **SetLockState** call so that any changes made to the message before the **SetLockState** call was received are saved.

The MAPI spooler unlocks the message before calling the **IMsgStore::FinishedMsg** method.

See Also

IMsgStore::AbortSubmit method, **IMsgStore::FinishedMsg** method

IMsgStore::SetReceiveFolder

The **IMsgStore::SetReceiveFolder** method sets the receive folder for a particular message class.

HRESULT SetReceiveFolder(
  LPTSTR *lpszMessageClass***,**
  ULONG *ulFlags***,**
  ULONG *cbEntryID***,**
  LPENTRYID *lpEntryID*
)

Parameters

lpszMessageClass
  Input parameter pointing to a string naming the message class for which the receive folder should be set. If a client application passes NULL or an empty string in the *lpszMessageClass* parameter, it sets the message class's receive folder to the default for the message store provider.

ulFlags
  Input parameter containing a bitmask of flags that controls the type of the text in the passed-in strings. The following flag can be set:

  MAPI_UNICODE
    Indicates the passed-in strings are in Unicode format. If the MAPI_UNICODE flag is not set, the strings are in ANSI format.

cbEntryID
  Input parameter containing the size, in bytes, of the entry identifier pointed to by the *lpEntryID* parameter.

lpEntryID
  Input parameter pointing to the entry identifier of the folder to set as the receive folder. Passing NULL in *lpEntryID* indicates that the current receive folder setting is replaced with the message store's default.

Return Value

S_OK
  The call succeeded and has returned the expected value or values.

Remarks

Message store providers implement the **IMsgStore::SetReceiveFolder** method to set or change the receive folder for a particular message class. With **SetReceiveFolder**, an application can, by using successive calls, specify a different receive folder for each message class defined or specify that incoming messages for multiple message classes all go to the same folder. A client can, for example, have its own class of messages arrive in its own folder. For instance, a fax application can designate a folder where the store provider places incoming faxes and one where the provider places outgoing faxes.

If an error occurs while calling **SetReceiveFolder**, the receive folder setting remains unchanged.

If **SetReceiveFolder** changes the receive folder setting with *lpEntryID* set to NULL, indicating the default receive folder should be set, **SetReceiveFolder** returns S_OK even if there was no existing setting for the indicated message class.

IMsgStore::StoreLogoff

The **IMsgStore::StoreLogoff** method enables the orderly logoff of a message store under client application control.

HRESULT StoreLogoff(
  ULONG FAR * *lpulFlags*
)

Parameters

lpulFlags
  Input-output parameter containing a bitmask of flags that controls logoff from the message store. On input, all flags set for this parameter are mutually exclusive; a client must specify only one flag per call. A client can set the following flags on input:

  LOGOFF_ABORT
    Indicates any transport provider activity for this store should be stopped before logoff. Control is returned to the client after activity is stopped. If any transport provider activity is taking place, the logoff does not occur and no change in the behavior of the MAPI spooler or transport providers occurs. If transport provider activity is quiet, the MAPI spooler releases the store.

  LOGOFF_NO_WAIT
    Indicates the message store should not wait for messages from transport providers before closing. All outbound mail that is ready to be sent, is sent; if this store contains the default Inbox, any in-process message is received, and then further reception is disabled. When all activity is completed, the MAPI spooler releases the store, and control is returned to the client immediately.

  LOGOFF_ORDERLY
    Indicates the message store should not wait for information from transport providers before closing. Any message being processed by the store is completed, and no new messages are processed. When all activity is completed, the MAPI spooler releases the store, and control is returned to the store provider immediately.

  LOGOFF_PURGE
    Works the same as LOGOFF_NO_WAIT but also calls the **IXPLogon::FlushQueues** method or the **IMAPIStatus::FlushQueues** method for the appropriate transport providers. The LOGOFF_PURGE flag returns control to the client after completion.

  LOGOFF_QUIET
    Indicates that if any transport provider activity is taking place, the logoff does not occur.

On output the following flags can be returned:

  LOGOFF_INBOUND
    Indicates that there are inbound messages arriving.

  LOGOFF_OUTBOUND
    Indicates that there are outbound messages in the process of being sent.

  LOGOFF_OUTBOUND_QUEUE
    Indicates that there are outbound messages pending, that is, in the Outbox.

Return Value

S_OK
  The call succeeded and has returned the expected value or values.

Remarks

Client applications call the **IMsgStore::StoreLogoff** method to exert some control over how transport providers log off when the message store object is released. After **StoreLogoff** returns with S_OK, any further calls to **StoreLogoff** are ignored.

The client can either set requirements about message store and transport provider interaction at message store release by setting the appropriate flags, or it can allow MAPI either to stop sending messages at message store release or to complete sending before message store release.

A client is only allowed to control transport provider logoff if it is the only implementation using the message store. If another client is still using the store, the store object is immediately released and control is returned to the client.

Message store providers implement **StoreLogoff** as a way to record the flags that the client requires passed to the **IMAPISupport::StoreLogoffTransports** method. Store providers should not actually make the **StoreLogoffTransports** call until the reference count on the message store object drops to zero, during the final release of the message store object. Multiple calls to **StoreLogoffTransports** simply overwrite the saved flags.

By default, if no clients call **StoreLogoff** before the reference count on the message store reaches zero, the store provider should set the LOGOFF_ABORT flag in the *ulFlags* parameter that it passes on the call to **StoreLogoffTransports**, which conducts logoff as described earlier.

See Also

IMAPIStatus::FlushQueues method, **IMAPISupport::StoreLogoffTransports** method, **IXPLogon::FlushQueues** method

IMsgStore::Unadvise

The **IMsgStore::Unadvise** method removes an object's registration for notification of message store changes previously established with a call to the **IMsgStore::Advise** method.

HRESULT Unadvise(
  ULONG *ulConnection*
)

Parameters

ulConnection
  Input parameter containing the number of the registration connection returned by a call to **IMsgStore::Advise**.

Return Value

S_OK
  The call succeeded and has returned the expected value or values.

Remarks

Message store providers implement the **IMsgStore::Unadvise** method to release the pointer to the advise sink object passed in the *lpAdviseSink* parameter in the previous call to **IMsgStore::Advise**, thereby canceling a notification registration. As part of discarding the pointer to the advise sink object, the object's **IUnknown::Release** method is called. Generally, **Release** is called during the **Unadvise** call, but if another thread is in the process of calling the **IMAPIAdviseSink::OnNotify** method for the advise sink object, the **Release** call is delayed until the **OnNotify** method returns.

See Also

IMAPIAdviseSink::OnNotify method, **IMsgStore::Advise** method

IMSLogon : IUnknown

The **IMSLogon** interface is used to access resources in a message store logon object. The message store logon object is the part of an open message store provider that is called directly by MAPI. There is a one-to-one correspondence between the message store logon object called by MAPI and the message store object called by client applications; you can think of the logon and store objects as one object exposing two interfaces. The two objects are created together and freed together.

At a Glance

Specified in header file:	MAPISPI.H
Object that supplies this interface:	Message store logon object
Corresponding pointer type:	LPMSLOGON
Implemented by:	Message store providers
Called by:	MAPI

Vtable Order

GetLastError	Returns a **MAPIERROR** structure containing information about the last error that occurred for a message store logon object.
Logoff	Logs off a message store provider.
OpenEntry	Opens a folder or message object and returns a pointer to the object to provide further access.
CompareEntryIDs	Compares two entry identifiers to determine if they refer to the same object.
Advise	Registers a message store provider for notifications about changes within the message store.
Unadvise	Removes an object's registration for notification of message store changes previously established with a call to the **IMSLogon::Advise** method.
OpenStatusEntry	Opens a status object.

IMSLogon::Advise

The **IMSLogon::Advise** method registers a message store provider for notifications about changes within the message store.

HRESULT Advise(
  ULONG *cbEntryID*,
  LPENTRYID *lpEntryID*,
  ULONG *ulEventMask*,
  LPMAPIADVISESINK *lpAdviseSink*,
  ULONG FAR * *lpulConnection*
)

Parameters

cbEntryID
  Input parameter containing the size, in bytes, of the entry identifier pointed to by the *lpEntryID* parameter.

lpEntryID
  Input parameter pointing to the entry identifier of the object about which notifications should be generated. This object can be a folder, a message, or any other object in the message store. Alternatively, if MAPI sets the *cbEntryID* parameter to zero and passes NULL for *lpEntryID*, the advise sink provides notifications about changes to the entire message store.

ulEventMask
  Input parameter containing an event mask of the types of notification events occurring for the object about which MAPI will generate notifications. The mask filters specific cases. Each event type has a structure associated with it that holds additional information about the event. The following table lists the possible event types along with their corresponding structures.

Notification event type	Corresponding structure
`fnevCriticalError`	**ERROR_NOTIFICATION**
`fnevNewMail`	**NEWMAIL_NOTIFICATION**
`fnevObjectCreated`	**OBJECT_NOTIFICATION**
`fnevObjectDeleted`	**OBJECT_NOTIFICATION**
`fnevObjectModified`	**OBJECT_NOTIFICATION**
`fnevObjectCopied`	**OBJECT_NOTIFICATION**
`fnevObjectMoved`	**OBJECT_NOTIFICATION**
`fnevSearchComplete`	**OBJECT_NOTIFICATION**
`fnevStatusObjectModified`	**STATUS_OBJECT_NOTIFICATION**

lpAdviseSink
  Input parameter pointing to an advise sink object to be called when an event occurs for the session object about which notification has been requested. This advise sink object must have already been allocated.

lpulConnection
  Output parameter pointing to a variable that upon a successful return holds the connection number for the notification registration. The connection number must be nonzero.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

MAPI_E_NO_SUPPORT

The operation is not supported by MAPI or by one or more service providers.

Remarks

Message store providers implement the **IMSLogon::Advise** method to register an object for notification callbacks. Whenever a change occurs to the indicated object, the provider checks to see what event mask bit was set in the *ulEventMask* parameter and thus what type of change occurred. If a bit is set, then the provider calls the **IMAPIAdviseSink::OnNotify** method for the advise sink object indicated by the *lpAdviseSink* parameter to report the event. Data passed in the notification structure to the **OnNotify** routine describes the event.

The call to **OnNotify** can occur during the call that changes the object, or at any following time. On systems that support multiple threads of execution, the call to **OnNotify** can occur on any thread. For a way to turn a call to **OnNotify** that might happen at an inopportune time into one that is safer to handle, a client application should use the **HrThisThreadAdviseSink** function.

To provide notifications, the message store provider implementing **Advise** needs to keep a copy of the pointer to the *lpAdviseSink* advise sink object; to do so, it calls the **IUnknown::AddRef** method for the advise sink to maintain its object pointer until notification registration is canceled with a call to the **IMSLogon::Unadvise** method. The **Advise** implementation should assign a connection number to the notification registration and call **AddRef** on this connection number before returning it in the *lpulConnection* parameter. Service providers can release the advise sink object before the registration is canceled, but they must not release the connection number until **Unadvise** has been called.

After a call to **Advise** has succeeded and before **Unadvise** has been called, providers must be prepared for the advise sink object to be released. A provider should therefore release its advise sink object after **Advise** returns unless it has a specific long-term use for it.

For more information about the notification process, see About Notification.

See Also

HrThisThreadAdviseSink function, **IMAPIAdviseSink::OnNotify** method, **IMSLogon::Unadvise** method, **NOTIFICATION** structure

IMSLogon::CompareEntryIDs

The **IMSLogon::CompareEntryIDs** method compares two entry identifiers to determine if they refer to the same object. MAPI only passes this call to a service provider if the unique identifiers (UIDs) in both entry identifiers to be compared are handled by that provider.

HRESULT CompareEntryIDs(
  ULONG *cbEntryID1***,**
  LPENTRYID *lpEntryID1***,**
  ULONG *cbEntryID2***,**
  LPENTRYID *lpEntryID2***,**
  ULONG *ulFlags***,**
  ULONG FAR * *lpulResult*
)

Parameters

cbEntryID1
  Input parameter containing the size, in bytes, of the entry identifier pointed to by the *lpEntryID1* parameter.

lpEntryID1
  Input parameter pointing to the first entry identifier to be compared.

cbEntryID2
  Input parameter containing the size, in bytes, of the entry identifier pointed to by the *lpEntryID2* parameter.

lpEntryID2
  Input parameter pointing to the second entry identifier to be compared.

ulFlags
  Reserved; must be zero.

lpulResult
  Output parameter pointing to a variable where the returned result of the comparison is stored; this variable is TRUE if the two entry identifiers refer to the same object, and FALSE otherwise.

Return Value

S_OK
  The call succeeded and has returned the expected value or values.

Remarks

Message store providers implement the **IMSLogon::CompareEntryIDs** method to compare two entry identifiers for a given entry within a message store to determine whether they refer to the same object. If the two entry identifiers refer to the same object, then **CompareEntryIDs** sets the *lpulResult* parameter to TRUE; if they refer to different objects, **CompareEntryIDs** sets *lpulResult* to FALSE.

CompareEntryIDs is useful because an object can have more than one valid entry identifier; such a situation can occur, for example, after a new version of a message store provider is installed.

IMSLogon::GetLastError

The **IMSLogon::GetLastError** method returns a **MAPIERROR** structure containing information about the last error that occurred for the message store object.

HRESULT GetLastError(
  HRESULT *hResult*,
  ULONG *ulFlags*,
  LPMAPIERROR FAR * *lppMAPIError*
)

Parameters

hResult
  Input parameter containing the result returned for the last call for the message store object that returned an error.

ulFlags
  Input parameter containing a bitmask of flags that controls the type of the returned strings. The following flag can be set:

  MAPI_UNICODE
    Indicates the strings in the **MAPIERROR** structure returned in the *lppMAPIError* parameter are in Unicode format. If the MAPI_UNICODE flag is not set, the strings are in ANSI format.

lppMAPIError
  Output parameter pointing to a pointer to the returned **MAPIERROR** structure containing version, component, and context information for the error. The *lppMAPIError* parameter can be set to NULL if there is no **MAPIERROR** structure to return.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

MAPI_E_BAD_CHARWIDTH
  Either the MAPI_UNICODE flag was set and the implementation does not support Unicode, or MAPI_UNICODE was not set and the implementation only supports Unicode.

Remarks

Use the **IMSLogon::GetLastError** method to retrieve information to display in a message to the user regarding the last error returned from a method call for the message store object.

To release all the memory allocated by MAPI for the returned **MAPIERROR** structure, client applications need only call the **MAPIFreeBuffer** function.

The return value from **GetLastError** must be S_OK for an application to make use of the **MAPIERROR** structure. Even if the return value is S_OK, a **MAPIERROR** structure might not be returned. If the implementation cannot determine what the last error was, or if a **MAPIERROR** structure is not available for that error, **GetLastError** returns a pointer to NULL in *lppMAPIError* instead.

See Also

MAPIERROR structure, **MAPIFreeBuffer** function

IMSLogon::Logoff

The **IMSLogon::Logoff** method logs off a message store provider.

HRESULT Logoff(
  ULONG FAR * *lpulFlags*
)

Parameters

lpulFlags
  Reserved; must be a pointer to zero.

Return Value

S_OK
  The call succeeded and has returned the expected value or values.

Remarks

Message store providers implement the **IMSLogon::Logoff** method to forcibly shut down a message store provider. **IMSLogon::Logoff** is called in the following situations:

- While MAPI is logging off a client after a call to the **IMAPISession::Logoff** method.
- While MAPI is logging off a message store provider. In this case, **IMSLogon::Logoff** is called as part of MAPI's processing the **IUnknown::Release** method of the support object that the message store provider creates while it is processing an **IMsgStore::StoreLogoff** or **IUnknown::Release** method call on a message store object.

See Also

IMAPISession::Logoff method, **IMAPISupport : IUnknown** interface, **IMsgStore::StoreLogoff** method, **IMSProvider::Logon** method, **MAPIFreeBuffer** function

IMSLogon::OpenEntry

The **IMSLogon::OpenEntry** method opens a folder or message object and returns a pointer to the object to provide further access.

HRESULT OpenEntry(
  ULONG *cbEntryID*,
  LPENTRYID *lpEntryID*,
  LPCIID *lpInterface*,
  ULONG *ulOpenFlags*,
  ULONG FAR * *lpulObjType*,
  LPUNKNOWN FAR * *lppUnk*
)

Parameters

cbEntryID
  Input parameter containing the size, in bytes, of the entry identifier pointed to by the *lpEntryID* parameter.

lpEntryID
  Input parameter pointing to the address of the entry identifier of the folder or message object to open.

lpInterface
  Input parameter pointing to the interface identifier (IID) for the object. Passing NULL indicates the object is cast to the standard interface for such an object. The *lpInterface* parameter can also be set to an identifier for an appropriate interface for the object.

ulOpenFlags
  Input parameter containing a bitmask of flags that controls how the object is opened. The following flags can be set:

  MAPI_BEST_ACCESS
    Indicates the object should be opened with the maximum network permissions allowed for the user and the maximum client application access. For example, if the client has read/write access, the object is opened with read/write access; if the client has read-only access, the object is opened with read-only access. The client can retrieve the access level by getting the PR_ACCESS_LEVEL property.

  MAPI_DEFERRED_ERRORS
    Indicates the call is allowed to succeed even if the underlying object is not accessible to the calling application. If the object is not accessible, some subsequent call to the object might return an error.

  MAPI_MODIFY
    Requests read/write access. By default, objects are created with read-only access, and clients should not work on the assumption that read/write access has been granted.

lpulObjType
  Output parameter pointing to a variable where the type of the opened object is stored.

lppUnk
  Output parameter pointing to a variable where the pointer to the opened object is stored.

Return Value

S_OK
  The call succeeded and has returned the expected value or values.

Remarks

MAPI calls the **IMSLogon::OpenEntry** method to open a folder or a message in a message store. MAPI passes in the entry identifier of the object to open; the message store provider should return a pointer providing further access to the object in the *lppUnk* parameter.

Before MAPI calls **IMSLogon::OpenEntry**, it first determines that the given message or folder entry identifier matches one registered by this message store provider. For more information on how store providers register entry identifiers, see **IMAPISupport::SetProviderUID**.

IMSLogon::OpenEntry is identical to the **IMsgStore::OpenEntry** method of the message store object but is called by MAPI when processing an **IMAPISession::OpenEntry** method call instead of being called by a client. Objects opened using **IMSLogon::OpenEntry** should be treated exactly the same as objects opened with the message store object; in particular, objects opened using this call should be invalidated when the message store object is released.

See Also

IMAPISupport::SetProviderUID method, **IMsgStore::OpenEntry** method

IMSLogon::OpenStatusEntry

The **IMSLogon::OpenStatusEntry** method opens a status object.

HRESULT OpenStatusEntry(
  LPCIID *lpInterface***,**
  ULONG *ulFlags***,**
  ULONG FAR * *lpulObjType***,**
  LPVOID FAR * *lppEntry*
)

Parameters

lpInterface
  Input parameter pointing to the interface identifier (IID) for the status object to open. Passing NULL indicates the standard interface for the object is returned, in this case the **IMAPIStatus** interface. The *lpInterface* parameter can also be set to an identifier for an appropriate interface for the object.

ulFlags
  Input parameter containing a bitmask of flags that controls how the status object is opened. The following flag can be set:

  MAPI_MODIFY
    Requests read/write access. By default, objects are created with read-only access, and client applications should not work on the assumption that read/write access has been granted.

lpulObjType
  Output parameter pointing to a variable where the type of the opened object is stored.

lppEntry
  Output parameter pointing to a variable where the pointer to the opened object is stored.

Return Value

S_OK
  The call succeeded and has returned the expected value or values.

Remarks

Message store providers implement the **IMSLogon::OpenStatusEntry** method to open a status object. This status object is then used to enable clients to call **IMAPIStatus** methods; for example, clients can use the **IMAPIStatus::SettingsDialog** method to reconfigure the message store logon session or the **IMAPIStatus::ValidateState** method to validate the state of the message store logon session.

See Also

IMAPIStatus : IMAPIProp interface, **IMAPIStatus::SettingsDialog** method, **IMAPIStatus::ValidateState** method

IMSLogon::Unadvise

The **IMSLogon::Unadvise** method removes an object's registration for notification of message store changes previously established with a call to the **IMSLogon::Advise** method.

HRESULT Unadvise(
  ULONG *ulConnection*
)

Parameters

ulConnection
  Input parameter containing the number of the registration connection returned by a call to **IMSLogon::Advise**.

Return Value

S_OK
  The call succeeded and has returned the expected value or values.

Remarks

Message store providers implement the **IMSLogon::Unadvise** method to release the pointer to the advise sink object passed in the *lpAdviseSink* parameter in the previous call to **IMSLogon::Advise**, thereby canceling a notification registration. As part of discarding the pointer to the advise sink object, the object's **IUnknown::Release** method is called. Generally, **Release** is called during the **Unadvise** call, but if another thread is in the process of calling the **IMAPIAdviseSink::OnNotify** method for the advise sink object, the **Release** call is delayed until the **OnNotify** method returns.

See Also

IMAPIAdviseSink::OnNotify method, **IMSLogon::Advise** method

IMSProvider : IUnknown

The **IMSProvider** interface provides access to a message store provider through a message store provider object. This message store provider object is returned at provider logon by the message store provider's **MSProviderInit** entry point function. The message store provider object is primarily used by client applications and the MAPI spooler to open message stores.

MAPI uses one message store provider object for all the message stores opened by the store provider implementing the object for the current MAPI session. If a second MAPI session logs onto any open stores, MAPI calls **MSProviderInit** a second time to create a new message store provider object for that session to use.

A message store provider object must contain the following to operate correctly:

- An *lpMalloc* memory-allocation pointer for use by all stores opened using this provider object.
- The *lpfAllocateBuffer, lpfAllocateMore,* and *lpfFreeBuffer* routine pointers to the **MAPIAllocateBuffer**, **MAPIAllocateMore**, and **MAPIFreeBuffer** memory allocation functions.
- A linked list of all the stores opened using this provider object and not yet closed.

At a Glance

Specified in header file:	MAPISPI.H
Object that supplies this interface:	Message store provider object
Corresponding pointer type:	LPMSPROVIDER
Implemented by:	Message store providers
Called by:	MAPI, the MAPI spooler

Vtable Order

Shutdown	Closes down a message store provider in an orderly fashion.
Logon	Logs MAPI on to one instance of a message store provider.
SpoolerLogon	Logs the MAPI spooler on to a message store.
CompareStoreIDs	Compares two message store entry identifiers to determine if they refer to the same store object.

IMSProvider::CompareStoreIDs

The **IMSProvider::CompareStoreIDs** method compares two message store entry identifiers to determine if they refer to the same store object.

HRESULT CompareStoreIDs(
  ULONG *cbEntryID1*,
  LPENTRYID *lpEntryID1*,
  ULONG *cbEntryID2*,
  LPENTRYID *lpEntryID2*,
  ULONG *ulFlags*,
  ULONG FAR * *lpulResult*
)

Parameters

cbEntryID1
 Input parameter containing the size, in bytes, of the entry identifier pointed to by the *lpEntryID1* parameter.

lpEntryID1
 Input parameter pointing to the first entry identifier to be compared.

cbEntryID2
 Input parameter containing the size, in bytes, of the entry identifier pointed to by the *lpEntryID2* parameter.

lpEntryID2
 Input parameter pointing to the second entry identifier to be compared.

ulFlags
 Reserved; must be zero.

lpulResult
 Output parameter pointing to a variable where the returned result of the comparison is stored; this variable is TRUE if the two entry identifiers refer to the same object, and FALSE otherwise.

Return Value

S_OK
 The call succeeded and has returned the expected value or values.

Remarks

MAPI calls the **IMSProvider::CompareStoreIDs** method while processing a call to the **IMAPISession::OpenMsgStore** method. **CompareStoreIDs** is called at this point to determine which profile section, if any, is associated with the message store being opened. A **CompareStoreIDs** call can be made when no message stores are open for a particular store provider. In addition, MAPI also calls **CompareStoreIDs** while processing a store provider call to the **IMAPISupport::OpenProfileSection** method.

The entry identifiers compared by **CompareStoreIDs** are both for the current store provider's dynamic-link library (DLL) and are both unwrapped store entry identifiers. For more information on wrapping store entry identifiers, see **IMAPISupport::WrapStoreEntryID**.

Comparing entry identifiers is useful because an object can have more than one valid entry identifier; such a situation can occur, for example, after a new version of a message store provider is installed.

See Also

IMAPISession::OpenMsgStore method, **IMAPISupport::OpenProfileSection** method, **IMAPISupport::WrapStoreEntryID** method

IMSProvider::Logon

The **IMSProvider::Logon** method logs MAPI onto one instance of a message store provider.

HRESULT Logon(
  LPMAPISUP *lpMAPISup*,
  ULONG *ulUIParam*,
  LPTSTR *lpszProfileName*,
  ULONG *cbEntryID*,
  LPENTRYID *lpEntryID*,
  ULONG *ulFlags*,
  LPCIID *lpInterface*,
  ULONG FAR * *lpcbSpoolSecurity*,
  LPBYTE FAR * *lppbSpoolSecurity*,
  LPMAPIERROR FAR * *lppMAPIError*,
  LPMSLOGON FAR * *lppMSLogon*,
  LPMDB FAR * *lppMDB*
)

Parameters

lpMAPISup
: Input parameter pointing to the current MAPI support object for the message store.

ulUIParam
: Input parameter containing the handle of the parent window for any dialog boxes or windows this method displays.

lpszProfileName
: Input parameter pointing to a string containing the name of the profile being used for store provider logon. This string can be displayed in dialog boxes, written out to a log file, or simply ignored. It must be in Unicode format if the MAPI_UNICODE flag is set in the *ulFlags* parameter.

cbEntryID
: Input parameter containing the size, in bytes, of the entry identifier pointed to by the *lpEntryID* parameter.

lpEntryID
: Input parameter pointing to the entry identifier for the message store. Passing NULL in *lpEntryID* indicates that a message store has not yet been selected and that dialog boxes enabling the user to select a message store can be presented.

ulFlags
: Input parameter containing a bitmask of flags that controls how the logon is performed. The following flags can be set:

MAPI_DEFERRED_ERRORS
: Indicates the call is allowed to succeed even if the underlying object is not accessible to the calling implementation. If the object is not accessible, some subsequent call to the object might return an error.

MAPI_UNICODE
: Indicates the passed-in strings are in Unicode format. If MAPI_UNICODE is not set, the strings are in ANSI format.

MDB_NO_DIALOG
: Prevents display of logon dialog boxes. If this flag is set, the error value MAPI_E_LOGON_FAILED is returned if logon is unsuccessful. If this flag is not set, the message store provider can prompt the user to correct a name or password, to insert a disk, or to perform other actions necessary to establish connection to the store.

MDB_NO_MAIL

Indicates the message store should not be used for sending or receiving mail. The flag signals MAPI not to notify the MAPI spooler that this message store is being opened. If this flag is set, and the message store is tightly coupled with a transport provider, then the provider does not need to call the **IMAPISupport::SpoolerNotify** method.

MDB_TEMPORARY
Logs on the store so that information can be retrieved programmatically from the profile section, without use of dialog boxes. This flag instructs MAPI that the store is not to be added to the message store table and that the store cannot be made permanent. If this flag is set, message store providers do not need to call the **IMAPISupport::ModifyProfile** method.

MDB_WRITE
Requests read/write access.

lpInterface
Input parameter pointing to the interface identifier (IID) for the message store to log on to. Passing NULL indicates the MAPI interface for the message store is returned – that is, the **IMsgStore** interface. The *lpInterface* parameter can also be set to an identifier for an appropriate interface for the message store, for example IID_IUnknown or IID_IMAPIProp.

lpcbSpoolSecurity
Output parameter pointing to the variable where the store provider returns the size, in bytes, of the validation data in the *lppbSpoolSecurity* parameter.

lppbSpoolSecurity
Output parameter pointing to a variable where the pointer to the returned validation data is stored. This validation data is provided so the **IMSProvider::SpoolerLogon** method can log the MAPI spooler on to the same store as the message store provider.

lppMAPIError
Output parameter pointing to a pointer to the returned **MAPIERROR** structure, if any, containing version, component, and context information for an error. The *lppMAPIError* parameter can be set to NULL if there is no **MAPIERROR** structure to return.

lppMSLogon
Output parameter pointing to a variable where the pointer to the message store logon object for MAPI to log on to is stored.

lppMDB
Output parameter pointing to a variable where the pointer to the message store object for the MAPI spooler and client applications to log on to is stored.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

MAPI_E_FAILONEPROVIDER
Indicates this provider cannot log on, but this error should not disable the service.

MAPI_E_LOGON_FAILED
A logon session could not be established.

MAPI_E_UNCONFIGURED
The profile does not contain enough information for the logon to complete. When this value is returned, MAPI calls the message store provider's message-service entry point function.

MAPI_E_USER_CANCEL
The user canceled the operation, typically by clicking the **Cancel** button in a dialog box.

MAPI_E_UNKNOWN_CPID
Indicates the server is not configured to support the client's code page.

MAPI_E_UNKNOWN_LCID
Indicates the server is not configured to support the client's locale information.

MAPI_W_ERRORS_RETURNED

The call succeeded, but the message store provider has error information available. To test for this warning, use the **HR_FAILED** macro. When this warning is returned, the call should be handled as successful. To get the error information from the provider, call the **IMAPISession::GetLastError** method.

Remarks

MAPI calls the **IMSProvider::Logon** method to do the majority of processing necessary to obtain access to a message store. Message store providers validate any user credentials necessary to access a particular store and return a message store object in the *lppMDB* parameter that the MAPI spooler and client applications can log onto.

In addition to the message store object returned for client and MAPI spooler use, the provider also returns a message store logon object for MAPI's use in controlling the opened store. The message store logon object and the message store object should be tightly linked inside the message store provider so each can affect the other. Usage of the store object and the logon object should be identical; there should be a one-to-one correspondence between the logon object and the store object such that the objects act as if they are one object exposing two interfaces. The two objects should also be created together and freed together.

The MAPI support object, created by MAPI and passed to the provider in the *lpMAPISup* parameter, provides access to functions in MAPI required by the provider. These include functions that save and retrieve profile information, access address books, and so on. The *lpMAPISup* pointer can be different for each store that is opened. While processing calls for a message store after logon, the store provider should use the *lpMAPISup* variable specific to that store. For any **Logon** call that opens a message store and succeeds in creating a message store logon object, the provider must save a pointer to the MAPI support object in the store logon object and must call the **IUnknown::AddRef** method to add a reference for the support object.

The *ulUIParam* parameter should be used if the provider presents dialog boxes during the **Logon** call. However, dialog boxes should not be presented if *ulFlags* contains the MDB_NO_DIALOG flag. If a user interface is called but *ulFlags* does not allow it, or if for some other reason a user interface cannot be displayed, the provider should return MAPI_E_LOGON_FAILED. If **Logon** displays a dialog box and the user cancels logon, typically by clicking the dialog box's **Cancel** button, the provider should return MAPI_E_USER_CANCEL.

The *lpEntryID* parameter can either be NULL or point to an unwrapped store entry identifier previously created by this message store. If *lpEntryID* points to an unwrapped entry identifier, that entry identifier can come from one of several places:

- It can be an entry identifier that the store provider previously wrapped and wrote to the profile section as a PR_ENTRYID property.
- It can be an entry identifier that the provider previously wrapped and returned to a calling client as a PR_STORE_ENTRYID property.
- It can be an entry identifier that the provider previously wrapped and returned to a calling client as the PR_ENTRYID property of a message store object.

In any of these cases, it is possible that the entry identifier was created on a different computer than the one currently being used.

When *lpEntryID* is not NULL, it should contain all of the information needed to identify and locate the message store.. This information can include network volume names, phone numbers, user account names, and so on. If the connection to the store cannot be made using the data in the entry identifier, then the store provider should display a dialog box that enables the user to select the store to be opened. A dialog box might be required, for example, if a server has been renamed, an account name has changed, or portions of the network are not running.

When *lpEntryID* is NULL, the message store to use has not yet been selected. The provider can still access a store without displaying a dialog box if it supports further methods to specify the store. For example, the provider can check its initialization file, or it can look for additional properties that were placed in its or its message service's profile section at configuration.

If a provider finds that all the required information is not in the profile, it should return MAPI_E_UNCONFIGURED so that MAPI calls the provider's message service entry point function to enable the user to select a store, or even to create one, and to enter an account name and password as needed. MAPI automatically creates a new profile section for a new store; this new profile section can be temporary or permanent, depending on how it has been added. If the store provider calls the **IMAPISupport::ModifyProfile** method, the new profile section becomes permanent and the store is added to the list of message stores returned by the **IMAPISession::GetMsgStoresTable** method.

The *lpInterface* parameter specifies the IID of the interface required for the newly opened store object. Passing NULL in *lpInterface* specifies that the MAPI message store interface, **IMsgStore**, is required. Passing the message store object IID, IID_IMsgStore, also specifies that **IMsgStore** is required. If IID_IUnknown is passed in *lpInterface*, the provider should open the store using whatever interface derived from **IUnknown** that is best for the provider; again, this is typically **IMsgStore**. When IID_IUnknown is passed, after the store open operation succeeds the calling implementation uses the **IUnknown::QueryInterface** method to select an interface.

The **IMSProvider::Logon** call should return sufficient information, such as a path to the store and credentials for accessing the store, to allow the MAPI spooler to log onto the same store the store provider does without presenting a dialog box. The parameters *lpcbSpoolSecurity* and *lppbSpoolSecurity* are used to return this information. The provider allocates the memory for this data by passing a pointer to a buffer in the **MSProviderInit** function's *lpfAllocateBuffer* parameter; the provider places the size of this buffer in *lpcbSpoolSecurity*.

MAPI frees this buffer when appropriate. If the MAPI spooler's logon to the store can be accomplished from the information in the profile section alone, the provider can return NULL in *lppbSpoolSecurity* and zero for the information's size in *lpcbSpoolSecurity*. The MAPI spooler logon occurs as part of a different process than the store logon; because the buffer holding the passed information gets copied between processes, it might not be in memory at the same location for the MAPI spooler process as for the store provider process. Therefore, a provider shouldn't put addresses into this buffer. For more information on MAPI spooler logon, see **IMSProvider::SpoolerLogon**.

Most store providers use the **IMAPISession::OpenProfileSection** method of the support object passed in the *lpMAPISup* parameter for saving and retrieving user credentials and options. **OpenProfileSection** enables a store provider to save additional arbitrary information in a profile section and associate it with logon for a particular resource. For example, a store provider can save the user account name and password associated with a resource and any paths or other necessary information needed to access that resource for its logon.

Properties with property identifiers 0x6600 through 0x67FF are secure properties available to the provider for its own use to store private data in profile sections. For more information on the uses of properties in profile section objects, see **IProfSect : IMAPIProp**.

In addition to any private data in properties with identifiers 0x6600 through 0x67FF, the store provider should provide information for the PR_DISPLAY_NAME property in its profile section. It should place in PR_DISPLAY_NAME the display name of the provider itself, an identifying string displayed to users so they can distinguish this message store from others they might have access to − for example, Microsoft Personal Information Store. PR_DISPLAY_NAME commonly contains a server name, user account name, or path.

Some profile section properties are visible in the message store table; others are visible during setup, installation, and configuration of the MAPI subsystem. The provider typically provides information for these visible properties both for a new profile section, which does not yet include saved credentials or private information, and when it finds that property information has changed. For more information on profile sections, see **IMAPISupport::OpenProfileSection**.

After successfully logging on a user, and before returning to MAPI, the store provider should create the array of properties for the status row for the resource and call **IMAPISupport::ModifyStatusRow**.

Logon calls that open message stores already open for the current MAPI session skip much of the processing described previously. These calls do not create status rows, do not return message store logon objects, do not call **AddRef** for the MAPI support object, and do not return data for MAPI spooler logon. These calls do return S_OK and do return a message store object with the interface requested.

To detect such calls, the provider should maintain a list in the message store provider object of stores already open for this provider object. When processing a **Logon** call, the provider should scan this list of open stores and determine if the store to be logged onto is already open. If it is, user credentials do not need to be checked and dialog box display should be avoided if possible. If dialog boxes must be displayed, the provider should check information returned to see whether a store has been opened a second time. In addition, the provider should check for duplicate openings using *lpEntryID* at the beginning of **Logon** call processing.

Standard processing for a **Logon** call that accesses an open store is as follows:

1. The store provider calls **AddRef** for the existing store object if the new interface being requested is the same as the interface for the existing store. Otherwise, it calls **QueryInterface** to get the new interface. If the new interface isn't one the store supports, the provider should return the error value MAPI_E_INTERFACE_NOT_SUPPORTED.
2. The provider returns a pointer to the required interface of the existing store object in *lppMDB*.
3. The provider returns NULL in *lppMSLogon*.
4. The provider should not open the profile for the support object passed in the call. Neither should it register a provider unique identifier, register a status row, nor return MAPI spooler logon data.
5. The provider should not call **AddRef** for the support object, because it does not require a pointer to the object.

Whenever possible, providers should return appropriate error and warning strings for **Logon** calls because doing so greatly eases the burden of users in figuring out why something did not work. To do so, a provider sets the members in the **MAPIERROR** structure. MAPI looks for, uses, and releases the **MAPIERROR** structure if it is returned by a provider.

Memory for this **MAPIERROR** should be allocated using the buffer passed in *lpfAllocateBuffer* on the **MSProviderInit** call. Any error strings contained in the returned structure should be in Unicode format if MAPI_UNICODE is set in the **Logon** *ulFlags;* otherwise, they should be in the ANSI character set.

For most error values returned from **Logon**, MAPI disables the message services to which the failing provider belongs. MAPI will not call any providers belonging to those services for the rest of the life of the MAPI session. In contrast, when **Logon** returns the MAPI_E_FAILONEPROVIDER error value from its logon, MAPI does not disable the message service to which the provider belongs. **Logon** should return MAPI_E_FAILONEPROVIDER if it encounters an error that does not warrant disabling the entire service for the life of the session. For example, a provider might return this error when it does not allow the display of a user interface and a required password is unavailable.

If a provider returns MAPI_E_UNCONFIGURED from its logon, MAPI will call the provider's message service entry function and then retry the logon. MAPI passes MSG_SERVICE_CONFIGURE as the context, to give the service a chance to configure itself. If the client has chosen to allow a user interface on the logon, the service can present its configuration property sheet so the user can enter configuration information.

For more information on using the **HR_FAILED** macro, see Using Macros for Error Handling.

See Also

IMAPISession::GetMsgStoresTable method, **IMAPISession::OpenMsgStore** method, **IMAPISession::OpenProfileSection** method, **IMAPISupport::ModifyProfile** method, **IMAPISupport::ModifyStatusRow** method, **IMsgStore : IMAPIProp** interface, **IMSProvider::SpoolerLogon** method, **IProfSect : IMAPIProp** interface, **MAPIERROR** structure,

MSProviderInit function

IMSProvider::Shutdown

The **IMSProvider::Shutdown** method closes down a message store provider in an orderly fashion.

```
HRESULT Shutdown(
   ULONG FAR * lpulFlags
  )
```

Parameters

lpulFlags
Reserved; must be a pointer to zero.

Return Value

S_OK
The call succeeded and has returned the expected value or values.

Remarks

MAPI calls the **IMSProvider::Shutdown** method just before releasing the message store provider object. MAPI releases all logon objects for a provider before calling **Shutdown** for that provider.

IMSProvider::SpoolerLogon

The **IMSProvider::SpoolerLogon** method logs the MAPI spooler onto a message store.

HRESULT SpoolerLogon(
  LPMAPISUP *lpMAPISup*,
  ULONG *ulUIParam*,
  LPTSTR *lpszProfileName*,
  ULONG *cbEntryID*,
  LPENTRYID *lpEntryID*,
  ULONG *ulFlags*,
  LPCIID *lpInterface*,
  ULONG *cbSpoolSecurity*,
  LPBYTE *lpbSpoolSecurity*,
  LPMAPIERROR FAR * *lppMAPIError*,
  LPMSLOGON FAR * *lppMSLogon*,
  LPMDB FAR * *lppMDB*
)

Parameters

lpMAPISup
  Input parameter pointing to the MAPI support object for the message store.

ulUIParam
  Input parameter containing the handle of the parent window for any dialog boxes or windows this method displays.

lpszProfileName
  Input parameter pointing to a string containing the name of the profile being used for the MAPI spooler logon. This string can be displayed in dialog boxes, written out to a log file, or simply ignored. It must be in Unicode format if the MAPI_UNICODE flag is set in the *ulFlags* parameter.

cbEntryID
  Input parameter containing the size, in bytes, of the entry identifier pointed to by the *lpEntryID* parameter.

lpEntryID
  Input parameter pointing to the entry identifier for the message store. Passing NULL in the *lpEntryID* parameter indicates that a message store has not yet been selected and that dialog boxes enabling the user to select a message store can be presented.

ulFlags
  Input parameter containing a bitmask of flags that controls how the logon is performed. The following flags can be set:

  MAPI_DEFERRED_ERRORS
    Indicates the call is allowed to succeed even if the underlying object is not accessible to the calling implementation. If the object is not accessible, some subsequent call to the object might return an error.

  MAPI_UNICODE
    Indicates the passed-in strings are in Unicode format. If MAPI_UNICODE is not set, the strings are in ANSI format.

  MDB_NO_DIALOG
    Prevents display of logon dialog boxes. If this flag is set, the error value MAPI_E_LOGON_FAILED is returned if logon is unsuccessful. If this flag is not set, the message store provider can prompt the user to correct a name or password, to insert a disk, or to perform other actions necessary to establish connection to the store.

  MDB_WRITE

Requests read/write access.

lpInterface
Input parameter pointing to the interface identifier (IID) for the message store to log onto. Passing NULL indicates the MAPI interface for the message store is returned – that is, the **IMsgStore** interface. The *lpInterface* parameter can also be set to an identifier for an appropriate interface for the message store, for example IID_IUnknown or IID_IMAPIProp.

cbSpoolSecurity
Input parameter pointing to the variable containing the size, in bytes, of validation data in the *lppbSpoolSecurity* parameter.

lpbSpoolSecurity
Input parameter pointing to a variable containing a pointer to validation data. The **SpoolerLogon** method uses this data to log the MAPI spooler onto the same store as the message store provider previously logged onto using the **IMSProvider::Logon** method.

lppMAPIError
Output parameter pointing to a pointer to the returned **MAPIERROR** structure, if any, containing version, component, and context information for an error. The *lppMAPIError* parameter can be set to NULL if there is no **MAPIERROR** structure to return.

lppMSLogon
Output parameter pointing to a variable where the pointer to the message store logon object for MAPI to log onto is stored.

lppMDB
Output parameter pointing to a variable where the pointer to the message store object for the MAPI spooler and client applications to log onto is stored.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

MAPI_E_UNCONFIGURED
The profile does not contain enough information for the logon to complete. When this value is returned, MAPI calls the message store provider's message service entry point function.

MAPI_W_ERRORS_RETURNED
The call succeeded, but the message store provider has error information available. To test for this warning, use the **HR_FAILED** macro. When this warning is returned, the call should be handled as successful. To get the error information from the provider, call the **IMAPISession::GetLastError** method.

Remarks

The MAPI spooler calls the **IMSProvider::SpoolerLogon** method to log onto a message store. The MAPI spooler should use the message store object returned by the message store provider in the *lppMDB* parameter during and after logon.

For consistency with the **IMSProvider::Logon** method, the provider also returns a message store logon object in the *lppMSLogon* parameter. Usage of the store object and the logon object are identical for usual store logon, because there is a one-to-one correspondence between the logon object and the store object; you can think of the logon and store objects as one object exposing two interfaces. The two objects are created together and freed together.

The store provider should internally mark the returned message store object to indicate that the store is being used by the MAPI spooler. Some of the methods for this store object behave differently than for the message store object provided to client applications. Keeping this internal mark is the most common way of triggering the behavior specific to the MAPI spooler.

For more information on using the **HR_FAILED** macro, see Using Macros for Error Handling.

See Also

IMSProvider::Logon method, **MAPIERROR** structure

IPersistMessage : IUnknown

The **IPersistMessage** interface is implemented by form objects to save, initialize, and load messages to and from forms. **IPersistMessage** works similarly to the OLE **IPersistStorage** interface; for more information on the **IPersistStorage** methods, and on working with storage objects in general, see the *OLE Programmer's Reference*.

At a Glance

Specified in header file:	MAPIFORM.H
Object that supplies this interface:	Persist message object
Corresponding pointer type:	LPPERSISTMESSAGE
Implemented by:	Form objects
Called by:	Form viewers

Vtable Order

GetLastError	Returns a **MAPIERROR** structure containing information about the last error that occurred for a message object.
GetClassID	Returns a form's message class identifier.
IsDirty	Checks a form for changes made since the form was last saved.
InitNew	Provides a form with a base message on which to build a new message.
Load	Loads a form from a specified message.
Save	Saves a revised form back to the message from which it was loaded or created.
SaveCompleted	Returns a message to a form after a save, submission, or other operation.
HandsOffMessage	Causes a message to release its message object.

IPersistMessage::GetClassID

The **IPersistMessage::GetClassID** method returns a form's message class identifier.

HRESULT GetClassID(
  LPCLSID *lpClassID*
)

Parameters

lpClassID
  Input parameter pointing to the returned class identifier.

Return Value

S_OK
  The call succeeded and has returned the expected value or values.

Remarks

For more information on working with class identifiers of storage objects, see the documentation for the **IPersistStorage** methods in the *OLE Programmer's Reference.*

IPersistMessage::GetLastError

The **IPersistMessage::GetLastError** method returns a **MAPIERROR** structure containing information about the last error that occurred in a form's message object.

```
HRESULT GetLastError(
  HRESULT hResult,
  ULONG ulFlags,
  LPMAPIERROR FAR * lppMAPIError
 )
```

Parameters

hResult
: Input parameter containing the result returned for the last call for the message object that returned an error.

ulFlags
: Input parameter containing a bitmask of flags that controls the type of the returned strings. The following flag can be set:

  MAPI_UNICODE
  : Indicates the strings in the **MAPIERROR** structure returned in the *lppMAPIError* parameter are in Unicode format. If the MAPI_UNICODE flag is not set, the strings are in ANSI format.

lppMAPIError
: Output parameter pointing to a pointer to the returned **MAPIERROR** structure containing version, component, and context information for the error. The *lppMAPIError* parameter can be set to NULL if there is no **MAPIERROR** structure to return.

Return Values

S_OK
: The call succeeded and has returned the expected value or values.

MAPI_E_BAD_CHARWIDTH
: Either the MAPI_UNICODE flag was set and the implementation does not support Unicode, or MAPI_UNICODE was not set and the implementation only supports Unicode.

Remarks

Use the **IMAPITable::GetLastError** method to retrieve information to display in a message to the user regarding the last error returned from a method call for a form's message object.

To release all the memory allocated by MAPI for the **MAPIERROR** structure, implementations need only call the **MAPIFreeBuffer** function.

The return value from **GetLastError** must be S_OK for an implementation to make use of the **MAPIERROR** structure. Even if the return value is S_OK, a **MAPIERROR** structure might not be returned. If the implementation cannot determine what the last error was, or if a **MAPIERROR** structure is not available for that error, **GetLastError** returns a pointer to NULL in *lppMAPIError* instead.

See Also

MAPIERROR structure, **MAPIFreeBuffer** function

IPersistMessage::HandsOffMessage

The **IPersistMessage::HandsOffMessage** method causes a form to release its message object.

HRESULT HandsOffMessage()

Parameters

None

Return Values

S_OK
    The call succeeded and has returned the expected value or values.

Remarks

For more information on working with the hands-off state of storage objects, see the *OLE Programmer's Reference.*

IPersistMessage::InitNew

The **IPersistMessage::InitNew** method provides a form with a base message on which to build a new message.

HRESULT InitNew(
  LPMAPIMESSAGESITE *pMessageSite*,
  LPMESSAGE *pMessage*
)

Parameters

pMessageSite
  Input parameter pointing to the message site the form uses to compose a new message.

pMessage
  Input parameter pointing to the message the form uses to compose a new message.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

Remarks

Form viewers call the **IPersistMessage::InitNew** method to set up a form with a message site and a message so as to compose a new message within the form. When a message is loaded into a form with **InitNew**, you can assume that the following required properties, and no others, have been set by form server implementations:

PR_DELETE_AFTER_SUBMIT
PR_IMPORTANCE
PR_ORIGINATOR_DELIVERY_REPORT_REQUESTED
PR_PRIORITY
PR_READ_RECEIPT_REQUESTED
PR_SENSITIVITY
PR_SENTMAIL_ENTRYID

For more information on initializing new storage objects, see the *OLE Programmer's Reference.*

IPersistMessage::IsDirty

The **IPersistMessage::IsDirty** method checks a form for changes made since the form was last saved.

HRESULT IsDirty()

Parameters

None

Return Values

S_OK
: The form has had changes made since it was last saved.

S_FALSE
: The form has not had changes made since it was last saved.

Remarks

For more information, see the documentation for the **IPersistStorage** methods in the *OLE Programmer's Reference.*

IPersistMessage::Load

The **IPersistMessage::Load** method loads a form from a specified message.

HRESULT Load(
  LPMESSAGESITE *pMessageSite*,
  LPMESSAGE *pMessage*,
  ULONG *ulMessageStatus*,
  ULONG *ulMessageFlags*
)

Parameters

pMessageSite
  Input parameter pointing to the specified message to be loaded.

pMessage
  Input parameter pointing to the message from which the message is loaded.

ulMessageStatus
  Input parameter containing a bitmask of client-defined or provider-defined flags, copied from the message's PR_MSG_STATUS property, that provides information on the state of the message.

ulMessageFlags
  Input parameter containing a bitmask of flags, copied from the message's PR_MESSAGE_FLAGS property, that provide further information on the state of the message.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

Remarks

Form viewers call the **IPersistMessage::Load** method to load a form from a specified message. **Load** is used to read an existing message. Flags and status bits set in the existing message's PR_MESSAGE_FLAGS and PR_MSG_STATUS properties are preserved in the new message.

For more information on loading storage objects, see the *OLE Programmer's Reference.*

See Also

PR_MESSAGE_FLAGS property, PR_MSG_STATUS property

IPersistMessage::Save

The **IPersistMessage::Save** method saves a revised form back to the message from which it was loaded or created.

HRESULT Save(
  LPMESSAGE *pMessage***,**
  ULONG *fSameAsLoad*
)

Parameters

pMessage
  Input parameter pointing to a message.

fSameAsLoad
  Input parameter containing a variable that is TRUE if the message the *pMessage* parameter points to is the message from which the form was loaded or created, and FALSE otherwise.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

Remarks

Form viewers call the **IPersistMessage::Save** method to save a revised form back to the message from which it was loaded or created. The form must not commit changes to this message; changes are committed by the form server implementation that calls **Save**.

For more information on saving storage objects, see the documentation on the **IPersistStorage** methods in the *OLE Programmer's Reference.*

IPersistMessage::SaveCompleted

The **IPersistMessage::SaveCompleted** method returns a message to a form after a save, submission, or other operation.

```
HRESULT SaveCompleted(
  LPMESSAGE pMessage
 )
```

Parameters

pMessage
Input parameter pointing to the message that was acted upon.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

Remarks

Form viewers call the **IPersistMessage::SaveCompleted** method at the end of a save request or after the failure of a dismissive operation such as a save operation or deletion. **SaveCompleted** returns the saved message to the form in which it was composed. Note that there is no reason to suppose the **IPersistMessage** interface used is the same interface as held for the message at the beginning of the save or other operation. However, the interface is for the same message object.

For more information on saving storage objects, see the documentation on the **IPersistStorage** methods in the *OLE Programmer's Reference.*

IProfAdmin : IUnknown

The **IProfAdmin** interface supports administration of profiles.

At a Glance

Specified in header file:	MAPIX.H
Object that supplies this interface:	Profile object
Corresponding pointer type:	LPPROFADMIN
Implemented by:	MAPI
Called by:	Client applications

Vtable Order

GetLastError	Returns a **MAPIERROR** structure containing information about the last error that occurred for a profile object.
GetProfileTable	Returns a table listing all profiles associated with a particular client application.
CreateProfile	Creates a new profile.
DeleteProfile	Deletes a profile.
ChangeProfilePassword	Changes the password for a profile.
CopyProfile	Copies a profile.
RenameProfile	Renames a profile.
SetDefaultProfile	Sets the default profile for a client application.
AdminServices	Enumerates and makes configuration changes to the message services in a profile.

IProfAdmin::AdminServices

The **IProfAdmin::AdminServices** method enumerates and makes configuration changes to the message services in a profile.

HRESULT AdminServices(
  LPTSTR *lpszProfileName***,**
  LPTSTR *lpszPassword***,**
  ULONG *ulUIParam***,**
  ULONG *ulFlags***,**
  LPSERVICEADMIN FAR * *lppServiceAdmin*
)

Parameters

lpszProfileName
  Input parameter pointing to a string containing name of the profile affected. The *lpszProfileName* parameter must not be NULL.

lpszPassword
  Input parameter pointing to a string containing the profile's password. If a client application passes NULL in the *lpszPassword* parameter and the profile requires a password to open, the profile provider displays a dialog box prompting the user for the password if the MAPI_DIALOG flag is set in the *ulFlags* parameter.

ulUIParam
  Input parameter containing the handle of the parent window for any dialog boxes or windows this method displays.

ulFlags
  Input parameter containing a bitmask of flags that controls how message service configurations are changed. The following flags can be set:

  MAPI_DIALOG
    Displays a dialog box prompting the user for the profile password. If this flag is not set, no dialog box is displayed.

  MAPI_UNICODE
    Indicates the passed-in strings are in Unicode format. If the MAPI_UNICODE flag is not set, the strings are in ANSI format.

lppServiceAdmin
  Output parameter pointing to a variable where a pointer to the returned message service administration object is stored. This message service administration object is used to change message service settings in the profile.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

MAPI_E_LOGON_FAILED
  The specified profile doesn't exist, or the password was wrong and a dialog box could not be displayed to the user requesting the correct password because MAPI_DIALOG was not set in *ulFlags*.

MAPI_E_USER_CANCEL
  The user canceled the operation, typically by clicking the **Cancel** button in a dialog box.

Remarks

Client applications call the **IProfAdmin::AdminServices** method to obtain a message service administration object so as to make configuration changes to the message services within a profile. Clients that only perform configuration should use **IProfAdmin::AdminServices** rather than the **IMAPISession::AdminServices** method because the **IProfAdmin** method creates no session object and loads no service providers. To make other changes, clients should call **IMAPISession::AdminServices**.

Clients calling **IProfAdmin::AdminServices** must specify an existing profile in *lpszProfileName*. If the specified profile does not exist, the call returns MAPI_E_LOGON_FAILED.

Profile providers are required to support names and passwords up to 64 characters long. Profile providers are required to support the following characters in profile name and password strings:

- All alphanumeric characters, including accent characters and the underscore character.
- Embedded spaces, but not leading or trailing spaces.

Profile providers can also support additional characters in profile and password names.

Profile providers are not required to implement support for profile passwords. Additionally, profile passwords are not supported on all operating system platforms; on platforms that do not support profile passwords, *lpszPassword* can be NULL or a pointer to a zero-length string. Currently, Microsoft Windows NT and Microsoft Windows 95 do not support passwords; Microsoft Windows version 3.1 does.

See Also

IMAPISession::AdminServices method

IProfAdmin::ChangeProfilePassword

The **IProfAdmin::ChangeProfilePassword** method changes the password for a profile.

HRESULT ChangeProfilePassword(
  LPTSTR *lpszProfileName*,
  LPTSTR *lpszOldPassword*,
  LPTSTR *lpszNewPassword*,
  ULONG *ulFlags*
)

Parameters

lpszProfileName
Input parameter pointing to a string containing the name of the profile whose password is to be changed.

lpszOldPassword
Input parameter pointing to a string containing the original password.

lpszNewPassword
Input parameter pointing to a string containing the new password.

ulFlags
Input parameter containing a bitmask of flags that controls the type of the passed-in strings. The following flag can be set:

MAPI_UNICODE
Indicates the passed-in strings are in Unicode format. If the MAPI_UNICODE flag is not set, the strings are in ANSI format.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

MAPI_E_LOGON_FAILED
The password is incorrect.

MAPI_E_NO_SUPPORT
The operating system does not support passwords.

Remarks

Client applications call the **IProfAdmin::ChangeProfilePassword** method to replace one profile password with another. **ChangeProfilePassword** does not display a user interface.

Profile providers are not required to implement support for profile passwords. **ChangeProfilePassword** returns MAPI_E_NO_SUPPORT if the operating system does not support passwords. Currently, Windows NT Server and Windows 95 do not support passwords; Windows version 3.1 does.

IProfAdmin::CopyProfile

The **IProfAdmin::CopyProfile** method copies a profile.

HRESULT CopyProfile(
  LPTSTR *lpszOldProfileName***,**
  LPTSTR *lpszOldPassword***,**
  LPTSTR *lpszNewProfileName***,**
  ULONG *ulUIParam***,**
  ULONG *ulFlags*
)

Parameters

lpszOldProfileName
  Input parameter pointing to a string containing the name of the profile to copy.

lpszOldPassword
  Input parameter pointing to a string containing the password of the profile to copy.

lpszNewProfileName
  Input parameter pointing to a string containing the name of the new profile to create.

ulUIParam
  Input parameter containing the handle of the parent window for any dialog boxes or windows this method displays.

ulFlags
  Input parameter containing a bitmask of flags that controls how the profile is copied. The following flags can be set:

  MAPI_DIALOG
    Displays a dialog box prompting the user for the correct password of the profile to copy. If this flag is not set, no dialog box is displayed.

  MAPI_UNICODE
    Indicates the passed-in strings are in Unicode format. If the MAPI_UNICODE flag is not set, the strings are in ANSI format.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

MAPI_E_ACCESS_DENIED
  The new profile name is the same as that of an existing profile.

MAPI_E_LOGON_FAILED
  The password for the profile to copy is incorrect, and a dialog box could not be displayed to the user requesting the correct password because MAPI_DIALOG was not set in the *ulFlags* parameter.

MAPI_E_NOT_FOUND
  The specified profile does not exist.

MAPI_E_USER_CANCEL
  The user canceled the operation, typically by clicking the **Cancel** button in a dialog box.

Remarks

Client applications call the **IProfAdmin::CopyProfile** method to make a copy of the profile indicated in *lpszOldProfileName*. **CopyProfile** names the copy using the string given in the *lpszNewProfileName* parameter. Copying a profile leaves the copy with the same password as the original.

Profile providers are required to support names and passwords up to 64 characters long. Profile providers are required to support the following characters in profile name and password strings:

- All alphanumeric characters, including accent characters and the underscore character.
- Embedded spaces, but not leading or trailing spaces.

Profile providers can also support additional characters in profile and password names.

Profile providers are not required to implement support for profile passwords. Additionally, profile passwords are not supported on all operating system platforms; on platforms that do not support profile passwords, *lpszOldPassword* can be NULL or a pointer to a zero-length string. Currently, Windows NT Server and Windows 95 do not support passwords; Windows version 3.1 does.

If a client passes NULL in *lpszOldPassword* and the profile to copy requires a password to open, the profile provider must display a dialog box prompting the user to provide a password. If the wrong password is supplied in *lpszOldPassword* and MAPI_DIALOG is not set in *ulFlags*, the call returns MAPI_E_LOGON_FAILED instead of prompting the user to provide the password.

IProfAdmin::CreateProfile

The **IProfAdmin::CreateProfile** method creates a new profile.

HRESULT CreateProfile(
  LPTSTR *lpszProfileName***,**
  LPTSTR *lpszPassword***,**
  ULONG *ulUIParam***,**
  ULONG *ulFlags*
)

Parameters

lpszProfileName
  Input parameter pointing to a string containing the name of the new profile.

lpszPassword
  Input parameter pointing to a string containing the password of the new profile.

ulUIParam
  Input parameter containing the handle of the parent window for any dialog boxes or windows this method displays.

ulFlags
  Input parameter containing a bitmask of flags that controls how the profile is created. The following flags can be set:

  MAPI_DEFAULT_SERVICES
    Indicates MAPI should populate the new profile with message services as indicated by the [Default Services] section in the MAPISVC.INF file.

  MAPI_DIALOG
    Displays each service provider's configuration property sheets. If this flag is not set, then all the message services added by this call are unconfigured.

  MAPI_UNICODE
    Indicates the passed-in strings are in Unicode format. If the MAPI_UNICODE flag is not set, the strings are in ANSI format.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

MAPI_E_NO_ACCESS
  The specified new profile already exists.

Remarks

Client applications call the **IProfAdmin::CreateProfile** method to create a new profile. **CreateProfile** can be used during client installation, at which point it can read data from a configuration file to fill its input parameters. **CreateProfile** is also used by clients that enable users to create new profiles; in such a case, the parameters receive input from a dialog box displayed by the client.

If the MAPI_DEFAULT_SERVICES flag is set in the *ulFlags* parameter, **CreateProfile** calls the message service entry point function for each message service in the [Default Services] section in the MAPISVC.INF file. The message service entry point function is called with the *ulContext* parameter set to MSG_SERVICE_CREATE. If both the MAPI_DIALOG and MAPI_DEFAULT_SERVICES flags are set in the **CreateProfile** method's *ulFlags* parameter, then the values in the *ulUIParam* and *ulFlags* parameters are also passed when the message service entry point function is called. The message service entry point functions are only called after all available information from the MAPISVC.INF file has been added to the profile. Service providers should display their configuration property sheets so that the user can configure the message service.

Profile providers are required to support names and passwords up to 64 characters long. Profile providers are required to support the following characters in profile name and password strings:

- All alphanumeric characters, including accent characters and the underscore character.
- Embedded spaces, but not leading or trailing spaces.

Profile providers can also support additional characters in profile and password names.

Profile providers are not required to implement support for profile passwords. Additionally, profile passwords are not supported on all operating system platforms; on platforms that do not support profile passwords, *lpszPassword* can be NULL or a pointer to a zero-length string. Currently, Windows NT Server and Windows 95 do not support passwords; Windows version 3.1 does.

If a profile with the same name as passed in the *lpszProfileName* parameter already exists, **CreateProfile** returns MAPI_E_NO_ACCESS.

See Also

IMsgServiceAdmin::ConfigureMsgService method, **IMsgServiceAdmin::CreateMsgService** method, **MSGSERVICEENTRY** function prototype

IProfAdmin::DeleteProfile

The **IProfAdmin::DeleteProfile** method deletes a profile.

HRESULT DeleteProfile(
  LPTSTR *lpszProfileName***,**
  ULONG *ulFlags*
)

Parameters

lpszProfileName
  Input parameter pointing to a string containing the name of the profile to be deleted.

ulFlags
  Input parameter containing a bitmask of flags that controls the type of the passed-in string. The following flag can be set:

  MAPI_UNICODE
    Indicates the passed-in string is in Unicode format. If the MAPI_UNICODE flag is not set, the string is in ANSI format.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

MAPI_E_NOT_FOUND
  The specified profile does not exist.

Remarks

Client applications call the **IProfAdmin::DeleteProfile** method to delete a profile. If the profile indicated in the *lpszProfileName* parameter does not exist, **DeleteProfile** returns MAPI_E_NOT_FOUND. If the profile to delete is in use by a client when **DeleteProfile** is called, **DeleteProfile** returns S_OK but does not delete the profile immediately. Instead, MAPI marks the profile for deletion and deletes it after all clients have logged off the profile.

The message service entry point function is called for each message service in the MAPISVC.INF file before each service is removed from the profile. The message service entry point function is called with the MSG_SERVICE_DELETE value set in its *ulContext* parameter . First the function deletes the service, and then it deletes the service's profile section. The message service entry point function is not called again after the service has been deleted.

No password is required to delete a profile.

See Also

IMsgServiceAdmin::DeleteMsgService method, **MSGSERVICEENTRY** function prototype

IProfAdmin::GetLastError

The **IProfAdmin::GetLastError** method returns a **MAPIERROR** structure containing information about the last error that occurred for a profile object.

HRESULT GetLastError(
  HRESULT *hResult***,**
  ULONG *ulFlags***,**
  LPMAPIERROR FAR * *lppMAPIError*
)

Parameters

hResult
: Input parameter containing the result returned for the last call for the profile object that returned an error.

ulFlags
: Input parameter containing a bitmask of flags that controls the type of the returned strings. The following flag can be set:

  MAPI_UNICODE
  : Indicates the strings in the **MAPIERROR** structure returned in the *lppMAPIError* parameter are in Unicode format. If the MAPI_UNICODE flag is not set, the strings are in ANSI format.

lppMAPIError
: Output parameter pointing to a pointer to the returned **MAPIERROR** structure containing version, component, and context information for the error. The *lppMAPIError* parameter can be set to NULL if there is no **MAPIERROR** structure to return.

Return Values

S_OK
: The call succeeded and has returned the expected value or values.

MAPI_E_BAD_CHARWIDTH
: Either the MAPI_UNICODE flag was set and the implementation does not support Unicode, or MAPI_UNICODE was not set and the implementation only supports Unicode.

Remarks

Client applications call the **IProfAdmin::GetLastError** method to retrieve information to display in a message to the user regarding the last error returned from a method call for the profile object.

To release all the memory allocated by MAPI for the returned **MAPIERROR** structure, clients need only call the **MAPIFreeBuffer** function.

The return value from **GetLastError** must be S_OK for a client to make use of the **MAPIERROR** structure. Even if the return value is S_OK, a **MAPIERROR** structure might not be returned. If the implementation cannot determine what the last error was, or if a **MAPIERROR** structure is not available for that error, **GetLastError** returns a pointer to NULL in *lppMAPIError* instead.

See Also

MAPIERROR structure, **MAPIFreeBuffer** function

IProfAdmin::GetProfileTable

The **IProfAdmin::GetProfileTable** method returns a table listing all profiles associated with a particular client application.

HRESULT GetProfileTable(
  ULONG *ulFlags***,**
  LPMAPITABLE FAR * *lppTable*
)

Parameters

ulFlags
  Input parameter containing a bitmask of flags that controls the type of the strings returned in the table's default column set. The following flag can be set:

  MAPI_UNICODE
    Indicates the returned strings are in Unicode format. If the MAPI_UNICODE flag is not set, the strings are in ANSI format.

lppTable
  Output parameter pointing to a variable where the pointer to the returned profile table object is stored.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

Remarks

A client application calls the **IProfAdmin::GetProfileTable** method to get a table containing rows listing information for each profile that has been created for use with that client. The columns of the profile table contain current information for the following properties:

  PR_DISPLAY_NAME
  PR_DEFAULT_PROFILE

Profiles that have been deleted, or that are in use but have been marked for deletion, are not returned in the profile table. Once a profile table has been returned, it does not reflect changes being made to the profile, such as the addition or deletion of profiles. Calls to the **IMAPITable::Advise** method for the profile table return S_OK, but no changes are made to the table.

If no profile exists, **GetProfileTable** does not return an error but returns a table object supporting the **IMAPITable** interface. If a client calls the **IMAPITable::QueryRows** method on that table, zero rows are returned.

Setting the MAPI_UNICODE flag in the *ulFlags* parameter:

- Sets the string type to Unicode for data returned for the initial active columns of the profile table by the **IMAPITable::QueryColumns** method. The initial active columns for a profile table are those columns **QueryColumns** returns before the service provider that contains the table calls the **IMAPITable::SetColumns** method.
- Sets the string type to Unicode for data returned for the initial active rows of the profile table by the **IMAPITable::QueryRows** method. The initial active rows for a profile table are those rows **QueryRows** returns before the provider that contains the table calls **SetColumns**.
- Controls the property types of the sort order returned by the **IMAPITable::QuerySortOrder** method before the provider that contains the profile table calls the **IMAPITable::SortTable** method.

See Also

IMAPITable : IUnknown interface, **MAPILogonEx** function

IProfAdmin::RenameProfile

The **IProfAdmin::RenameProfile** method renames a profile.

HRESULT RenameProfile(
  LPTSTR *lpszOldProfileName***,**
  LPTSTR *lpszOldPassword***,**
  LPTSTR *lpszNewProfileName***,**
  ULONG *ulUIParam***,**
  ULONG *ulFlags*
)

Parameters

lpszOldProfileName
Input parameter pointing to a string containing the current name of the profile to rename.

lpszOldPassword
Input parameter pointing to a string containing the password of the profile to rename. If a client application passes NULL in the *lpszOldPassword* parameter, and the profile requires a password to open, the profile provider displays a dialog box prompting the user for the password.

lpszNewProfileName
Input parameter pointing to a string containing the name of the profile to rename.

ulUIParam
Input parameter containing the handle of the parent window for any dialog boxes or windows this method displays.

ulFlags
Input parameter containing a bitmask of flags that controls how the renaming operation is performed. The following flags can be set:

MAPI_DIALOG
Displays a dialog box prompting the user for the profile password. If this flag is not set, no dialog box is displayed.

MAPI_UNICODE
Indicates the passed-in strings are in Unicode format. If the MAPI_UNICODE flag is not set, the strings are in ANSI format.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

MAPI_E_LOGON_FAILED
The profile password is incorrect.

MAPI_E_USER_CANCEL
The user canceled the operation, typically by clicking the **Cancel** button in a dialog box.

Remarks

Client applications call the **IProfAdmin::RenameProfile** method to rename a profile. **RenameProfile** takes the profile name in the *lpszOldProfileName* parameter and replaces it with the profile name in the *lpszNewProfileName* parameter. Renaming the profile does not change its password. If the profile to rename is in use by a client when **RenameProfile** is called, **RenameProfile** returns S_OK but does not rename the profile immediately. Instead, MAPI marks the profile for renaming and renames it after all clients have logged off the profile.

Profile providers are required to support names and passwords up to 64 characters long. Profile providers are required to support the following characters in profile name and password strings:

- All alphanumeric characters, including accent characters and the underscore character.
- Embedded spaces, but not leading or trailing spaces.

Profile providers can also support additional characters in profile and password names.

Profile providers are not required to implement support for profile passwords. Additionally, profile passwords are not supported on all operating system platforms; on platforms that do not support profile passwords, *lpszOldPassword* can be NULL or a pointer to a zero-length string. Currently, Windows NT Server and Windows 95 do not support passwords; Windows version 3.1 does.

If a client passes NULL in *lpszOldPassword* and the profile to rename requires a password to open, the profile provider must display a dialog box prompting the user to provide a password. If the wrong password is supplied in *lpszOldPassword* and MAPI_DIALOG is not set in *ulFlags*, the call returns MAPI_E_LOGON_FAILED instead of prompting the user to provide the password.

IProfAdmin::SetDefaultProfile

The **IProfAdmin::SetDefaultProfile** method sets the default profile for a client application.

HRESULT SetDefaultProfile(
  LPTSTR *lpszProfileName*,
  ULONG *ulFlags*
)

Parameters

lpszProfileName
  Input parameter pointing to a string containing the name of the new default profile. If a client application passes either NULL or a pointer to a zero-length string in the *lpszProfileName* parameter, there will be no default profile and any existing default profile setting is removed.

ulFlags
  Input parameter containing a bitmask of flags that controls the type of the passed-in string. The following flag can be set:

  MAPI_UNICODE
    Indicates the passed-in string is in Unicode format. If the MAPI_UNICODE flag is not set, the string is in ANSI format.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

MAPI_E_NOT_FOUND
  The specified profile does not exist.

Remarks

Client applications call the **IProfAdmin::SetDefaultProfile** method to set the name of the profile to use as the default when a particular client logs on to a MAPI session. This profile is the one used when clients pass the MAPI_USE_DEFAULT flag when calling the **MAPILogonEx** function. **SetDefaultProfile** also changes the PR_DEFAULT_PROFILE property within the profile.

See Also

IProfAdmin::GetProfileTable method, **MAPILogonEx** function, PR_DEFAULT_PROFILE property

IProfSect : IMAPIProp

The **IProfSect** interface is used to work with properties of profile section objects by calling methods of the **IMAPIProp** interface. **IProfSect** does not have any unique methods of its own, but you can call the **IMAPIProp** methods on a profile section object with the following considerations.

The profile section object does not support a transaction model, so all changes made to a profile section following calls to the **IMAPIProp::CopyProps** and **IMAPIProp::CopyTo** methods occur immediately. Calls to the **IMAPIProp::SaveChanges** method succeed, but don't actually save any changes. One consequence of this implementation is that when property sheets or dialog boxes work on a profile section object directly, changes made by the user occur instantly. Service providers should implement their property sheets and dialog boxes with copies of their profile section objects. The following steps describe how to provide this type of implementation:

1. Open the profile section with the **IMAPISupport::OpenProfileSection** or **IProviderAdmin::OpenProfileSection** methods.
2. Call the API function **CreateIProp** to retrieve a property data object.
3. Call the profile section's **IMAPIProp::CopyTo** method to copy properties from the profile section to the property data object.
4. Call **IMAPISupport::DoConfigPropSheet** method to display the configuration user interface, passing a pointer to the property data object for the *lpConfigData* parameter.
5. When the user saves changes to configuration properties in the property sheet or dialog box, call the **IMAPIProp::CopyTo** method to copy the properties from the property data object to the profile section.

Profile section objects do not support named properties. **IMAPIProp::GetIDsFromNames** and **IMAPIProp::GetNamesFromIDs** return MAPI_E_NO_SUPPORT if called for a profile section object and attempts to set properties with identifiers in the range above 0x8000 with **IMAPIProp::SetProps** returns PT_ERROR as the property type.

Profile sections reserve the identifier range 0X67F0 to 0X67FF for secure properties. Service providers can use this range to store passwords and other provider-specific credentials. Properties in this range are not returned in the complete list of properties when NULL is passed in the *lpPropTag* parameter of the **IMAPIProp::GetProps** method, nor are they returned in the *lppPropTagArray* parameter of the **IMAPIProp::GetPropList** method. Secure properties must be requested specifically by their identifiers.

MAPI also furnishes a hard-coded profile section named MUID_PROFILE_INSTANCE, with PR_SEARCH_KEY as its single property. In many cases, the PR_PROFILE_NAME property found on any profile section is sufficient, but sometimes is not adequate because a deleted profile could be succeeded by another with the same name. To cover this situation, this key is guaranteed by MAPI to be unique among all profiles created.

At a Glance

Specified in header file:	MAPIX.H
Object that supplies this interface:	Profile section object
Corresponding pointer type:	LPPROFSECT
Implemented by:	Profile providers
Transaction model:	Non-transacted
Called by:	Client applications and service providers

Vtable Order

No unique methods

Required Properties

PR_OBJECT_TYPE	Read only
PR_PROFILE_NAME	Read only

IPropData : IMAPIProp

Service providers use the **IPropData** interface to add object type properties to an object and to get and set access rights for objects and properties. The **IPropData** interface is derived from the **IMAPIProp** interface and does not have any additional methods or object types of its own. To use an **IPropData** interface, call the **CreateIProp** function. For more information about using the **IPropData** interface to manage data, see About Object and Property Access.

At a Glance

Specified in header file:	MAPIUTIL.H
Object that supplies this interface:	Property data
Corresponding pointer type:	LPPROPDATA
Implemented by:	MAPI
Transaction model:	Non-transacted
Called by:	Client applications, service providers

Vtable Order

HrSetObjAccess	Sets the access rights for an object.
HrSetPropAccess	Sets the access rights or modification flags for the specified properties.
HrGetPropAccess	Returns the current access rights for the specified properties.
HrAddObjProps	Adds properties of type PT_OBJECT to an object.

IPropData::HrAddObjProps

The **IPropData::HrAddObjProps** method adds properties of type PT_OBJECT to an object.

HRESULT HrAddObjProps(
  LPSPropTagArray *lpPropTagArray*,
  LPSPropProblemArray FAR * *lppProblems*
)

Parameters

lpPropTagArray
  Input parameter pointing to an **SPropTagArray** structure containing an array of property tags indicating the properties to add.

lppProblems
  Output parameter pointing to a variable where the pointer to a returned **SPropProblemArray** structure is stored. This structure holds information on properties that couldn't be accessed and should be checked when the method returns. If a pointer to NULL is passed in the *lppProblems* parameter, no **SPropProblemArray** is returned even if some specified properties were not added to the object.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

MAPI_E_INVALID_TYPE
  A property type other than PT_OBJECT was passed in the **SPropTagArray** structure.

MAPI_E_NO_ACCESS
  The object has been set not to allow read/write access.

MAPI_W_PARTIAL_COMPLETION
  Some, but not all, of the properties were added.

Remarks

Client applications and service providers call the **IPropData::HrAddObjProps** method to add properties to an object. The properties being added in the **SPropTagArray** structure in the *lpPropTagArray* parameter must be of type PT_OBJECT. If properties of types other than PT_OBJECT are added, the call returns MAPI_E_INVALID_TYPE.

If the **HrAddObjProps** call returns MAPI_W_PARTIAL_COMPLETION and a pointer to an **SPropProblemArray** structure in *lppProblems*, check the returned **SPropProblemArray** structure to find out which properties were not added. The **SPropProblemArray** structure must be freed by calling the **MAPIFreeBuffer** function.

If the object to add properties to has previously been set to disallow read/write access, **HrAddObjProps** returns MAPI_E_NO_ACCESS. To obtain read/write access, a client or provider should first call the **IPropData::SetObjAccess** method passing in the IPROP_READWRITE flag in the *ulAccess* parameter, then call **HrAddObjProps** to add properties to the object.

See Also

MAPIFreeBuffer function, **SPropProblemArray** structure, **SPropTagArray** structure

IPropData::HrGetPropAccess

The **IPropData::HrGetPropAccess** method returns the current access rights for the specified properties.

```
HRESULT HrGetPropAccess(
  LPSPropTagArray FAR * lppPropTagArray,
  ULONG FAR * FAR * lprgulAccess
 )
```

Parameters

lppPropTagArray
Input-output parameter that on input contains an **SPropTagArray** structure containing an array of property tags indicating the properties for which to return access rights. If a pointer to NULL is passed in the *lppPropTagArray* parameter, then access rights for all properties of the object are sought. On output, *lppPropTagArray* points to a variable where the returned **SPropTagArray** structure is stored. This returned **SPropTagArray** contains the access rights for the specified properties.

lprgulAccess
Output parameter pointing to a variable where an array of bitmasks of flags is returned. Each bitmask indicates the access rights of one of the individual properties returned in the **SPropTagArray** structure. For each property tag, the following flags can be returned:

IPROP_CLEAN
Indicates the property hasn't been modified.

IPROP_DIRTY
Indicates the property has been modified.

IPROP_READONLY
Indicates the property is read-only.

IPROP_READWRITE
Indicates the property is read/write.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

Remarks

Client applications and service providers call the **IPropData::HrGetPropAccess** method to get the access rights for each property of an object. **HrGetPropAccess** is also used to check whether a property has been modified or deleted. If a property requested has been deleted, it is not returned in the **SPropTagArray** structure; zero is returned instead. If a client or provider passes a pointer to NULL in the *lppPropTagArray* parameter, the deleted property is returned in the array. If a property has been modified, its IPROP_DIRTY flag is set.

See Also

SPropTagArray structure

IPropData::HrSetObjAccess

The **IPropData::HrSetObjAccess** method sets the access rights for an object.

HRESULT HrSetObjAccess(
  ULONG *ulAccess*
)

Parameters

ulAccess
  Input parameter containing a bitmask of flags that sets the access rights of the object. One of the following flags can be set:
  IPROP_READONLY
    Sets the object's properties to read-only.
  IPROP_READWRITE
    Sets the object's properties to read/write.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

Remarks

Client application and service providers call the **IPropData::HrSetObjAccess** method to set the access rights for an entire object. By default, all MAPI objects have read-only access set when they are created. To set access rights on individual properties, a client or provider first sets the access rights for the object by calling **HrSetObjAccess** with the IPROP_READWRITE flag set in the *ulAccess* parameter. It then calls the **IPropData::HrSetPropAccess** method to set the access rights for individual properties.

See Also

IPropData::HrGetPropAccess method, **IPropData::HrSetPropAccess** method

IPropData::HrSetPropAccess

The **IPropData::HrSetPropAccess** method sets the access rights or modification flags for the specified properties.

HRESULT HrSetPropAccess(
  LPSPropTagArray *lpPropTagArray*,
  ULONG FAR * *rgulAccess*
)

Parameters

lpPropTagArray
: Input parameter pointing to an **SPropTagArray** structure containing an array of property tags. These tags indicate the properties affected by the access flags in the array of bitmasks in the *rgulAccess* parameter.

rgulAccess
: Input parameter containing an array of bitmasks of flags used to set the access rights of the properties listed in the **SPropTagArray** structure in the *lpPropTagArray* parameter. For each property tag, the following flags can be set:

  IPROP_CLEAN
  : Sets the individual property to an unmodified state.

  IPROP_DIRTY
  : Sets the individual property to a modified state.

  IPROP_READONLY
  : Sets the individual property to read-only.

  IPROP_READWRITE
  : Sets the individual property to read/write.

Return Values

S_OK
: The call succeeded and has returned the expected value or values.

MAPI_E_NO_ACCESS
: An attempt was made to modify a read-only object or an attempt to access an object for which the user has insufficient permissions.

Remarks

By default, all MAPI properties have read/write access set when they are created. Client applications and service providers call the **IPropData::HrSetPropAccess** method to change the access rights for each individual property in the **SPropTagArray** structure in the *lpPropTagArray* parameter. For each property, there is a corresponding entry in the *rgulAccess* array that is a bitmask . There are two pairs of valid values for these bits; each pair is mutually exclusive. The pairs of valid values are as follows:

IPROP_READONLY	IPROP_CLEAN
IPROP_READWRITE	IPROP_DIRTY

HrSetPropAccess returns MAPI_E_INVALID_PARAMETER when callers try to ignore the mutual exclusivity of these values and pass IPROP_READONLY with IPROP_READWRITE or IPROP_CLEAN with IPROP_DIRTY.

See Also

SPropTagArray structure

IProviderAdmin : IUnknown

The **IProviderAdmin** interface is used to manage the service providers within a message service. A calling process can get a pointer to an **IProviderAdmin** interface in two ways: by calling the **IMsgServiceAdmin::AdminProviders** method or from within a provider's message service entry point function. Client applications calling methods of the **IProviderAdmin** interface are not allowed to create or delete providers; all such changes made to a message service must be made from within the context of a message service entry point function.

Most message services do not allow providers to be added or deleted while the profile is in use. The results are unpredictable if an implementation makes changes to the profile section of a service that doesn't support changes.

At a Glance

Specified in header file:	MAPIDEFS.H
Object that supplies this interface:	Provider administration object
Corresponding pointer type:	LPPROVIDERADMIN
Implemented by:	MAPI
Called by:	Client applications, service providers

Vtable Order

GetLastError	Returns a **MAPIERROR** structure containing information about the last error that occurred for a provider administration object.
GetProviderTable	Returns a table listing the service providers in a message service.
CreateProvider	Adds a service provider to a message service.
DeleteProvider	Deletes a service provider from a message service.
OpenProfileSection	Opens a section of the current profile and returns a pointer that provides further access to the profile object.

IProviderAdmin::CreateProvider

The **IProviderAdmin::CreateProvider** method adds a service provider to a message service.

HRESULT CreateProvider(
  LPTSTR *lpszProvider*,
  ULONG *cValues*,
  LPSPropValue *lpProps*,
  ULONG *ulUIParam*,
  ULONG *ulFlags*,
  MAPIUID FAR * *lpUID*
)

Parameters

lpszProvider
  Input parameter pointing to a string containing the name of the provider to add.

cValues
  Input parameter containing the number of property values in the **SPropValue** structure pointed to by the *lpProps* parameter.

lpProps
  Input parameter pointing to an **SPropValue** structure containing the property values of the properties associated with this provider object.

ulUIParam
  Input parameter containing the handle of the parent window for any dialog boxes or windows this method displays. The *ulUIParam* parameter is used if the MAPI_DIALOG flag is set in the *ulFlags* parameter.

ulFlags
  Input parameter containing a bitmask of flags that controls the type of the passed-in string. The following flag can be set:

  MAPI_UNICODE
    Indicates the passed-in string is in Unicode format. If the MAPI_UNICODE flag is not set, the string is in ANSI format.

lpUID
  Output parameter pointing to the **MAPIUID** structure holding the MAPI unique identifier (MAPIUID) for the provider to add.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

MAPI_E_USER_CANCEL
  The user canceled the operation, typically by clicking the **Cancel** button in a dialog box.

Remarks

Client applications and message-service entry point functions call the **IProviderAdmin::CreateProvider** method to add a provider to a message service. The string in the *lpszProvider* parameter must name a provider that belongs to the message service. MAPI does not verify that the name matches the name of a provider in the service; if the passed name doesn't match a service name, the call succeeds, but the results are unpredictable. Most message services do not allow providers to be added or deleted while the profile is in use.

The message service entry point function for the message service is called with the MSG_SERVICE_PROVIDER_CREATE value set in its *ulContext* parameter. If MAPI_DIALOG is set in the **CreateProvider** method's *ulFlags* parameter, then the values in the *ulUIParam* and *ulFlags* parameters are also passed when the entry point function is called. Service providers should display their configuration property sheets so the user can configure the message service.

The message service entry point function is only called after all available information from the MAPISVC.INF file has been added to the profile.

See Also

MAPIUID structure, **MSGSERVICEENTRY** function prototype, **SPropValue** structure

IProviderAdmin::DeleteProvider

The **IProviderAdmin::DeleteProvider** method deletes a service provider from a message service.

HRESULT DeleteProvider(
  LPMAPIUID *lpUID*
)

Parameters

lpUID
  Input parameter pointing to the **MAPIUID** structure holding the MAPI unique identifier (MAPIUID) for the provider to delete.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

MAPI_E_NOT_FOUND
  The MAPIUID was not recognized.

Remarks

Client applications and message service entry point functions call the **IProviderAdmin::DeleteProvider** method to delete from a message service the provider indicated by the unique identifier in the *lpUID* parameter. Most message services do not allow providers to be added or deleted while the profile is in use.

If the provider to delete is in use by an implementation when **DeleteProvider** is called, **DeleteProvider** returns S_OK but does not delete the provider immediately. Instead, MAPI marks the provider for deletion and deletes it after all implementations have logged off the provider.

The message service entry point function is called for the message service before the provider is removed from the service. The message service entry point function is called with the MSG_SERVICE_PROVIDER_DELETE value set in its *ulContext* parameter. First the function deletes the provider, and then it deletes the provider's profile section. The message service entry point function is not called again after the provider has been deleted.

See Also

MAPIUID structure, **MSGSERVICEENTRY** function prototype

IProviderAdmin::GetLastError

The **IProviderAdmin::GetLastError** method returns a **MAPIERROR** structure containing information about the last error that occurred for a provider administration object.

HRESULT GetLastError(
  HRESULT *hResult***,**
  ULONG *ulFlags***,**
  LPMAPIERROR FAR * *lppMAPIError*
)

Parameters

hResult
  Input parameter containing the result returned for the last call for the provider administration object that returned an error.

ulFlags
  Input parameter containing a bitmask of flags that controls the type of the returned strings. The following flag can be set:

  MAPI_UNICODE
    Indicates the strings in the **MAPIERROR** structure returned in the *lppMAPIError* parameter are in Unicode format. If the MAPI_UNICODE flag is not set, the strings are in ANSI format.

lppMAPIError
  Output parameter pointing to a pointer to the returned **MAPIERROR** structure containing version, component, and context information for the error. The *lppMAPIError* parameter can be set to NULL if there is no **MAPIERROR** structure to return.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

MAPI_E_BAD_CHARWIDTH
  Either the MAPI_UNICODE flag was set and the implementation does not support Unicode, or MAPI_UNICODE was not set and the implementation only supports Unicode.

Remarks

Client applications and message service entry point functions call the **IProviderAdmin::GetLastError** method to retrieve information to display in a message to the user regarding the last error returned from a method call for the provider administration object.

To release all the memory allocated by MAPI for the returned **MAPIERROR** structure, clients need only call the **MAPIFreeBuffer** function.

The return value from **GetLastError** must be S_OK for a client to make use of the **MAPIERROR** structure. Even if the return value is S_OK, a **MAPIERROR** structure might not be returned. If the implementation cannot determine what the last error was, or if a **MAPIERROR** structure is not available for that error, **GetLastError** returns a pointer to NULL in *lppMAPIError* instead.

See Also

MAPIERROR structure, **MAPIFreeBuffer** function

IProviderAdmin::GetProviderTable

The **IProviderAdmin::GetProviderTable** method returns a table listing the service providers in a message service.

HRESULT GetProviderTable(
  ULONG *ulFlags***,**
  LPMAPITABLE FAR * *lppTable*
)

Parameters

ulFlags
  Input parameter containing a bitmask of flags that controls the type of the strings returned in the provider table's default column set. The following flag can be set:

  MAPI_UNICODE
    Indicates the returned strings are in Unicode format. If the MAPI_UNICODE flag is not set, the strings are in ANSI format.

lppTable
  Output parameter pointing to a variable where the pointer to the returned provider table object is stored.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

Remarks

Client applications and message service entry point functions call the **IProviderAdmin::GetProviderTable** method to get a pointer to a table object that lists all of the address book, message store, transport, and message hook providers currently installed as part of a message service. The columns of the provider table contain the current information for the following properties:

PR_DISPLAY_NAME
PR_INSTANCE_KEY
PR_PROVIDER_DISPLAY
PR_PROVIDER_DLL_NAME
PR_PROVIDER_UID
PR_RESOURCE_TYPE
PR_SERVICE_NAME
PR_SERVICE_UID

The following properties are computed only for transport providers:

PR_PROVIDER_ORDINAL
PR_RESOURCE_FLAGS

The provider table's PR_PROVIDER_ORDINAL property can be used to restrict sort operations on the table. The first transport provider in the list has PR_PROVIDER_ORDINAL set to 0, the next provider to 1, and so on; this functionality enables a client to retrieve the table with the list of providers set to the correct order.

Providers that have been deleted, or are in use but have been marked for deletion, are not returned in the provider table. Once a provider table has been returned, it does not reflect changes being made to the profile, such as the addition or deletion of providers. Calls to the **IMAPITable::Advise** method for the provider table return S_OK, but no changes are made to the table.

If no provider exists, **GetProviderTable** does not return an error but returns a table object supporting the **IMAPITable** interface. If the **IMAPITable::QueryRows** method is called on that table, zero rows are returned.

Setting the MAPI_UNICODE flag in the *ulFlags* parameter does the following:

- Sets the string type to Unicode for data returned for the initial active columns of the provider table by the **IMAPITable::QueryColumns** method. The initial active columns for a provider table are those columns the **QueryColumns** method returns before the provider that contains the table calls the **IMAPITable::SetColumns** method.
- Sets the string type to Unicode for data returned for the initial active rows of the provider table by **QueryRows**. The initial active rows for a provider table are those rows **QueryRows** returns before the provider that contains the table calls **SetColumns**.
- Controls the property types of the sort order returned by the **IMAPITable::QuerySortOrder** method before the provider that contains the provider table calls the **IMAPITable::SortTable** method.

GetProviderTable may return extra sections associated with the messaging services in the table in addition to the service providers themselves. Extra sections are those added to the profile using the "Sections" keyword of MAPISVC.INF, or added by providers using the PR_SERVICE_EXTRA_UIDS property of the messaging service profile section. If your application needs to process only service providers, it can identify the extra sections by the fact that the PR_RESOURCE_TYPE column contains a property type of PT_ERROR. This column must be present on service provider sections. Note that this only applies to **IProviderAdmin::GetProviderTable**, and not **IMsgServiceAdmin::GetProviderTable**, which does not include extra sections in the table.

See Also

IMAPITable::QueryColumns method, **IMAPITable::QueryRows** method, **IMAPITable::QuerySortOrder** method, **IMAPITable::SetColumns** method, **IMsgServiceAdmin::GetProviderTable** method

IProviderAdmin::OpenProfileSection

The **IProviderAdmin::OpenProfileSection** method opens a section of the current profile and returns a pointer that provides further access to the profile object.

HRESULT OpenProfileSection(
  LPMAPIUID *lpUID*,
  LPCIID *lpInterface*,
  ULONG *ulFlags*,
  LPPROFSECT FAR * *lppProfSect*
)

Parameters

lpUID
  Input parameter pointing to the **MAPIUID** structure holding the MAPI unique identifier (MAPIUID) for the profile section. Client applications must not pass NULL for the *lpUID* parameter. A service provider can pass NULL to get the MAPIUID when calling from its message service entry point function.

lpInterface
  Input parameter pointing to the interface identifier (IID) for the profile section. Passing NULL indicates the identifier for the profile section object interface for the object, IID_IProfSect, is used. The *lpInterface* parameter can also be set to an identifier for an appropriate interface for the object, for example IID_IMAPIProp or IID_IProfSect.

ulFlags
  Input parameter containing a bitmask of flags that controls how the profile section is opened. The following flags can be set:

  MAPI_DEFERRED_ERRORS
    Indicates the call is allowed to succeed even if the underlying object is not accessible to the calling process. If the object is not accessible, some subsequent call to the object might return an error.

  MAPI_MODIFY
    Requests read/write access. By default, objects are created with read-only access; providers should not work on the assumption that read/write access has been granted. Clients are not allowed read/write access to provider sections of the profile.

lppProfSect
  Output parameter pointing to a variable where the pointer to the returned profile section object is stored.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

MAPI_E_NO_ACCESS
  An attempt was made to modify a read-only profile section or an attempt to access an object for which the user has insufficient permissions.

MAPI_E_NOT_FOUND
  The requested object does not exist.

Remarks

Client applications and message service entry point functions call the **IProviderAdmin::OpenProfileSection** method to open a profile section for reading information from and writing information to the active profile for the session. A profile section object supporting the **IProfSect** interface is returned in the *lppProfSect* parameter. Default behavior is to open the profile section as read-only, unless the call sets the MAPI_MODIFY flag in the *ulFlags* parameter. Clients cannot open profile sections belonging to providers using the **IProviderAdmin::OpenProfileSection** method.

More than one method call can open a profile section with read-only access at a time, but only one method call can open a profile section with read/write access at a time. If any other process has the profile section open, a read/write open operation fails and returns MAPI_E_NO_ACCESS. A read-only open operation fails if the section is open for writing.

If an **OpenProfileSection** call opens a nonexistent profile section by passing MAPI_MODIFY in *ulFlags*, the call creates the section. If an **OpenProfileSection** call attempts to open a nonexistent section with read-only access, it returns MAPI_E_NOT_FOUND.

See Also

IMAPIProp : IUnknown interface, **IProfSect : IMAPIProp** interface, **MAPIUID** structure

ISpoolerHook : IUnknown

The **ISpoolerHook** interface enables a messaging hook provider to reroute messages before they go to their destination.

At a Glance

Specified in header file:	MAPIHOOK.H
Object that supplies this interface:	Messaging hook provider object
Corresponding pointer type:	LPSPOOLERHOOK
Implemented by:	Messaging hook providers
Called by:	The MAPI spooler

Vtable Order

InboundMsgHook	Informs the MAPI spooler it should allow the messaging hook provider to reroute a message from the Inbox to another folder.
OutboundMsgHook	Informs the MAPI spooler it should allow the messaging hook provider to reroute a message from the Sent Items folder to another folder.

ISpoolerHook::InboundMsgHook

The **ISpoolerHook::InboundMsgHook** method informs the MAPI spooler it should allow the messaging hook provider to reroute a message from the Inbox to another folder.

HRESULT InboundMsgHook(
  LPMESSAGE *lpMessage***,**
  LPMAPIFOLDER *lpFolder***,**
  LPMDB *lpMDB***,**
  ULONG FAR * *lpulFlags***,**
  ULONG FAR * *lpcbEntryID***,**
  LPBYTE FAR * *lppEntryID*
)

Parameters

lpMessage
Input parameter pointing to the message to reroute.

lpFolder
Input parameter pointing to the parent folder of the message in its message store.

lpMDB
Input parameter pointing to the message store containing the folder and message.

lpulFlags
Output parameter containing a bitmask of flags that controls how the MAPI spooler responds to the hook for the message indicated in the *lpMessage* parameter. The following flags can be set:

HOOK_CANCEL
Indicates any subsequent hook functions should not be called for this message. If a hook closes, moves, or deletes the message, the messaging hook provider should set this flag.

HOOK_DELETE
Indicates the message should be deleted without being moved.

lpcbEntryID
Input-output parameter pointing to a variable containing the size, in bytes, of the entry identifier pointed to by the *lppEntryID* parameter.

lppEntryID
Input-output parameter pointing to a variable where is stored the pointer to the entry identifier of the folder where the message will be moved.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

Remarks

Messaging hook providers implement the **ISpoolerHook::InboundMsgHook** method to reroute a message from the default Inbox to another folder. Before moving a message to another folder or message store, a messaging hook provider should call the **IUnknown::QueryInterface** method for the message object to make sure the provider can get an interface for the message that is compatible with the provider's implementation.

Before rerouting a message, a messaging hook provider must replace the passed-in entry identifier in *lppEntryID* with the entry identifier of the new target folder. The MAPI spooler moves the message to the indicated folder for the provider, unless another hook function replaces the folder entry identifier. If a hook requires that its operation be the final action on the message, it can set the HOOK_CANCEL flag in the *lpulFlags* parameter before returning.

If a provider replaces the *lppEntryID* entry identifier, it must call the **MAPIFreeBuffer** function to free the previous one. The copy of the entry identifier the provider stores in *lppEntryID* should be allocated using the **MAPIAllocateBuffer** function.

If a messaging hook provider's implementation must move a message to another folder itself, it should close the message, place zero in the *lpcbEntryID* parameter, and free the *lppEntryID* entry identifier, if *lppEntryID* is not already NULL. The hook then sets *lppEntryID* to NULL and places a pointer to the message's new parent folder in *lpFolder*. Messaging hook providers that move the message must set HOOK_CANCEL in *lpulFlags*.

If a hook deletes a message, it should close the message, place zero in *lpcbEntryID*, and place NULL in *lppEntryID* after freeing the existing entry identifier if need be. It then deletes the message and returns HOOK_CANCEL in *lpulFlags*. Alternatively, it can combine the HOOK_CANCEL and HOOK_DELETE flags in *lpulFlags* using the logical OR operator.

The MAPI spooler calls hook providers in the order in which they are specified in the provider section of the profile, as it does transport providers. The MAPI spooler releases the messaging hook provider object at session shutdown. If a provider called the **IUnknown::AddRef** method for a session at initialization, it should call the **IUnknown::Release** method to release the session and any objects, such as message stores, it opened and maintained during the session.

ISpoolerHook::OutboundMsgHook

The **ISpoolerHook::OutboundMsgHook** method informs the MAPI spooler it should allow the messaging hook provider to reroute a message from the Sent Items folder to another folder.

HRESULT OutboundMsgHook(
  LPMESSAGE *lpMessage***,**
  LPMAPIFOLDER *lpFolder***,**
  LPMDB *lpMDB***,**
  ULONG FAR * *lpulFlags***,**
  ULONG FAR * *lpcbEntryID***,**
  LPBYTE FAR * *lppEntryID*
)

Parameters

lpMessage
Input parameter pointing to the message to reroute.

lpFolder
Input parameter pointing to the parent folder of the message in its message store.

lpMDB
Input parameter pointing to the message store containing the folder and message.

lpulFlags
Output parameter containing a bitmask of flags that controls how the MAPI spooler responds to the hook for the message indicated in the *lpMessage* parameter. The following flags can be set:

HOOK_CANCEL
Indicates any subsequent hook functions should not be called for this message. If a hook closes, moves, or deletes the message, the messaging hook provider should set this flag.

HOOK_DELETE
Indicates the message should be deleted without being moved.

lpcbEntryID
Input-output parameter pointing to a variable containing the size, in bytes, of the entry identifier pointed to by the *lppEntryID* parameter.

lppEntryID
Input-output parameter pointing to a variable where the pointer to the entry identifier of the folder where the message will be moved is stored.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

Remarks

Messaging hook providers implement the **ISpoolerHook::OutboundMsgHook** method to reroute a message from the default Sent Items folder to another folder. Before moving a message to another folder or message store, a messaging hook provider should call the **IUnknown::QueryInterface** method for the message object to make sure the provider can get an interface for the message that is compatible with the provider's implementation.

Before rerouting a message, a messaging hook provider must replace the passed-in entry identifier in *lppEntryID* with the entry identifier of the new target folder. The MAPI spooler moves the message to the indicated folder for the provider, unless another hook function replaces the folder entry identifier. If a hook requires that its operation be the final action on the message, it can set the HOOK_CANCEL flag in the *lpulFlags* parameter before returning.

If a provider replaces the *lppEntryID* entry identifier, it must call the **MAPIFreeBuffer** function to free the previous one. The copy of the entry identifier the provider stores in *lppEntryID* should be allocated using the **MAPIAllocateBuffer** function.

If a messaging hook provider's implementation must move a message to another folder itself, it should close the message, place zero in the *lpcbEntryID* parameter, and free the *lppEntryID* entry identifier, if *lppEntryID* is not already NULL. The hook then sets *lppEntryID* to NULL and places a pointer to the message's new parent folder in *lpFolder*. Messaging hook providers that move the message must set HOOK_CANCEL in *lpulFlags.*

If a hook deletes a message, it should close the message, place zero in *lpcbEntryID*, and place NULL in *lppEntryID* after freeing the existing entry identifier if need be. It then deletes the message and returns HOOK_CANCEL in *lpulFlags*. Alternatively, it can combine the HOOK_CANCEL and HOOK_DELETE flags in *lpulFlags* using the logical OR operator.

The MAPI spooler calls hook providers in the order in which they are specified in the provider section of the profile, as it does transport providers. The MAPI spooler releases the messaging hook provider object at session shutdown. If a provider called the **IUnknown::AddRef** method for a session at initialization, it should call the **IUnknown::Release** method to release the session and any objects, such as message stores, it opened and maintained during the session.

ITableData : IUnknown

The **ITableData** interface provides utility methods for working with tables.

At a Glance

Specified in header file:	MAPIUTIL.H
Object that supplies this interface:	Table data object
Corresponding pointer type:	LPTABLEDATA
Implemented by:	MAPI
Called by:	Client applications, service providers

Vtable Order

HrGetView	Creates a new view for a table.
HrModifyRow	Modifies a row in a table, or adds a row to a table.
HrDeleteRow	Deletes a row from a table.
HrQueryRow	Returns all properties of a specified row in a table and its row index in that table.
HrEnumRow	Returns the properties contained in a row of a table.
HrNotify	Finds a particular table row so as to send a notification about that row.
HrInsertRow	Inserts a row into a table.
HrModifyRows	Modifies multiple rows in a table, or adds multiple rows to a table.
HrDeleteRows	Deletes multiple rows from a table.

ITableData::HrDeleteRow

The **ITableData::HrDeleteRow** method deletes a row from a table.

HRESULT HrDeleteRow(
  LPSPropValue *lpSPropValue*
)

Parameters

lpSPropValue
  Input parameter pointing to an **SPropValue** structure that holds the property value for the property that indicates the index number of the row to delete. This property value must contain the same index column value as was passed for the *ulPropTagIndexColumn* parameter of the call to the **CreateTable** function when the table was created.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

MAPI_E_NOT_FOUND
  The **SPropValue** structure passed does not match a corresponding row in the table.

Remarks

Client applications and service providers call the **ITableData::HrDeleteRow** method to delete a table row. To perform this deletion, **HrDeleteRow** takes in the *lpSPropValue* parameter a property value indicating the row's index number and uses this index to locate and delete the row from the underlying table and from any open views for the table. This property value must contain the same index column value as was passed for the **CreateTable** *ulPropTagIndexColumn* when the table was created. If no row with that index number exists, **HrDeleteRow** returns MAPI_E_NOT_FOUND.

After the row is deleted, notifications are sent to all implementations that have an open view on the table and that have registered to receive notifications for table modifications.

Deleting a row does not reduce the columns available to existing views or subsequently opened views, even if the deleted row was the last row with a value for a specific column.

See Also

CreateTable function, **ITableData::HrDeleteRows** method, **ITableData::HrModifyRow** method, **SPropValue** structure, **TABLE_NOTIFICATION** structure

ITableData::HrDeleteRows

The **ITableData::HrDeleteRows** method deletes multiple rows from a table.

HRESULT HrDeleteRows(
  ULONG *ulFlags***,**
  LPSRowSet *lprowsetToDelete***,**
  ULONG FAR * *cRowsDeleted*
)

Parameters

ulFlags
  Input parameter containing a bitmask of flags that controls how table rows are deleted. The following flag can be set:

  TAD_ALL_ROWS
    Deletes all rows from the underlying table data and all corresponding table views and sends a single TABLE_RELOAD notification to all table views registered for notifications.

lprowsetToDelete
  Input parameter pointing to an **SRowSet** structure containing a counted array of index properties, each of which indicates a row to delete. To be deleted, each row must have an index property that is unique among all rows for this table. The *lprowsetToDelete* parameter can be NULL if the TAD_ALL_ROWS flag is set in the *ulFlags* parameter.

cRowsDeleted
  Output parameter containing a variable where the returned number of rows deleted is stored.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

Remarks

Client applications and service providers call the **ITableData::HrDeleteRows** method to find and delete multiple table rows. To perform this deletion, **HrDeleteRows** locates and deletes each row corresponding to an index property in the **SRowSet** structure passed in *lprowsetToDelete*. The property values in *lprowsetToDelete* must contain the same index column values as passed for the *ulPropTagIndexColumn* parameter of the call to the **CreateTable** function when the table was created.

HrDeleteRows returns in the *cRowsDeleted* parameter the number of rows actually deleted. It does not return an error for rows that were not found. For example, if none of the rows requested can be found, zero is returned in *cRowsDeleted*. If TAD_ALL_ROWS is set in *ulFlags*, all rows in the table are deleted.

After the rows are deleted, a single notification is sent to each implementation that has an open view on the table and that has registered to receive notifications for table modifications.

Deleting rows does not reduce the columns available to existing table views or subsequently opened table views, even if the rows deleted were the last with values for a specific column.

See Also

CreateTable function, **ITableData::HrDeleteRow** method, **ITableData::HrModifyRows** method, **SRowSet** structure, **TABLE_NOTIFICATION** structure

ITableData::HrEnumRow

The **ITableData::HrEnumRow** method returns the properties contained in a row of a table.

HRESULT HrEnumRow(
  ULONG *ulRowNumber***,**
  LPSRow FAR * *lppSRow*
)

Parameters

ulRowNumber
  Input parameter indicating the number of the row for which to return properties. The value in the *ulRowNumber* parameter can be any value from 0 through *n* - 1, where *n* is the total number of rows in the table.

lppSRow
  Output parameter pointing to a variable where a pointer to the returned **SRow** structure holding property information about the row is stored.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

Remarks

Client applications and service providers call the **ITableData::HrEnumRow** method to return the full property set for the row indicated in *ulRowNumber*. Rows can be enumerated with multiple calls to this method. The rows are returned based on the chronological order that they were added to the table. This chronological order is maintained for the life of the table object.

If the row number indicated in *ulRowNumber* does not exist, NULL is returned in the **SRow** structure in the *lppSRow* parameter and the method returns S_OK.

MAPI allocates memory for the returned **SRow** structure using the **MAPIAllocateBuffer** function when the table is created. The calling process must release this memory by calling the **MAPIFreeBuffer** function when done with the **SRow**.

See Also

MAPIAllocateBuffer function, **MAPIFreeBuffer** function, **SRow** structure

ITableData::HrGetView

The **ITableData::HrGetView** method creates a new view for a table.

HRESULT HrGetView(
  LPSSortOrderSet *lpSSortOrderSet*,
  CALLERRELEASE FAR * *lpfCallerRelease*,
  ULONG *ulCallerData*,
  LPMAPITABLE FAR * *lppMAPITable*
)

Parameters

lpSSortOrderSet
: Input parameter pointing to an **SSortOrderSet** structure holding the default sort order for the view. If NULL is passed in the *lpSSortOrderSet* parameter, no initial sorting is done.

lpfCallerRelease
: Input parameter pointing to a callback function to call when the view is released. This callback function is passed the data contained in the *ulCallerData* parameter. If NULL is passed in the *lpfCallerRelease* parameter, no callback is made.

ulCallerData
: Input parameter containing 32-bit data that the calling process requires saved with the new view and returned in the release callback.

lppMAPITable
: Output parameter pointing to a variable where the pointer to the newly created view is stored. The table view pointed to by the *lppMAPITable* parameter is the table that a service provider passes to a client application.

Return Values

S_OK
: The call succeeded and has returned the expected value or values.

Remarks

Client applications and service providers call the **ITableData::HrGetView** method to create a new view for a table. All rows and columns of the table are initially visible in the view; no restriction is initially applied to the table. When a sort order is specified in *lpSSortOrderSet*, the view is sorted according to the specified order and the cursor is placed at the beginning of the first row.

When the calling process requires more complex initial conditions for a view than can be produced by sorting, it should set a sort order, set a restriction, and then return the view to the client application.

When the table object returned by **HrGetView** is released, the callback function, based on the **CALLERRELEASE** function prototype, that is specified in the *lpfCallerRelease* parameter is called with the data contained in the *ulCallerData* parameter, a pointer to the table data object the view applies to, and the pointer to the table object being released.

See Also

CALLERRELEASE function prototype, **IMAPITable : IUnknown** interface, **SSortOrderSet** structure

ITableData::HrInsertRow

The **ITableData::HrInsertRow** method inserts a row into a table.

HRESULT HrInsertRow(
  ULONG *uliRow***,**
  LPSRow *lpSRow*
)

Parameters

uliRow
  Input parameter containing the number of the table row before which the new row is inserted. The *uliRow* parameter can hold row numbers from 0 through *n*, where *n* is the total number of rows in the table; passing *n* in *uliRow* results in the row being appended to the table's end.

lpSRow
  Input parameter pointing to an **SRow** structure containing all properties for the row being inserted.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

MAPI_E_INVALID_PARAMETER
  A row already exists with the same index column.

Remarks

Client applications and service providers call the **ITableData::HrInsertRow** method to insert a row into a table. Passing a value of 0 through *n* - 1 in the *uliRow* parameter results in the new row being inserted above the table row with the given number. Passing a value of *n* in *uliRow* results in the new row being appended to the table's end.

One of the property columns in the **SRow** structure in the *lpSRow* parameter must be a column holding the index property of the row to be inserted, which uniquely identifies the row within the table. If this value for the index column already exists within the table, **HrInsertRow** returns MAPI_E_INVALID_PARAMETER. If there is no current row with this index value, **HrInsertRow** adds the new row. The property columns in the **SRow** structure do not have to be in the same order as the property columns in the table.

After the row is inserted, notifications are sent to all implementations that have an open view on the table and that have registered to receive notifications for table modifications. Notifications are not sent if the rows have been restricted out of the view on the table.

See Also

SRow structure, **TABLE_NOTIFICATION** structure

ITableData::HrModifyRow

The **ITableData::HrModifyRow** method modifies a row in a table, or adds a row to a table.

```
HRESULT HrModifyRow(
   LPSRow lpSRow
  )
```

Parameters

lpSRow
: Input parameter pointing to an **SRow** structure containing the set of properties for the row to add or modify. The indicated row must contain the same index column as passed in the *ulPropTagIndexColumn* parameter of the call to the **CreateTable** function when the table was created.

Return Values

S_OK
: The call succeeded and has returned the expected value or values.

MAPI_E_INVALID_PARAMETER
: The passed-in row does not have an index column.

Remarks

Client applications and service providers call the **ITableData::HrModifyRow** method to modify one row of a table or add one row to a table. The **SRow** structure specified in the *lpSRow* parameter contains the properties for the row to add or modify.

One of the property columns in the **SRow** structure must be a column holding the index property of the row to modify or add; this index column must be the same as passed for *ulPropTagIndexColumn* in the **CreateTable** call that created the table. **HrModifyRow** replaces with the new row any row in the table that has the same index value as in the **SRow** structure. If there is no current row with this index value, **HrModifyRow** adds the new row to the end of the table.

The property columns in the **SRow** structuredo not have to be in the same order as the property columns in the table. The **SRow** structure can also include properties for which there are no current columns in the table; MAPI adds new columns to the table as needed.

The row is modified or inserted for all views on the table object. Doing so can involve adding the row to or removing it from a view based on a restriction in effect for the view. Columns added because the **SRow** structure includes properties for which there are no current columns become available to existing views but are not included in existing views' currently active columns. However, added columns are active in views opened after their addition. After a row is inserted, notifications are sent to all implementations that have an open view on the table and that have registered to receive notifications for table modifications.

Any text or string properties placed in a client table must be in the client's character set, whether ANSI or Unicode. MAPI's table implementation does not handle character set conversions.

See Also

SRow structure, **TABLE_NOTIFICATION** structure

ITableData::HrModifyRows

The **ITableData::HrModifyRows** method modifies multiple rows in a table, or adds multiple rows to a table.

HRESULT HrModifyRows(
  ULONG *ulFlags***,**
  LPSRowSet *lpSRowSet*
)

Parameters

ulFlags
  Reserved; must be zero.

lpSRowSet
  Input parameter pointing to an **SRowSet** structure containing the set of rows to add or modify.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

MAPI_E_INVALID_PARAMETER
  One or more of the passed-in rows does not have an index column. If this error is returned, no rows are changed.

Remarks

Client applications and service providers call the **ITableData::HrModifyRows** method to modify the table rows indicated in the **SRowSet** structure passed in the *lpSRowSet* parameter. If no current table rows match the rows in the **SRowSet**, **HrModifyRows** adds the indicated rows to the table. **HrModifyRows** replaces with the **SRowSet** structure rows any rows in the table that have the same index values. If there are no current rows with the same values, **HrModifyRows** adds the new rows to the table's end.

Each row in the **SRowSet** structure must have a property column holding the index property of that row, which uniquely identifies the row within the table. These property columns must contain the same index properties as passed in the *ulPropTagIndexColumn* parameter of the **CreateTable** function call that the table was created. If one of the rows has a different index column, **HrModifyRows** returns MAPI_E_INVALID PARAMETER and no rows are changed.

The property columns in the **SRowSet** structure do not have to be in the same order as the property columns in the table. The **SRowSet** structure can also include properties for which there are no current columns in the table; MAPI adds new columns to the table as needed.

If any views are open for the table, the added or modified rows are inserted or moved appropriately in each view. Doing so can involve adding the rows to or removing them from a view based on a restriction in effect for the view. Columns added because the **SRowSet** structure includes properties for which there are no current columns become available to existing views but are not included in existing views' currently active columns. However, added columns are active in views opened after their addition. After the rows are removed, a single notification is sent to each implementation that has an open view on the table and that has registered to receive notifications for table modifications.

Notifications to views on the table object are made in one batch, but if more than eight notifications are to be sent a single TABLE_CHANGED notification is sent instead.

Any text or string properties placed in a client table must be in the client's character set, whether ANSI or Unicode. MAPI's table implementation does not handle character set conversions.

See Also

SRowSet structure

ITableData::HrNotify

The **ITableData::HrNotify** method finds a particular table row so as to send a notification about that row.

HRESULT HrNotify(
  ULONG *ulFlags***,**
  ULONG *cValues***,**
  LPSPropValue *lpSPropValue*
)

Parameters

ulFlags
  Reserved; must be zero.

cValues
  Input parameter containing the number of property values in the **SPropValue** structure pointed to by the *lpSPropValue* parameter.

lpSPropValue
  Input parameter pointing to an **SPropValue** structure containing the property values of the properties used to locate a particular row. The located row's properties must exactly match those passed in the **SPropValue** structure.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

Remarks

Client applications and service providers call the **ITableData::HrNotify** method to get a detailed notification of type `fnevTableModifed` for the table row whose property values match those passed in *lpSPropValue*, even when the values within that row haven't changed. If the property values for the identified row have not changed, the existing row information is sent in the `fnevTableModifed` notification. In the case of a change to a display table row for an edit control, a client should reload the data associated with the control.

See Also

SPropValue structure, **TABLE_NOTIFICATION** structure

ITableData::HrQueryRow

The **ITableData::HrQueryRow** method returns all properties of a specified row in a table and its row index in that table.

HRESULT HrQueryRow(
  LPSPropValue *lpSPropValue***,**
  LPSRow FAR * *lppSRow***,**
  ULONG FAR * *lpuliRow*
)

Parameters

lpSPropValue
  Input parameter pointing to an **SPropValue** structure that holds the property value for the index property specifying the row for which to return properties.

lppSRow
  Output parameter pointing to a variable where a pointer to the returned **SRow** structure holding all properties for the specified row is stored.

lpuliRow
  Output parameter pointing to the returned index number for the returned row. This number is the index property of the row, which uniquely identifies the row within the table. If no row index number is required, NULL should be passed in the *lpuliRow* parameter.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

MAPI_E_INVALID_PARAMETER
  The **SPropValue** structure passed in does not contain an index column.

Remarks

Client applications and service providers call the **ITableData::HrQueryRow** method to retrieve the full property set for the row indicated by the index property in the *lpSPropValue* parameter. **HrQueryRow** also returns the row index for this row within the table.

MAPI uses, but does not modify, the passed-in **SPropValue** structure. If the calling client or provider allocated memory for this structure, the client or provider must free this memory after the **HrQueryRow** call returns.

MAPI allocates memory for the returned **SRow** structure using the **MAPIAllocateBuffer** function when the table is created. The calling client or provider must release this memory by calling the **MAPIFreeBuffer** function when done with the **SRow** structure.

See Also

MAPIAllocateBuffer function, **MAPIFreeBuffer** function, **SPropValue** structure, **SRow** structure

ITnef : IUnknown

The **ITnef** interface provides methods for encapsulating those MAPI properties not supported by a messaging system into binary streams that can accompany messages through transport provider handling and through gateways. The format used for this encapsulation is Transport-Neutral Encapsulation Format (TNEF). The target transport provider can then, on receiving a message including TNEF-encapsulated properties, decode the encapsulation to retrieve all the properties of the original message.

At a Glance

Specified in header file:	TNEF.H
Object that supplies this interface:	TNEF object
Corresponding pointer type:	LPTNEF
Implemented by:	MAPI
Called by:	Transport providers, message store providers, gateways

Vtable Order

AddProps	Allows the calling service provider or gateway to add properties to include in the encapsulation of a message or an attachment.
ExtractProps	Extracts the properties from a TNEF encapsulation.
Finish	Finishes processing for all TNEF operations that are queued and waiting.
OpenTaggedBody	Opens a stream interface on the text of an encapsulated message.
SetProps	Sets the value of one or more properties for an encapsulated message or attachment without modifying the original message or attachment.
EncodeRecips	Encodes a view for a message's recipient table in the TNEF data stream for the message.
FinishComponent	Processes individual components from a message one at a time into a TNEF stream.

ITnef::AddProps

The **ITnef::AddProps** method allows the calling service provider or gateway to add properties to include in the encapsulation of a message or an attachment.

HRESULT AddProps(
  ULONG *ulFlags*,
  ULONG *ulElemID*,
  LPVOID *lpvData*,
  LPSPropTagArray *lpPropList*
)

Parameters

ulFlags
  Input parameter containing a bitmask of flags that controls how properties are included in or excluded from encapsulation. The following flags can be set:

  TNEF_PROP_ATTACHMENTS_ONLY
    Encodes only the properties in the *lpPropList* parameter that are part of attachments within the message.

  TNEF_PROP_CONTAINED
    Encodes only properties from the attachment specified by the *ulElemID* parameter. If the *lpvData* parameter is non-null, then the data pointed to is written into the attachment's encapsulation in the transport file indicated in the PR_ATTACH_TRANSPORT_NAME property.

  TNEF_PROP_CONTAINED_TNEF
    Encodes only properties from the message or attachment specified by *ulElemID*. If this flag is set, the value in *lpvData* must be an ISTREAMTNEF pointer.

  TNEF_PROP_EXCLUDE
    Encodes all properties not specified in *lpPropList*.

  TNEF_PROP_INCLUDE
    Encodes all properties specified in *lpPropList*.

  TNEF_PROP_MESSAGE_ONLY
    Encodes only those properties specified in *lpPropList* that are part of the message itself.

ulElemID
  Input parameter containing an attachment's PR_ATTACH_NUM property, which holds a number that uniquely identifies the attachment within its parent message. The *ulElemID* parameter is used when special handling is requested for an attachment. The *ulElemID* parameter should be zero unless the TNEF_PROP_CONTAINED or TNEF_PROP_CONTAINTAINED_TNEF flag is set in the *ulFlags* parameter.

lpvData
  Input parameter pointing to attachment data used to replace the data of the attachment specified in *ulElemID*. The *lpvData* parameter should be NULL unless TNEF_PROP_CONTAINED or TNEF_PROP_CONTAINTAINED_TNEF is set in *ulFlags*.

lpPropList
  Input parameter pointing to the list of properties to include in or exclude from encapsulation.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

Remarks

Transport providers, message store providers, and gateways call the **ITnef::AddProps** method to list properties to be included in or excluded from the TNEF encapsulation of a message or an attachment. Using successive calls, the provider or gateway can specify a list of properties to add and encode or to exclude from being encoded. Providers and gateways can also use **AddProps** to provide information on any special handling attachments should be given.

AddProps is only supported for TNEF objects opened with the TNEF_ENCODE flag for the **OpenTnefStream** or **OpenTnefStreamEx** function.

Note that no actual TNEF encoding happens for **AddProps** until the **ITnef::Finish** method is called. This functionality means that pointers passed into **AddProps** must remain valid until after the call to **Finish** is made. At that point, all objects and data passed in with **AddProps** calls can be released or freed.

See Also

ITnef::Finish method, **OpenTnefStream** function, **OpenTnefStreamEx** function, PR_ATTACH_TRANSPORT_NAME property

ITnef::EncodeRecips

The **ITnef::EncodeRecips** method encodes a view for a message's recipient table in the TNEF data stream for the message.

HRESULT EncodeRecips(
  ULONG *ulFlags***,**
  LPMAPITABLE *lpRecipientTable*
)

Parameters

ulFlags
  Reserved; must be zero.

lpRecipientTable
  Input parameter pointing to the recipient table for which the view is encoded. The *lpRecipientTable* parameter can be NULL.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

Remarks

Transport providers, message store providers, and gateways call the **ITnef::EncodeRecips** method to perform TNEF encoding for a particular recipient table view. Such encoding is useful, for instance, if a provider or gateway requires a particular column set, sort order, or restriction for the recipient table.

A provider or gateway passes the table view to be encoded in *lpRecipientTable*. The TNEF implementation encodes the recipient table with the given view, using the given column set, sort order, restriction, and position. If a provider or gateway passes NULL in *lpRecipientTable*, TNEF gets the recipient table from the message being encoded, using the **IMessage::GetRecipientTable** method, and processes every row of the table into the TNEF stream using the table's current settings.

Calling **EncodeRecips** with NULL in *lpRecipientTable* thus encodes all message recipients and is equivalent to calling the **ITnef::AddProps** method with the TNEF_PROP_INCLUDE flag in its *ulFlags* parameter and the PR_MESSAGE_RECIPIENTS property in its *lpPropList* parameter.

See Also

IMessage::GetRecipientTable method, **ITnef::AddProps** method, PR_MESSAGE_RECIPIENTS property

ITnef::ExtractProps

The **ITnef::ExtractProps** method extracts the properties from a TNEF encapsulation.

HRESULT ExtractProps(
  ULONG *ulFlags*,
  LPSPropTagArray *lpPropList*,
  LPSTnefProblemArray FAR * *lpProblems*
)

Parameters

ulFlags
  Input parameter containing a bitmask of flags that controls how properties are decoded. The following flags can be set:
  TNEF_PROP_EXCLUDE
    Decodes all properties not specified in the *lpPropList* parameter.
  TNEF_PROP_INCLUDE
    Decodes all properties specified in *lpPropList*.

lpPropList
  Input parameter pointing to the list of properties to include in or exclude from the decoding operation.

lpProblems
  Output parameter pointing to a variable where a pointer to a returned **STnefProblemArray** structure is stored. The **STnefProblemArray** indicates which properties were not encoded properly, if any. If NULL is passed in the *lpProblems* parameter, no property problem array is returned.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

MAPI_E_CORRUPT_DATA
  Data being decoded into a stream is corrupted.

Remarks

Transport providers, message store providers, and gateways call the **ITnef::ExtractProps** method to extract – that is, decode – properties from the encapsulation of a message or an attachment that was passed to the **OpenTnefStream** function. The calling provider or gateway can specify a list of properties to decode. Providers and gateways can also use **ExtractProps** to provide information on any special handling attachments should be given.

Once decoding is done, the original message passed into **OpenTnefStream** is repopulated with the decoded properties. Subsequent **ExtractProps** calls go back to the message and extract the new list of properties.

Unlike the **ITnef::AddProps** method, which queues requested actions until the **ITnef::Finish** method is called, properties are decoded when the **ExtractProps** call is made. For that reason, the target message for encapsulation decoding should be relatively empty. Existing properties in the target message are overwritten by encapsulated properties.

ExtractProps is only supported for objects opened with the TNEF_DECODE flag for the **OpenTnefStream** or **OpenTnefStreamEx** function.

The TNEF implementation reports TNEF stream encoding problems without halting the **ExtractProps** process. The **STnefProblemArray** structure returned in *lpProblems* indicates which TNEF attributes or MAPI properties, if any, could not be processed. The value returned in the **scode** member of the

STnefProblemArray indicates the specific problem. The provider or gateway can work on the assumption that all properties or attributes for which **ExtractProps** does not return a problem report were processed successfully.

One exception is that if, during the decoding of a TNEF stream, a property in the MAPI encapsulation block cannot be processed and leaves the stream unreliable; then decoding of the encapsulation block is halted and a problem is reported. The problem array for this type of problem contains 0L for the **ulPropTag** member, `attMAPIProps` or `attAttachment` for the **ulAttribute** member, and MAPI_E_UNABLE_TO_COMPLETE for the **scode** member. Note that the decoding of the stream is not halted, just the decoding of the MAPI encapsulation block. The stream decoding continues with the next attribute block.

If a provider or gateway does not work with problem arrays, it can pass NULL in *lpProblems;* in this case, no problem array is returned.

The value returned in *lpProblems* is only valid if the call returns S_OK. When S_OK is returned, the provider or gateway should check the values returned in the **STnefProblemArray** structure. If an error occurs on the call, then the **STnefProblemArray** structure is not filled in and the calling provider or gateway should not use or free the structure. If no error occurs on the call, the calling provider or gateway must release the memory for the **STnefProblemArray** by calling the **MAPIFreeBuffer** function.

See Also

ITnef::AddProps method, **ITnef::Finish** method, **ITnef::SetProps** method, **MAPIFreeBuffer** function, **OpenTnefStream** function, **OpenTnefStreamEx** function, **STnefProblemArray** structure

ITnef::Finish

The **ITnef::Finish** method finishes processing for all TNEF operations that are queued and waiting.

HRESULT Finish(
  ULONG *ulFlags***,**
  WORD FAR * *lpKey***,**
  LPSTnefProblemArray FAR * *lpProblem*
)

Parameters

ulFlags
  Reserved; must be zero.

lpKey
  Output parameter pointing to the PR_ATTACH_NUM key property of an attachment. The TNEF encapsulation object uses this key to match an attachment to its attachment placement tag within a message. This key should be unique across messages.

lpProblem
  Output parameter pointing to a variable where a pointer to a returned **STnefProblemArray** structure is stored. The **STnefProblemArray** indicates which properties were not encoded properly, if any. If NULL is passed in the *lpProblem* parameter, no property problem array is returned.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

Remarks

Transport providers, message store providers, and gateways call the **ITnef::Finish** method to perform the encoding of all properties for which encoding was requested in calls to the **ITnef::AddProps** and **ITnef::SetProps** methods. If the TNEF object to receive encoding was opened with the TNEF_ENCODE flag for the **OpenTnefStream** or **OpenTnefStreamEx** function, all properties processed in the **Finish** call are encoded into the encapsulation stream passed to that object.

After the **Finish** call, the pointer to the encapsulation stream is set to the end of the TNEF data. If the provider or gateway is to use the TNEF stream data, it must reset the stream pointer to the beginning of the TNEF stream data.

Finish is only supported on objects opened with TNEF_ENCODE set for **OpenTnefStream** or **OpenTnefStreamEx**.

The TNEF implementation reports TNEF stream encoding problems without halting the **Finish** process. The **STnefProblemArray** structure returned in *lpProblem* indicates which TNEF attributes or MAPI properties, if any, could not be processed. The value returned in the **scode** member of the **STnefProblemArray** indicates the specific problem. The provider or gateway can work on the assumption that all properties or attributes for which **Finish** does not return a problem report were processed successfully.

If a provider or gateway does not work with problem arrays, it can pass NULL in *lpProblem;* in this case, no problem array is returned.

The value returned in *lpProblem* is only valid if the call returns S_OK. When S_OK is returned, the provider or gateway should check the values returned in the **STnefProblemArray** structure. If an error occurs on the call, then the **STnefProblemArray** structure is not filled in and the calling provider or gateway should not use or free the structure. If no error occurs on the call, the calling provider or gateway must release the memory for the **STnefProblemArray** by calling the **MAPIFreeBuffer** function.

See Also

ITnef::AddProps method, **MAPIFreeBuffer** function, **OpenTnefStream** function, **OpenTnefStreamEx** function, PR_ATTACH_NUM property, **STnefProblemArray** structure

ITnef::FinishComponent

The **ITnef::FinishComponent** method processes individual components from a message one at a time into a TNEF stream.

HRESULT FinishComponent(
  ULONG *ulFlags***,**
  ULONG *ulComponentID***,**
  LPSPropTagArray *lpCustomPropList***,**
  LPSPropValue *lpCustomProps***,**
  LPSPropTagArray *lpPropList***,**
  LPSTnefProblemArray FAR * *lppProblems*
)

Parameters

ulFlags
  Input parameter containing a bitmask of flags that controls which component has processing finished. One or the other of the following flags must be set:

  TNEF_COMPONENT_ATTACHMENT
    Indicates an attachment object is encoded; the *ulComponentID* parameter contains the PR_ATTACH_NUM property of the attachment processed.

  TNEF_COMPONENT_MESSAGE
    Indicates a message object is encoded.

ulComponentID
  Input parameter containing either zero to indicate processing for a message or the PR_ATTACH_NUM property of the attachment to be processed. If the TNEF_COMPONENT_MESSAGE flag is set in the *ulFlags* parameter, *ulComponentID* must be zero.

lpCustomPropList
  Input parameter pointing to an **SPropTagArray** structure that holds property tags identifying the properties passed in the *lpCustomProps* parameter. There must be a one-to-one correspondence between each property value in *lpCustomProps* and a property tag in the *lpCustomPropList* parameter.

lpCustomProps
  Input parameter pointing to an **SPropValue** structure containing property values for the properties to encode.

lpPropList
  Input parameter pointing to an **SPropTagArray** structure containing property tags for the properties to encode.

lppProblems
  Output parameter pointing to a variable where a pointer to a returned **STnefProblemArray** structure is stored. The **STnefProblemArray** structure indicates which properties were not encoded properly, if any. If NULL is passed in the *lppProblems* parameter, no property problem array is returned.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

Remarks

Transport providers, message store providers, and gateways call the **ITnef::FinishComponent** method to perform TNEF processing for one component, either a message or an attachment, as indicated by the flag set in the *ulFlags* parameter.

For component processing to be enabled, the calling provider or gateway must have passed the TNEF_COMPONENT_ENCODING flag in *ulFlags* for the **OpenTnefStream** function or **OpenTnefStreamEx** function that opened the object to receive encoding.

Passing values in *lpCustomPropList* and *lpCustomProps* performs component encoding equivalent to that done by the **ITnef::SetProps** method. Passing a value in the *lpPropList* parameter performs component encoding equivalent to that done by the **ITnef::AddProps** method with the TNEF_PROP_INCLUDE flag set in *ulFlags*. Passing such values enables you to perform encodings with a single call rather than multiple calls.

The TNEF implementation reports TNEF stream encoding problems without halting the **FinishComponent** process. The **STnefProblemArray** structure returned in *lppProblems* indicates which TNEF attributes or MAPI properties, if any, could not be processed. The value returned in the **scode** member of the **STnefProblemArray** indicates the specific problem. The provider or gateway can work on the assumption that all properties or attributes for which **FinishComponent** does not return a problem report were processed successfully.

If a provider or gateway does not work with problem arrays, it can pass NULL in *lppProblems;* in this case, no problem array is returned.

The value returned in *lppProblems* is only valid if the call returns S_OK. When S_OK is returned, the provider or gateway should check the values returned in the **STnefProblemArray** structure. If an error occurs on the call, then the **STnefProblemArray** structure is not filled in and the calling provider or gateway should not use or free the structure. If no error occurs on the call, the calling provider or gateway must release the memory for the **STnefProblemArray** by calling the **MAPIFreeBuffer** function.

See Also

ITnef::AddProps method, **ITnef::SetProps** method, **MAPIFreeBuffer** function, **OpenTnefStream** function, **OpenTnefStreamEx** function, **SPropTagArray** structure, **STnefProblemArray** structure

ITnef::OpenTaggedBody

The **ITnef::OpenTaggedBody** method opens a stream interface on the text of an encapsulated message.

HRESULT OpenTaggedBody(
  LPMESSAGE *lpMessage***,**
  ULONG *ulFlags***,**
  LPSTREAM FAR * *lppStream*
)

Parameters

lpMessage
: Input parameter pointing to the message the stream is associated with. This message is not required to be the same message passed in the call to the **OpenTnefStream** or **OpenTnefStreamEx** function beginning TNEF encoding.

ulFlags
: Input parameter containing a bitmask of flags that controls how stream interface is opened. The following flags can be set:

  MAPI_CREATE
  : Indicates that the property does not exist, it should be created. If the property does exist, the current data in the property should be discarded. When an implementation sets the MAPI_CREATE flag, it should also set the MAPI_MODIFY flag.

  MAPI_MODIFY
  : Requests read/write access. The default interface is read-only. MAPI_MODIFY must be set whenever MAPI_CREATE is set.

lppStream
: Output parameter pointing to a variable in which is stored the pointer to a stream object that contains the text from the PR_BODY property of the passed-in encapsulated message and that supports the **IStream** interface.

Return Values

S_OK
: The call succeeded and has returned the expected value or values.

Remarks

Transport providers, message store providers, and gateways call the **ITnef::OpenTaggedBody** method to open a stream interface on the text of an encapsulated message (that is, on a TNEF object).

As part of its processing, **ITnef::OpenTaggedBody** either inserts or parses attachment tags that indicate the position of any attachments or OLE objects within the message text. The attachment tags are in the following format:

[[*attachment name* **:** *n* **in** *attachment container name* **]]**

where *attachment name* describes the attachment object; *n* is a number identifying the attachment that is part of a sequence, incrementing from the value passed in the *lpKey* parameter of the **OpenTnefStream** or **OpenTnefStreamEx** function; and *attachment container name* describes the physical component where the attachment object resides.

OpenTaggedBody reads out message text and inserts an attachment tag wherever an attachment object originally appeared in the text. The original message text is not changed.

When a message that has tags is passed to a stream, the tags are stripped out and the attachment objects are relocated in the position of the tags in the stream.

See Also

OpenTnefStream function, **OpenTnefStreamEx** function, PR_BODY property

ITnef::SetProps

The **ITnef::SetProps** method sets the value of one or more properties for an encapsulated message or attachment without modifying the original message or attachment.

HRESULT SetProps(
  ULONG *ulFlags*,
  ULONG *ulElemID*,
  ULONG *cValues*,
  LPSPropValue *lpProps*
)

Parameters

ulFlags
  Input parameter containing a bitmask of flags that controls how properties' values are set. The following flag can be set:

  TNEF_PROP_CONTAINED
    Encodes only properties from the message or attachment specified by the *ulElemID* parameter.

ulElemID
  Input parameter containing an attachment's PR_ATTACH_NUM property, which holds a number that uniquely identifies the attachment within its parent message.

cValues
  Input parameter containing the number of property values in the **SPropValue** structure pointed to by the *lpProps* parameter.

lpProps
  Input parameter pointing to an **SPropValue** structure containing the property values of the properties to set.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

Remarks

Transport providers, message store providers, and gateways call the **ITnef::SetProps** method to set properties to include in the encapsulation of a message or an attachment without modifying the original message or attachment. Any properties set with this call override existing properties in the encapsulated message.

SetProps is only supported for encapsulated messages (that is, TNEF objects) opened with the TNEF_ENCODE flag for the **OpenTnefStream** or **OpenTnefStreamEx** function. Any number of properties can be set with this call.

Note that no actual TNEF encoding for **SetProps** happens until after the **ITnef::Finish** method is called. This functionality means that pointers passed into **SetProps** must remain valid until after the call to **Finish** is made. At that point, all objects and data passed into **SetProps** calls can be released or freed.

See Also

ITnef::Finish method, **OpenTnefStream** function, **OpenTnefStreamEx** function, PR_ATTACH_NUM property, **SPropValue** structure

IXPLogon : IUnknown

The **IXPLogon** interface is used to provide the MAPI spooler access within a transport provider.

At a Glance

Specified in header file:	MAPISPI.H
Object that supplies this interface:	Transport logon object
Corresponding pointer type:	LXPLOGON
Implemented by:	Transport providers
Called by:	The MAPI spooler

Vtable Order

AddressTypes	Indicates to the MAPI spooler what types of recipients a transport provider can handle.
RegisterOptions	Informs the messaging system about the options provided by a transport provider for a messaging address type.
TransportNotify	Signals in a transport provider session the occurrence of an event for the MAPI spooler about which the provider requested notification.
Idle	Calls a transport provider at a point when the system is idle to perform low-priority operations.
TransportLogoff	Terminates a transport provider session with the MAPI spooler.
SubmitMessage	Indicates to a transport provider the MAPI spooler has a message for the provider to deliver.
EndMessage	Informs a transport provider the MAPI spooler has completed its sending pass for the provider.
Poll	Checks whether a transport provider has one or more incoming messages available.
StartMessage	Indicates the transfer of an incoming message from the transport provider to the MAPI spooler.
OpenStatusEntry	Opens a status object.
ValidateState	Has a transport provider check its external status.
FlushQueues	Requests that transport operations occur quickly.

IXPLogon::AddressTypes

The **IXPLogon::AddressTypes** method indicates to the MAPI spooler what types of recipients a transport provider can handle.

HRESULT AddressTypes(
  ULONG FAR * *lpulFlags***,**
  ULONG FAR * *lpcAdrType***,**
  LPTSTR FAR * FAR * *lpppszAdrTypeArray***,**
  ULONG FAR * *lpcMAPIUID***,**
  LPUID FAR * FAR * *lpppUIDArray*
)

Parameters

lpulFlags
  Output parameter containing a bitmask of flags that controls the type of the returned strings. The following flag can be set:

  MAPI_UNICODE
    Indicates the returned strings are in Unicode format. If the MAPI_UNICODE flag is not set, the strings are in ANSI format.

lpcAdrType
  Output parameter pointing to a variable containing the number of entries in the array pointed to by the *lpppAdrTypeArray* parameter.

lpppszAdrTypeArray
  Output parameter pointing to a variable where the transport provider places an array of pointers to strings that identify recipient types.

lpcMAPIUID
  Output parameter pointing to a variable containing the number of entries in the array pointed to by the *lpppUIDArray* parameter.

lpppUIDArray
  Output parameter pointing to a variable where the transport provider places an array of pointers to **MAPIUID** structures that identify recipient types.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

Remarks

The MAPI spooler calls the **IXPLogon::AddressTypes** method immediately after a transport provider returns from a call to the **IXPProvider::TransportLogon** method so the transport provider can indicate what types of recipients it can handle. To indicate this, the transport provider should pass in the *lpppszAdrTypeArray* parameter a pointer to an array of pointers to strings, or pass in the *lpppUIDArray* parameter a pointer to an array of pointers to **MAPIUID** structures, or pass values in both parameters.

These two arrays are used for different identification processes. MAPI and the MAPI spooler use the **MAPIUID** structures in the *lpppUIDArray* array to identify those recipient entry identifiers that are directly handled by the transport provider or by the messaging system to which the transport provider connects. Neither MAPI nor the MAPI spooler performs expansion of addresses with entry identifiers containing any of these **MAPIUID** structures; these structures are only used for recipient type identification.

The MAPI spooler uses each of the strings in the *lpppszAdrTypeArray* parameter for a comparison test when deciding which transport provider should handle which recipients for an outbound message. If a message recipient's PR_ADDRTYPE property exactly matches a string identifying one of the messaging address types supplied by the transport provider, the provider can handle that recipient.

In the event multiple transport providers can handle the same type of recipient, MAPI selects a transport provider based on the transport priority order indicated in the client application's profile. To determine which transport provider to use, the MAPI spooler scans all provider-specified **MAPIUID** structures in priority order, then all provider-specified address type values in priority order. The first transport provider to match a particular recipient in this scan gets the first opportunity to handle this recipient. If that provider does not handle the recipient, the MAPI spooler continues the scan so as to find a transport provider for any recipient not yet handled. The scan continues until no further matches are found, at which point a nondelivery report is generated for any recipient that was not handled.

If the provider always supports a particular set of recipient types, the address type and **MAPIUID** arrays passed by the transport provider can be static. If the transport provider dynamically constructs these arrays, it can use the support object that was passed in the call to **TransportLogon** directly previous to allocate memory, although this is not strictly necessary.

The memory used for the address type and **MAPIUID** arrays should remain allocated until the final call to the **IXPLogon::TransportLogoff** method is performed, at which time the transport provider can free the memory if necessary. The contents of these arrays should not be altered by the transport provider after returning from the **TransportLogoff** call.

A transport provider that can handle any type of recipient can return NULL in *lpppszAdrTypeArray*. LAN-based messaging systems that support a variety of gateways commonly do this. Such a transport provider should be installed last in the MAPI and MAPI spooler priority order of transport providers within the profile.

A transport provider that does not support outbound messages dispatched to it based on address type should return a single zero-length string in *lpppAdrTypeArray*. If a transport provider supports no recipient types, it should pass NULL for the **MAPIUID** and an empty string for the address type.

For more information on working with address types, see Displaying and Editing Addresses with Simple MAPI.

See Also

IXPLogon::TransportLogoff method, **IXPProvider::TransportLogon** method, **MAPIUID** structure

IXPLogon::EndMessage

The **IXPLogon::EndMessage** method informs a transport provider the MAPI spooler has completed its sending pass for the provider.

HRESULT EndMessage(
  ULONG *ulMsgRef***,**
  ULONG FAR * *lpulFlags*
)

Parameters

ulMsgRef
  Input parameter containing a 32-bit reference value, specific to a message, obtained in an earlier call to the **IXPLogon::SubmitMessage** method.

lpulFlags
  Output parameter containing a bitmask of flags that indicates to the MAPI spooler what it should do with the message. If no flags are set, the message has been sent. The following flags can be set:

  END_DONT_RESEND
    Indicates the transport provider has all the information it needs about this message for now. When the transport provider requires more information or when it has completed sending the message, it notifies the MAPI spooler by calling the **IMAPISupport::SpoolerNotify** method with the NOTIFY_SENTDEFERRED flag and by passing the message's entry identifier.

  END_RESEND_LATER
    Indicates that the transport provider isn't sending the message now for reasons that are not error conditions and that it should be called again later to send the message.

  END_RESEND_NOW
    Indicates the transport provider needs to restart the message passed to it in an **IMessage::SubmitMessage** method call.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

Remarks

The MAPI spooler calls the **IXPLogon::EndMessage** method after completing the processing involved in providing extended delivery or nondelivery information.

Once this call returns, the value in the *ulMsgRef* parameter is no longer valid for this message. The transport provider can reuse the same value on a future message.

All objects opened by the transport provider during the transfer of a message should be released before returning from the **EndMessage** call, with the exception of the message object the MAPI spooler passes to the transport provider. The message object passed by the MAPI spooler is invalid after the **EndMessage** call.

See Also

IMAPISupport::SpoolerNotify method, **IMessage::SubmitMessage** method, **IXPLogon::SubmitMessage** method

IXPLogon::FlushQueues

The **IXPLogon::FlushQueues** method requests that transport operations occur quickly.

HRESULT FlushQueues(
  ULONG *ulUIParam***,**
  ULONG *cbTargetTransport***,**
  LPENTRYID *lpTargetTransport***,**
  ULONG *ulFlags*
)

Parameters

ulUIParam
  Input parameter containing the handle of the parent window for any dialog boxes or windows this method displays.

cbTargetTransport
  Reserved; must be zero.

lpTargetTransport
  Reserved; must be NULL.

ulFlags
  Input parameter containing a bitmask of flags that controls how message queue flushing is done. The following flags can be set:

  FLUSH_DOWNLOAD
    Indicates the inbound message queue or queues should be flushed.

  FLUSH_FORCE
    Indicates the transport provider should process this request if possible, even if doing so is time-consuming.

  FLUSH_NO_UI
    Indicates the transport provider should not display a user interface.

  FLUSH_UPLOAD
    Indicates the outbound message queue or queues should be flushed.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

Remarks

The MAPI spooler calls the **IXPLogon::FlushQueues** method to advise the transport provider that the MAPI spooler is about to begin processing messages. The transport provider should call the **IMAPISupport::ModifyStatusRow** method to set an appropriate bit for its state in the PR_STATUS_CODE property of its status row. After updating its status row, the transport provider should return S_OK for the **FlushQueues** call. The MAPI spooler then starts sending messages, with the operation being synchronous to the MAPI spooler.

To support its implementation of the **IMAPIStatus::FlushQueues** method, the MAPI spooler calls **IXPLogon::FlushQueues** for all logon objects for active transport providers running in a profile session. When a transport provider's **FlushQueues** method is called as a result of a client application calling **IMAPIStatus::FlushQueues**, the message processing occurs asynchrousously to the client.

See Also

IMAPIStatus::FlushQueues method

IXPLogon::Idle

The **IXPLogon::Idle** method calls a transport provider at a point when the system is idle to perform low-priority operations.

```
HRESULT Idle(
    ULONG ulFlags
  )
```

Parameters

ulFlags
Reserved; must be zero.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

Remarks

The MAPI spooler periodically calls the **IXPLogon::Idle** method during times when the system is idle, if requested by passing the XP_LOGON_SP flag in the call to the **IXPProvider::TransportLogon** method that opened the current session. At times when the system is idle, the transport provider can perform background operations that are not appropriate during other calls, or that need to occur on a regular basis.

See Also

IXPProvider::TransportLogon method

IXPLogon::OpenStatusEntry

The **IXPLogon::OpenStatusEntry** method opens a status object.

HRESULT OpenStatusEntry(
LPCIID *lpInterface*,
ULONG *ulFlags*,
ULONG FAR * *lpulObjType*,
LPMAPISTATUS FAR * *lppEntry*
)

Parameters

lpInterface
Input parameter pointing to an interface identifier (IID) for the transport logon object. Passing NULL indicates the **IMAPIStatus** interface is returned. The *lpInterface* parameter can also be set to an identifier for an interface for the object.

ulFlags
Input parameter containing a bitmask of flags that controls how the status object is opened. The following flag can be set:

MAPI_MODIFY
Requests read/write access. The default interface is read-only.

lpulObjType
Output parameter pointing to a variable where the type of the opened object is stored.

lppEntry
Output parameter pointing to a variable where the pointer to the opened status object is stored.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

Remarks

The MAPI spooler calls the **IXPLogon::OpenStatusEntry** method when a client application calls an **OpenEntry** method for the entry identifier in the transport provider's subsystem status table row. **OpenStatusEntry** opens an object with the **IMAPIStatus** interface associated with this particular transport provider logon. This object is then used to enable client applications to call **IMAPIStatus** methods, for example to reconfigure the logon session – using the **IMAPIStatus::SettingsDialog** method – or to validate the state of the logon session – using the **IMAPIStatus::ValidateState** method.

See Also

IMAPIStatus : IMAPIProp interface

IXPLogon::Poll

The **IXPLogon::Poll** method checks whether a transport provider has one or more incoming messages available.

```
HRESULT Poll(
  ULONG FAR * lpulIncoming
 )
```

Parameters

lpulIncoming
  Output parameter containing a value indicating the existence of incoming messages. A nonzero value indicates that there are incoming messages.

Returned Values

S_OK
  The call succeeded and has returned the expected value or values.

Remarks

The MAPI spooler periodically calls the **IXPLogon::Poll** method if the transport provider indicates it must be polled for new messages; it does so by passing the LOGON_SP_POLL flag to the call to the **IXPProvider::TransportLogon** method at the beginning of a session. If the transport provider indicates in response to the **Poll** call that there are one or more inbound messages available for it to process, the MAPI spooler calls the **IXPLogon::StartMessage** method to allow the provider to process the first incoming message. The transport provider indicates incoming messages by setting the value in **Poll**'s *lpulIncoming* parameter to nonzero.

See Also

IXPLogon::StartMessage method, **IXPProvider::TransportLogon** method

IXPLogon::RegisterOptions

The **IXPLogon::RegisterOptions** method informs the messaging system about the options provided by a transport provider for a messaging address type.

HRESULT RegisterOptions(
    ULONG FAR * *lpulFlags***,**
    ULONG FAR * *lpcOptions***,**
    LPOPTIONDATA FAR * *lppOptions*
)

Parameters

lpulFlags
  Output parameter containing a bitmask of flags that controls the type of the returned strings. The following flag can be set:

  MAPI_UNICODE
    Indicates the returned strings are in Unicode format. If the MAPI_UNICODE flag is not set, the strings are in ANSI format.

lpcOptions
  Output parameter pointing to a variable containing the number of options contained in the structure returned in the *lppOptions* parameter.

lppOptions
  Output parameter pointing to a variable where a pointer to the returned **OPTIONDATA** structure is stored. The **OPTIONDATA** structure contains information for a particular messaging address type.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

If anything other than S_OK is returned, the provider is logged off.

Remarks

The MAPI spooler calls the **IXPLogon::RegisterOptions** method to get the options for messages and recipients supported by a transport provider for a particular messaging address type. These options are then registered with MAPI so they can be displayed in options dialog boxes.

RegisterOptions returns in the *lppOptions* parameter pointers to one or two **OPTIONDATA** structures for each supported messaging address type, depending on whether the provider is registered for both recipient and message options, recipient options only, or message options only. If a provider is registered for both option types, **RegisterOptions** writes one structure containing option information for recipients and one containing option information for messages. For each structure, the **ulFlags** member indicates whether the options apply to a recipient or a message.

For an example of the use of **OPTIONDATA**, consider a transport provider that handles recipients for both Microsoft Mail Server and Microsoft Mail Server for the Macintosh. If the provider is registered for both recipient and message options, it provides two pairs of **OPTIONDATA** structures, one pair for each platform. The MAPI spooler can use these structures to determine what options are valid for each platform. Once it has this option information, the MAPI spooler prompts the user with a dialog box to retrieve the setting the user wants for each option.

MAPI also uses the options registered on the **RegisterOptions** call to resolve message and recipient options. MAPI does so by using a callback function that the transport provider supplies; this callback function, declared with the **OPTIONCALLBACK** function prototype defined in MAPIDEFS.H, receives a wrapped **IMAPIProp** interface that manages the provider's message and recipient properties.

The provider is responsible for memory management. If memory is allocated for one or more **OPTIONDATA** structures during this call, the provider should free the memory upon logoff.

See Also

IXPLogon::RegisterOptions method, **OPTIONCALLBACK** function prototype, **OPTIONDATA** structure

IXPLogon::StartMessage

The **IXPLogon::StartMessage** method initiates the transfer of an incoming message from the transport provider to the MAPI spooler.

HRESULT StartMessage(
  ULONG *ulFlags*,
  LPMESSAGE *lpMessage*,
  ULONG FAR * *lpulMsgRef*
)

Parameters

ulFlags
  Reserved; must be zero.

lpMessage
  Input parameter pointing to a message object with read/write access, representing the incoming message, which is used by the transport provider to access and manipulate that message. This object remains valid until after the transport provider returns from the call to the **IXPLogon::StartMessage** method.

lpulMsgRef
  Output parameter pointing to a variable in which the transport provider returns the 32-bit reference value it assigned to this message. This value is initialized to 1 by the MAPI spooler before returning it to the transport provider.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

Remarks

The MAPI spooler calls the **IXPLogon::StartMessage** method to initiate the transfer of an incoming message from the transport provider to the MAPI spooler. Before the transport provider starts to use the message object, it should store a message reference in the *lpulMsgRef* parameter for potential use by a call to the **IXPLogon::TransportNotify** method.

During a **StartMessage** call, the MAPI spooler processes methods for objects opened during the transfer of the message and also processes any attachments. This processing can take a long time. Transport providers on 16-bit Windows platforms should call the **IMAPISupport::SpoolerYield** callback function for the MAPI spooler frequently during this processing to release CPU time for other system tasks.

All recipients in the recipient table created by the transport provider for the message must contain all required addressing properties. If necessary, the provider can construct a custom recipient to represent a particular recipient. However, if the provider can produce a recipient entry that includes more information, it should do so. For example, in the case where a transport provider has enough information about an address book provider's recipient format that it can build a valid entry identifier for a recipient for that format, it should build the entry identifier.

If any nontransmittable properties are received, the transport provider should not store them in the new message. However, the provider should store in the message all transmittable properties required for the message.

If the incoming message is to be a delivery report or a nondelivery report and the transport provider is unable to use the **IMAPISupport::StatusRecips** method to generate the report from the original message, the provider should itself populate the passed message with the appropriate properties.

To save the incoming message in the appropriate MAPI message store after processing, the transport provider calls the **IMAPIProp::SaveChanges** method. If the transport provider doesn't have any messages to pass to the MAPI spooler, it can stop the incoming message by returning from the **StartMessage** call without calling **SaveChanges**.

All objects opened by the transport provider during a **StartMessage** call should be released before returning. However, the provider should not release the message object originally passed by the MAPI spooler in the *lpMessage* parameter.

If an error is returned from **StartMessage**, the message in process is released without having changes saved and is lost. The transport provider in such a case should pass the flag NOTIFY_CRITICAL_ERROR with a call to the **IMAPISupport::SpoolerNotify** method and call the **IXPLogon::Poll** method to notify the MAPI spooler that it is in a severe error condition.

For more information, see Interacting with the MAPI Spooler.

See Also

IMAPIProp::SaveChanges method, **IMAPISupport::SpoolerNotify** method, **IMAPISupport::SpoolerYield** method, **IMAPISupport::StatusRecips** method, **IMessage::GetRecipientTable** method, **IXPLogon::Poll** method, **IXPLogon::TransportNotify** method

IXPLogon::SubmitMessage

The **IXPLogon::SubmitMessage** method indicates to a transport provider the MAPI spooler has a message for the provider to deliver.

HRESULT SubmitMessage(
  ULONG *ulFlags***,**
  LPMESSAGE *lpMessage***,**
  ULONG FAR * *lpulMsgRef***,**
  ULONG FAR * *lpulReturnParm*
)

Parameters

ulFlags
  Input parameter containing a bitmask of flags that controls how the message is submitted. The following flag can be set:

  BEGIN_DEFERRED
    Indicates the MAPI spooler is calling a transport provider with a message whose entry identifier was passed from the transport provider using the **IMAPISupport::SpoolerNotify** method's NOTIFY_SENTDEFERRED flag.

lpMessage
  Input parameter pointing to a message object with read/write access, representing the message to deliver, which is used by the transport provider to access and manipulate that message. This object remains valid until after the transport provider returns from a subsequent call to the **IXPLogon::EndMessage** method.

lpulMsgRef
  Output parameter pointing to a variable in which the transport provider returns the 32-bit reference value it assigned to this message. The MAPI spooler passes this reference value in subsequent calls for this message. The value is initialized to zero by the MAPI spooler before returning it to the transport provider.

lpulReturnParm
  Output parameter pointing to a variable that corresponds to the MAPI_E_WAIT or MAPI_E_NETWORK_ERROR error value returned by the **IXPLogon::SubmitMessage** method.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

MAPI_E_BUSY
  The transport provider can't handle the message because it is performing another operation. A provider should use this return value to indicate that no processing occurred and that the MAPI spooler should not call **EndMessage**. The MAPI spooler tries the **SubmitMessage** call again later.

MAPI_E_CANCEL
  Indicates that although on a previous **SpoolerNotify** call the transport provider requested the MAPI spooler resubmit the message, conditions have since changed and the message should not be resent. The MAPI spooler goes on to handle something else.

MAPI_E_NETWORK_ERROR
  A network error prevented successful completion of the operation. The *lpulReturnParm* parameter should be set to the number of seconds before the MAPI spooler resubmits the message.

MAPI_E_NOT_ME
  The transport provider cannot handle this message. The MAPI spooler should try to find another transport provider for it. A provider should use this return value to indicate that no processing occurred and that the MAPI spooler should not call **EndMessage**.

MAPI_E_WAIT
A temporary problem prevents the transport provider from handling the message. The *lpulReturnParm* parameter should be set to the number of seconds before the MAPI spooler resubmits the message.

Remarks

The MAPI spooler calls the **IXPLogon::SubmitMessage** method when it has a message for the transport provider to carry. The message is passed to the transport provider using the *lpMessage* parameter.

If the provider is ready to accept the message, it should return a reference value by using the *lpulMsgRef* parameter, process the passed object, and return the appropriate value, usually S_OK. If the provider is not prepared to handle the transfer, it should return an error value and, optionally, another MAPI return value in *lpulReturnParm* to indicate how long the MAPI spooler should wait before resubmitting the message.

A transport provider's implementation of this method can:

- Put the message into an internal queue to wait for transmission, possibly copying it to local storage, and return.
- Attempt to perform the actual transmission and return when the transmission has completed, either successfully or unsuccessfully.
- Determine whether or not to send the message after checking the resource involved. In this case, if the resource is free, the provider can lock the resource, prepare the message, and submit it. If the resource is busy, the provider can prepare the message and defer sending to a later time.

The preferred technique depends on the transport provider and the expected number of processes competing for system resources.

During a **SubmitMessage** call, the transport provider controls the transfer of message data from the message object. However, the transport provider should assign a 32-bit reference value to the message, to which it returns a pointer in *lpulMsgRef*, before transferring data. It does so because at any point during the process the MAPI spooler can call the **IXPLogon::TransportNotify** method with the NOTIFY_CANCEL_MESSAGE flag set to signal the provider it should release any open objects and stop message transfer.

The transport provider should not send any nontransmittable properties of the message. When it finds such a property, it should go on to process the next one. The provider should make every effort not to display MAPI_P1 recipient information as part of the transmitted message content, using such recipient information only for addressing purposes. MAPI_P1 recipients are internally-generated recipients for use in resends, they should not be transmitted. Instead, use the other recipients for transmitting recipient information. The purpose of this arrangement is to permit resend recipients to see the exact same recipient table as the original recipients.

During a **SubmitMessage** call, the MAPI spooler processes methods for objects opened during the transfer of the message and also processes any attachments. This processing can take a long time. Transport providers running on 16-bit Windows platforms should call the **IMAPISupport::SpoolerYield** callback function for the MAPI spooler frequently during this processing to release CPU time for other system tasks.

All message recipients are visible in the recipient table of the message originally passed by the MAPI spooler. The transport provider should process only those recipients that it can handle based on entry identifier, address type, or both, and that do not already have their PR_RESPONSIBILITY property set to TRUE. If PR_RESPONSIBILITY is already set to TRUE, another transport provider handles that recipient. When the provider has completed sufficient processing of a recipient to determine whether it can handle messages for that recipient, it should set that recipient's PR_RESPONSIBILITY property to TRUE in the passed message. Usually, the provider makes this determination after message delivery is complete.

Typically, the transport provider does not return from a **SubmitMessage** call until it has completed the transfer of message data. If no error is returned, the next call from the MAPI spooler to the provider is a call to the **IXPLogon::EndMessage** method.

If an error is returned from **SubmitMessage**, the MAPI spooler releases the message in process without saving changes. If the transport provider requires message changes be saved, it must call the **IMAPIProp::SaveChanges** method on the message before returning.

In case of errors occurring because of transport problems, the MAPI spooler retains the message but delays resubmitting it to the transport provider based on the value returned in *lpulRetunParm*. The transport provider must fill in that value if its return from **SubmitMessage** is MAPI_E_WAIT or MAPI_E_NETWORK_ERROR. If a severe error condition is occurring, the transport provider must call the **IMAPISupport::SpoolerNotify** method with the NOTIFY_CRITICAL_ERROR flag.

See Also

IMAPIProp::SaveChanges method, **IMAPISupport::SpoolerNotify** method, **IMAPISupport::SpoolerYield** method, **IXPLogon::EndMessage** method, **IXPLogon::TransportNotify** method

IXPLogon::TransportLogoff

The **IXPLogon::TransportLogoff** method terminates a transport provider session with the MAPI spooler.

```
HRESULT TransportLogoff(
   ULONG ulFlags
  )
```

Parameters

ulFlags
Reserved; must be zero.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

Whether or not the transport provider returns a value other than S_OK, the provider's logged off.

Remarks

The MAPI spooler calls the **IXPLogon::TransportLogoff** method to terminate a transport provider session for a particular user. Before calling **TransportLogoff**, the MAPI spooler discards any data about supported messaging address types for this session passed in the **IXPLogon::AddressTypes** method.

The transport provider should be prepared to accept a call to **TransportLogoff** at any time. If a message is in process, the provider should stop the sending process.

The transport provider should release all resources allocated for its current session. If it has allocated any memory for this session with the **MAPIAllocateBuffer** function, it should free the memory by using the **MAPIFreeBuffer** function. Any memory allocated by the transport provider to satisfy calls to the **IXPLogon::AddressTypes**, **IXPLogon::RegisterOptions**, and **IMAPISession::MessageOptions** methods can be safely released at this time.

Usually, a provider should, on completing a **TransportLogoff** call, first invalidate its logon object by calling the **IMAPISupport::MakeInvalid** method and then release its support object. The provider's implementation of **TransportLogoff** should release the support object last, because when the support object is released, the MAPI spooler can also release the provider object itself.

See Also

IMAPISession::MessageOptions method, **IMAPISupport::MakeInvalid** method, **IMAPISupport::SpoolerYield** method, **IXPLogon::AddressTypes** method, **IXPLogon::RegisterOptions** method, **MAPIAllocateBuffer** function, **MAPIFreeBuffer** function

IXPLogon::TransportNotify

The **IXPLogon::TransportNotify** method signals in a transport provider session the occurrence of an event for the MAPI spooler about which the provider requested notification.

HRESULT TransportNotify(
  ULONG FAR * *lpulFlags***,**
  LPVOID FAR * *lppvData*
)

Parameters

lpulFlags
  Input-output parameter containing a bitmask of flags that signals notification events. The following flags can be set by the MAPI spooler on input and must be returned unchanged on output:

  NOTIFY_ABORT_DEFERRED
    Notifies the transport provider that a message for which it accepted responsibility is being canceled. Only transport providers that support deferral must support this flag. The *lppvData* parameter points to the entry identifier of the canceled message. Messages that have not been processed by the MAPI spooler can still be canceled by calling the **IMsgStore::AbortSubmit** method.

  NOTIFY_BEGIN_INBOUND
    Indicates inbound messages can now be accepted for this transport provider session. The MAPI spooler regularly calls the **IXPLogon::Poll** method if the transport provider set the flag LOGON_SP_POLL with the **IXPProvider::TransportLogon** call at logon. Once the NOTIFY_BEGIN_INBOUND flag is set, the MAPI spooler honors the NOTIFY_NEWMAIL flag passed in the call to the **IMAPISupport::SpoolerNotify** method. The status table row for the transport provider session should be updated before returning by calling the **IMAPISupport::ModifyStatusRow** method. NOTIFY_BEGIN_INBOUND is mutually exclusive with the NOTIFY_END_INBOUND flag.

  NOTIFY_BEGIN_INBOUND_FLUSH
    Signals the transport provider to cycle through inbound messages as quickly as possible. To comply with this notification, the transport provider should set the flag STATUS_INBOUND_FLUSH in the PR_STATUS_CODE property of its status table row as soon as it can, using **ModifyStatusRow**. Until the end of this inbound messaging cycle, which is when the provider determines it has downloaded all it can or when it has received an **IXPLogon::TransportNotify** method call with the flag NOTIFY_END_INBOUND_FLUSH set, the provider should not call the **IMAPISupport::SpoolerYield** method or otherwise give up cycles to the operating system that can be used to speed delivery of incoming messages. At the end of the inbound flush operation, the provider should use **SpoolerNotify** to clear the STATUS_INBOUND_FLUSH flag in the PR_STATUS_CODE property of its status row.

  NOTIFY_BEGIN_OUTBOUND
    Indicates outbound operations can now occur for this transport provider session. If there are any messages to be sent for this provider, a call to the **IXPLogon::SubmitMessage** method follows. The status table row for this session should be updated before returning. The NOTIFY_BEGIN_OUTBOUND flag is mutually exclusive with the NOTIFY_END_OUTBOUND flag.

  NOTIFY_BEGIN_OUTBOUND_FLUSH
    Works similarly to the NOTIFY_BEGIN_INBOUND_FLUSH flag but refers to outbound messages, and the appropriate status flag is STATUS_OUTBOUND_FLUSH.

  NOTIFY_CANCEL_MESSAGE
    Indicates the MAPI spooler must cancel transfer of the message for which the *lppvData* parameter points to the 32-bit value obtained with the **IXPLogon::SubmitMessage** method call. The NOTIFY_CANCEL_MESSAGE flag can be set without the transport provider having returned

from the **SubmitMessage**, **IXPLogon::StartMessage**, or **IXPLogon::EndMessage** method call associated with the message. The transport provider must return from the entry point that is handling the canceled message as soon as possible.

For an in-process incoming message, the transport provider should retain the incoming message wherever it is presently stored and pass it at the next convenient time. The message object data already stored for the incoming message is discarded.

If the transport provider did not update this 32-bit value at **StartMessage** or **SubmitMessage** time, it is 0 for outbound messages and 1 for inbound messages.

NOTIFY_END_INBOUND
Indicates inbound operations must cease for this transport provider session. The MAPI spooler ceases to use **Poll** and ignores NOTIFY_NEWMAIL for this session. In-process messages should not be aborted. The status table row for the transport session should be updated by calling **ModifyStatusRow** before returning. NOTIFY_END_INBOUND is mutually exclusive with NOTIFY_BEGIN_INBOUND.

NOTIFY_END_INBOUND_FLUSH
Notifies the transport provider to come out of flush mode. The transport provider should not stop downloading, but downloading should be done in the background. The status table row for the transport session should be updated by calling **ModifyStatusRow** when the transport provider can comply with this notification.

NOTIFY_END_OUTBOUND
Indicates outbound operations must cease for this transport provider session. The MAPI spooler ceases to call **SubmitMessage** and ignores the **SpoolerNotify** flag NOTIFY_READYTOSEND. If there is an in-process message, it should not be stopped; to stop a message, use the NOTIFY_CANCEL_MESSAGE flag. The status table row for this session should be updated by calling **ModifyStatusRow** before returning. NOTIFY_END_INBOUND is mutually exclusive with NOTIFY_BEGIN_OUTBOUND.

NOTIFY_END_OUTBOUND_FLUSH
Works similarly to NOTIFY_END_INBOUND_FLUSH but refers to outbound messages, and the appropriate status flag is STATUS_OUTBOUND_FLUSH.

lppvData
Output parameter pointing to a variable where a pointer to event-specific data is stored. For more information on what *lppvData* holds, see the description for the *lpulFlags* parameter.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

Remarks

The MAPI spooler calls the **IXPLogon::TransportNotify** method to signal to the transport provider the occurrence of events about which notification has been requested. These events include the MAPI spooler's requiring the cancellation of transfer for a message, the beginning or ending of inbound or outbound transport operations, and the beginning or ending of an operation to clear an incoming or outgoing message queue.

When the user tries to cancel a message that the transport provider has previously deferred, the MAPI spooler calls **TransportNotify** passing in both the NOTIFY_ABORT_DEFERRED and NOTIFY_CANCEL_MESSAGE flags in *ulFlags*. If the MAPI spooler is logging off and still has messages in the queue, it passes only NOTIFY_ABORT_DEFERRED in *ulFlags* when it calls **TransportNotify**.

The provider must synchronize access to its data on this call, because the MAPI spooler can invoke this method from another thread of execution or from a procedure for a different window.

See Also

IMAPISupport::SpoolerNotify method, **IMAPISupport::SpoolerYield** method, **IMsgStore::AbortSubmit** method, **IXPLogon::EndMessage** method, **IXPLogon::Poll** method, **IXPLogon::StartMessage** method, **IXPLogon::SubmitMessage** method, **IXPLogon::TransportNotify** method, **IXPProvider::TransportLogon** method

IXPLogon::ValidateState

The **IXPLogon::ValidateState** method has a transport provider check its external status.

HRESULT ValidateState(
  ULONG *ulUIParam*,
  ULONG *ulFlags*
)

Parameters

ulUIParam
  Input parameter containing the handle of the parent window for any dialog boxes or windows this method displays.

ulFlags
  Input parameter containing a bitmask of flags that controls how the status check is done and its results. The following flags can be set:

  ABORT_XP_HEADER_OPERATION
    Indicates the user canceled the operation, typically by clicking the **Cancel** button in a dialog box. The transport provider has the option to continue working on the operation, or it can abort the operation and return MAPI_E_USER_CANCELED.

  CONFIG_CHANGED
    Validates the state of currently loaded transport providers by causing the MAPI spooler to call their **IXPLogon::AddressTypes** and **IMAPISession::MessageOptions** methods. This flag also provides the MAPI spooler an opportunity to correct critical transport-provider failures without forcing client applications to log off and then log on again.

  FORCE_XP_CONNECT
    Indicates the user selected a connect operation. When this flag is used with the REFRESH_XP_HEADER_CACHE or PROCESS_XP_HEADER_CACHE flag, the connect action occurs without caching.

  FORCE_XP_DISCONNECT
    Indicates the user selected a disconnect operation. When this flag is used with REFRESH_XP_HEADER_CACHE or PROCESS_XP_HEADER_CACHE, the disconnect action occurs without caching.

  PROCESS_XP_HEADER_CACHE
    Indicates that entries in the header cache table should be processed, that all messages marked with the MSGSTATUS_REMOTE_DOWNLOAD flag should be downloaded, and that all messages marked with the MSGSTATUS_REMOTE_DELETE flag should be deleted. Messages that have both MSGSTATUS_REMOTE_DOWNLOAD and MSGSTATUS_REMOTE_DELETE set should be moved.

  REFRESH_XP_HEADER_CACHE
    Indicates that a new list of message headers should be downloaded, and that all message status marking flags should be cleared.

  SUPPRESS_UI
    Prevents the transport provider from displaying a user interface.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

MAPI_E_BUSY
  Another operation is in progress; it should be allowed to complete, or it should be stopped, before this operation is attempted.

MAPI_E_NO_SUPPORT

The remote transport provider involved does not support a user interface, and the client application should display the dialog box itself.

MAPI_E_USER_CANCEL

The user canceled the operation, typically by choosing the Cancel button in a dialog box.

Remarks

The MAPI spooler calls the **IXPLogon::ValidateState** method to support calls to the **IMAPIStatus::ValidateState** method for the status object. The transport provider should respond to the **IXPLogon::ValidateState** call exactly as if the MAPI spooler had opened a status object for the current logon session and then called **ValidateState** on that object.

To support its implementation of **IMAPIStatus::ValidateState**, the MAPI spooler calls **IXPLogon::ValidateState** on all logon objects for all active transport providers running in a profile session.

See Also

IMAPISession::MessageOptions method, **IMAPIStatus::ValidateState** method, **IXPLogon::AddressTypes** method

IXPProvider : IUnknown

The **IXPProvider** interface is used to initialize a transport provider object and to shut down the object when it is no longer needed.

At a Glance

Specified in header file:	MAPISPI.H
Object that supplies this interface:	Transport provider object
Corresponding pointer type:	LPXPROVIDER
Implemented by:	Transport providers
Called by:	The MAPI spooler

Vtable Order

Shutdown	Closes down a transport provider in an orderly fashion.
TransportLogon	Establishes a session in which a user logs onto a transport provider.

IXPProvider::Shutdown

The **IXPProvider::Shutdown** method closes down a transport provider in an orderly fashion.

```
HRESULT Shutdown (
  ULONG FAR * lpulFlags
 )
```

Parameters

lpulFlags
Reserved; must be zero.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

Remarks

The MAPI spooler calls the **IXPProvider::Shutdown** method just prior to releasing a transport provider object. Before calling **Shutdown**, MAPI releases all logon objects for a provider.

See Also

XPProviderInit function

IXPProvider::TransportLogon

The **IXPProvider::TransportLogon** method establishes a session in which a user logs on to a transport provider.

HRESULT TransportLogon(
  LPMAPISUP *lpMAPISup*,
  ULONG *ulUIParam*,
  LPTSTR *lpszProfileName*,
  ULONG FAR * *lpulFlags*,
  LPMAPIERROR FAR * *lppMAPIError*,
  LPXPLOGON FAR * *lppXPLogon*
)

Parameters

lpMAPISup
: Input parameter pointing to the transport provider's support object for callback functions within MAPI for this session. This object remains valid until the transport provider releases it.

ulUIParam
: Input parameter containing the handle of the parent window for any dialog boxes or windows this method displays. The *ulUIParam* parameter can be non-null, for example when the LOGON_SETUP flag is set in the *lpulFlags* parameter.

lpszProfileName
: Input parameter pointing to a string containing the profile name of the user. The *lpszProfileName* parameter is used primarily when a dialog box must be presented.

lpulFlags
: Input-output parameter containing a bitmask of flags that controls how the logon session is established. The following flags can be set on input by the MAPI spooler:

LOGON_NO_CONNECT
: Indicates the user account is logging on to this transport provider for purposes other than transmission and reception of messages. The transport provider should not attempt to make any connections.

LOGON_NO_DIALOG
: Indicates no dialog box should be displayed even if the currently saved user credentials are invalid or insufficient for logon.

LOGON_NO_INBOUND
: Indicates the transport provider does not need to initialize for reception of messages and should not accept incoming messages. The MAPI spooler can use the **IXPLogon::TransportNotify** method to signal the transport provider to enable inbound message processing.

LOGON_NO_OUTBOUND
: Indicates the transport provider does not need to initialize for sending messages, as the MAPI spooler does not provide any. If a client application requires a connection to a remote provider during the composition of a message so that it can make **IXPLogon::AddressTypes** or **IXPLogon::RegisterOptions** method calls, the transport provider should make the connection. The MAPI spooler can use **TransportNotify** to signal the transport provider when outbound operations can begin.

MAPI_UNICODE
: Indicates the passed-in string for the profile name is in Unicode format. If the MAPI_UNICODE flag is not set, the string is in ANSI format.

The following flags can be set on output by the transport provider:

LOGON_SP_IDLE

Requests that the MAPI spooler frequently call the transport provider's **IXPLogon::Idle** method for idle-time processing.

LOGON_SP_POLL

Requests that the MAPI spooler frequently call the **IXPLogon::Poll** method on the returned logon object to check for new messages. If this flag is not set, the MAPI spooler only checks for new messages when the transport provider calls the **IMAPISupport::SpoolerNotify** method. A transport provider effectively becomes send-only by not setting this flag and by not notifying the MAPI spooler of message receipt.

LOGON_SP_RESOLVE

Requests that the MAPI spooler resolve to full addresses all message addresses for recipients not supported by this transport provider, so that the transport provider can construct a reply path for all recipients.

MAPI_UNICODE

Indicates the returned strings in the **MAPIERROR** structure, if any, are in Unicode format. If the MAPI_UNICODE flag is not set, the strings are in ANSI format.

lppMAPIError

Output parameter pointing to a pointer to the returned **MAPIERROR** structure, if any, containing version, component, and context information for the error. The *lppMAPIError* parameter can be set to NULL if there is no **MAPIERROR** structure to return.

lppXPLogon

Output parameter pointing to a variable where the pointer to the returned transport-provider logon object is stored.

Return Values

S_OK

The call succeeded and has returned the expected value or values.

MAPI_E_FAILONEPROVIDER

Indicates this provider cannot log on, but this error should not disable the service.

MAPI_E_UNCONFIGURED

The profile does not contain enough information for the logon to complete. MAPI calls the provider's message service entry point function.

MAPI_E_UNKNOWN_CPID

Indicates the server is not configured to support the client's code page.

MAPI_E_UNKNOWN_LCID

Indicates the server is not configured to support the client's locale information.

MAPI_E_USER_CANCEL

The user canceled the operation, typically by clicking the **Cancel** button in a dialog box.

Remarks

The MAPI spooler calls the **IXPProvider::TransportLogon** method to establish a logon session for a user.

Most transport providers use the **IMAPISupport::OpenProfileSection** method provided with the support object pointed to by the *lpMAPISup* parameter for saving and retrieving user identity information, server addresses, and credentials. Using **OpenProfileSection** enables a transport provider to save arbitrary information and associate it with a logon to a particular resource. For example, a provider can use **OpenProfileSection** to save the account name and password associated with a particular session and any server names or other necessary information needed to access resources for that session. MAPI hides information associated with a resource from outside access. The profile section made available through *lpMAPISup* is segregated by the MAPI spooler so data related to this user context is separated from data for other contexts.

The transport provider must call the **IUnknown::AddRef** method for the support object and keep a copy of the pointer to this object as part of the provider logon object.

The profile display name in *lpszProfileName* is provided so the transport provider can use it in error messages or logon dialog boxes. If the provider retains this name, it must be copied to storage allocated by the provider.

Transport providers that are tightly coupled with other service providers might need to do additional work at logon to establish the proper credentials required for operations between companion providers.

Usually, transport providers are opened when the user first logs on to a profile. Because the first logon to a profile thus generally precedes logon to any message store, the MAPI spooler usually calls **TransportLogon** with both the LOGON_NO_INBOUND and LOGON_NO_OUTBOUND flags set in *lpulFlags*. Later, when the appropriate message stores are available in the profile session, the MAPI spooler calls **TransportNotify** to initiate inbound and outbound operations for the transport provider.

Passing the LOGON_NO_CONNECT flag in *lpulFlags* signals offline operation of the transport provider. This flag indicates no external connection is to be made; if the transport provider cannot establish a session without an external connection, it should return an error value for the logon.

A transport provider should set the LOGON_SP_IDLE flag in *lpulFlags* at initialization time to support any scheduled connections and to handle automatic operations, such as automatic message downloading, timed message downloading, or timed message submission. If this flag is set, the MAPI spooler calls **Idle** when system idle time occurs to initiate such operations. The MAPI spooler does not call **Idle** at regular intervals; rather, it is called only during true idle time, so providers should not work on any assumption about how frequently their **Idle** methods will be called. Providers that support idle-time operations should supply the correct configuration user interface in their provider property sheet.

If the transport provider logon succeeds, the provider should return in the *lppXPLogon* parameter a pointer to a logon object for the MAPI spooler to use for further provider access. If **TransportLogon** displays a logon dialog box and the user cancels logon, typically by clicking the **Cancel** button in the dialog box, the provider should return MAPI_E_USER_CANCEL.

For most error values returned from **TransportLogon**, MAPI disables the message services to which the failing provider belongs. MAPI will not call any providers belonging to that service for the rest of the life of the MAPI session. In contrast, when **TransportLogon** returns the MAPI_E_FAILONEPROVIDER error value from its logon, MAPI does not disable the message service to which the provider belongs. **TransportLogon** should return MAPI_E_FAILONEPROVIDER if it encounters an error that does not warrant disabling the entire service for the life of the session.

If a provider returns MAPI_E_UNCONFIGURED from its logon, MAPI will call the provider's message service entry function and then retry the logon. MAPI passes MSG_SERVICE_CONFIGURE as the context, to give the service a chance to configure itself. If the client has chosen to allow a user interface on the logon, the service can present its configuration property sheet so the user can enter configuration information.

If the provider finds that all the required information is not in the profile, it should return MAPI_E_UNCONFIGURED so that MAPI calls the provider's message-service entry point function.

See Also

IMAPISupport::OpenProfileSection method, **IMAPISupport::SpoolerNotify** method, **IXPLogon::AddressTypes** method, **IXPLogon::Idle** method, **IXPLogon::Poll** method, **IXPLogon::RegisterOptions** method, **IXPLogon::TransportNotify** method, **MAPIERROR** structure

MAPI Functions and Related Macros

The following alphabetic entries contain documentation for MAPI Functions. For more information on using functions that may not be supported in future versions of MAPI, see Using Functions.

ACCELERATEABSDI

The **ACCELERATEABSDI** function prototype defines a callback function that accelerates the addition of names to an address book.

At a Glance

Specified in header file:	MAPIDEFS.H
Implemented by:	MAPI
Called by:	Client applications

BOOL (STDMETHODCALLTYPE ACCELERATEABSDI)(
 ULONG *ulUIParam*,
 LPVOID *lpvmsg*
 );

Parameters

ulUIParam
 Input parameter specifying an implementation-specific 32-bit value used for passing user interface information to a function. In Microsoft Windows applications, *ulUIParam* is the parent window handle for a dialog box and is of type HWND, cast to a **ULONG**. A value of zero indicates there is no parent window.

lpvmsg
 Input parameter pointing to a Windows message.

Return Values

A function with the **ACCELERATEABSDI** prototype returns TRUE if it handles the message.

Remarks

An **ACCELERATEABSDI** prototype-based function is used only if the client has set the DIALOG_SDI flag in the *ulFlags* member of the **ADRPARM** structure.

A client application calls a function based on this prototype in its Windows message loop during execution of a modeless address book dialog box for the **IAddrBook::Address** method. This calling is terminated when MAPI calls a function based on the **DISMISSMODELESS** function prototype.

ABProviderInit

The **ABProviderInit** function initializes an address book provider for operation.

At a Glance

Specified in header file:	MAPISPI.H
Implemented by:	Address book providers
Called by:	MAPI

HRESULT ABProviderInit(
  HINSTANCE *hInstance*,
  LPMALLOC *lpMalloc*,
  LPALLOCATEBUFFER *lpAllocateBuffer*,
  LPALLOCATEMORE *lpAllocateMore*,
  LPFREEBUFFER *lpFreeBuffer*,
  ULONG *ulFlags*,
  ULONG *ulMAPIVer*,
  ULONG FAR * *lpulProviderVer*,
  LPABPROVIDER FAR * *lppABProvider*
 );

Parameters

hInstance
  Input parameter containing the instance of the address book provider's dynamic-link library (DLL) that MAPI used when it linked.

lpMalloc
  Input parameter pointing to a memory allocator object exposing the OLE **IMalloc** interface. The address book provider may need to use this allocation method when working with certain interfaces such as **IStream**.

lpAllocateBuffer
  Input parameter pointing to the **MAPIAllocateBuffer** function, to be used to allocate memory.

lpAllocateMore
  Input parameter pointing to the **MAPIAllocateMore** function, to be used to allocate additional memory where required.

lpFreeBuffer
  Input parameter pointing to the **MAPIFreeBuffer** function, to be used to free memory.

ulFlags
  Input parameter containing a bitmask of flags. The following flag can be set:

  MAPI_NT_SERVICE
    Indicates the provider is being loaded in the context of a Windows NT service, a special type of process without access to any user interface.

ulMAPIVer
  Input parameter containing the version number of the service provider interface that MAPI.DLL uses. For the current version number, see the MAPISPI.H header file.

lpulProviderVer
  Output parameter pointing to the version number of the service provider interface that this address book provider uses.

lppABProvider
  Output parameter pointing to a pointer to the initialized address book provider object.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

Remarks

MAPI calls the entry point function **ABProviderInit** to initialize a newly loaded address book provider. The provider must implement **ABProviderInit** in compliance with the **ABPROVIDERINIT** function prototype, also specified in MAPISPI.H. MAPI defines **ABPROVIDERINIT** to use the standard MAPI initialization call type, STDMAPIINITCALLTYPE, which causes **ABProviderInit** to follow the CDECL calling convention.

The address book provider should use the functions pointed to by *lpAllocateBuffer*, *lpAllocateMore*, and *lpFreeBuffer* for most memory allocation and deallocation. In particular, the provider must use these functions to allocate memory for use by client applications when calling object interfaces such as **IMAPIProp::GetProps** and **IMAPITable::QueryRows**. If the provider expects to use the OLE memory allocator later, it should call the **IUnknown::AddRef** method for the allocator object pointed to by the *lpMalloc* parameter.

For more information on entry point functions, see About Provider DLL Entry Point Functions.

See Also

IABProvider : IUnknown interface, **HPProviderInit** function, **MSProviderInit** function, **XPProviderInit** function

BuildDisplayTable

The **BuildDisplayTable** function creates a display table from the property page data contained in a DTPAGE structure.

At a Glance

Specified in header file:	MAPIUTIL.H
Implemented by:	MAPI
Called by:	Service providers

STDAPI BuildDisplayTable(
  LPALLOCATEBUFFER *lpAllocateBuffer***,**
  LPALLOCATEMORE *lpAllocateMore***,**
  LPFREEBUFFER *lpFreeBuffer***,**
  LPVOID *lpvReserved***,**
  HINSTANCE *hInstance***,**
  UINT *cPages***,**
  LPDTPAGE *lpPage***,**
  ULONG *ulFlags***,**
  LPMAPITABLE * *lppTable***,**
  LPTABLEDATA * *lppTblData*
 );

Parameters

lpAllocateBuffer
  Input parameter pointing to the **MAPIAllocateBuffer** function, to be used to allocate memory.

lpAllocateMore
  Input parameter pointing to the **MAPIAllocateMore** function, to be used to allocate additional memory where required.

lpFreeBuffer
  Input parameter pointing to the **MAPIFreeBuffer** function, to be used to free memory.

lpvReserved
  Reserved; must be zero.

hInstance
  Input parameter containing an instance of a MAPI object from which **BuildDisplayTable** retrieves resources.

cPages
  Input parameter containing the number of **DTPAGE** structures pointed to by the *lpPage* parameter.

lpPage
  Input parameter pointing to one or more **DTPAGE** structures that contain information about the display table to be built.

ulFlags
  Input parameter containing a bitmask of flags. The following flag can be set:

  MAPI_UNICODE
    Indicates the passed-in strings are in Unicode format. If the MAPI_UNICODE flag is not set, the strings are in ANSI format.

lppTable
  Input parameter pointing to the **IMAPITable** interface for the display table.

lppTblData
  Input-output parameter pointing to the **ITableData** interface for the object that contains table data for the display table. On input, the pointer is NULL to indicate that the existing table data object has

been released. On output, the pointer indicates a display table returned by **BuildDisplayTable**.

Remarks

The *lpAllocateBuffer*, *lpAllocateMore*, and *lpFreeBuffer* input parameters point to the **MAPIAllocateBuffer**, **MAPIAllocateMore**, and **MAPIFreeBuffer** functions, respectively. If a client application calls **BuildDisplayTable**, it passes in these pointers to the functions named as listed. If a service provider calls **BuildDisplayTable**, it passes the pointers to these functions it received in its initialization call or retrieved by calling the **IMAPISupport::GetMemAllocRoutines** method.

Anything that can be read from the dialog resource, will be. This includes:

- The page title (*ulbLpszLabel* member of the **DTBLPAGE** structure), read from the dialog title in the resource.
- All control titles (*ulbLpszLabel* member of other control structures), read from the control text in the resource.

BuildDisplayTable overwrites anything passed in the input control structures with information from the dialog resource which means the caller of **BuildDisplayTable** cannot dynamically specify page or control titles. Callers who need to do that can choose to have **BuildDisplayTable** return to them the table data object in *lppTableData* and change rows in it; but they might find it simpler to build the display table by hand in a table data object instead.

CALLERRELEASE

The **CALLERRELEASE** function prototype defines a callback function that releases a table data object.

At a Glance

Specified in header file:	MAPIUTIL.H
Implemented by:	Client applications and service providers
Called by:	MAPI

void CALLERRELEASE(
  ULONG *ulCallerData***,**
  LPTABLEDATA *lpTblData***,**
  LPMAPITABLE *lpVue*
 );

Parameters

ulCallerData
  Input parameter containing 32-bit data about the table data object to release.

lpTblData
  Input parameter pointing to the **ITableData : IUnknown** interface for the table data object to release.

lpVue
  Input parameter pointing to the **IMAPITable : IUnknown** interface for the table whose data is released. This table interface is an interface for the table object passed in the *lppMAPITable* parameter of the **ITableData::HrGetView** method that created the object to release.

Return Values

None

Remarks

Client applications use the **CALLERRELEASE** function prototype to create a callback function to release a view on the underlying table data object without having to keep track of that object. After all views on a table data object are released, implementations of **ITableData::HrGetView** call the **CALLERRELEASE** function and then release the table data object underlying the views.

ChangeIdleRoutine

This function may not be supported in future versions of MAPI.

The **ChangeIdleRoutine** function changes some or all of the characteristics of an idle function based on the **FNIDLE** function prototype.

At a Glance

Specified in header file:	MAPIUTIL.H
Implemented by:	MAPI
Called by:	Client applications and service providers

VOID ChangeIdleRoutine(
  FTG *ftg*,
  PFNIDLE *pfnIdle*,
  LPVOID *pvIdleParam*,
  short *priIdle*,
  ULONG *csecIdle*,
  USHORT *iroIdle*,
  USHORT *ircIdle*
 );

Parameters

ftg
  Input parameter containing a function tag that identifies the idle function.

pfnIdle
  Input parameter pointing to the idle function.

pvIdleParam
  Input parameter pointing to a new block of memory that the calling implementation allocates for the idle function.

priIdle
  Input parameter containing a value representing a new priority for the idle function. Possible priorities for implementation-defined functions are greater than or less than zero, but not zero. A value of zero is reserved for a user event such as a mouse click.

  Values greater than zero represent priorities for background tasks that have a higher priority than user events and are dispatched as part of the standard message pump loop. Values less than zero represent priorities for idle tasks that only run during message-pump idle time. Examples of priorities are: 1 for foreground submission, -1 for power-edit character insertion, and -3 for downloading new messages.

csecIdle
  Input parameter containing a new time, in hundredths of a second, to apply to the idle function. The meaning of the initial time value varies, depending on what is passed in the *iroIdle* parameter. It can be:

- The minimum period of user inaction that must elapse before the MAPI idle engine calls the idle function for the first time, if the FIROWAIT flag is set in *iroIdle*. After this time passes, the idle engine can call the idle function as often as necessary.
- The minimum interval between calls to the idle function, if the FIROINTERVAL flag is set in *iroIdle*.

iroIdle
  Input parameter containing a bitmask of flags indicating new options for the idle function. The following flags can be set:

FIRODISABLED

Indicates that the idle function is initially disabled when registered. The default action is to enable the idle function when the **FtgRegisterIdleRoutine** function registers it.

FIROINTERVAL

Indicates that the time specified by the *csecIdle* parameter is the minimum interval between successive calls to the idle function.

FIROWAIT

Indicates that the time specified by the *csecIdle* parameter is the minimum period of user inaction that must elapse before the MAPI idle engine calls the idle function for the first time. After this time passes, the idle engine can call the idle function as often as necessary.

ircIdle

Input parameter containing a bitmask of flags used to indicate the changes to be made to the idle function. The following flags can be set:

FIRCCSEC

Indicates a change to the time associated with the idle function (that is, a change indicated by the value passed in the *csecIdle* parameter).

FIRCIRO

Indicates a change to the options for the idle function (that is, a change indicated by the value passed in the *iroIdle* parameter).

FIRCPFN

Indicates a change to the idle function pointer (that is, a change indicated by the value passed in the *pfnIdle* parameter).

FIRCPRI

Indicates a change to the priority of the idle function (that is, a change indicated by the value passed in the *priIdle* parameter).

FIRCPV

Indicates a change to the memory block of the idle function (that is, a change indicated by the value passed in the *pvIdleParam* parameter).

CheckParameters

The **CheckParameters** macro calls an internal function to check debugging parameters on service provider methods called by MAPI.

At a Glance

Specified in header file:	MAPIVAL.H
Implemented by:	MAPI
Called by:	Service providers

HRESULT CheckParameters(
  METHODS *eMethod***,**
  LPVOID *First*
 );

Parameters

eMethod
  (Input) Specifies, by enumeration, the method to validate.

First
  (Input) Pointer to the first argument on the stack.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

Remarks

The **CheckParameters** macro has been superseded by the **CheckParms** macro. **CheckParameters** does not work correctly on RISC platforms and is now prevented from compiling on them. It still compiles and works correctly on Intel platforms, but **CheckParms** is recommended on all platforms.

CheckParms

The **CheckParms** macro calls an internal function to check debugging parameters on service provider methods called by MAPI.

At a Glance

Specified in header file:	MAPIVAL.H
Implemented by:	MAPI
Called by:	Service providers

HRESULT CheckParms(
  METHODS *eMethod***,**
  LPVOID *First*
 );

Parameters

eMethod
  (Input) Specifies, by enumeration, the method to validate.

First
  (Input) Pointer to the first argument on the stack.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

Remarks

In contrast to the **ValidateParms** and **UIValidateParms** macros, the **CheckParms** macro does not perform a full parameter validation. Parameters passed between MAPI and service providers are assumed to be correct, so **CheckParms** performs a debug validation only.

For more information on parameter validation, see Validating Parameters to Interface Methods.

CloseIMsgSession

This function may not be supported in future versions of MAPI.

The **CloseIMsgSession** function closes a message session.

At a Glance

Specified in header file:	IMESSAGE.H
Implemented by:	MAPI
Called by:	Client applications and service providers

VOID CloseIMsgSession(
  LPMSGSESS *lpMsgSess*
 );

Parameters

lpMsgSess
  Input parameter pointing to the message session object obtained using the **OpenIMsgSession** function at the start of the message session.

Remarks

A client application uses the **CloseIMsgSession** function along with the **OpenIMsgSession** function to wrap the creation of messages inside a message session. When a client application closes this session, it also closes all messages created within the session. Using sessions obtained with the **IMessage** interface is optional.

A message session keeps track of all messages opened during the session, in addition to all the tables and attachments within the messages. When a client calls **CloseIMsgSession**, it closes all these objects.

CreateIProp

The **CreateIProp** function creates a property data object, that is, an **IPropData** object.

At a Glance

Specified in header file:	MAPIUTIL.H
Implemented by:	MAPI
Called by:	Client applications and service providers

SCODE CreateIProp(
  LPCIID *lpInterface*,
  ALLOCATEBUFFER FAR * *lpAllocateBuffer*,
  ALLOCATEMORE FAR * *lpAllocateMore*,
  FREEBUFFER FAR * *lpFreeBuffer*,
  LPVOID *lpvReserved*,
  LPPROPDATA FAR * *lppPropData*
);

Parameters

lpInterface
Input parameter pointing to an interface identifier (IID) for the property data object.

lpAllocateBuffer
Input parameter pointing to the **MAPIAllocateBuffer** function, to be used to allocate memory.

lpAllocateMore
Input parameter pointing to the **MAPIAllocateMore** function, to be used to allocate additional memory where required.

lpFreeBuffer
Input parameter pointing to the **MAPIFreeBuffer** function, to be used to free memory.

lpvReserved
Reserved; must be zero.

lppPropData
Output parameter pointing to a variable where the returned property data object is stored.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

MAPI_E_INTERFACE_NOT_SUPPORTED
The requested interface is not supported for this object.

Remarks

The *lpAllocateBuffer*, *lpAllocateMore*, and *lpFreeBuffer* input parameters point to the **MAPIAllocateBuffer**, **MAPIAllocateMore**, and **MAPIFreeBuffer** functions, respectively. If a client application calls the **CreateIProp** function, it passes in these parameters pointers to the functions named as listed. If a service provider calls **CreateIProp**, it passes the pointers to these functions it received in its initialization call or retrieved with a call to the **IMAPISupport::GetMemAllocRoutines** method.

CreateTable

The **CreateTable** function creates structures and an object handle for the **ITableData** interface that can be used to create table views.

At a Glance

Specified in header file:	MAPIUTIL.H
Implemented by:	MAPI
Called by:	Client applications and service providers

SCODE CreateTable(
  LPCIID *lpInterface*,
  ALLOCATEBUFFER FAR * *lpAllocateBuffer*,
  ALLOCATEMORE FAR * *lpAllocateMore*,
  FREEBUFFER FAR * *lpFreeBuffer*,
  LPVOID *lpvReserved*,
  ULONG *ulTableType*,
  ULONG *ulPropTagIndexColumn*,
  LPSPropTagArray *lpSPropTagArrayColumns*,
  LPTABLEDATA FAR * *lppTableData*
);

Parameters

lpInterface
Input parameter pointing to an interface identifier (IID) for the table data object. Passing NULL in the *lpInterface* parameter indicates that the table data object returned in the *lppTableData* parameter is cast to the standard interface for a table data object; the valid interface identifier is IID_IMAPITableData.

lpAllocateBuffer
Input parameter pointing to the **MAPIAllocateBuffer** function, to be used to allocate memory.

lpAllocateMore
Input parameter pointing to the **MAPIAllocateMore** function, to be used to allocate additional memory where required.

lpFreeBuffer
Input parameter pointing to the **MAPIFreeBuffer** function, to be used to free memory.

lpvReserved
Reserved; must be zero.

ulTableType
Input parameter containing a table type that is available to a client application or service provider as part of the **IMAPITable::GetStatus** return data on its table views. Possible values are:

TBLTYPE_DYNAMIC
The table's contents are dynamic and can change as the underlying data changes.

TBLTYPE_KEYSET
The rows within the table are fixed, but the values within these rows are dynamic and can change as the underlying data changes.

TBLTYPE_SNAPSHOT
The table is static and the contents do not change when the underlying data changes.

ulPropTagIndexColumn
Input parameter containing the index number of the column for use when changing table data.

lpSPropTagArrayColumns

Input parameter pointing to an **SPropTagArray** structure containing an array of property tags indicating the properties required in the table for which the object holds data.

lppTableData

Output parameter pointing to a variable where the pointer to the returned table data object is stored.

Return Values

S_OK

The call succeeded and has returned the expected value or values.

Remarks

The *lpAllocateBuffer*, *lpAllocateMore*, and *lpFreeBuffer* input parameters point to the **MAPIAllocateBuffer**, **MAPIAllocateMore**, and **MAPIFreeBuffer** functions, respectively. If a client application calls the **CreateTable** function, it passes in these parameters pointers to the functions named as listed. If a service provider calls **CreateTable**, it passes the pointers to these functions that it received in its initialization call or retrieved with a call to the **IMAPISupport::GetMemAllocRoutines** method.

See Also

IMAPITable : IUnknown interface

DeinitMapiUtil

The **DeinitMapiUtil** function releases utility functions called explicitly by the **ScInitMapiUtil** function or implicitly by the **MAPIInitialize** function.

At a Glance

Specified in header file:	MAPIUTIL.H
Implemented by:	MAPI
Called by:	Client applications

DeinitMapiUtil(VOID);

Parameters

None

Remarks

The **ScInitMapiUtil** and **DeinitMapiUtil** functions cooperate to call and release select utility functions, as opposed to the **MAPIInitialize** function, which calls core as well as utility functions. When **ScInitMapiUtil** calls utility functions, it also initializes the necessary memory.

When use of the functions called by **ScInitMapiUtil** is complete, **DeinitMapiUtil** must be explicitly called to release them. In contrast, **MAPIInitialize** implicitly calls **DeinitMapiUtil**.

See Also

MAPIUninitialize function

DeregisterIdleRoutine

This function may not be supported in future versions of MAPI.

The **DeregisterIdleRoutine** function removes a client application or service provider function based on the **FNIDLE** function prototype from the idle table.

At a Glance

Specified in header file:	MAPIUTIL.H
Implemented by:	MAPI
Called by:	Client applications and service providers

VOID DeregisterIdleRoutine(
  FTG *ftg*
 );

Parameters

ftg
  Input parameter containing a function tag that identifies the idle function to be removed.

Remarks

A client application or service provider can only remove an idle function (that is, a function based on the **FNIDLE** function prototype) from the idle table if the function is not active. MAPI does not verify that the idle function is in a state from which it can be exited.

A client or provider can use the idle function itself to make the call to the **DeregisterIdleRoutine** function. The idle function is deregistered when the function returns.

After the idle function is deregistered, the idle engine does not call it again. Any implementation that calls **DeregisterIdleRoutine** must deallocate any memory blocks to which it passed pointers for the idle engine to use in its original call to the **FtgRegisterIdleRoutine** function.

DISMISSMODELESS

The **DISMISSMODELESS** function prototype defines a callback function that MAPI calls when it has dismissed a modeless address book dialog box. This call is necessary so that the client application will stop calling the accelerator function based on the **ACCELERATEABSDI** function prototype when the modeless dialog box is not active.

At a Glance

Specified in header file:	MAPIDEFS.H
Implemented by:	Client applications
Called by:	MAPI

void (STDMETHODCALLTYPE DISMISSMODELESS)(
  ULONG *ulUIParam***,**
  LPVOID *lpvContext*
 );

Parameters

ulUIParam
  Input parameter specifying an implementation-specific 32-bit value typically used for passing user interface information to a function. For example, in Microsoft Windows this parameter is the parent window handle for the dialog box and is of type HWND (cast to a ULONG). A value of zero is always valid.

lpvContext
  Input parameter pointing to an arbitrary value passed to the callback function when MAPI calls it. This value can represent an address of significance to the client application. Typically, for C++ code, *lpvContext* is a pointer to the address of a C++ object.

See Also

ADRPARM structure

EnableIdleRoutine

This function may not be supported in future versions of MAPI.

The **EnableIdleRoutine** function enables or disables an idle function based on the **FNIDLE** function prototype.

At a Glance

Specified in header file:	MAPIUTIL.H
Implemented by:	MAPI
Called by:	Client applications and service providers

VOID EnableIdleRoutine(
  FTG *ftg*,
  BOOL *fEnable*
);

Parameters

ftg
  Input parameter containing a function tag that identifies the idle function to be enabled or disabled.

fEnable
  Input parameter containing TRUE if the idle engine should enable the idle function, or FALSE if the idle engine should disable it.

See Also

FtgRegisterIdleRoutine function

FBadColumnSet

This function may not be supported in future versions of MAPI.

The **FBadColumnSet** function tests the validity of one or more table column sets for use by a service provider in a subsequent call to the **IMAPITable::SetColumns** method.

At a Glance

Specified in header file:	MAPIVAL.H
Implemented by:	MAPI
Called by:	Service providers

ULONG FBadColumnSet(
  LPSPropTagArray *lpptaCols*
 );

Parameters

lpptaCols
  Input parameter pointing to an **SPropTagArray** structure containing an array of property tags defining the table column sets to validate.

Return Values

TRUE
  The specified column sets are invalid.

FALSE
  The specified column sets are valid.

Remarks

A service provider calls the **FBadColumnSet** function. This function treats columns of type PT_ERROR as invalid and columns of type PT_NULL as valid.

FBadEntryList

This function may not be supported in future versions of MAPI.

The **FBadEntryList** function validates a list of MAPI entry identifiers.

At a Glance

Specified in header file:	MAPIVAL.H
Implemented by:	MAPI
Called by:	Service providers

BOOL FBadEntryList(
  LPENTRYLIST *lpEntryList*
 );

Parameters

lpEntryList
  Input parameter pointing to an **ENTRYLIST** structure defining the list of entry identifiers to be validated.

Return Values

TRUE
  One or more of the listed entry identifiers are invalid.

FALSE
  All of the listed entry identifiers are valid.

Remarks

The **FBadEntryList** function determines if the entry identifier list has been correctly generated. An example of an invalid entry identifier is an identifier for which memory has been incorrectly allocated or an identifier of an incorrect size.

FBadProp

This function may not be supported in future versions of MAPI.

The **FBadProp** function validates a specified property.

At a Glance

Specified in header file:	MAPIVAL.H
Implemented by:	MAPI
Called by:	Service providers

ULONG FBadProp(
  LPSPropValue *lpprop*
 );

Parameters

lpprop
  Input parameter containing an **SPropValue** structure defining the property to be validated.

Return Values

TRUE
  The specified property is invalid.

FALSE
  The specified property is valid.

Remarks

A service provider can call the **FBadProp** function for several reasons, for example to prepare for a call to the **IMAPIProp::SetProps** method setting a property. **FBadProp** validates the specified property depending on the property type. For example, if the property is Boolean, **FBadProp** ensures that its value is either TRUE or FALSE. If the property is binary, **FBadProp** checks its pointer and size and makes sure that it is allocated correctly.

See Also

FBadPropTag function

FBadPropTag

This function may not be supported in future versions of MAPI.

The **FBadPropTag** function validates a specified property tag.

At a Glance

Specified in header file:	MAPIVAL.H
Implemented by:	MAPI
Called by:	Service providers

ULONG FBadPropTag(
  ULONG *ulPropTag*
 );

Parameters

ulPropTag
  Input parameter containing the property tag to be validated.

Return Values

TRUE
  The specified property tag is not a valid MAPI property tag.

FALSE
  The specified property tag is a valid MAPI property tag.

Remarks

A service provider calls the **FBadPropTag** function.

See Also

FBadProp function

FBadRestriction

This function may not be supported in future versions of MAPI.

The **FBadRestriction** function validates a restriction used to limit a table.

At a Glance

Specified in header file:	MAPIVAL.H
Implemented by:	MAPI
Called by:	Service providers

ULONG FBadRestriction(
  LPSRestriction *lpres*
 );

Parameters

lpres
  Input parameter containing an **SRestriction** structure defining the restriction to be validated.

Return Values

TRUE
  The specified restriction or any of its subrestrictions is invalid.

FALSE
  The specified restriction and its subrestrictions are valid.

Remarks

A service provider calls the **FBadRestriction** function. Once validated, a restriction can be passed in calls to the **IMAPITable::Restrict** method to restrict a table to certain rows, to the **IMAPITable::FindRow** method to locate a table row, and to methods of the **IMAPIContainer** interface to perform a restriction on a container object.

FBadRgIpNameID ▸

This function may not be supported in future versions of MAPI.

The **FBadRgIpNameID** function validates an array of pointers that specify name identifier structures and verifies their allocation.

At a Glance

Specified in header file:	MAPIVAL.H
Implemented by:	MAPI
Called by:	Service providers

BOOL FBadRgIpNameID(
  LPMAPINAMEID FAR * *lppNameId***,**
  ULONG *cNames*
 );

Parameters

lppNameId
  Input parameter pointing to an array of **MAPINAMEID** structures defining the name identifiers.

cNames
  Input parameter containing the number of name identifiers in the array pointed to by the *lppNameId* parameter.

Return Values

TRUE
  One or more of the specified name identifiers is invalid.

FALSE
  The specified name identifiers are valid.

Remarks

See Also

IMAPIProp::GetIDsFromNames method, **IMAPIProp::GetNamesFromIDs** method

FBadRgLpszW

This function may not be supported in future versions of MAPI.

The **FBadRgLpszW** function validates all strings in an array of Unicode strings.

At a Glance

Specified in header file:	MAPIVAL.H
Implemented by:	MAPI
Called by:	Service providers

BOOL FBadRgLpszW(
  LPWSTR FAR * *lppszW*,
  ULONG *cStrings*
 );

Parameters

lppszW
  Input parameter pointing to an array of null-terminated Unicode strings.

cStrings
  Input parameter containing the number of strings in the array pointed to by the *lppszW* parameter.

Return Values

TRUE
  One or more of the strings in the specified array are invalid.

FALSE
  The strings in the specified array are valid.

FBadRow

This function may not be supported in future versions of MAPI.

The **FBadRow** function validates a row in a table.

At a Glance

Specified in header file:	MAPIVAL.H
Implemented by:	MAPI
Called by:	Service providers

ULONG FBadRow(
  LPSRow *lprow*
  );

Parameter

lprow
  Input parameter pointing to an **SRow** structure identifying the row to be validated.

Return Values

TRUE
  The specified row is invalid.

FALSE
  The specified row is valid.

Remarks

A service provider calls the **FBadRow** function.

See Also

FBadRowSet function

FBadRowSet

This function may not be supported in future versions of MAPI.

The **FBadRowSet** function validates all table rows included in a set of table rows.

At a Glance

Specified in header file:	MAPIVAL.H
Implemented by:	MAPI
Called by:	Service providers

BOOL FBadRowSet(
  LPSRowSet *lpRowSet*
 );

Parameter

lpRowSet
  Input parameter pointing to an **SRowSet** structure identifying the row set to be validated. If the pointer is NULL, the structure is invalid.

Return Values

TRUE
  A row of the specified row set is invalid or the row set itself is invalid.

FALSE
  The rows of the specified row set and the row itself are valid.

Remarks

A service provider calls the **FBadRowSet** function.

See Also

FBadRow function

FBadSortOrderSet

This function may not be supported in future versions of MAPI.

The **FBadSortOrderSet** function validates a sort order set by verifying its memory allocation.

At a Glance

Specified in header file:	MAPIVAL.H
Implemented by:	MAPI
Called by:	Service providers

ULONG FBadSortOrderSet(
  LPSSortOrderSet *lpsos*
 );

Parameter

lpsos
  Input parameter pointing to an **SSortOrderSet** structure identifying the sort order set to be validated.

Return Values

TRUE
  The specified sort order set is invalid.

FALSE
  The specified sort order set is valid.

Remarks

A service provider calls the **FBadSortOrderSet** function. This function can be used to prepare for a call to a sort method such as the **IMAPITable::SortTable** method.

FBinFromHex

This function may not be supported in future versions of MAPI.

The **FBinFromHex** function converts a string representation of a hexadecimal number to binary data.

At a Glance

Specified in header file:	MAPIUTIL.H
Implemented by:	MAPI
Called by:	Client applications and service providers

BOOL FBinFromHex(
  LPTSTR *sz***,**
  LPBYTE *pb*
 );

Parameters

sz
  Input parameter pointing to the null-terminated string to be converted. Valid characters include the hexadecimal characters zero through nine and both uppercase and lowercase characters A through F.

pb
  Output parameter pointing to a variable where the returned binary number is stored.

Return Values

TRUE
  The string was successfully converted into a binary number.

FALSE
  The input string contains invalid ASCII hexadecimal characters.

See Also

ScBinFromHexBounded function

FEqualNames

This function may not be supported in future versions of MAPI.

The **FEqualNames** function determines whether two MAPI name identifiers are equal.

At a Glance

Specified in header file:	MAPIUTIL.H
Implemented by:	MAPI
Called by:	Client applications and service providers

BOOL FEqualNames(
LPMAPINAMEID *lpName1*,
LPMAPINAMEID *lpName2*
);

Parameters

lpName1
Input parameter pointing to a **MAPINAMEID** structure defining the first name to be tested.

lpName2
Input parameter pointing to a **MAPINAMEID** structure defining the second name to be tested.

Return Values

TRUE
The two name identifiers are equal.

FALSE
The two name identifiers are not equal.

Remarks

The **FEqualNames** function is useful because name identifiers are represented as structures, which cannot be compared by simple binary methods. The testing process is case-sensitive for strings.

FNIDLE

This function may not be supported in future versions of MAPI.

The **FNIDLE** function prototype defines an idle function that the MAPI idle engine calls periodically according to priority. The specific functionality of the idle function is defined by the client or provider.

At a Glance

Specified in header file:	MAPIUTIL.H
Implemented by:	Client applications and service providers
Called by:	MAPI

BOOL (STDAPICALLTYPE FNIDLE)(
LPVOID *lpvContext*
);

Parameters

lpvContext
Input parameter specifying a pointer to a block of memory. The idle function can use this value as a pointer to a state buffer for length operations.

Return Values

FALSE
An idle function with the **FNIDLE** prototype should always return FALSE.

Remarks

A client application or service provider must call the idle engine function **Idle_InitDLL** before it can register its own idle function with a call to the **FtgRegisterIdleRoutine** function. Then the client application or provider can use these other idle engine functions as needed during idle operations:

- **EnableIdleRoutine**
- **ChangeIdleRoutine**
- **DeregisterIdleRoutine**
- **Idle_DeInitDLL**
- **FIsIdleExit**
- **FDoNextIdleTask**

FPropCompareProp

This function may not be supported in future versions of MAPI.

The **FPropCompareProp** function compares two properties using a binary relational operator.

At a Glance

Specified in header file:	MAPIUTIL.H
Implemented by:	MAPI
Called by:	Client applications and service providers

BOOL FPropCompareProp(
  LPSPropValue *lpSPropValue1***,**
  ULONG *ulRelOp***,**
  LPSPropValue *lpSPropValue2*
 );

Parameters

lpSPropValue1
  Input parameter pointing to an **SPropValue** structure defining the first property for comparison.

ulRelOp
  Input parameter containing the relational operator to use in the comparison.

lpSPropValue2
  Input parameter pointing to an **SPropValue** structure defining the second property for comparison.

Return Values

TRUE
  The function has succeeded in comparing the input properties.

FALSE
  The function has not succeeded in comparing the input properties.

Remarks

The order of comparison is *lpSPropValue1*, *ulRelOp*, *lpSPropValue2*. If the property types of the properties compared do not match, the **FPropCompareProp** function determines that they are not equal but that they are otherwise incomparable.

The comparison method depends on the property types included with the **SPropValue** property definitions and a fuzzy level heuristic also provided by the calling implementation. The **FPropContainsProp** function can be used to prepare restrictions for generating a table.

FPropContainsProp

This function may not be supported in future versions of MAPI.

The **FPropContainsProp** function compares two property values, generally strings or binary arrays, to see if one value contains the other.

At a Glance

Specified in header file:	MAPIUTIL.H
Implemented by:	MAPI
Called by:	Client applications and service providers

BOOL FPropContainsProp(
  LPSPropValue *lpSPropValueDst***,**
  LPSPropValue *lpSPropValueSrc***,**
  ULONG *ulFuzzyLevel*
 );

Parameters

lpSPropValueDst
: Input parameter pointing to an **SPropValue** structure defining the property value that might contain the property value pointed to by the *lpSPropValueSrc* parameter.

lpSPropValueSrc
: Input parameter pointing to an **SPropValue** structure defining the property value that **FPropContainsProp** is seeking within the property value pointed to by the *lpSPropValueDst* parameter.

ulFuzzyLevel
: Input parameter containing the fuzzy level to use in the comparison. Possible fuzzy level values are:

FL_FULLSTRING
: Works identically to TNEF_PROP_CONTAINED_TNEF; requires an exact match.

FL_IGNORECASE
: Indicates the comparison deals only with properties of type PT_STRING8. When this value is set, **FPropContainsProp** makes the comparison in case-insensitive fashion.

FL_IGNORENONSPACE
: Indicates the comparison deals only with properties of type PT_STRING8. When this value is set, **FPropContainsProp** makes the comparison so as to ignore Unicode-defined nonspacing characters, for example diacritical marks.

FL_LOOSE
: Indicates the comparison deals only with properties of type PT_STRING8. When this value is set, a service provider performs as many fuzzy level heuristics of types FL_IGNORECASE and FL_IGNORESPACE as it has been designed to handle.

FL_PREFIX
: Indicates the comparison deals with properties of types PT_STRING8 and PT_BINARY. When this value is set, **FPropContainsProp** compares the values of the two properties only through the length of the property indicated by the *lpSPropValueSrc* parameter.

FL_SUBSTRING
: Indicates the comparison deals with properties of types PT_STRING8 and PT_BINARY. When this value is set, **FPropContainsProp** checks to see if the property value indicated by the *lpSPropValueSrc* parameter is contained as a substring in the other property.

Return Values

TRUE

In the following cases:

- FL_FULLSTRING is set for the fuzzy level and the values of the source and destination properties are equivalent.
- FL_SUBSTRING is set for the fuzzy level and the property indicated by the *lpSPropValueSrc* parameter is contained as a substring in the property indicated by the *lpSPropValueDst* parameter.
- For PT_BINARY properties not falling into one of the categories listed for the *ulFuzzyLevel* parameter, the property indicated by *lpSPropValueSrc* is contained as a byte sequence in the property indicated by *lpSPropValueDst*.

FL_FULLSTRING, FL_SUBSTRING and FL_PREFIX are mutually exclusive. Only one of them can be set where anywhere from zero to all three of the other possible values can be set: FL_IGNORECASE, FL_IGNORENONSPACE, and FL_LOOSE.

FALSE

The properties being compared are not both of the same type, one or both of the properties is not of either the PT_STRING8 or PT_BINARY type, or the input fuzzy level is not one of those listed for *ulFuzzyLevel*.

Remarks

For comparisons of properties of type PT_STRING8 not covered by one of the values for *ulFuzzyLevel*, the **FPropContainsProp** function compares the input property values as fuzzy level one.

The comparison method **FPropContainsProp** uses depends on the property types included with the **SPropValue** property definitions and a fuzzy level heuristic also provided by the calling implementation. **FPropContainsProp** can be used to prepare restrictions for generating a table.

See Also

FPropCompareProp function

FPropExists

This function may not be supported in future versions of MAPI.

The **FPropExists** function searches for a given property tag in an **IMAPIProp** interface or an interface derived from **IMAPIProp**, such as **IMessage** or **IMAPIFolder**.

At a Glance

Specified in header file:	MAPIUTIL.H
Implemented by:	MAPI
Called by:	Client applications and service providers

BOOL FPropExists(
  LPMAPIPROP *pobj***,**
  ULONG *ulPropTag*
 );

Parameters

pobj
  Input parameter pointing to the **IMAPIProp** interface or interface derived from **IMAPIProp** within which to search for the property tag.

ulPropTag
  Input parameter containing the property tag for which to search.

Return Values

TRUE
  The function found a match for the given property tag.

FALSE
  The function did not find a match for the given property tag.

Remarks

If the given property tag has type PT_UNSPECIFIED, the **FPropExists** function finds a match only for the property identifier. Otherwise, the match is for the entire property tag, including the type.

FreePadrlist

The **FreePadrlist** function destroys an **ADRLIST** structure and frees associated memory, including memory allocated for all member structures.

At a Glance

Specified in header file:	MAPIUTIL.H
Implemented by:	MAPI
Called by:	Client applications and service providers

void FreePadrlist(
  LPADRLIST *padrlist*
 );

Parameter

padrlist
  Input parameter pointing to the **ADRLIST** structure to be destroyed.

Remarks

MAPI calls the **MAPIFreeBuffer** function to free every entry in the **ADRLIST** structure before freeing the complete structure. Therefore all such entries must have been allocated using the **MAPIAllocateBuffer** and **MAPIAllocateMore** functions.

FreeProws

The **FreeProws** function destroys an **SRowSet** structure and frees associated memory, including memory allocated for all structure members.

At a Glance

Specified in header file:	MAPIUTIL.H
Implemented by:	MAPI
Called by:	Client applications and service providers

void FreeProws(
   LPSRowSet *prows*
  );

Parameter

prows
   Input parameter pointing to the **SRowSet** structure to be destroyed.

Remarks

Usually, MAPI assigns the **MAPIAllocateBuffer** function as the function to free memory, but it can designate any other comparable function if necessary.

FtAddFt

This function may not be supported in future versions of MAPI.

The **FtAddFt** function adds one unsigned 64-bit integer to another.

At a Glance

Specified in header file:	MAPIUTIL.H
Implemented by:	MAPI
Called by:	Client applications and service providers

FILETIME FtAddFt(
  FILETIME *Addend1*,
  FILETIME *Addend2*
 );

Parameters

Addend1
  Input parameter containing a **FILETIME** structure defining the first unsigned 64-bit integer to be added.

Addend2
  Input parameter containing a **FILETIME** structure defining an unsigned 64-bit integer to be added to the value indicated by the *Addend1* parameter.

Remarks

The **FtAddFt** function returns a **FILETIME** structure containing the sum of the two integers.

FtgRegisterIdleRoutine

This function may not be supported in future versions of MAPI.

The **FtgRegisterIdleRoutine** function adds a function based on the **FNIDLE** function prototype to the idle table.

At a Glance

Specified in header file:	MAPIUTIL.H
Implemented by:	MAPI
Called by:	Client applications and service providers

FtgRegisterIdleRoutine(
  PFNIDLE *pfnIdle***,**
  LPVOID *pvIdleParam***,**
  short *priIdle***,**
  ULONG *csecIdle***,**
  USHORT *iroIdle*
 );

Parameters

pfnIdle
  Input parameter pointing to the idle function.

pvIdleParam
  Input parameter pointing to a block of memory that the idle engine should use when it calls the idle function.

priIdle
  Input parameter containing the initial priority that the calling implementation requests for the idle function. Possible priorities for implementation-defined functions are greater than or less than zero, but not zero. The zero priority is reserved for a user event (for example, a mouse click or a WM_PAINT message).

  Priorities greater than zero represent background tasks that have a higher priority than user events and are dispatched as part of the standard message pump loop. Priorities less than zero represent idle tasks that only run during message-pump idle time. Examples of priorities are: 1 for foreground submission, -1 for power-edit character insertion, and -3 for downloading new messages.

csecIdle
  Input parameter containing an initial time value, in hundredths of a second, that the calling implementation requests to be used in specifying idle function parameters. The meaning of the initial time value varies, depending on what is passed in the *iroIdle* parameter. It can be:

- The minimum period of user inaction that must elapse before the MAPI idle engine calls the idle function for the first time, if the FIROWAIT flag is set in *iroIdle*. After this time passes, the idle engine can call the idle function as often as necessary.
- The minimum interval between calls to the idle function, if the FIROINTERVAL flag is set in *iroIdle*.

iroIdle
  Input parameter containing a bitmask of flags used to set initial options for the idle function. The following flags can be set:

  FIRODISABLED
    Indicates that the idle function is initially disabled when registered. The default action is to enable the idle function when **FtgRegisterIdleRoutine** registers it.

  FIROINTERVAL

Indicates that the time specified by the *csecIdle* parameter is the minimum interval between successive calls to the idle function.

FIROWAIT

Indicates that the time specified by the *csecIdle* parameter is the minimum period of user inaction that must elapse before the MAPI idle engine calls the idle function for the first time. After this time passes, the idle engine can call the idle function as often as necessary.

Remarks

To later make changes to a registered idle function, a client application or service provider can call the **ChangeIdleRoutine** function. To remove an idle function from the idle table, a client application or provider calls the **DeregisterIdleRoutine** function.

When all foreground tasks for the platform become idle, the idle engine calls the highest-priority idle function in the idle table that is ready to execute.

The **FtgRegisterIdleRoutine** function returns a function tag identifying the idle function that **FtgRegisterIdleRoutine** has added to the idle table. If **FtgRegisterIdleRoutine** cannot register the idle function for the client or service provider, for example because of memory problems, it returns NULL.

FtMulDw

This function may not be supported in future versions of MAPI.

The **FtMulDw** function multiplies an unsigned 64-bit integer indicating a time value by an unsigned 32-bit integer in doubleword format.

At a Glance

Specified in header file:	MAPIUTIL.H
Implemented by:	MAPI
Called by:	Client applications and service providers

FILETIME FtMulDw(
  DWORD *Multiplier*,
  FILETIME *Multiplicand*
 );

Parameters

Multiplier
  Input parameter containing an unsigned 32-bit integer in doubleword format.

Multiplicand
  Input parameter containing a **FILETIME** structure defining a 64-bit integer time value to be multiplied by the value in the *Multiplier* parameter.

Remarks

The **FtMulDw** function returns a **FILETIME** structure containing the product of the input values.

FtMulDwDw

This function may not be supported in future versions of MAPI.

The **FtMulDwDw** function multiplies one unsigned 32-bit integer in doubleword format by another.

At a Glance

Specified in header file:	MAPIUTIL.H
Implemented by:	MAPI
Called by:	Client applications and service providers

FILETIME FtMulDwDw(
  DWORD *Multiplicand***,**
  DWORD *Multiplier*
 );

Parameters

Multiplicand
  Input parameter containing an unsigned 32-bit integer in doubleword format to be multiplied by the value in the *Multiplier* parameter.

Multiplier
  Input parameter containing an unsigned 32-bit integer in doubleword format.

Remarks

The **FtMulDwDw** function returns a **FILETIME** structure containing the product of the input values.

FtNegFt

This function may not be supported in future versions of MAPI.

The **FtNegFt** function computes the two's complement of an unsigned 64-bit integer indicating a time value.

At a Glance

Specified in header file:	MAPIUTIL.H
Implemented by:	MAPI
Called by:	Client applications and service providers

FILETIME FtNegFt(
  FILETIME *ft*
 );

Parameters

ft
  Input parameter containing a **FILETIME** structure containing the unsigned 64-bit integer indicating a time value for which to compute the two's complement.

Remarks

The **FtNegFt** function returns a **FILETIME** structure containing the two's complement of the input value.

FtSubFt

This function may not be supported in future versions of MAPI.

The **FtSubFt** function subtracts one unsigned 64-bit integer indicating a time value from another.

At a Glance

Specified in header file:	MAPIUTIL.H
Implemented by:	MAPI
Called by:	Client applications and service providers

FILETIME FtSubFt(
  FILETIME *Minuend***,**
  FILETIME *Subtrahend*
 );

Parameters

Minuend
  Input parameter containing a **FILETIME** structure defining the unsigned 64-bit integer from which the value in the *Subtrahend* parameter is subtracted.

Subtrahend
  Input parameter containing a **FILETIME** structure defining an unsigned 64-bit integer that this function subtracts from the value indicated by the *Minuend* parameter.

Remarks

The **FtSubFt** function returns a **FILETIME** structure containing the results of the subtraction.

GetAttribIMsgOnIStg

The **GetAttribIMsgOnIStg** function retrieves the attributes of the properties of a particular object.

At a Glance

Specified in header file:	IMESSAGE.H
Implemented by:	MAPI
Called by:	Client applications and message store providers

HRESULT GetAttribIMsgOnIStg(
  LPVOID *lpObject***,**
  LPSPropTagArray *lpPropTagArray***,**
  LPSPropAttrArray FAR * *lppPropAttrArray*
 );

Parameters

lpObject
  Input parameter, obtained from the **OpenIMsgOnIStg** function, pointing to the object for which property attributes are being retrieved.

lpPropTagArray
  Input parameter pointing to an **SPropTagArray** structure containing an array of property tags indicating the properties for which attributes are being retrieved.

lppPropAttrArray
  Output parameter pointing to a variable where the returned **SPropAttrArray** structure is stored. This **SPropAttrArray** structure contains the retrieved property attributes.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

MAPI_W_ERRORS_RETURNED
  The call succeeded overall, but one or more properties could not be accessed and were returned with a property type of PT_ERROR.

Remarks

The **GetAttribIMsgOnIStg** function is used to make message properties read-only when required by the **IMessage** schema. The sample message store provider uses it for this purpose. For more information, see Messages.

In the *lppPropAttrArray* parameter the number and position of the attributes correspond to the number and position of the property tags in the *lpPropTagArray* parameter.

See Also

IMAPIProp : IUnknown interface, **SetAttribIMsgOnIStg** function

GetInstance

This function may not be supported in future versions of MAPI.

The **GetInstance** function copies one value within a multivalued property to a single-valued property of the same type.

At a Glance

Specified in header file:	MAPIUTIL.H
Implemented by:	MAPI
Called by:	Client applications and service providers

VOID GetInstance(
  LPSPropValue *pvalMv*,
  LPSPropValue *pvalSv*,
  ULONG *uliInst*
 );

Parameters

pvalMv
  Input parameter pointing to an **SPropValue** structure defining a multivalued property.

pvalSv
  Input parameter pointing to a single-valued property to receive data.

uliInst
  Input parameter containing the instance number (that is, the array element) of the value being copied from the structure indicated by the *pvalMv* parameter.

Remarks

If the value copied is too large for the allocated memory, the **GetInstance** function only copies pointers rather than allocating new memory.

HexFromBin

This function may not be supported in future versions of MAPI.

The **HexFromBin** function converts a binary number into a string representation of a hexadecimal number.

At a Glance

Specified in header file:	MAPIUTIL.H
Implemented by:	MAPI
Called by:	Client applications and service providers

void HexFromBin(
  LPBYTE *pb*,
  int *cb*,
  LPTSTR *sz*
 );

Parameters

pb
  Input parameter pointing to the binary data to be converted.

cb
  Input parameter containing the size, in bytes, of the *pb* parameter.

sz
  Output parameter pointing to a variable where the returned null-terminated string representing the binary data is stored.

Remarks

This function takes a pointer to a unit of binary data whose size is indicated by the *cb* parameter. It returns back in the *sz* string, within (2**cb*)+1 bytes of memory, a representation of this binary information in hexadecimal numbers. If the byte value is 10, for example, the hexadecimal string will be 0A, so one byte becomes two bytes long in the string.

HPProviderInit

The **HPProviderInit** function initializes a message hook provider for operation.

At a Glance

Specified in header file:	MAPIHOOK.H
Implemented by:	Messaging hook providers
Called by:	MAPI

HRESULT HPProviderInit(
  LPMAPISESSION *lpSession*,
  HINSTANCE *hInstance*,
  LPALLOCATEBUFFER *lpAllocateBuffer*,
  LPALLOCATEMORE *lpAllocateMore*,
  LPFREEBUFFER *lpFreeBuffer*,
  LPMAPIUID *lpSectionUID*,
  ULONG *ulFlags*,
  LPSPOOLERHOOK FAR * *lppSpoolerHook*
 );

Parameters

lpSession
Input parameter pointing to a copy of the object representing the MAPI spooler session. Because the session object is a copy, any component installed as part of the message hook provider must be considered "trusted code." The message hook provider should call the **IUnknown::AddRef** method for the session object.

hInstance
Input parameter containing an instance of the message hook provider's dynamic-link library (DLL) that MAPI used when it linked.

lpAllocateBuffer
Input parameter pointing to the **MAPIAllocateBuffer** function, to be used by the message hook provider to allocate memory.

lpAllocateMore
Input parameter pointing to the **MAPIAllocateMore** function, to be used by the message hook provider to allocate additional memory where required.

lpFreeBuffer
Input parameter pointing to the **MAPIFreeBuffer** function, to be used by the message hook provider to free memory.

lpSectionUID
Input parameter pointing to the MAPI unique identifier (MAPIUID) of the message hook provider's profile section. **HPProviderInit** can open this identifier using a session-level call to the **IMAPISupport::OpenProfileSection** method. However, because MAPI and the MAPI spooler control some properties in the session, the provider should use the range of provider-specific property identifiers for storage and retrieval of profile section properties.

ulFlags
Input parameter containing a bitmask of flags used to control whether the message hook provider is called for incoming or outgoing messages. The following flags can be set:

HOOK_INBOUND
Indicates that the message hook provider processes messages inbound to the MAPI spooler.

HOOK_OUTBOUND
Indicates that the message hook provider processes messages outbound from the MAPI spooler.

MAPI_NT_SERVICE
Indicates the provider is being loaded in the context of a Windows NT service, a special type of process without access to any user interface.

lppSpoolerHook
Output parameter pointing to a pointer to the initialized message hook provider object.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

Remarks

To initialize a message hook provider, MAPI calls the function named **HPProviderInit**, based on the **HPPROVIDERINIT** function prototype defined in MAPIHOOK.H, from the message hook provider's DLL. The message hook provider must use its implementation of **HPProviderInit** to respond to the MAPI initialization call.

The message hook provider must also define **HPProviderInit** using the CDECL calling convention. CDECL definition is required for each service-provider initialization function to ensure the function can work with the current version of the service provider interface, even if the number of function parameters used is not the number set for that function in the current version of the interface. MAPI provides the **HPPROVIDERINIT** function prototype to help define **HPProviderInit** as CDECL. The **HPPROVIDERINIT** function prototype has a standard MAPI initialization call type, STDMAPIINITCALLTYPE.

The *lpAllocateBuffer*, *lpAllocateMore*, and *lpFreeBuffer* input parameters point to the **MAPIAllocateBuffer**, **MAPIAllocateMore**, and **MAPIFreeBuffer** functions, respectively, for use by the message-hook provider DLL. The provider should use these pointers for memory allocation or deallocation. In particular, the provider must use these pointers when allocating memory for use by clients in calling object interfaces. For example, two provider interface methods that typically allocate memory are **IMAPIProp::GetProps** and **IMAPITable::QueryRows**.

If the provider needs to use mutex objects or critical sections, it should set them up during initialization using **HPProviderInit**. A mutex object should be owned by the message-hook provider object created by this function.

For more information on using the OLE method **IUnknown::AddRef**, see Implementing the IUnknown Interface.

For more information on using **ABProviderInit**, see the information on using the **MSProviderInit**, **ABProviderInit**, and **XPProviderInit** functions in Initializing the Transport Provider and About Provider DLL Entry Point Functions.

See Also

ABProviderInit function, **IMAPISession : IUnknown** interface, **MSProviderInit** function, **XPProviderInit** function

HrAddColumnsEx

The **HrAddColumnsEx** function adds or moves columns to the beginning of an existing table.

At a Glance

Specified in header file:	MAPIUTIL.H
Implemented by:	MAPI
Called by:	Client applications and service providers

HRESULT HrAddColumnsEx(
  LPMAPITABLE *lptbl*,
  LPSPropTagArray *lpproptagColumnsNew*,
  LPALLOCATEBUFFER *lpAllocateBuffer*,
  LPFREEBUFFER *lpFreeBuffer*,
  void (FAR * *lpfnFilterColumns*) **(LPSPropTagArray** *ptaga*)
 );

Parameters

lptbl
  (Input) Pointer to the MAPI table affected.

lpproptagColumnsNew
  (Input) Pointer to an **SPropTagArray** structure containing the array of property tags for the properties to be added or moved to the beginning of the table.

lpAllocateBuffer
  (Input) Pointer to the **MAPIAllocateBuffer** function, to be used to allocate memory.

lpFreeBuffer
  (Input) Pointer to the **MAPIFreeBuffer** function, to be used to free memory.

lpfnFilterColumns
  (Input) Pointer to a callback function furnished by the caller. If the *lpfnFilterColumns* parameter is set to NULL, no callback is made.

ptaga
  (Input parameter to the callback function pointed to by *lpfnFilterColumns*) Pointer to an **SPropTagArray** structure containing the array of new property tags for the columns moved within or added to the table.

Return Values

S_OK
  The call succeeded and the specified columns were moved or added.

Remarks

The **HrAddColumnsEx** function allows the caller to furnish a callback function to filter the original property tags. For example, the caller might want to convert strings from property type PT_UNICODE to PT_STRING8. **HrAddColumnsEx** first adds or moves the specified columns, then calls the callback function if one is furnished, and finally calls **IMAPITable::SetColumns**. The callback function can change data within the property tag array but cannot add new tags.

The properties passed to **HrAddColumnsEx** using the *lpproptagColumnsNew* parameter will be the first properties listed on subsequent calls to the **IMAPITable::QueryRows** method. If any table properties are undefined when **QueryRows** is called, they will have the property type PT_NULL and the property identifier zero.

The *lpAllocateBuffer* and *lpFreeBuffer* input parameters point to the **MAPIAllocateBuffer** and **MAPIFreeBuffer** functions, respectively. If a client calls **HrAddColumnsEx**, it passes in these parameters pointers to the functions named as listed. If a service provider calls **HrAddColumnsEx**, it passes the pointers to these functions it received in its initialization call or retrieved by calling the **IMAPISupport::GetMemAllocRoutines** method.

See Also

IMAPITable::QueryColumns method

HrAllocAdviseSink

This function may not be supported in future versions of MAPI.

The **HrAllocAdviseSink** function creates an advise sink object, given a context specified by the calling implementation and a callback function to be triggered by an event notification.

At a Glance

Specified in header file:	MAPIUTIL.H
Implemented by:	MAPI
Called by:	Client applications and service providers

STDAPI HrAllocAdviseSink(
  LPNOTIFCALLBACK *lpfnCallback***,**
  LPVOID *lpvContext***,**
  LPMAPIADVISESINK FAR * *lppAdviseSink*
 );

Parameters

lpfnCallback
  Input parameter pointing to the callback function defined by a client application or service provider that MAPI is to call when a notification event occurs for the newly created advise sink.

lpvContext
  Input parameter pointing to an arbitrary value passed to the callback function when MAPI calls it. This value can represent an address of significance to the client or provider. Typically, for C++ code, the *lpvContext* parameter represents a pointer to the address of an object.

lppAdviseSink
  Output parameter pointing to a pointer to an advise sink object.

Remarks

To use the **HrAllocAdviseSink** function, a client application or provider creates an object to receive notifications, creates a notification callback function based on the **NOTIFCALLBACK** function prototype that goes with that object, and passes a pointer to the object in the **HrAllocAdviseSink** function as the *lpvContext* value. Doing so performs a notification; and as part of the notification process, MAPI calls the callback function with the object pointer as the context.

MAPI implements its notification engine asynchronously. In C++, the notification callback can be an object method. If the object generating the notification is not present, the client or provider requesting notification must keep a separate reference count for that object for the object's advise sink.

HrAllocAdviseSink should be used sparingly; it is safer for clients to create their own advise sinks.

HrComposeEID

This function may not be supported in future versions of MAPI.

The **HrComposeEID** function creates a compound entry identifier for an object, usually a message in a message store.

At a Glance

Specified in header file:	MAPIUTIL.H
Implemented by:	MAPI
Called by:	Client applications

HrComposeEID(
  LPMAPISESSION *psession***,**
  ULONG *cbStoreRecordKey***,**
  LPBYTE *pStoreRecordKey***,**
  ULONG *cbMsgEID***,**
  LPENTRYID *pMsgEID***,**
  ULONG FAR * *pcbEID***,**
  LPENTRYID FAR * *ppEID*
 );

Parameters

psession
  Input parameter pointing to the session in use by the client application.

cbStoreRecordKey
  Input parameter containing the size, in bytes, of the record key of the message store holding the message or other object. If zero is passed in the *cbStoreRecordKey* parameter, the *ppEID* parameter points to a copy of the object's entry identifier.

pStoreRecordKey
  Input parameter pointing to the record key of the message store containing the message or other object.

cbMsgEID
  Input parameter containing the size, in bytes, of the entry identifier of the message or other object.

pMsgEID
  Input parameter pointing to the entry identifier of the object.

pcbEID
  Output parameter pointing to a variable where the size, in bytes, of the returned identifier is stored.

ppEID
  Output parameter pointing to a variable where the returned data is stored. If the value of the *cbStoreRecordKey* parameter is greater than zero, the *ppEID* parameter points to a pointer to the compound entry identifier that is created. If *cbStoreRecordKey* is zero, *ppEID* points to a pointer to a copy of the object's entry identifier.

Remarks

If the message or other object for which the compound entry identifier is being created resides in a message store, the identifier is created from the object's entry identifier and the store's record key. If the object is not in a store (that is, if the byte count for the store record key passed in *cbStoreRecordKey* is zero), the object's entry identifier is simply copied.

The **HrComposeEID** function, primarily for use with CMC, enables CMC-based applications to work with objects in multiple stores through the use of compound entry identifiers.

See Also

HrComposeMsgID function, **HrDecomposeEID** function, **HrDecomposeMsgID** function

HrComposeMsgID

This function may not be supported in future versions of MAPI.

The **HrComposeMsgID** function creates an ASCII entry identifier string for an object, usually a message in a message store.

At a Glance

Specified in header file:	MAPIUTIL.H
Implemented by:	MAPI
Called by:	Client applications

HrComposeMsgID(
LPMAPISESSION *psession*,
ULONG *cbStoreRecordKey*,
LPBYTE *pStoreRecordKey*,
ULONG *cbMsgEID*,
LPENTRYID *pMsgEID*,
LPTSTR FAR * *pszMsgID*
);

Parameters

psession
Input parameter pointing to the session in use by the client application.

cbStoreRecordKey
Input parameter containing the size, in bytes, of the record key of the message store containing the message or other object. If zero is passed in the *cbStoreRecordKey* parameter, the *pszMsgID* parameter points to a copy of the entry identifier converted to text.

pStoreRecordKey
Input parameter pointing to the record key of the message store containing the message or other object.

cbMsgEID
Input parameter containing the size, in bytes, of the entry identifier of the message or other object.

pMsgEID
Input parameter pointing to the entry identifier of the object.

pszMsgID
Output parameter pointing to a variable where the returned data is stored. If the *cbStoreRecordKey* parameter is greater than zero, the *pszMsgID* parameter points to a compound entry identifier converted to text. If *cbStoreRecordKey* is zero, it points to a noncompound entry identifier converted to text.

Remarks

If the message or other object for which the compound entry identifier is being created resides in a message store, the identifier string is created from the object's entry identifier and the store's record key. If the object is not in a store (that is, if the byte count for the store record key passed in the *cbStoreRecordKey* parameter is zero), the object's entry identifier is simply copied and converted into a string.

The **HrComposeMsgID** function enables client applications based on Simple MAPI and OLE to work with objects in multiple stores through the use of compound entry identifiers.

Calling **HrComposeMsgID** is essentially equivalent to calling the **HrComposeEID** function and then the **HrSzFromEntryID** function.

See Also

HrDecomposeEID function, **HrDecomposeMsgID** function

HrDecomposeEID

This function may not be supported in future versions of MAPI.

The **HrDecomposeEID** function takes apart the compound entry identifier of an object, usually a message in a message store, into the entry identifier of that object within the store and that store's entry identifier.

At a Glance

Specified in header file:	MAPIUTIL.H
Implemented by:	MAPI
Called by:	Client applications

HrDecomposeEID(
  LPMAPISESSION *psession*,
  ULONG *cbEID*,
  LPENTRYID *pEID*,
  ULONG FAR * *pcbStoreEID*,
  LPENTRYID FAR * *ppStoreEID* ,
  ULONG FAR * *pcbMsgEID*,
  LPENTRYID FAR * *ppMsgEID*
 );

Parameters

psession
  Input parameter pointing to the session in use by the client application.

cbEID
  Input parameter containing the size, in bytes, of the compound entry identifier to be taken apart.

pEID
  Input parameter pointing to the compound entry identifier to be taken apart.

pcbStoreEID
  Output parameter pointing to a variable where the returned size, in bytes, of the entry identifier of the message store containing the object is stored. If the *pEID* parameter points to a noncompound entry identifier, then the *pcbStoreEID* parameter points to zero.

ppStoreEID
  Output parameter pointing to a variable where the returned entry identifier of the message store containing the object is stored. If the *pEID* parameter points to a noncompound entry identifier, NULL is returned in the *ppStoreEID* parameter.

pcbMsgEID
  Output parameter pointing to a variable where the returned size, in bytes, of the entry identifier of the object is stored. If the *pEID* parameter points to a noncompound entry identifier, then the *pcbMsgEID* parameter is equal to the value of the *cbEID* parameter.

ppMsgEID
  Output parameter pointing to a variable where the returned entry identifier of the object is stored. If *pEID* points to a noncompound entry identifier, memory is copied so that the pointer in *pEID* is equal to the pointer to the pointer in the *ppMsgEID* parameter.

Remarks

If the identifier specified by the *pEID* parameter is compound, it is split into the entry identifier of the object within its message store and that store's entry identifier. Noncompound entry identifier strings are simply copied. The compound identifier taken apart is usually one created by the **HrComposeEID** function.

The memory that holds the *pEID* parameter is released upon successful completion of this function. The calling implementation is responsible for freeing memory for the output parameters.

See Also

HrDecomposeMsgID function

HrDecomposeMsgID

This function may not be supported in future versions of MAPI.

The **HrDecomposeMsgID** function takes apart the compound entry-identifier string of an object, usually a message in a message store, into the entry identifier of that object within the store and that store's entry identifier.

At a Glance

Specified in header file:	MAPIUTIL.H
Implemented by:	MAPI
Called by:	Client applications

HrDecomposeMsgID(
  LPMAPISESSION *psession*,
  LPTSTR *szMsgID*,
  ULONG FAR * *pcbStoreEID*,
  LPENTRYID FAR * *ppStoreEID* ,
  ULONG FAR * *pcbMsgEID*,
  LPENTRYID FAR * *ppMsgEID*
 );

Parameters

psession
  Input parameter pointing to the session in use by the client application.

szMsgID
  Input parameter containing the entry identifier string of the object.

pcbStoreEID
  Output parameter pointing to a variable where the returned size, in bytes, of the entry identifier string of the message store containing the object is stored. If the *szMsgID* parameter points to a noncompound entry-identifier string, then the *pcbStoreEID* parameter points to zero.

ppStoreEID
  Output parameter pointing a variable where the returned entry identifier of the message store containing the object is stored. If the *szMsgID* parameter points to a noncompound entry identifier, NULL is returned in the *ppStoreEID* parameter.

pcbMsgEID
  Output parameter pointing to a variable storing the returned size, in bytes, of the entry identifier string of the object (within its store). If the *szMsgID* parameter contains a noncompound entry-identifier string, then the *pcbMsgEID* parameter is equal to the value of the *cbEID* parameter.

ppMsgEID
  Output parameter pointing to a variable where the returned entry identifier string of the object (within its store) is stored.

Remarks

Noncompound entry-identifier strings are accepted. The compound identifier string taken apart is usually one created by the **HrComposeMsgID** function.

See Also

HrDecomposeEID function

HrDispatchNotifications

The **HrDispatchNotifications** function forces dispatching of all queued notifications.

At a Glance

Specified in header file:	MAPIUTIL.H
Implemented by:	MAPI
Called by:	Client applications and service providers

```
HRESULT HrDispatchNotifications(
  ULONG ulFlags
 );
```

Parameters

ulFlags
Reserved; must be zero. .

Remarks

The **HrDispatchNotifications** function causes MAPI to dispatch all notifications that are currently queued in the MAPI notification engine without waiting for a message dispatch. This can have a beneficial effect on memory utilization.

HrEntryIDFromSz

This function may not be supported in future versions of MAPI.

The **HrEntryIDFromSz** function creates an entry identifier from an ASCII-encoded string, and allocates memory for the entry identifier.

At a Glance

Specified in header file:	MAPIUTIL.H
Implemented by:	MAPI
Called by:	Client applications

HrEntryIDFromSz(
  LPTSTR *sz***,**
  ULONG FAR * *pcb***,**
  LPENTRYID FAR * *ppentry*
 );

Parameters

sz
  Input parameter pointing to the ASCII-encoded string from which to create an entry identifier.

pcb
  Output parameter pointing to the size, in bytes, of the entry identifier pointed to by the *ppentry* parameter.

ppentry
  Output parameter pointing to a pointer to the returned **ENTRYID** structure containing the new entry identifier.

See Also

HrSzFromEntryID function

HrGetOneProp

This function may not be supported in future versions of MAPI.

The **HrGetOneProp** function retrieves the value of a single property from an **IMAPIProp** interface or an interface derived from **IMAPIProp**.

At a Glance

Specified in header file:	MAPIUTIL.H
Implemented by:	MAPI
Called by:	Client applications and service providers

HrGetOneProp(
  LPMAPIPROP *pmp*,
  ULONG *ulPropTag*,
  LPSPropValue FAR * *ppprop*
 );

Parameters

pmp
  Input parameter pointing to the interface from which the property value is to be retrieved.

ulPropTag
  Input parameter containing the property tag of the property to be retrieved.

ppprop
  Output parameter pointing to a pointer to the returned **SPropValue** structure defining the retrieved property value.

Return Values

MAPI_E_NOT_FOUND
  The requested object does not exist.

Remarks

Unlike the **IMAPIProp::GetProps** method, the **HrGetOneProp** function does not return any warning. Because it retrieves only one property, it simply either succeeds or fails. To retrieve multiple properties, a client application or service provider should call **GetProps**.

See Also

HrSetOneProp function

HrIStorageFromStream

The **HrIStorageFromStream** function layers an **IStorage** interface onto an **IStream** object.

At a Glance

Specified in header file:	MAPIUTIL.H
Implemented by:	MAPI
Called by:	Client applications and service providers

HRESULT HrIStorageFromStream(
  LPUNKNOWN *lpUnkIn*,
  PIID *lpInterface*,
  ULONG *ulFlags*,
  LPSTORAGE FAR * *lppStorageOut*
 );

Parameters

lpUnkIn
  Input parameter pointing to the **IUnknown** object that implements the stream object.

lpInterface
  Input parameter pointing to the interface identifier (IID) for the object that implements the stream object. Any of the following values can be passed in the *lpInterface* parameter: NULL, **IID_IStream**, or **IID_ILockBytes**. Passing NULL in *lpInterface* is the same as passing **IID_IStream**.

ulFlags
  Input parameter containing a bitmask of flags that controls how the storage is to be created relative to the stream. The default setting is that the storage has read-only access and occurs within the stream starting at position zero. The following flags can be set:

  STGSTRM_CREATE
    Creates a new storage object for the stream object.

  STGSTRM_CURRENT
    Starts storage at the current position of the stream.

  STGSTRM_MODIFY
    Allows the calling service provider to write to the returned storage.

  STGSTRM_RESET
    Starts storage at position zero.

lppStorageOut
  Output parameter pointing to a pointer to the returned **IStorage** object.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

Remarks

Store providers support the **HrIStorageFromStream** function using the **IStorage** interface for attachments. Store providers must implement the **IStream** interface. **HrIStorageFromStream** provides the **IStorage** interface for the **IStream** object. It is possible to pass either an **ILockBytes** or an **IStream** interface in *lpUnkIn*.

HrQueryAllRows

The **HrQueryAllRows** function retrieves all rows of a table.

At a Glance

Specified in header file:	MAPIUTIL.H
Implemented by:	MAPI
Called by:	Client applications and service providers

HRESULT HrQueryAllRows(
  LPMAPITABLE *ptable*,
  LPSPropTagArray *ptaga*,
  LPSRestriction *pres*,
  LPSSortOrderSet *psos*,
  LONG *crowsMax*,
  LPSRowSet FAR * *pprows*
 );

Parameters

ptable
  Input parameter pointing to the MAPI table from which rows are retrieved.

ptaga
  Input parameter pointing to an **SPropTagArray** structure containing an array of property tags indicating the properties that identify each column in the table. These tags are used to select the table columns to be retrieved. If the *ptaga* parameter is NULL, **HrQueryAllRows** retrieves the column set of the current table view passed in the *ptable* parameter.

pres
  Input parameter pointing to an **SRestriction** structure containing retrieval restrictions. If the *pres* parameter is NULL, the calling implementation makes no restrictions.

psos
  Input parameter pointing to an **SSortOrderSet** structure identifying the sort order of the columns to be retrieved. If the *psos* parameter is NULL, the default sort order for the table is used.

crowsMax
  Input parameter containing the maximum number of rows to be retrieved. If the value of the *crowsMax* parameter is zero, no limit on the number of rows retrieved is set.

pprows
  Output parameter pointing to the returned **SRowSet** structure containing the retrieved table rows.

Return Values

S_OK
  The call retrieved the expected rows of a table.

MAPI_E_TABLE_TOO_BIG
  The number of rows in the table is larger than the number passed for the *crowsMax* parameter.

Remarks

When querying all rows of a table, a client application or service provider should call **HrQueryAllRows** instead of the **IMAPITable::QueryRows** method.

See Also

IMAPITable : IUnknown interface

HrSetOneProp

This function may not be supported in future versions of MAPI.

The **HrSetOneProp** function changes one property of an object.

At a Glance

Specified in header file:	MAPIUTIL.H
Implemented by:	MAPI
Called by:	Client applications and service providers

HrSetOneProp(
  LPMAPIPROP *pmp***,**
  LPSPropValue *pprop*
 );

Parameters

pmp
  Input parameter pointing to an **IMAPIProp** interface or an interface derived from **IMAPIProp**.

pprop
  Input parameter pointing to the **SPropValue** structure defining the property to be changed.

Remarks

Unlike the **IMAPIProp::SetProps** method, the **HrSetOneProp** function does not return anyarning. Because it sets only one property, it simply either succeeds or fails. To change multiple properties, use the faster **SetProps**.

See Also

HrGetOneProp function

HrSzFromEntryID

This function may not be supported in future versions of MAPI.

The **HrSzFromEntryID** function encodes an entry identifier into an ASCII string, and allocates memory for the string.

At a Glance

Specified in header file:	MAPIUTIL.H
Implemented by:	MAPI
Called by:	Client applications

HrSzFromEntryID(
  ULONG *cb*,
  LPENTRYID *pentry*,
  LPTSTR FAR * *psz*
 );

Parameters

cb
  Input parameter containing the size, in bytes, of the entry identifier pointed to by the *pentry* parameter.

pentry
  Input parameter pointing to an **ENTRYID** structure defining the entry identifier to be encoded.

psz
  Output parameter pointing to a variable where the returned ASCII-encoded string is stored.

Remarks

To use Simple MAPI to look at messages in a folder other than the Inbox folder, which is the only folder recognized by Simple MAPI, a client application or service provider must use MAPI methods. A client or provider calls these methods to enumerate the messages in the folder by looking through the contents table for the folder, which stores message entry identifiers. When a client or provider finds the entry identifier for a particular required message, it can call the **HrSzFromEntryID** function to convert the identifier to a string that is a Simple MAPI message identifier so the message can be manipulated in Simple MAPI. Then the client or provider can call the Simple MAPI **MAPIReadMail** function to retrieve the required information about the message.

See Also

HrComposeEID function, **HrEntryIDFromSz** function

HrThisThreadAdviseSink

The **HrThisThreadAdviseSink** function creates an advise sink that wraps an existing advise sink for thread safety.

At a Glance

Specified in header file:	MAPIUTIL.H
Implemented by:	MAPI
Called by:	Client applications

HrThisThreadAdviseSink(
  LPMAPIADVISESINK *lpAdviseSink*,
  LPMAPIADVISESINK FAR * *lppAdviseSink*
 );

Parameters

lpAdviseSink
Input parameter pointing to the advise sink to be wrapped.

lppAdviseSink
Output parameter pointing a pointer to a new advise sink that wraps the advise sink pointed to by the *lpAdviseSink* parameter.

Remarks

The purpose of the wrapper is to ensure that notification is called on the same thread that called the **HrThisThreadAdviseSink** function. This function is used to protect notification callbacks that must run on a particular thread.

Client applications should use **HrThisThreadAdviseSink** to restrict when notifications are generated. This happens when calls are made to the **IMAPIAdviseSink::OnNotify** method of the advise sink object passed by the client in a previous **Advise** call. If notifications can be generated arbitrarily, a notification implementation might make a client multithreaded when multithreading is inappropriate. For example, this result is troublesome when an implementation has been created using a library, such as one of the Microsoft Foundation Class Libraries, that does not support multithreaded calls. In this situation, a client is difficult to test and is prone to error.

HrThisThreadAdviseSink ensures that the **OnNotify** methods calls occur at these appropriate times:

- During the processing of any call to any MAPI method.
- When window messages are being processed.

When **HrThisThreadAdviseSink** is implemented, any calls to the new advise sink's **OnNotify** method on any thread cause the original notification method to be executed on the thread in which **HrThisThreadAdviseSink** was called.

For more information on notification and advise sinks, see About Notification and Implementing an Advise Sink Object.

HrValidateIPMSubtree

This function may not be supported in future versions of MAPI.

The **HrValidateIPMSubtree** function adds one or more standard interpersonal message (IPM) folders to a message store.

At a Glance

Specified in header file:	MAPIUTIL.H
Implemented by:	MAPI
Called by:	Client applications

HrValidateIPMSubtree(
  LPMDB *lpMDB*,
  ULONG *ulFlags*,
  ULONG FAR * *lpcValues*,
  LPSPropValue FAR * *lppProps*,
  LPMAPIERROR FAR * *lppMapiError*
 );

Parameters

lpMDB
Input parameter pointing to the message store object to which to add the folder or folders.

ulFlags
Input parameter containing a bitmask of flags used to control how the folders are created. The following flags can be set:

MAPI_FORCE_CREATE
Indicates the folder or folders created be verified before creation, even if message store properties indicate that they are valid. A client application or service provider typically sets this flag when an error indicates that the structure of the created folder has been damaged.

MAPI_FULL_IPM_TREE
Indicates the following folders should be created in addition to the two folders that are always created: a folder titled Folder and a folder titled Common Views, and under the IPM-subtree root folder the Inbox, Outbox and Sent Items folders. The MAPI_FULL_IPM_TREE flag is typically set when the message store in which the folders are created is the default store.

lpcValues
Output parameter pointing to the number of values in the **SPropValue** array in the *lppProps* parameter is stored. The value of the *lpcValues* parameter can be zero if *lppProps* is NULL.

lppProps
Output parameter pointing to a pointer to an array of **SPropValue** structures indicating values for PR_VALID_FOLDER_MASK and related properties is stored. The IPM Inbox's entry identifier is included as part of this **SPropValue** array; this entry identifier has a special property tag coded as PROP_TAG(PT_BINARY, PROP_ID_NULL). The *lppProps* parameter can be NULL, indicating that the calling implementation does not require that a **SPropValue** array be returned.

lppMapiError
Output parameter pointing to a variable where the pointer to the returned **MAPIERROR** structure containing version, component, and context information for an error is stored. The *lppMAPIError* parameter can be set to NULL if there is no **MAPIERROR** structure to return.

Remarks

MAPI uses the **HrValidateIPMSubtree** function internally to construct the standard IPM folder tree in a message store when the store is first opened, or when a store is made the default store. This function can also be used by client applications to validate or repair standard message folders.

HrValidateIPMSubtree always creates a root folder in the IPM subtree, the Deleted Items folder in the root directory of the IPM subtree, and a finder folder in the message store root directory.

HrValidateIPMSubtree sets the PR_VALID_FOLDER_MASK property to indicate whether each IPM folder it creates has a valid entry identifier. The following entry identifier properties of the message store are set to the entry identifiers of the corresponding folders and returned in the *lppProps* parameter along with PR_VALID_FOLDER_MASK:

PR_COMMON_VIEWS_ENTRYID
PR_FINDER_ENTRYID
PR_IPM_OUTBOX_ENTRYID
PR_IPM_SENTMAIL_ENTRYID
PR_IPM_SUBTREE_ENTRYID
PR_IPM_WASTEBASKET_ENTRYID
PR_VIEWS_ENTRYID
PROP_TAG macro (PT_BINARY, PROP_ID_NULL, a placeholder tag for the IPM Inbox).

See Also

IMAPISession::OpenMsgStore method

LAUNCHWIZARDENTRY

The **LAUNCHWIZARDENTRY** function prototype defines a function that starts the Profile Wizard application.

At a Glance

Specified in header file:	MAPIWZ.H
Implemented by:	MAPI
Called by:	Client applications

HRESULT LAUNCHWIZARDENTRY(
  HWND *hParentWnd*,
  ULONG *ulFlags*,
  LPCTSTR FAR * *lppszServiceNameToAdd*,
  ULONG *cbBufferMax*,
  LPTSTR *lpszNewProfileName*
 );

Parameters

hParentWnd
Input parameter specifying a handle to the caller's parent window. If the caller does not have a parent window, the *hParentWnd* parameter should be NULL.

ulFlags
Input parameter containing a bitmask of flags indicating options for the Profile Wizard. The following flags can be set:

MAPI_PW_ADD_SERVICE_ONLY
Indicates that the Profile Wizard is to add a single service to the default profile and not show the page for selecting services.

MAPI_PW_FIRST_PROFILE
Indicates that the profile to be created is the first one for this workstation.

MAPI_PW_HIDE_SERVICES_LIST
Indicates that the Profile Wizard's page for selecting services should be hidden.

MAPI_PW_LAUNCHED_BY_CONFIG
Indicates that the Profile Wizard was launched by the Control Panel configuration application.

MAPI_PW_PROVIDER_UI_ONLY
Indicates that only the provider's configuration dialog boxes should be displayed and the Profile Wizard's pages should not appear. The MAPI_PW_PROVIDER_UI_ONLY flag can only be set if the MAPI_PW_ADD_SERVICE_ONLY flag is set.

lppszServiceNameToAdd
Input parameter specifying a pointer to a string containing the service name to be added if the *ulFlags* parameter is set to MAPI_PW_ADD_SERVICE_ONLY.

cbBufferMax
Input parameter specifying the size of the buffer pointed to by the *lpszNewProfileName* parameter.

lpszNewProfileName
Output parameter specifying a pointer to a buffer where the LAUNCHSERVICEENTRY function returns the name of the created profile.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

MAPI_E_CALL_FAILED

An error of unexpected or unknown origin prevented the operation from completing.

Remarks

The implementation of the **LAUNCHWIZARDENTRY** function prototype is the entry point into the MAPI Profile Wizard application. MAPI calls this entry point **LaunchWizard**.

When the *ulFlags* parameter is set to MAPI_PW_ADD_SERVICE_ONLY, the following rules apply:

- The flag MAPI_PW_LAUNCHED_BY_CONFIG causes the welcome page to be hidden.
- The flags MAPI_PW_PROVIDER_UI_ONLY & MAPI_PW_HIDE_SERVICES_LIST are useful only when there is no default profile. When a default profile does not exist, these flags determine the Profile Wizard page to be shown.
- When the flag MAPI_PW_ADD_SERVICE_ONLY is set and a default profile exists, this indicates that none of the Profile Wizard pages should be shown.

If the caller specifies MAPI_PW_ADD_SERVICE_ONLY and the service is already in the default profile (and only one such service can be in the profile at a time), the Profile Wizard does not add the provider. An error value is returned indicating that the provider was already in the default profile.

If the provider supports the wizard pages, it must allow programmatic configuration of the profile.

See Also

MSGSERVICEENTRY function prototype, **WIZARDENTRY** function prototype

LPFNBUTTON

The **LPFNBUTTON** function prototype defines a callback function that MAPI calls to activate an optional button control in an address book dialog box. This button is typically a Details button.

At a Glance

Specified in header file:	MAPIDEFS.H
Implemented by:	Service providers
Called by:	MAPI

SCODE (
  STDMETHODCALLTYPE FAR * LPFNBUTTON)(ULONG *ulUIParam*,
  LPVOID *lpvContext*,
  ULONG *cbEntryID*,
  LPENTRYID *lpSelection*,
  ULONG *ulFlags*
 );

Parameters

ulUIParam
  Input parameter containing the handle of the parent windows for any dialog boxes or windows this function displays.

lpvContext
  Input parameter pointing to an arbitrary value passed to the callback function when MAPI calls it. This value can represent an address of significance to the client application. Typically, for C++ code, *lpvContext* represents a pointer to the address of a C++ object.

cbEntryID
  Input parameter containing the size, in bytes, of the entry identifier pointed to by the *lpSelection* parameter.

lpSelection
  Input parameter pointing to the entry identifier defining the selection within the dialog box.

ulFlags
  Reserved; must be zero.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

Remarks

Client applications call a callback function based on the **LPFNBUTTON** prototype to define a button in a details dialog box. The client passes a pointer to the callback function in calls to the **IAddrBook::Details** method.

Service providers call a hook function based on the **LPFNBUTTON** prototype to define a button in a details dialog box. The provider passes a pointer to this hook function in calls to the **IMAPISupport::Details** method.

In both cases, when the dialog box is displayed and the user chooses the defined button, MAPI calls **LPFNBUTTON**.

See Also

BuildDisplayTable function

LPropCompareProp

This function may not be supported in future versions of MAPI.

The **LPropCompareProp** function compares two property values to determine if they are equal.

At a Glance

Specified in header file:	MAPIUTIL.H
Implemented by:	MAPI
Called by:	Client applications and service providers

LONG LPropCompareProp(
  LPSPropValue *lpSPropValueA***,**
  LPSPropValue *lpSPropValueB*
 );

Parameters

lpSPropValueA
  Input parameter pointing to an **SPropValue** structure defining the first property value to be compared.

lpSPropValueB
  Input parameter pointing to an **SPropValue** structure defining the second property value to be compared.

Remarks

Use the **LPropCompareProp** function only if the types of the two properties to be compared are the same.

Before calling **LPropCompareProp**, a client application or service provider must first retrieve the properties for comparison with a call to the **IMAPIProp::GetProps** method. When a client or provider calls **LPropCompareProp**, the function first examines the property tags to ensure that the comparison of property values is valid. The function then compares the property values, returning an appropriate value.

If the property values are unequal, **LPropCompareProp** determines which one is the greater. The properties that **LPropCompareProp** compares do not have to belong to the same object.

LPropCompareProp returns one of the following values for most property types:

- Less than zero if the value indicated by the *lpSPropValueA* parameter is less than that indicated by the *lpSPropValueB* parameter.
- Greater than zero if the value indicated by *lpSPropValueA* is greater than that indicated by *lpSPropValueB*.
- Zero if the value indicated by *lpSPropValueA* equals the value indicated by *lpSPropValueB*.

For property types that have no intrinsic ordering, such as Boolean or error types, the **LPropCompareProp** function returns an undefined value if the two property values are not equal. This undefined value is nonzero and consistent across calls.

MAPIAdminProfiles

The **MAPIAdminProfiles** function creates a profile administration object.

At a Glance

Specified in header file:	MAPIX.H
Implemented by:	MAPI
Called by:	Client applications

HRESULT MAPIAdminProfiles(
  ULONG *ulFlags***,**
  LPPROFADMIN FAR * *lppProfAdmin*
 );

Parameters

ulFlags
  Input parameter containing a bitmask of flags indicating options for the service entry function. The following flag can be set:

  MAPI_UNICODE
    Indicates the passed-in strings are in Unicode format. If the MAPI_UNICODE flag is not set, the strings are in ANSI format.

lppProfAdmin
  Output parameter pointing to a a pointer to the new profile administration object.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

See Also

IProfAdmin::CreateProfile method

MAPIAllocateBuffer

The **MAPIAllocateBuffer** function allocates a memory buffer.

At a Glance

Specified in header file:	MAPIX.H
Implemented by:	MAPI
Called by:	Client applications and service providers

SCODE MAPIAllocateBuffer(
  ULONG *cbSize***,**
  LPVOID FAR * *lppBuffer*
 );

Parameters

cbSize
  Input parameter containing the size, in bytes, of the buffer to be allocated.

lppBuffer
  Output parameter pointing to a variable where the returned allocated buffer is stored.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

Remarks

During **MAPIAllocateBuffer** call processing, the calling implementation acquires a block of memory from the operating system. The memory buffer is allocated on an even-numbered byte address. On platforms where long integer access is more efficient, the operating system allocates the buffer on an address whose size in bytes is a multiple of four.

Calling the **MAPIFreeBuffer** function releases the memory buffer allocated by **MAPIAllocateBuffer**, by calling the **MAPIAllocateMore** function and any buffers linked to it, when the memory is no longer needed.

MAPIAllocateMore

The **MAPIAllocateMore** function allocates a memory buffer that is linked to another buffer previously allocated with the **MAPIAllocateBuffer** function.

At a Glance

Specified in header file:	MAPIX.H
Implemented by:	MAPI
Called by:	Client applications and service providers

SCODE MAPIAllocateMore(
  ULONG *cbSize***,**
  LPVOID *lpObject***,**
  LPVOID FAR * *lppBuffer*
 );

Parameters

cbSize
  Input parameter containing the size, in bytes, of the new buffer to be allocated.

lpObject
  Input parameter pointing to an existing MAPI buffer allocated using **MAPIAllocateBuffer**.

lppBuffer
  Output parameter pointing to a variable where the returned, newly allocated buffer is stored.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

Remarks

During **MAPIAllocateMore** call processing, the calling implementation acquires a block of memory from the operating system. The memory buffer is allocated on an even-numbered byte address. On platforms where long integer access is more efficient, the operating system allocates the buffer on an address whose size in bytes is a multiple of four.

The only way to release a buffer allocated with **MAPIAllocateMore** is to pass the buffer pointer specified in the *lpObject* parameter to the **MAPIFreeBuffer** function. The link between the memory buffers allocated with **MAPIAllocateBuffer** and **MAPIAllocateMore** enables **MAPIFreeBuffer** to release both buffers with a single call.

MAPIDeInitIdle

This function may not be supported in future versions of MAPI.

The **MAPIDeInitIdle** function shuts down the DLL for the idle engine.

At a Glance

Specified in header file:	MAPIUTIL.H
Implemented by:	MAPI
Called by:	Client applications and service providers

VOID MAPIDeInitIdle(void)

Parameters

None.

Remarks

A client application or service provider should call this function when it no longer needs the idle engine, for example, when it is about to stop processing.

See Also

MAPIInitIdle, **FNIDLE**

MAPIFreeBuffer

The **MAPIFreeBuffer** function frees a memory buffer allocated with a call to the **MAPIAllocateBuffer** function or the **MAPIAllocateMore** function.

At a Glance

Specified in header file:	MAPIX.H
Implemented by:	MAPI
Called by:	Client applications and service providers

ULONG MAPIFreeBuffer(
  LPVOID *lpBuffer*
 );

Parameters

lpBuffer
  Input parameter pointing to a previously allocated memory buffer. If NULL is passed in the *lpBuffer* parameter, **MAPIFreeBuffer** does nothing.

Return Values

S_OK
  The call succeeded and freed the memory requested. MAPIFreeBuffer can also return S_OK on already freed locations or if memory block is not allocated with **MAPIAllocateBuffer** and **MAPIAllocateMore**.

Remarks

Usually, when a client application or service provider calls **MAPIAllocateBuffer** or **MAPIAllocateMore**, the operating system constructs in one contiguous memory buffer one or more complex structures with multiple levels of pointers. When a MAPI function or method creates a buffer with such contents, a client can later free all the structures contained in the buffer by passing to **MAPIFreeBuffer** the pointer to the buffer returned by the MAPI function that created the buffer. For a service provider to free a memory buffer using **MAPIFreeBuffer**, it must pass the pointer to that buffer returned with the provider's support object.

The call to **MAPIFreeBuffer** to free a particular buffer must be made as soon as a client or provider is finished using this buffer. Simply calling the **MAPILogoff** function at the end of a MAPI session does not automatically release memory buffers.

A client or service provider should operate on the assumption that the pointer passed in *lpBuffer* is invalid after a successful return from **MAPIFreeBuffer**. If the pointer indicates either a memory block not allocated by the messaging system through **MAPIAllocateBuffer** or **MAPIAllocateMore** or a free memory block, the behavior of **MAPIFreeBuffer** is undefined.

Note Passing a null pointer to **MAPIFreeBuffer** makes application cleanup code simpler and smaller because **MAPIFreeBuffer** can initialize pointers to NULL and then free them in the cleanup code without having to test them first.

MAPIFreeBuffer is exported, with a slightly different syntax, by both Simple MAPI and MAPI. For Simple MAPI, it is an entry point function.

See Also

IMAPISupport::GetMemAllocRoutines method

MAPIGetDefaultMalloc

The **MAPIGetDefaultMalloc** function retrieves the address of the default MAPI memory allocation function.

At a Glance

Specified in header file:	MAPIUTIL.H
Implemented by:	MAPI
Called by:	Client applications and service providers

LPMALLOC MAPIGetDefaultMalloc()

Parameters

None.

Remarks

The **MAPIGetDefaultMalloc** function returns a pointer to the default MAPI memory allocation function.

MAPIInitialize

The **MAPIInitialize** function increments the MAPI subsystem reference count and initializes global data for the MAPI DLL.

At a Glance

Specified in header file:	MAPIX.H
Implemented by:	MAPI
Called by:	Client applications

HRESULT MAPIInitialize(
LPVOID *lpMapiInit*
);

Parameters

lpMapiInit
Input parameter pointing to a **MAPIINIT_0** structure. The *lpMapiInit* parameter can be set to NULL.

Return Values

S_OK
The MAPI subsystem was initialized successfully.

Remarks

The **MAPIInitialize** function increments the MAPI reference count for the MAPI subsystem, and the **MAPIUninitialize** function decrements the internal reference count. Thus, the number of calls to one function must equal the number of calls to the other. **MAPIInitialize** returns S_OK if MAPI has not been previously initialized.

A client or service provider must call **MAPIInitialize** before making any other MAPI call. Failure to do so causes client or service provider calls to return the MAPI_E_NOT_INITIALIZED value.

When calling **MAPIInitialize** from a multithreaded application, set the *lpMapiInit* parameter to a **MAPIINIT_0** structure that is declared as follows:

MAPIINIT_0 MAPIINIT= { 0, MAPI_MULTITHREAD_NOTIFICATIONS}

and call:

MAPIInitialize (&MAPIINIT);

When this structure is declared, MAPI creates a separate thread to handle the notification window, which continues until the initialize reference count falls to zero. If the *ulflags* member is set to MAPI_NT_SERVICE, the provider is being loaded in the context of a Windows NT service.

MAPIInitialize does not return any extended error information. Unlike most other MAPI calls, the meanings of its return values are strictly defined to correspond to the particular step of the initialization that failed:

Step 1 Checks parameters and flags

MAPI_E_INVALID_PARAMETER or MAPI_E_UNKNOWN_FLAGS. Caller passed invalid parameter or flag.

Step 2 Initializes registry keys required by MAPI and confirms the type of operating system. This step only happens if the client process is runnning as a service under Windows NT and sets the MAPI_NT SERVICE flag in the MAPIINIT_0.

MAPI_E_TOO_COMPLEX. (Windows NT only) The calling process is an NT service and registry keys required by MAPI could not be initalized.

Additional information may be available in the application event log.

Step 3a Checks for compatibility between the current versions of OLE and MAPI.

MAPI_E_VERSION. The version of OLE installed on the workstation is not compatible with this version of MAPI.

Step 3b Initializes OLE.

During this step only, this function can return an error code not listed here. Any error *not* listed here should be assumed to come from the OLE function **CoInitialize**.

Step 4 Initializes per-process global variables

MAPI_E_SESSION_LIMIT MAPI sets up context specific to the current process. Failures may occur on Win16 if the number of processes exceeds a certain number, or on any system if available memory is exhausted.

Step 5 Initializes shared global variables of all processes

MAPI_E_NOT_ENOUGH_RESOURCES. Not enough system resources were available to complete the operation

Step 6 Initializes the notification engine, creates its window and its thread if requested by the MAPI_MULTITHREAD_NOTIFICATIONS flag.

MAPI_E_INVALID_OBJECT May fail if system resources are exhausted.

Step 7 Loads and initializes the profile provider. Verifies that **MAPIInitialize** can access the registry key where profile data are stored.

MAPI_E_NOT_INITIALIZED The profile provider has encountered an error. In a service running under Windows NT version 3.51, the profile provider must first initialize the registry subtree called HKEY_CURRENT_USER, this occurs in Step 2. This process is subject to additional failures, including lack of access rights to the registry.

MAPIInitIdle

This function may not be supported in future versions of MAPI.

The **MAPIInitIdle** function initializes the DLL for the idle engine. A client application or service provider must call **MAPIInitIdle** before calling any other idle engine function.

At a Glance

Specified in header file:	MAPIUTIL.H
Implemented by:	MAPI
Called by:	Client applications and service providers

LONG MAPIInitIdle(
  LPVOID *lpvReserved*
 );

Parameter

lpvReserved
  Reserved; must be zero.

Remarks

The **MAPIInitIdle** function returns zero if initialization is successful, and -1 otherwise. When **MAPIInitIdle** is called multiple times, only the first call succeeds.

See Also

MAPIDeinitIdle, **FNIDLE**

MAPILogonEx ▶

The **MAPILogonEx** function logs a client application onto a session with the messaging system.

At a Glance

Specified in header file:	MAPIX.H
Implemented by:	MAPI
Called by:	Client applications

HRESULT MAPILogonEx(
  ULONG *ulUIParam*,
  LPTSTR *lpszProfileName*,
  LPTSTR *lpszPassword*,
  FLAGS *flFlags*,
  LPMAPISESSION FAR * *lppSession*
 );

Parameters

ulUIParam
: Input parameter containing the handle to the window to which the logon dialog box is modal. If no dialog box is displayed during the call, the *ulUIParam* parameter is ignored. This parameter can be zero.

lpszProfileName
: Input parameter pointing to a string containing the name of the profile to use when logging on. This string is limited to 64 characters.

lpszPassword
: Input parameter pointing to a string containing the password of the profile. The *lpszPassword* parameter can be NULL whether or not the *lpszProfileName* parameter is NULL. This string is limited to 64 characters.

flFlags
: Input parameter containing a bitmask of flags used to control how logon is performed. The following flags can be set:

MAPI_ALLOW_OTHERS
: Indicates that the shared session should be returned, allowing subsequent clients to acquire the session without providing any user credentials.

MAPI_FORCE_DOWNLOAD
: Indicates an attempt should be made to download all of the user's messages before returning. If the MAPI_FORCE_DOWNLOAD flag is not set, messages can be downloaded in the background after the call to **MAPILogonEx** returns.

MAPI_LOGON_UI
: Indicates that a dialog box should be displayed to prompt the user for logon information if required. When the MAPI_LOGON_UI flag is not set, the calling client does not display a logon dialog box and returns an error value if the user is not logged on. MAPI_LOGON_UI and MAPI_PASSWORD_UI are mutually exclusive.

MAPI_NEW_SESSION
: Indicates an attempt should be made to create a new MAPI session rather than acquire the shared session. If the MAPI_NEW_SESSION flag is not set, **MAPILogonEx** uses an existing shared session even if the *lpszprofileName* parameter is not NULL.

MAPI_NO_MAIL
: Indicates that MAPI should not inform the MAPI spooler of the session's existence. The result is that no messages can be sent or received within the session except through a tightly-coupled store and transport pair. A calling client sets this flag when either configuration work must be done

or the client is browsing the available message stores.

MAPI_PASSWORD_UI
Indicates that a dialog box should be displayed to prompt the user for the profile password. MAPI_PASSWORD_UI cannot be set if MAPI_LOGON_UI is set because the calling client can only present one of the two dialog boxes. This dialog box does not allow the profile name to be changed; the *lpszProfileName* parameter must be non-NULL

MAPI_SERVICE_UI_ALWAYS
Indicates that **MAPILogonEx** should display a configuration dialog box for each message service in the profile. The dialog boxes are displayed after the profile has been chosen but before any message service is logged on. The MAPI common dialog box for logon also contains a check box that requests the same operation.

MAPI_TIMEOUT_SHORT
Indicates the logon should fail if blocked for more than a few seconds.

MAPI_UNICODE
Indicates the passed-in strings are in Unicode format. If the MAPI_UNICODE flag is not set, the strings are in ANSI format.

MAPI_USE_DEFAULT
Indicates the messaging subsystem should substitute the profile name of the default profile for the *lpszProfileName* parameter. The MAPI_EXPLICIT_PROFILE flag is ignored unless *lpszProfileName* is NULL or empty.

lppSession
Output parameter pointing to a pointer to the MAPI session interface.

Return Values

S_OK
The logon succeeded.

MAPI_E_LOGON_FAILED
The logon did not succeed, either because one or more of the parameters to **MAPILogonEx** were invalid or because there were too many sessions open already.

MAPI_E_TIMEOUT
MAPI serializes all logons through a mutex. This returns if the MAPI_TIMEOUT_SHORT flag was set and another thread held the mutex.

MAPI_E_USER_CANCEL
The user canceled the operation, typically by choosing the Cancel button in a dialog box.

Remarks

MAPI client applications call the **MAPILogonEx** function to log onto a session with the messaging system. All strings passed in and returned to and from MAPI calls are null-terminated and must be specified in the current character set or code page of the calling client or provider's operating system.

The *lpszProfileName* parameter is ignored if there is an existing previous session that called **MapiLogonEx** with the MAPI_ALLOW_OTHERS flag set and if the flag MAPI_NEW_SESSION is not set. If the *lpszProfileName* parameter is NULL or points to an empty string, and the *flFlags* parameter includes the MAPI_LOGON_UI flag, the **MAPILogonEx** function generates a logon dialog box with an empty field for the profile name.

When logging onto a specific profile, a client should pass the MAPI_NEW_SESSION flag into **MAPILogonEx** in addition to the profile name. Otherwise, if another client has established a shared session by logging on with MAPI_ALLOW_OTHERS, the client will be logged onto the shared session instead of to the profile requested.

The MAPI_EXPLICIT_PROFILE flag will not cause the default profile name to be used when *lpszProfileName* is NULL or empty unless the MAPI_USE_DEFAULT flag is also present.

The MAPI_NO_MAIL flag has several effects that result in the following when not using the MAPI spooler:

- No messages can be sent or delivered by the MAPI spooler during this session.
- Server based stores might still send or deliver messages.
- Messages sent or delivered by server based stores will not be processed by any hook providers.
- Per-message and per-recipient options for transports will not be available.
- The status table will not contain entries for transport providers, and any transport functionality dependent on status objects (such as configuration) will not be available.
- The message spooler row in the status table will contain the STATUS_FAILURE value.
- Piggybacked logons will be allowed, but those logons will not cause the previous logon to receive status object updates.

See Also

IMAPISession::GetMsgStoresTable method, **IMAPISession::OpenMsgStore** method

MAPIOpenFormMgr

The **MAPIOpenFormMgr** function opens an **IMAPIFormMgr** interface on a form library provider object in the context of an existing session.

At a Glance

Specified in header file:	MAPIFORM.H
Implemented by:	MAPI
Called by:	Client applications

MAPIOpenFormMgr(
  LPMAPISESSION *pSession*,
  LPMAPIFORMMGR FAR * *ppmgr*
 );

Parameters

pSession
 Input parameter pointing to the session in use by the client application.

ppmgr
 Output parameter pointing to the returned **IMAPIFormMgr** interface.

Remarks

After a client application makes a call to the **MAPIOpenFormMgr** function, most subsequent forms-related interactions take place through the form library provider or an interface returned by the form library provider. Among other things, the **IMAPIFormMgr** interface allows the client to work with message handlers and perform resolutions between message classes and form libraries.

MAPIOpenLocalFormContainer

The **MAPIOpenLocalFormContainer** function returns an interface pointer to the local form library.

At a Glance

Specified in header file:	MAPIFORM.H
Implemented by:	MAPI
Called by:	Client applications

MAPIOpenLocalFormContainer(
  LPMAPIFORMCONTAINER FAR * *ppfcnt*
 );

Parameters

ppfcnt
  Output parameter pointing to a pointer to the local form library interface.

Remarks

The interface to which a pointer is returned can be used by third-party installation programs to install application-specific forms into the library without the program first having to log onto MAPI.

MAPIUninitialize

The **MAPIUninitialize** function decrements the reference count, cleans up, and deletes per-instance global data for the MAPI DLL.

At a Glance

Specified in header file:	MAPIX.H
Implemented by:	MAPI
Called by:	Client applications

void MAPIUninitialize()

Parameters

None.

Remarks

A client application calls the **MAPIUninitialize** function to end its interaction with MAPI, begun with a call to the **MAPIInitialize** function. After **MAPIUninitialize** is called, no other MAPI calls can be made by the client.

MAPIUninitialize decrements the reference count, and the corresponding **MAPIInitialize** function increments the reference count. Thus, the number of calls to one function must equal the number of calls to the other.

MapStorageSCode

This function may not be supported in future versions of MAPI.

The **MapStorageSCode** function maps an HRESULT return value from an OLE storage object to a MAPI return value of the SCODE type.

At a Glance

Specified in header file:	IMESSAGE.H
Implemented by:	MAPI
Called by:	Client applications and service providers

SCODE MapStorageSCode(
  SCODE *StgSCode*
 );

Parameters

StgSCode
  Input parameter containing the HRESULT return value from an OLE storage object to be mapped to a MAPI SCODE value.

Return Values

S_OK
  Indicates the call succeeded and returned the expected value.

MAPI_E_CALL_FAILED
  Indicates the function cannot find a matching value.

Remarks

MAPI provides this function for the internal use of MAPI components that base their message implementations on the message DLL. Because these components open OLE storage themselves, they must be able to map error values returned for problems with OLE storage to MAPI SCODE values.

For more information, see About Structured Storage.

MSGCALLRELEASE

The **MSGCALLRELEASE** function prototype defines a callback function that frees the **IStorage** interface after the final release of a top-level message that was opened with the **OpenIMsgOnIStg** function.

At a Glance

Specified in header file:	IMESSAGE.H
Implemented by:	Client applications and service providers
Called by:	MAPI

void(
  ULONG *ulCallerData***,**
  LPMESSAGE *lpMessage*
 );

Parameters

ulCallerData
  An input parameter that contains calling application information about the **IMessage** interface.

lpMessage
  An input parameter pointing to the top-level message and attachments that have been released.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

MSGSERVICEENTRY

The **MSGSERVICEENTRY** function prototype defines an optional service provider entry point to support message service configuration.

At a Glance

Specified in header file:	MAPISPI.H
Implemented by:	Message services
Called by:	MAPI

HRESULT MSGSERVICEENTRY(
  HINSTANCE *hInstance*,
  LPMALLOC *lpMalloc*,
  LPMAPISUP *lpMAPISup*,
  ULONG *ulUIParam*,
  ULONG *ulFlags*,
  ULONG *ulContext*,
  ULONG *cValues*,
  LPSPropValue *lpProps*,
  LPPROVIDERADMIN *lpProviderAdmin*,
  LPMAPIERROR FAR * *lppMapiError*
 );

Parameters

hInstance
  Input parameter specifying the handle of the instance of the service provider DLL. The handle is typically used to retrieve resources.

lpMalloc
  Input parameter pointing to a memory allocator object exposing the OLE **IMalloc** interface. The message service may need to use this allocation method when working with certain interfaces such as **IStream**.

lpMAPISup
  Input parameter pointing to an **IMAPISupport:IUnknown** interface implementation.

ulUIParam
  Input parameter specifying an implementation-specific 32-bit value used for passing user interface information to a function or zero. In Microsoft Windows applications, the *ulUIParam* parameter is the parent window handle for the configuration dialog box and is of type HWND (cast to a ULONG). A value of zero indicates that there is no parent window.

ulFlags
  Input parameter containing a bitmask of flags indicating options for the service entry function. The following flags can be set:

  MAPI_UNICODE
    Indicates the passed-in strings are in Unicode format. If the MAPI_UNICODE flag is not set, the strings are in ANSI format.

  MSG_SERVICE_UI_READ_ONLY
    Indicates the service's configuration user interface should display the current configuration but not allow the user to change it.

  SERVICE_UI_ALLOWED
    Permits a configuration dialog box to be displayed if necessary. When the SERVICE_UI_ALLOWED flag is set, the dialog box should be displayed only if the *lpProps* property value array is empty or does not contain a valid configuration. If

SERVICE_UI_ALLOWED is not set, a dialog box might still be displayed if the UI_SERVICE_ALWAYS flag is set.

UI_CURRENT_PROVIDER_FIRST

Requests that the configuration dialog box for the active provider be displayed on top of other dialog boxes.

UI_SERVICE_ALWAYS

Requires the message service to display a configuration dialog box. If the UI_SERVICE_ALWAYS flag is not set, a configuration dialog box might still be displayed if the SERVICE_UI_ALLOWED flag is set and valid configuration information is not available from the *lpProps* property value array. Either SERVICE_UI_ALLOWED or UI_SERVICE_ALWAYS must be set to allow a user interface to be displayed.

ulContext

Input parameter specifying the configuration operation that MAPI is currently performing. The *ulContext* parameter will contain one of the following values:

MSG_SERVICE_CONFIGURE

Indicates that changes to the service's configuration should be made in the profile. If the UI_SERVICE_ALWAYS flag is set, the service should display its configuration dialog box. The dialog box should also be displayed if the SERVICE_UI_ALLOWED flag is set and the *lpProps* parameter is empty or does not contain valid configuration data. If *lpProps* contains valid data, no dialog box should be displayed and the service should use this data for making the configuration change.

MSG_SERVICE_CREATE

Indicates the service is being added to a profile. If either the UI_SERVICE_ALWAYS or SERVICE_UI_ALLOWED flag is set, the service should display its configuration dialog box. If neither flag is set, the service should fail.

MSG_SERVICE_DELETE

Indicates the service is being removed from a profile. After receiving this event, the service should return S_OK.

MSG_SERVICE_INSTALL

Indicates the service has been installed to the user's workstation from a network, floppy disk, or other external medium. After receiving this event, the service usually returns S_OK.

MSG_SERVICE_PROVIDER_CREATE

Requests that the service create an additional instance of a provider. If the service supports this operation, it should call **IProviderAdmin::CreateProvider**. If the service does not support this operation, it can return MAPI_E_NO_SUPPORT.

MSG_SERVICE_PROVIDER_DELETE

Requests that the service delete a provider instance. If the service supports this operation, it should call **IProviderAdmin::DeleteProvider**. If the service does not support this operation, it can return MAPI_E_NO_SUPPORT.

MSG_SERVICE_UNINSTALL

Indicates the service is being removed. After receiving this event, the service can perform any clean up tasks that should be done before the service ends and then return with a success value. If the user cancels the removal, the service should return MAPI_E_USER_CANCEL.

cValues

Input parameter specifying the number of property values in the array pointed to by the *lpProps* parameter. The value of the *cValues* parameter is zero if MAPI is passing no property values.

lpProps

Input parameter pointing to an optional array of **SPropValue** structures indicating values for provider-supported properties that the function will use in configuring the message service. The function only uses this parameter if the *ulContext* parameter is set to MSG_SERVICE_CONFIGURE. This parameter is commonly used to pass the path to a file for a file-based service, such as a personal address book service. If the MSG_SERVICE_CONFIGURE flag is not passed in the *ulFlags* parameter, the *lpProps* parameter must be zero.

lpProviderAdmin
Input parameter pointing to an **IProviderAdmin:IUnknown** interface that the function can use to locate profile sections for a specific provider in the current message service.

lppMapiError
Output parameter pointing to a **MAPIERROR** structure. The structure is allocated with the **MAPIAllocateBuffer** function. All members are optional, although most structures will contain a valid error message string in the *lpszError* member. If the *lpszComponent* or *lpszError* members of the structure are present, their memory must eventually be freed by a single call to **MAPIFreeBuffer** on the base structure.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

MAPI_E_UNCONFIGURED
The service provider has not been configured.

MAPI_E_USER_CANCEL
The user canceled the operation, typically by choosing the Cancel button in a dialog box.

MAPI_E_NO_SUPPORT
The provider either does not support changes to its objects or does not support notification of changes.

MAPI_E_BAD_CHARWIDTH
Either the MAPI_UNICODE flag was set and the implementation does not support Unicode, or MAPI_UNICODE was not set and the implementation only supports Unicode.

Remarks

A function defined using the **MSGSERVICEENTRY** function prototype enables message services to configure themselves or to perform other service-specific actions. The function primarily furnishes a dialog box in which the user can change settings specific to the message service. It can also support programmatic configuration by using the property value array passed in the *lpProps* parameter. Programmatic configuration is optional unless the service supports the Profile Wizard, for which it is required.

MAPI calls this entry point from the Control Panel application. Typically, a client or the profile provider calls the **MSGSERVICEENTRY** function prototype in response to a user's request. These two applications call **IMsgServiceAdmin::CreateMsgService** or **IMsgServiceAdmin::ConfigureMsgService** and these methods call the service entry function.

MAPI places no restriction on the function name that a message service uses for the **MSGSERVICEENTRY** prototype but prefers the name **ServiceEntry**. There is no restriction on the ordinal for the function, and a single provider DLL can contain more than one function. However, only one of the functions can be named **ServiceEntry**.

MAPI enables a message service to use Windows 95-style property sheets for its configuration dialog boxes. The message service can use the **BuildDisplayTable** function and the **IMAPISupport::DoConfigPropsheet** method to simplify configuration dialog box implementation.

It is possible for a user to cancel a MSG_SERVICE_UNINSTALL operation. In this case, the **ServiceEntry** function should check with the user to verify that the service should not be removed and return MAPI_E_USER_CANCEL if the service remains installed.

A function based on the **MSGSERVICEENTRY** prototype returns one of the HRESULT values listed. MAPI forwards this value when responding to a client's call to **IMsgServiceAdmin::ConfigureMsgService**.

Message services that export a service entry function must include the PR_SERVICE_DLL_NAME and PR_SERVICE_ENTRY_NAME properties in the message service section of MAPISVC.INF.

MSProviderInit

The **MSProviderInit** function initializes a message store provider for operation.

At a Glance

Specified in header file:	MAPISPI.H
Implemented by:	Message store providers
Called by:	MAPI

HRESULT MSProviderInit(
  HINSTANCE *hInstance*,
  LPMALLOC *lpMalloc*,
  LPALLOCATEBUFFER *lpAllocateBuffer*,
  LPALLOCATEMORE *lpAllocateMore*,
  LPFREEBUFFER *lpFreeBuffer*,
  ULONG *ulFlags*,
  ULONG *ulMAPIVer*,
  ULONG FAR * *lpulProviderVer*,
  LPMSPROVIDER FAR * *lppMSProvider*
 );

Parameters

hInstance
Input parameter containing an instance of the message store provider's dynamic-link library (DLL) that MAPI used when it linked.

lpMalloc
Input parameter pointing to a memory allocator object exposing the OLE **IMalloc** interface. The message store provider may need to use this allocation method when working with certain interfaces such as **IStream**.

lpAllocateBuffer
Input parameter pointing to the **MAPIAllocateBuffer** function, to be used to allocate memory.

lpAllocateMore
Input parameter pointing to the **MAPIAllocateMore** function, to be used to allocate additional memory where required.

lpFreeBuffer
Input parameter pointing to the **MAPIFreeBuffer** function, to be used to free memory.

ulFlags
Input parameter containing a bitmask of flags. The following flag can be set:

MAPI_NT_SERVICE
Indicates the provider is being loaded in the context of a Windows NT service, a special type of process without access to any user interface.

ulMAPIVer
Input parameter containing the version number of the service provider interface that MAPI.DLL uses. For the current version number, see the MAPISPI.H header file.

lpulProviderVer
Output parameter pointing to the version number of the service provider interface that this message store provider uses.

lppMSProvider
Output parameter pointing to a pointer to the initialized message store provider object.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

Remarks

To initialize a message store provider, MAPI calls the function named **MSProviderInit**, based on the MSPROVIDERINIT function prototype defined in MAPISPI.H from the message store provider's DLL. The message store provider must use its implementation of **MSProviderInit** to respond to the MAPI initialization call.

The message store provider must also define the **MSProviderInit** function using the CDECL calling convention. CDECL definition is required for each service provider initialization function to ensure the function can work with the current version of the service provider interface, even if the number of function parameters used is not the number set for that function in the current version of the interface. MAPI provides the **MSPROVIDERINIT** prototype to help define **MSProviderInit** as CDECL. The **MSPROVIDERINIT** prototype has a standard MAPI initialization call type, STDMAPIINITCALLTYPE.

The *lpAllocateBuffer*, *lpAllocateMore and lpFreeBuffer* input parameters specify pointers to the **MAPIAllocateBuffer**, **MAPIAllocateMore**, and **MAPIFreeBuffer** functions, respectively, for use by the message-store provider DLL. The provider DLL should not be linked with MAPIX.DLL. Instead, it should use these pointers for memory allocation or deallocation. In particular, the provider must use these pointers when allocating memory for use by clients in calling object interfaces. For example, two provider interface methods that typically allocate memory are **IMAPIProp::GetProps** and **IMAPITable::QueryRows**.

The message store provider should retain information on the allocator pointers passed to it in *lpAllocateBuffer*, *lpAllocateMore*, and *lpFreeBuffer*. If the provider will use a memory allocator later, it should call the OLE method **IUnknown::AddRef** for the allocation object pointed to by the *lpMalloc* parameter.

For more information on using **MSProviderInit**, see the information on using the **MSProviderInit**, **ABProviderInit**, and **XPProviderInit** functions in Initializing the Transport Provider and About Provider DLL Entry Point Functions.

See Also

ABProviderInit function, **HPProviderInit** function, **IMSProvider : IUnknown** interface, **XPProviderInit** function

NOTIFCALLBACK

This function may not be supported in future versions of MAPI.

The **NOTIFCALLBACK** function prototype defines a callback function that MAPI calls to send an event notification. This callback function can only be used when wrapped in an advise sink object created by calling the **HrAllocAdviseSink** function.

At a Glance

Specified in header file:	MAPIDEFS.H
Implemented by:	Client applications and service providers
Called by:	MAPI

ULONG (STDAPICALLTYPE NOTIFCALLBACK) (
   LPVOID *lpvContext***,**
   ULONG *cNotification***,**
   LPNOTIFICATION *lpNotifications*
 );

Parameters

lpvContext
   Input parameter specifying a pointer to an arbitrary value passed to the callback function when MAPI calls it. This value can represent an address of significance to the client application or service provider. Typically, for C++ code, the *lpvContext* parameter represents a pointer to the address of a C++ object.

cNotification
   Input parameter specifying the number of event notifications in the array indicated by the *lpNotifications* parameter.

lpNotifications
   Output parameter specifying a pointer to the location where this function writes an array of **NOTIFICATION** structures containing the event notifications.

Remarks

The set of valid return values for the **NOTIFCALLBACK** function prototype depends on whether the function is implemented by a client application or a service provider. Clients should always return S_OK. Providers should return either S_OK or NOTIFY_CANCELED. If NOTIFY_CANCELED is returned, the caller of **IMAPISupport::Notify** will receive NOTIFY_CANCELED in the *lpUlFlags* parameter.

See Also

IMAPIAdviseSink::OnNotify method

OpenIMsgOnIStg

The **OpenIMsgOnIStg** function creates internal memory structures and an object handle for creation of a new message object.

At a Glance

Specified in header file:	IMESSAGE.H
Implemented by:	MAPI
Called by:	Client applications and service providers

SCODE OpenIMsgOnIStg(
  LPMSGSESS *lpMsgSess*,
  LPALLOCATEBUFFER *lpAllocateBuffer*,
  LPALLOCATEMORE *lpAllocateMore*,
  LPFREEBUFFER *lpFreeBuffer*,
  LPMALLOC *lpmalloc*,
  LPVOID *lpMapiSup*,
  LPSTORAGE *lpStg*,
  MSGCALLRELEASE FAR * *lpfMsgCallRelease*,
  ULONG *ulCallerData*,
  ULONG *ulFlags*,
  LPMESSAGE FAR * *lppMsg*
);

Parameters

lpMsgSess
Input parameter pointing to a message session object.

lpAllocateBuffer
Input parameter pointing to the **MAPIAllocateBuffer** function, to be used by the service provider to allocate memory.

lpAllocateMore
Input parameter pointing to the **MAPIAllocateMore** function, to be used by the service provider to allocate additional memory where required.

lpFreeBuffer
Input parameter pointing to the **MAPIFreeBuffer** function, to be used by the service provider to free memory.

lpMalloc
Input parameter pointing to a memory allocator object exposing the OLE **IMalloc** interface. The message object may need to use this allocation method when working with certain interfaces such as **IStream**.

lpMapiSup
Input parameter pointing to an optional MAPI support object used when a service provider calls **OpenIMsgOnIStg**.

lpStg
Input-output parameter pointing to an **IStorage** object that is open and has read/write access. Because messages do not support write-only access, **OpenIMsgOnIStg** does not accept a storage object opened with write-only access.

lpfMsgCallRelease
Input parameter pointing to a message-release callback function in the service provider DLL.

ulCallerData
Input parameter containing caller data to be written by the callback function indicated by the

lpfMsgCallRelease parameter.

ulFlags
Input parameter containing a bitmask of flags used to control whether the OLE **IStorage::Commit** method is called when the client application calls the **IMessage::SaveChanges** method. The following flag can be set:

IMSG_NO_ISTG_COMMIT
Controls whether the OLE method **IStorage::Commit** is called when the client calls **SaveChanges**.

lppMsg
Output parameter pointing to a pointer to the opened message object.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

Warning The correct process for defining a message attachment is to call the **IMAPIProp::OpenProperty** method with a source interface of **IMessage : IMAPIProp**. **IMAPIProp::OpenProperty** is also supported for message attachments when the source interface is the OLE interface **IStorage**. **IStorage** is supported to allow an easy way to put a Microsoft Word for Windows document into an attachment without converting the attachment to or from the OLE **IStream** interface. However, use of **IStorage** presents a danger to the predictability of **IMessage** when the client application or service provider passes a new attachment data pointer to **OpenIMsgOnIStg** and then releases objects in the wrong order.

Including the *lpMsgSess* parameter ensures that the new message is created within a session so that it can be closed when the session is closed. If *lpMsgSess* is NULL, a message is created independently of any session. If the client or service provider that created the message does not release the message or does not release open tables within the message, memory is leaked until the external application terminates.

The *lpAllocateBuffer*, *lpAllocateMore*, and *lpFreeBuffer* input parameters point to the **MAPIAllocateBuffer**, **MAPIAllocateMore**, and **MAPIFreeBuffer** functions, respectively, for use by the service provider DLL. The provider DLL should not be linked with MAPIX.DLL. Instead, it should use these pointers for memory allocation or deallocation. In particular, the provider must use these pointers when allocating memory for use by clients in calling object interfaces. For example, two provider interface methods that typically allocate memory are **IMAPIProp::GetProps** and **IMAPITable::QueryRows**. The *lpAllocateBuffer*, *lpAllocateMore*, and *lpFreeBuffer* pointers are optional when the **OpenIMsgOnIStg** function is called with a valid *lpMapiSup* parameter.

If a value is supplied for *lpMapiSup*, **IMessage** supports the MAPI_DIALOG and ATTACH_DIALOG flags by calling the **IMAPISupport::DoProgressDialog** method to supply a progress user interface for the **IMAPIProp::CopyTo** and **IMessage::DeleteAttach** methods. The **IMessage::ModifyRecipients** method attempts to convert short-term entry identifiers to long-term entry identifiers by calling the support method **IMAPISession::OpenAddressBook** and making calls on the resulting address book object. If zero is passed for *lpMapiSup*, **IMessage** ignores MAPI_DIALOG and ATTACH_DIALOG and stores short-term entry identifiers without conversion.

MAPI does not define the behavior of multiple open operations performed on a message subobject, such as an attachment, a stream, a message store, or another message. MAPI currently allows a subobject that is already open to be opened once more, but MAPI performs the open operation by incrementing the reference count for the existing open object and returning it to the client or provider that called the **IMessage::OpenAttach** method or **IMAPIProp::OpenProperty** method. Thus, the access requested for the first open operation on a subobject is the access provided for all subsequent open operations, regardless of the access requested by the operations.

Some clients of **IMessage** might call the OLE method **IStorage::Commit** after writing data beyond what **IMessage** itself writes to the storage object. To aid in this, the **IMessage** implementation guarantees to name all substorages. Therefore, if the client keeps its names out of that namespace, there will be no accidental collisions.

The callback function mentioned with the *lpfMsgCallRelease* parameter is optional; if provided, it should be based on the **MSGCALLRELEASE** function prototype. If *lpfMsgCallRelease* is supplied, the **IMessage** interface calls the callback function when the top-level message receives its last release call. IMSG.DLL will not use the **IStorage** object pointed to by the *lpStg* parameter after making this call.

See Also

OpenIMsgSession function

OpenIMsgSession

This function may not be supported in future versions of MAPI.

The **OpenIMsgSession** function groups the creation of messages within a session, so that closing the session also closes all messages created within that session.

At a Glance

Specified in header file:	IMESSAGE.H
Implemented by:	MAPI
Called by:	Client applications and service providers

SCODE OpenIMsgSession(
  LPMALLOC *lpMalloc*,
  ULONG *ulFlags*,
  LPMSGSESS FAR * *lppMsgSess*
 );

Parameters

lpMalloc
  Input parameter pointing to a memory allocator object exposing the OLE **IMalloc** interface. The client application may need to use this allocation method when working with certain interfaces such as **IStream**.

ulFlags
  Reserved; must be zero.

lppMsgSess
  Output parameter pointing to a variable where the returned message-session object is stored.

Remarks

To establish a message session, a client application should call the **OpenIMsgSession** function to obtain a message session pointer before calling the **OpenIMsgOnIStg** function function for the first time to create a new message object. When finished with a message session, the client should call the **CloseIMsgSession** function to end it.

OpenIMsgSession is used by clients that require the ability to handle several related messages as storage objects. If only a single message is to be open at a time, a client does not need to track multiple messages. In this case, it has little cause to create a message session and no reason to call **OpenIMsgSession**.

For more information on the use of OLE memory allocators, see *Inside OLE*, *Second Edition*, by Kraig Brockschmidt, and the *OLE Programmer's Reference*.

See Also

IMAPISession : IUnknown interface,

OpenStreamOnFile

The **OpenStreamOnFile** function allocates and initializes a stream object to hold the contents of a file.

At a Glance

Specified in header file:	MAPIUTIL.H
Implemented by:	MAPI
Called by:	Client applications and service providers

HRESULT OpenStreamOnFile(
  LPALLOCATEBUFFER *lpAllocateBuffer*,
  LPFREEBUFFER *lpFreeBuffer*,
  ULONG *ulFlags*,
  LPTSTR *szFileName*,
  LPTSTR *szPrefix*,
  LPTSTREAM FAR * *lppStream*
 );

Parameters

lpAllocateBuffer
  Input parameter pointing to the **MAPIAllocateBuffer** function, to be used by the service provider to allocate memory.

lpFreeBuffer
  Input parameter pointing to the **MAPIFreeBuffer** function, to be used by the provider to free memory.

ulFlags
  Input parameter containing a bitmask of flags used to control the creation of the stream object. All of the flags used with the OLE 2.0 methods **IStream::Read** and **IStream::Write**, in addition to the following MAPI-defined flag, can be used:

  SOF_UNIQUEFILENAME
    Creates in a temporary directory a new file that is accessible with the OLE **IStream** interface. To create a read-only stream interface, do not use this flag.

szFileName
  Input parameter containing the filename for which this function opens a stream object.

szPrefix
  Input parameter containing the prefix for the filename on which the **OpenStreamOnFile** function opens a stream object.

lppStream
  Output parameter pointing to a variable where the pointer to the returned stream object is stored.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

MAPI_E_NO_ACCESS
  The object could not be accessed due to insufficient user permissions or because read-only objects cannot be modified.

MAPI_E_NOT_FOUND
  The requested object does not exist.

Remarks

A client application or service provider calls the **OpenStreamOnFile** function to allocate and initialize a stream object to hold the contents of a file. The stream object, and thus the file, is then accessible through the OLE interface **IStream**. To free the stream object, the client or provider must call the OLE method **IStream::Release**, which is inherited from the **IUnknown** interface.

A 32-bit Windows program should call **OpenStreamOnFile** strictly as defined in the preceding "Syntax" section, using the name provided. However, a 16-bit Windows program can set its own name for this function using the **OPENSTREAMONFILE** function prototype. The prototype has exactly the same syntax as that provided for the function preceding, except that it designates a return value of HRESULT instead of STDMETHODIMP.

The *lpAllocateBuffer* and *lpFreeBuffer* input parameters point to the **MAPIAllocateBuffer** and **MAPIFreeBuffer** functions, respectively, for use by the service provider DLL. The provider DLL should not be linked with MAPIX.DLL. Instead, it should use these pointers for memory allocation or deallocation. In particular, the provider must use these pointers when allocating memory for use by clients in calling object interfaces. For example, two provider interface methods that typically allocate memory are **IMAPIProp::GetProps** and **IMAPITable::QueryRows**.

The SOF_UNIQUEFILENAME flag is used to create a temporary filename that is unique to the messaging system. If this flag is set, the *szFileName* parameter specifes the filename, and the *szPrefix* parameter contains a string that is used to designate the directory in which the file should be created. This string is prefixed to the filename. If the value in *szFileName* is NULL, the unique file will be created in the temporary directory that is returned from the Windows function **GetTempDir**. If the SOF_UNIQUEFILENAME flag is not set, *szPrefix* is ignored and *szFileName* should contain the fully qualified path to the file being opened or created. The file will be opened or created based on the other flags that are set in *ulFlags*.

OpenTnefStream

The **OpenTnefStream** function is called by a transport provider to initiate a MAPI Transport Neutral Encapsulation Format (TNEF) session.

At a Glance

Specified in header file:	TNEF.H
Implemented by:	MAPI
Called by:	Transport providers

HRESULT OpenTnefStream(
  LPVOID *lpvSupport*,
  LPSTREAM *lpStream*,
  LPTSTR *lpszStreamName*,
  ULONG *ulFlags*,
  LPMESSAGE *lpMessage*,
  WORD *wKey*,
  LPITNEF FAR * *lppTNEF*
 );

Parameters

lpvSupport
  Passes a support object or passes in NULL. If NULL, the value of the *lpadrbook* parameter should be non-null.

lpStream
  Input parameter specifying a pointer to a storage stream object (OLE **IStream** interface) providing a source or destination for a TNEF stream message.

lpszStreamName
  Input parameter pointing to the name of the data stream that the TNEF object uses. If the caller has set the TNEF_ENCODE flag (*ulFlags* parameter) in its call to **OpenTnefStream**, the *lpszName* parameter must specify a non-null pointer to a non-null string consisting of any characters considered valid for naming a file. MAPI does not allow string names including the characters "[", "]", or "**:**", even if the file system permits their use. The size of the string passed for *lpszName* must not exceed the value of MAX_PATH, the maximum length of a string containing a path name.

ulFlags
  Input parameter containing a bitmask of flags used to indicate the mode of the function. The following flags can be set:

  TNEF_BEST_DATA
    Indicates that all possible properties are mapped into their down-level attributes, but when there is a possible data loss due to the conversion to a down-level attribute, the property is also encoded in the encapsulations. NOTE: this will cause the duplication of information in the TNEF stream. TNEF_BEST_DATA is the default if no other modes are specified.

  TNEF_COMPATIBILITY
    Ensures backwards compatibility with the MAIL 3.0 client. TNEF streams encoded with this flag will map all possible properties into their corresponding down-level attribute. This mode also causes the defaulting of some properties that are required by down-level clients.

  TNEF_DECODE
    Indicates the TNEF object on the indicated stream is opened with read-only access. The transport provider must set this flag if it wants the function to initialize the object for subsequent decoding.

  TNEF_ENCODE
    Indicates the TNEF object on the indicated stream is opened for read/write access. The transport provider must set this flag if it wants the function to initialize the object for subsequent encoding.

TNEF_PURE
Encodes all properties into the MAPI encapsulation blocks. Therefore, a "pure" TNEF file will consist of, at most, attMAPIProps, attAttachment, attRenddata, and attRecipTable. This mode is ideal for use when no backwards compatibility is required.

lpMessage
Input parameter pointing to a message object as a destination for a decoded message with attachments or a source for an encoded message with attachments. Any properties of a destination message might be overwritten by the properties of an encoded message.

wKey
Input parameter specifying a search key that the TNEF object uses to match attachments to the text tags inserted in the message text. This value should be relatively unique across messages.

lppTNEF
Output parameter pointing to a variable where the new TNEF object is stored.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

Remarks

A TNEF object created by the **OpenTnefStream** function later calls the OLE method **IUnknown::AddRef** to add references for the support object, the stream object, and the message object. The transport provider can release the references for all three objects with a single call to the OLE method **IUnknown::Release** on the TNEF object.

OpenTnefStream allocates and initializes a TNEF object (**ITnef** interface) for the provider to use in encoding a MAPI message (**IMessage** interface) into a TNEF stream message. Alternatively, the function can set up the object for the provider to use in subsequent calls to **ITnef::ExtractProps** to decode a TNEF stream message into a MAPI message. To free the TNEF object and close the session, the transport provider must call the inherited **IUnknown::Release** method on the object.

A 32-bit Windows program should call **OpenTnefStream** strictly as defined in the "Syntax" section, using the name provided. However, a 16-bit Windows program can set its own name for this function using the **OPENTNEFSTREAM** function prototype. The prototype has exactly the same syntax as the formal function, except that it designates a return value of HRESULT instead of STDMETHODIMP.

This function is the original entry point for TNEF access and has been replaced by **OpenTnefStreamEx** but is still used for compatibility for those already using TNEF.

See Also

IMAPISupport : IUnknown interface, **IXPProvider::TransportLogon** method

OpenTnefStreamEx

The **OpenTnefStreamEx** function creates a TNEF object that can be used to encode or decode a message object into a TNEF data stream for use by transports or gateways and message stores. This is the entry-point for TNEF access.

At a Glance

Specified in header file:	TNEF.H
Implemented by:	MAPI
Called by:	Transport providers

HRESULT OpenTnefStreamEx(
  LPVOID *lpvSupport*,
  LPSTREAM *lpStreamName*,
  LPTSTR *lpszStreamName*,
  ULONG *ulFlags*,
  LPMESSAGE *lpMessage*,
  WORD *wKeyVal*,
  LPADRBOOK *lpAdressBook*,
  LPITNEF FAR * *lppTNEF*
 );

Parameters

lpvSupport
  Passes a support object or passes in NULL. If NULL, the *lpadrbook* parameter should be non-null.

lpStreamName
  Input parameter specifying a pointer to a storage stream object, such as an OLE **IStream** interface, providing a source or destination for a TNEF stream message.

lpszStreamName
  Input parameter pointing to the name of the data stream that the TNEF object uses. If the caller has set the TNEF_ENCODE flag (*ulFlags* parameter) in its call to **OpenTnefStream**, the *lpszName* parameter must specify a non-null pointer to a non-null string consisting of any characters considered valid for naming a file. MAPI does not allow string names including the characters "[", "]", or "**:**", even if the file system permits their use. The size of the string passed for the *lpszName* parameter must not exceed the value of MAX_PATH, the maximum length of a string containing a path name.

ulFlags
  Input parameter containing a bitmask of flags used to indicate the mode of the function. The following flags can be set:

  TNEF_BEST_DATA
    Indicates that all possible properties are mapped into their down-level attributes, but when there is a possible data loss due to the conversion to a down-level attribute, the property is also encoded in the encapsulations. NOTE: this will cause the duplication of information in the TNEF stream. TNEF_BEST_DATA is the default if no other modes are specified.

  TNEF_COMPATIBILITY
    Ensures backwards compatibility with the MAIL 3.0 client. TNEF streams encoded with this flag will map all possible properties into their corresponding down-level attribute. This mode also causes the defaulting of some properties that are required by down-level clients.

  TNEF_DECODE
    Indicates the TNEF object on the indicated stream is opened with read-only access. The transport provider must set this flag if it wants the function to initialize the object for subsequent decoding.

  TNEF_ENCODE

Indicates the TNEF object on the indicated stream is opened for read/write access. The transport provider must set this flag if it wants the function to initialize the object for subsequent encoding.

TNEF_PURE

Encodes all properties into the MAPI encapsulation blocks. Therefore, a "pure" TNEF file will consist of, at most, the attributes attMAPIProps, attAttachment, attRenddata, and attRecipTable. This mode is ideal for use when no backwards compatibility is required.

lpMessage

Input parameter pointing to a message object as a destination for a decoded message with attachments or a source for an encoded message with attachments. Any properties of a destination message can be overwritten by the properties of an encoded message.

wKeyVal

Input parameter specifying a search key that the TNEF object uses to match attachments to the text tags inserted in the message text. This value should be relatively unique across messages.

lpAdressBook

Input parameter pointing to an address book object used to get addressing information for entry identifiers.

lppTNEF

Output parameter pointing to a variable where the new TNEF object is stored.

Return Values

S_OK

The call succeeded and has returned the expected value or values.

Remarks

OpenTnefStreamEx is the recommended replacement for **OpenTnefStream**, the original entry point for TNEF access.

A TNEF object created by the **OpenTnefStreamEx** function later calls the OLE method **IUnknown::AddRef** to add references for the support object, the stream object, and the message object. The transport provider can release the references for all three objects with a single call to the OLE method **IUnknown::Release** on the TNEF object.

OpenTnefStreamEx allocates and initializes a TNEF object for the provider to use in encoding a MAPI message into a TNEF stream message. Alternatively, this function can set up the object for the provider to use in subsequent calls to **ITnef::ExtractProps** to decode a TNEF stream message into a MAPI message. To free the TNEF object and close the session, the transport provider must call the inherited **IUnknown::Release** method on the object.

A 32-bit Windows program should call **OpenTnefStreamEx** strictly as defined in the syntax, using the name provided. However, a 16-bit Windows program can set its own name for this function using the **OPENTNEFSTREAM** function prototype. The prototype has exactly the same syntax as the formal function, except that it designates a return value of HRESULT instead of STDMETHODIMP.

The base value for the *wKeyVal* parameter must not be zero and should not be the same for every call to **OpenTnefStreamEx**. Instead, use random numbers based on the system time from the run-time library's random number generator.

See Also

IMAPISupport : IUnknown interface, **IXPProvider::TransportLogon** method

OPTIONCALLBACK

The **OPTIONCALLBACK** function prototype defines a callback function that MAPI calls to retrieve a wrapped **IMAPIProp** interface that manages a transport provider's properties.

At a Glance

Specified in header file:	MAPISPI.H
Implemented by:	Transport providers
Called by:	MAPI

SCODE OPTIONCALLBACK(
  HINSTANCE *hInst*,
  LPMALLOC *lpMalloc*,
  ULONG *ulFlags*,
  ULONG *cbOptionData*,
  LPBYTE *lpbOptionData*,
  LPMAPISUP *lpMAPISup*,
  LPMAPIPROP *lpDataSource*,
  LPMAPIPROP FAR * *lppWrappedSource*,
  LPMAPIERROR FAR * *lppMAPIError*
 );

Parameters

hInst
Input parameter containing the *hinstance* value for this transport provider's dynamic link library (DLL) as returned from the **LoadLibrary** function call made by MAPI.

lpMalloc
Input parameter pointing to a memory allocator object exposing the OLE **IMalloc** interface. The transport provider may need to use this allocation method when working with certain interfaces such as **IStream**.

ulFlags
Input parameter specifying a bitmask of flags that controls what options are processed. The following flags can be set:

OPTION_TYPE_MESSAGE
Indicates message options.

OPTION_TYPE_RECIPIENT
Indicates recipient options.

cbOptionData
Input parameter containing the size, in bytes, of the data pointed to by the *lpbOptionData* parameter.

lpbOptionData
Input parameter pointing to an **OPTIONDATA** structure containing option data for the recipient or message. This **OPTIONDATA** structure was passed in the call to the **IXPLogon::RegisterOptions** method that registered the options.

lpMAPISup
Input parameter pointing to a MAPI support object the provider can use to call the methods of the **IMAPISupport : IUnknown** interface.

lpDataSource
Input parameter pointing to an **IMAPIProp : IUnknown** interface that MAPI wraps for the transport provider's use.

lppWrappedSource
Output parameter pointing to a variable where the pointer to the wrapped **IMAPIProp** interface

returned by the transport provider is stored.

lppMAPIError
Output parameter pointing to a pointer to the returned **MAPIERROR** structure, if any, containing version, component, and context information for the error.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

Remarks

MAPI calls the transport provider's **OPTIONCALLBACK** function if a transport provider has previously registered message options with the **IXPLogon::RegisterOptions** method. Transport providers that do not define message or recipient options do not need to implement this callback function.

MAPI passes a wrapped **IMAPIProp : IUnknown** interface in the *lpDataSource* parameter. The transport provider should build a display table, set any properties, and then pass that display table back to MAPI in a wrapped **IMAPIProp** interface in the *lppWrappedSource* parameter. MAPI uses this **IMAPIProp** interface to display properties in the message or recipient options dialog box that is displayed to users. When users make selections in the dialog box resulting in a call to the **IMAPIProp::OpenProperty** method for the PR_DETAILS_TABLE property, the transport provider gets the call and should display the display table. The transport provider must call the **IMAPIProp::SetProps** method followed by the **IMAPIProp::SaveChanges** method on any changes the user made to the display table.

For more information on how to create display tables, see About Display Tables.

PpropFindProp

This function may not be supported in future versions of MAPI.

The **PpropFindProp** function searches for a specified property in a property set.

At a Glance

Specified in header file:	MAPIUTIL.H
Implemented by:	MAPI
Called by:	Client applications and service providers

PpropFindProp(
  LPSPropValue *rgprop***,**
  ULONG *cprop***,**
  ULONG *ulPropTag*
 );

Parameters

rgprop
  Input parameter containing an array of **SPropValue** structures that define the properties to be searched.

cprop
  Input parameter containing the number of properties in the property set indicated by the *rgprop* parameter.

ulPropTag
  Input parameter containing the property tag for the property to search for in the property set indicated by the *rgprop* parameter.

Remarks

If the given property tag indicates a property of type PT_UNSPECIFIED, the **PpropFindProp** function finds a match only for the property identifier within the tag. Otherwise, it finds a match for the entire property tag, including the property type, and returns the property so identified.

PpropFindProp returns an **SPropValue** structure defining the property that matches the input property tag, and NULL if there is no match.

PreprocessMessage

The **PreprocessMessage** function prototype defines a function that preprocesses message contents or the format of a message.

At a Glance

Specified in header file:	MAPISPI.H
Implemented by:	Transport providers
Called by:	MAPI spooler

HRESULT PreprocessMessage(
LPVOID *lpvSession*,
LPMESSAGE *lpMessage*,
LPADRBOOK *lpAdrBook*,
LPMAPIFOLDER *lpFolder*,
LPALLOCATEBUFFER *AllocateBuffer*,
LPALLOCATEMORE *AllocateMore*,
LPFREEBUFFER *FreeBuffer*,
ULONG FAR * *lpcOutbound*,
LPMESSAGE FAR * FAR * *lpppMessage*,
LPADRLIST FAR * *lppRecipList*
);

Parameters

lpvSession
Input parameter pointing to the session to be used.

lpMessage
Input parameter pointing to the message to be preprocessed.

lpAdrBook
Input parameter pointing to the address book from which the user should select recipients for the message.

lpFolder
Input-output parameter pointing to a folder. On input, the *lpFolder* parameter points to the folder that contains messages to be preprocessed. On output, *lpFolder* points to the folder where preprocessed messages have been placed.

lpAllocateBuffer
Input parameter pointing to the **MAPIAllocateBuffer** function, to be used to allocate memory.

lpAllocateMore
Input parameter pointing to the **MAPIAllocateMore** function, to be used to allocate additional memory where required.

lpFreeBuffer
Input parameter pointing to the **MAPIFreeBuffer** function, to be used to free memory.

lpcOutbound
Output parameter pointing to a variable containing the number of messages in the array pointed to by the *lpppMessage* parameter.

lpppMessage
Output parameter pointing to a pointer to an array of pointers to preprocessed or otherwise generated messages.

lppRecipList
Output parameter pointing to a variable where a returned **ADRLIST** structure, listing preprocessor-detected recipients for which the message is undeliverable, is optionally stored. For more

information on the contents of this list, see the **IMAPISupport::StatusRecips** method.

Remarks

A transport-provider message preprocessor can present a progress indicator during message preprocessing. However, it should never present a dialog box requiring user interaction during message preprocessing.

When a preprocessor adds large amounts of data to an outbound message, certain procedures should be followed. This type of message can be stored in a server-based message store, causing the preprocessor to access a remote store, a time-consuming procedure. To avoid having to do so, the preprocessor should have an option that enables it to store data that takes a large amount of space in a local message store and to provide a reference to that local store in the message.

The preprocessor should not release any of the objects originally passed to the function based on **PreprocessMessage**.

Before the MAPI spooler can call a **PreprocessMessage** function, the transport provider must have registered the function in a call to the **IMAPISupport::RegisterPreprocessor** method. After calling a **PreprocessMessage** function, the spooler cannot continue submitting a message until the function returns.

The MAPI spooler owns the task of submitting messages. This means the original message is never placed in an array of message pointers and that a call to the **SubmitMessage** methods is never required.

See Also

IAddrBook : IUnknown interface, **IMAPIFolder : IMAPIContainer** interface, **IMAPISupport : IUnknown** interface

PropCopyMore

This function may not be supported in future versions of MAPI.

The **PropCopyMore** function copies a single property value from a source location to a destination location.

At a Glance

Specified in header file:	MAPIUTIL.H
Implemented by:	MAPI
Called by:	Client applications and service providers

SCODE PropCopyMore(
  LPSPropValue *lpSPropValueDest***,**
  LPSPropValue *lpSPropValueSrc***,**
  ALLOCATEMORE * *lpfAllocMore***,**
  LPVOID *lpvObject*
 );

Parameters

lpSPropValueDest
  Output parameter pointing to the location to which this function writes an **SPropValue** structure defining the copied property value.

lpSPropValueSrc
  Input parameter pointing to the **SPropValue** structure containing the property value to be copied.

lpfAllocMore
  Input parameter pointing to the **MAPIAllocateMore** function to be used to allocate additional memory if the destination location is not large enough to hold the property to be copied.

lpvObject
  Input parameter pointing to an object for which **MAPIAllocateMore** will allocate space if necessary.

Remarks

A client application or service provider can use the **PropCopyMore** function to copy a property out of a table that is about to be freed in order to use it elsewhere.

PropCopyMore does not need to allocate memory unless the property value copied is of a type, such as PT_STRING8, that does not fit in an **SPropValue** structure. For these large properties, the function allocates memory using the **MAPIAllocateMore** function to which a pointer is passed in the *lpfAllocMore* parameter.

Injudicious use of **PropCopyMore** fragments memory; consider using the **ScCopyProps** function instead.

RemovePreprocessInfo

The **RemovePreprocessInfo** function prototype defines a function that removes from a message preprocessed information written by a function based on the **PreprocessMessage** function prototype.

At a Glance

Specified in header file:	MAPISPI.H
Implemented by:	Transport providers
Called by:	MAPI spooler

HRESULT RemovePreprocessInfo(
  LPMESSAGE *lpMessage*
 );

Parameters

lpMessage
  Input parameter pointing to the preprocessed message from which information is to be removed.

Remarks

The MAPI spooler calls a **RemovePreprocessInfo** function. A transport provider registers the function based on **RemovePreprocessInfo** at the same time it registers the parallel function based on **PreprocessMessage** in a call to the **IMAPISupport::RegisterPreprocessor** method.

An image rendering suitable for fax transmission is an example of preprocessed information written by a function defined by the **PreprocessMessage** function prototype. The MAPI spooler usually calls a **RemovePreprocessInfo** function after sending a message containing preprocessed information.

RTFSync

The **RTFSync** function ensures that the Rich Text Format (RTF) message text matches the plain text version. It is necessary to call this function before reading the RTF version and after modifying the RTF version.

At a Glance

Specified in header file:	MAPIUTIL.H
Implemented by:	MAPI
Called by:	RTF-aware client applications and message store providers

HRESULT RTFSync(
  LPMESSAGE *lpMessage*,
  ULONG *ulFlags*,
  BOOL FAR * *lpfMessageUpdated*
 );

Parameters

lpMessage
  Input parameter pointing to the message to be updated.

ulFlags
  Input parameter containing a bitmask of flags used to indicate the RTF or plain text version of the message has changed. The following flags can be set:

  RTF_SYNC_BODY_CHANGED
    Indicates the plain text version of the message has changed.

  RTF_SYNC_RTF_CHANGED
    Indicates the RTF version has changed.

  All other bits in the *ulFlags* parameter are reserved for future use.

lpfMessageUpdated
  Output parameter pointing to a variable indicating whether there is an updated message. TRUE if there is an updated message, FALSE otherwise.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

Remarks

If the PR_RTF_IN_SYNC property is missing or is FALSE, before reading the PR_RTF_COMPRESSED property the **RTFSync** function should be called with the RTF_SYNC_BODY_CHANGED flag set.

If the STORE_RTF_OK flag is not set in the PR_STORE_SUPPORT_MASK property, this function should be called with the RTF_SYNC_RTF_CHANGED flag set after modifying PR_RTF_COMPRESSED.

If both PR_BODY and PR_RTF_COMPRESSED have been changed, the **RTFSync** function should be called with both flags set.

If the value of the *lpfMessageUpdated* parameter is set to TRUE, then the **IMAPIProp::SaveChanges** method should be called for the message. If **SaveChanges** is not called, the modifications will not be saved in the message.

Message store providers can use **RTFSync** to keep the PR_BODY and PR_RTF_COMPRESSED properties in sync.

For more information, see About Supporting RTF Text for Message Store Providers.

See Also

WrapCompressedRTFStream function

ScBinFromHexBounded

This function may not be supported in future versions of MAPI.

The **ScBinFromHexBounded** function converts the specified portion of a string representation of a hexadecimal number into a binary number.

At a Glance

Specified in header file:	MAPIUTIL.H
Implemented by:	MAPI
Called by:	Client applications and service providers

SCODE ScBinFromHexBounded(
  LPTSTR *sz*,
  LPBYTE *pb*,
  ULONG *cb*
 );

Parameters

sz
: Input parameter pointing to the null-terminated string to be converted. Valid characters include the hexadecimal characters 0 through 9 and both uppercase and lowercase characters a through f.

pb
: Output parameter pointing to a variable where the returned binary number is stored.

cb
: Input parameter containing the size, in bytes, of the *pb* parameter.

See Also

FBinFromHex function

ScCopyNotifications

This function may not be supported in future versions of MAPI.

The **ScCopyNotifications** function copies a group of event notifications to a single block of memory.

At a Glance

Specified in header file:	MAPIUTIL.H
Implemented by:	MAPI
Called by:	Client applications and service providers

SCODE ScCopyNotifications(
  int *cntf*,
  LPNOTIFICATION *rgntf*,
  LPVOID *pvDst*,
  ULONG FAR * *pcb*
 );

Parameters

cntf
  Input parameter containing the number of **NOTIFICATION** structures in the array indicated by the *rgntf* parameter.

rgntf
  Input parameter pointing to an array of **NOTIFICATION** structures defining the event notifications to be copied.

pvDst
  Output parameter pointing to a variable where the returned notifications are stored.

pcb
  Optional output parameter pointing to a variable where the size, in bytes, of the array pointed to by the *rgntf* parameter is stored. If not NULL, the *pcb* parameter is set to the number of bytes stored in the *pvDst* parameter.

Remarks

If NULL is passed in the *pcb* parameter, no copying is performed; if a non-null value is passed in *pcb*, the **ScCopyNotifications** function copies the size of the array and the array itself to a single block of memory. If *pcb* is not NULL, it is set to the number of bytes stored in the *pvDst* parameter. The *pvDst* parameter must be large enough to contain the entire array.

ScCopyProps

This function may not be supported in future versions of MAPI.

The **ScCopyProps** function copies the properties defined by an array of **SPropValue** structures to a new destination.

At a Glance

Specified in header file:	MAPIUTIL.H
Implemented by:	MAPI
Called by:	Client applications and service providers

SCODE ScCopyProps(
  int *cprop*,
  LPSPropValue *rgprop*,
  LPVOID *pvDst*,
  ULONG FAR * *pcb*
 );

Parameters

cprop
  Input parameter containing the number of properties to be copied.

rgprop
  Input parameter pointing to an array of **SPropValue** structures that define the properties to be copied. The *rgprop* parameter need not point to the beginning of the array, but it must point to the beginning of one of the **SPropValue** structures in the array.

pvDst
  Input parameter pointing to the initial position in memory to which this function copies the properties.

pcb
  Optional output parameter pointing to a variable where the size, in bytes, of the block of memory pointed to by the *pvDst* parameter is stored.

Remarks

The new array and its data reside in a buffer created with a single allocation, and the **ScRelocProps** function can be used to adjust the pointers in the individual **SPropValue** structures. Prior to this adjustment, the pointers are valid.

ScCopyProps maintains the original property order for the copied property array.

The *pcb* parameter is optional; if it is not NULL, it is set to the number of bytes stored in the *pvDst* parameter.

See Also

ScDupPropset function

ScCountNotifications

This function may not be supported in future versions of MAPI.

The **ScCountNotifications** function determines the size, in bytes, of an array of event notifications, and validates the memory associated with the array.

At a Glance

Specified in header file:	MAPIUTIL.H
Implemented by:	MAPI
Called by:	Client applications and service providers

SCODE ScCountNotifications(
  int *cntf*,
  LPNOTIFICATION *rgntf*,
  ULONG FAR * *pcb*
 );

Parameters

cntf
  Input parameter containing the number of **NOTIFICATION** structures in the array indicated by the *rgntf* parameter.

rgntf
  Input parameter pointing to the array of **NOTIFICATION** structures whose size is to be determined.

pcb
  Optional output parameter pointing to a variable where the size, in bytes, of the array pointed to by the *rgntf* parameter is stored. If NULL, **ScCountNotifications** validates the array of notifications.

Remarks

If NULL is passed in the *pcb* parameter, the **ScCountNotifications** function only validates the array of notifications but no counting is done; if a non-null value is passed in *pcb*, **ScCountNotifications** determines the size of the array and stores the result in *pcb*. The *pcb* parameter must be large enough to contain the entire array.

ScCountProps

This function may not be supported in future versions of MAPI.

The **ScCountProps** function determines the size, in bytes, of a property value array and validates the memory associated with the array.

At a Glance

Specified in header file:	MAPIUTIL.H
Implemented by:	MAPI
Called by:	Client applications and service providers

SCODE ScCountProps(
  int *cprop*,
  LPSPropValue *rgprop*,
  ULONG FAR * *pcb*
 );

Parameters

cprop
  Input parameter containing the number of properties in the array indicated by the *rgprop* parameter.

rgprop
  Input parameter pointing to a range in an array of **SPropValue** structures that defines the properties whose size is to be determined. This range does not necessarily start at the beginning of the array.

pcb
  Optional output parameter pointing to a variable where the size, in bytes, of the property array is stored.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

MAPI_E_INVALID_PARAMETER
  Indicates that a property in the property value array has an identifier of PROP_ID_NULL or PROP_ID_INVALID or that the property array contains a multivalued property with no property values.

Remarks

If NULL is passed in the *pcb* parameter, the **ScCountProps** function validates the array of notifications but no counting is done. If a non-null value is passed in *pcb*, the **ScCountNotifications** function determines the size of the array and stores the result in *pcb*. The *pcb* parameter must be large enough to contain the entire array.

As it is counting, **ScCountProps** validates the memory associated with the array. **ScCountProps** only works with properties about which MAPI has information.

See Also

PropCopyMore function

ScCreateConversationIndex

The **ScCreateConversationIndex** function indicates where in a message thread a message belongs.

At a Glance

Specified in header file:	MAPIUTIL.H
Implemented by:	MAPI
Called by:	Client applications and service providers

SCODE ScCreateConversationIndex(
  ULONG *cbParent***,**
  LPBYTE *lpbParent***,**
  ULONG FAR* *lpcbIndex***,**
  LPBYTE FAR * *lppbIndex*
 );

Parameters

cbParent
  Indicates a count of bytes in the parent conversation index.

lpbParent
  Indicates a pointer to bytes in the parent conversation index. This may be NULL if *cbParent* is zero.

lpcbIndex
  Indicates a pointer to the count of bytes in the new conversation index returned by the call.

lppbIndex
  Indicates a pointer to a pointer to the new conversation index returned by the call.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

ScDupPropset

The **ScDupPropset** function duplicates a property value array in a single block of MAPI memory combining the operations of the **ScCopyProps** and **ScCountProps** functions.

At a Glance

Specified in header file:	MAPIUTIL.H
Implemented by:	MAPI
Called by:	Client applications and service providers

SCODE ScDupPropset(
  int *cprop*,
  LPSPropValue *rgprop*,
  LPALLOCATEBUFFER *lpAllocateBuffer*,
  LPSPropValue FAR * *prgprop*
 );

Parameters

cprop
  Input parameter containing the number of property values in the array indicated by the *rgprop* parameter.

rgprop
  Input parameter pointing to an array of **SPropValue** structures defining the property values to be duplicated.

lpAllocateBuffer
  Input parameter pointing to the **MAPIAllocateBuffer** function, to be used to allocate memory for the duplicated array.

prgprop
  Output parameter pointing to the initial position in memory where the returned duplicated array of **SPropValue** structures is stored.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

ScInitMapiUtil

The **ScInitMapiUtil** function replaces **MAPIInitialize** when only select utility functions are being used.

At a Glance

Specified in header file:	MAPIUTIL.H
Implemented by:	MAPI
Called by:	Client applications

SCODE ScInitMapiUtil(
  ULONG *ulFlags*
 );

Parameters

ulFlags
  Reserved; must be zero.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

Remarks

The **ScInitMapiUtil** and **DeinitMapiUtil** functions cooperate to call and release select utility functions, as opposed to **MAPIInitialize**, which calls core as well as utility functions. When **ScInitMapiUtil** calls utility functions, it also initializes the necessary memory.

When use of the functions that **ScInitMapiUtil** has called is complete, **DeinitMapiUtil** must be explicitly called to release them. In contrast, **MAPIInitialize** implicitly calls **DeinitMapiUtil**.

See Also

MapiUninitialize function

ScLocalPathFromUNC

This function may not be supported in future versions of MAPI.

The **ScLocalPathFromUNC** function locates a local path counterpart to the given UNC path.

At a Glance

Specified in header file:	MAPIUTIL.H
Implemented by:	MAPI
Called by:	Client applications and service providers

SCODE ScLocalPathFromUNC(
  LPSTR *szUNC*,
  LPSTR *szLocal*,
  UINT *cchLocal*
 );

Parameters

szUNC
  Input parameter containing a path in the format \\[*server*]\[*share*]\[*path*] of a file or directory.

szLocal
  Output parameter containing a path in the format [*drive:*]\[*path*] of the same file or directory as for the *szUNC* parameter.

cchLocal
  Input parameter containing the size of the buffer for the output string.

See Also

ScUNCFromLocalPath function

ScMAPIXFromCMC

The **ScMAPIXFromCMC** function enables a client application to pass in a CMC session handle and get back a pointer to a MAPI session object.

At a Glance

Specified in header file:	MAPI.H
Implemented by:	MAPI
Called by:	Client applications

SCODE ScMAPIXFromCMC(
  CMC_SESSION *cmcsession*,
  ULONG *ulFlags*,
  LPCIID *lpInterface*,
  LPMAPISESSION FAR * *lppMAPISession*
);

Parameters

cmcsession
  Input parameter pointing to the CMC session in use.

ulFlags
  Reserved; must be zero.

lpInterface
  Input parameter indicating which MAPI interface is being returned.

lppMAPISession
  Output parameter pointing to a variable where the pointer to the MAPI session in use is stored.

Return Values

S_OK
  The call succeeded and converted the CMC session to a MAPI session.

Remarks

There are no inverse functions for either the **ScMAPIXFromCMC** or **ScMAPIXFromSMAPI** function, that is, a client cannot convert to a CMC session or Simple MAPI session from a MAPI session. Also, a client cannot convert from a CMC session to a Simple MAPI session.

For more information on this function, see Using Multiple Client Interfaces and Explicit Logon with Simple MAPI.

ScMAPIXFromSMAPI

The **ScMAPIXFromSMAPI** function enables a client application to pass in a Simple MAPI session handle and get back a pointer to a MAPI session object.

At a Glance

Specified in header file:	MAPI.H
Implemented by:	MAPI
Called by:	Client applications

SCODE ScMAPIXFromSMAPI(
  LHANDLE *lhSimpleSession***,**
  ULONG *ulFlags***,**
  LPCIID *lpInterface***,**
  LPMAPISESSION FAR * *lppMAPISession*
 );

Parameters

lhSimpleSession
  Input parameter pointing to a Simple MAPI session.

ulFlags
  Reserved; must be zero.

lpInterface
  Input parameter indicating which MAPI interface is to be returned.

lppMAPISession
  Output parameter pointing to a variable where a pointer for a specific MAPI session is stored.

Return Values

S_OK
  The call succeeded and has converted the SMAPI session to a MAPI session.

Remarks

There are no inverse functions for either the **ScMAPIXFromCMC** or **ScMAPIXFromSMAPI** function, that is, a client cannot convert to a CMC session or Simple MAPI session from a MAPI session. Also, a client cannot convert from a Simple MAPI session to a CMC session.

For more information on using this function, see Using Multiple Client Interfaces and Explicit Logon with Simple MAPI.

ScRelocNotifications

This function may not be supported in future versions of MAPI.

The **ScRelocNotifications** function adjusts a pointer within a specified event notification array.

At a Glance

Specified in header file:	MAPIUTIL.H
Implemented by:	MAPI
Called by:	Client applications and service providers

SCODE ScRelocNotifications(
  int *cntf*,
  LPNOTIFICATION *rgntf*,
  LPVOID *pvBaseOld*,
  LPVOID *pvBaseNew*,
  ULONG FAR * *pcb*
 );

Parameters

cntf
  Input parameter containing the number of **NOTIFICATION** structures in the array indicated by the *rgntf* parameter.

rgntf
  Input parameter pointing to the array of **NOTIFICATION** structures defining event notifications within which a pointer is to be adjusted.

pvBaseOld
  Input parameter pointing to the original base address of the array indicated by the *rgntf* parameter.

pvBaseNew
  Input parameter containing the location to which **ScRelocNotifications** writes the new base address of the array indicated by the *rgntf* parameter.

pcb
  Output parameter pointing to a variable where the size, in bytes, of the array indicated by the *pvBaseNew* parameter is stored.

Remarks

The *pcb* parameter to the **ScRelocNotifications** function is optional.

See Also

ScRelocProps function

ScRelocProps

This function may not be supported in future versions of MAPI.

The **ScRelocProps** function adjusts the pointers in a **SPropValue** array after the array and its data have been copied or moved to a new location.

At a Glance

Specified in header file:	MAPIUTIL.H
Implemented by:	MAPI
Called by:	Client applications and service providers

SCODE ScRelocProps(
  int *cprop*,
  LPSPropValue *rgprop*,
  LPVOID *pvBaseOld*,
  LPVOID *pvBaseNew*,
  ULONG FAR * *pcb*
 );

Parameters

cprop
  Input parameter containing the number of properties in the array pointed to by the *rgprop* parameter.

rgprop
  Input parameter pointing to an array of **SPropValue** structures for which pointers are to be adjusted.

pvBaseOld
  Input parameter pointing to the original base address of the array pointed to by the *rgprop* parameter.

pvBaseNew
  Input parameter pointing to the new base address of the array pointed to by the *rgprop* parameter.

pcb
  Optional output parameter pointing to a variable where the size, in bytes, of the array indicated by the *pvBaseNew* parameter is stored. If not NULL, the *pcb* parameter is set to the number of bytes stored in the pvD parameter.

Remarks

The **ScRelocProps** function operates on the assumption that the property value array for which pointers are adjusted was originally allocated in a single call similar to a call to the **ScCopyProps** function. If a client application or service provider is working with a property value that is built from disjointed blocks of memory, it should use **ScCopyProps** to copy properties instead.

ScRelocProps is used to maintain the validity of pointers in an **SPropValue** array. To maintain pointers' validity when writing such an array to and reading it from a disk, perform the following operations:

1. Before writing the array and data to a disk, call **ScRelocProps** on the array with the *pvBaseNew* parameter pointing to some standard value (zero, for instance).
2. After reading the array and data from a disk, call **ScRelocProps** on the array with the *pvBaseOld* parameter equal to the same standard value used in Step 1. The array and data must be read into a buffer created with a single allocation.
3. The *pcb* parameter to **ScRelocProps** is optional.

See Also

MAPIAllocateBuffer function, **ScCountProps** function, **ScDupPropset** function, **SCRelocNotifications** function

ScUNCFromLocalPath

This function may not be supported in future versions of MAPI.

The **ScUNCFromLocalPath** function locates a UNC path counterpart to the given local path.

At a Glance

Specified in header file:	MAPIUTIL.H
Implemented by:	MAPI
Called by:	Client applications and service providers

SCODE ScUNCFromLocalPath(
  LPSTR *szLocal*,
  LPSTR *szUNC*,
  UINT *cchUNC*
 );

Parameters

szLocal
  Input parameter containing a path in the format [*drive:*]\[*path*] of a file or directory.

szUNC
  Output parameter containing a path in the format \\[*server*]\[*share*]\[*path*] of the same file or directory as for the *szLocal* parameter.

cchUNC
  Input parameter containing the size of the buffer for the output string.

See Also

ScLocalPathFromUNC function

SERVICEWIZARDDLGPROC

The **SERVICEWIZARDDLGPROC** prototype function defines a callback function invoked by the Profile Wizard to allow a provider to react to user events when the provider's property sheets or pages are shown.

At a Glance

Specified in header file:	MAPIWZ.H
Implemented by:	Service providers
Called by:	MAPI

BOOL SERVICEWIZARDDLGPROC(
  HWND *hDlg*,
  UINT *wMsgID*,
  WPARAM *wParam*,
  LPARAM *lParam*
 );

Parameters

hDlg
  Input parameter containing a window handle to the Profile Wizard dialog box.

wMsgID
  Input parameter indicating the message to be processed. In addition to all the regular Windows messages expected by a modal dialog box, the following messages can be received:

  WM_CLOSE
    Called when the Profile Wizard has completed. The provider should do any cleanup such as deallocating any dynamically allocated memory.

  WM_COMMAND
    Called for all of the provider's controls, and in addition, WM_COMMAND is called for the Next { ID_NEXT }, and Prev { ID_PREV } buttons. When called with ID_NEXT or ID_PREV, the provider is responsible for hiding the old page's controls and showing the controls for the next or previous page.

  WM_INITDIALOG
    Called even after the dialog box exists to allow the provider to initialize the controls that the Profile Wizard has added to the dialog box.

  WIZ_QUERYNUMPAGES
    Called to ask for the number of pages that the provider needs to display. The provider should return the number of pages instead of TRUE or FALSE. For example, use the following return statement to indicate that three pages should to be displayed:

```
return (BOOL)3;
```

wParam
  Input parameter whose contents depend on the message specified in the *wMsgID* parameter.

lParam
  Input parameter whose contents depend on the message specified in the *wMsgID* parameter.

Remarks

When the user chooses the Next button, the **SERVICEWIZARDDLGPROC** function is called with the WM_COMMAND message and ID_NEXT in the *wParam* parameter. The following steps describe what occurs between the time the user chooses Next and the time the first provider's configuration pages are rendered.

1. The Profile Wizard hides any controls that are on the window.
2. The Profile Wizard adds the provider's controls (hidden) to the page.
3. The Profile Wizard calls **SERVICEWIZARDDLGPROC**, sending the WM_INITDIALOG message, so that the provider can initialize the controls.
4. The Profile Wizard calls **SERVICEWIZARDDLGPROC**, sending the WIZ_QUERYNUMPAGES message. The provider returns the number of pages that it will be showing.
5. The Profile Wizard calls **SERVICEWIZARDDLGPROC**, sending the WM_COMMAND message with the *wParam* parameter set to either ID_NEXT or ID_PREV. At this point, the provider either returns FALSE {error} or reveals its controls and returns TRUE {success}. If the Profile Wizard passes in ID_NEXT, the provider's first page is displayed. If ID_PREV is passed in, the last page is displayed.
6. The Profile Wizard calls the provider's **SERVICEWIZARDDLGPROC** function, sending the WM_COMMAND message with the *wParam* parameter set to either ID_NEXT or ID_PREV (depending on which button the user chose). The provider is responsible for showing or hiding its controls and writing its data to the **IMAPIProp** passed to the profile wizard to step through its sequence of pages. The provider should return TRUE if the next or previous page was successfully shown, and FALSE if neither the next nor previous page could be shown. The provider needs to be aware of when it is stepping outside of its sequence of pages, and respond appropriately by hiding its controls and writing its data to the profile.
7. If the user steps outside the provider's range of pages, the Profile Wizard deletes the provider's hidden controls from the dialog box and calls the next provider (or displays its next page if that was the last provider).

The value returned by **SERVICEWIZARDDLGPROC** is dependent on the message type sent. The recommended practice is to return TRUE if the provider processes the message and FALSE if the provider does not process the message.

SetAttribIMsgOnIStg

The **SetAttribIMsgOnIStg** function sets property attributes for properties of a particular object.

At a Glance

Specified in header file:	IMESSAGE.H
Implemented by:	MAPI
Called by:	Client applications and message store providers

HRESULT SetAttribIMsgOnIStg(
  LPVOID *lpObject***,**
  LPSPropTagArray *lpPropTags***,**
  LPSPropAttrArray *lpPropAttrs***,**
  LPSPropProblemArray FAR * *lppPropProblems*
);

Parameters

lpObject
  Input parameter pointing to the object for which property attributes are being set.

lpPropTags
  Input parameter pointing to an **SPropTagArray** structure containing an array of property tags indicating the properties for which property attributes are being set.

lpPropAttrs
  Input parameter pointing to an **SPropAttrArray** structure listing the property attributes to set.

lppPropProblems
  Output parameter pointing to a variable where the returned **SPropProblemArray** structure containing a set of property problems is stored. This structure identifies problems encountered if **SetAttribIMsgOnIStg** has been able to set some properties, but not all. If a pointer to NULL is passed in the *lppPropProblems* parameter, no property problem array is returned even if some properties were not set.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

MAPI_W_ERRORS_RETURNED
  The call succeeded overall, but one or more properties could not be accessed and were returned with a property type of PT_ERROR.

Remarks

A message store provider usually calls the **SetAttribIMsgOnIStg** function.

In the *lpPropAttrs* parameter, the number and position of the attributes must match the number and position of the property tags passed in the *lpPropTags* parameter.

The **SetAttribIMsgOnIStg** function is used to make message properties read-only when required by the **IMessage** schema. The sample message store provider uses it for this purpose. For more information, see Messages.

See Also

GetAttribIMsgOnIStg function

SzFindCh

This function may not be supported in future versions of MAPI.

The **SzFindCh** function searches for the first occurrence of a character in a string.

At a Glance

Specified in header file:	MAPIUTIL.H
Implemented by:	MAPI
Called by:	Client applications and service providers

SzFindCh (LPCTSTR *sz*, **USHORT** *ch***)**

Parameters

sz
Input parameter specifying a pointer to the string to be searched.

ch
Input parameter specifying the character for which this function searches.

Return Values

If successful, this function returns a pointer to the first occurrence of the character in the string. It returns a value of NULL otherwise.

Remarks

The **SzFindCh** function searches for an exact match only; it is sensitive to case and diacritical differences.

SzFindLastCh

This function may not be supported in future versions of MAPI.

The **SzFindLastCh** function searches for the last occurrence of a character in a string.

At a Glance

Specified in header file:	MAPIUTIL.H
Implemented by:	MAPI
Called by:	Client applications and service providers

SzFindLastCh (LPCTSTR *sz*, **USHORT** *ch***)**

Parameters

sz

Input parameter specifying a pointer to the string to be searched.

ch

Input parameter specifying the character for which this function searches.

Return Values

If successful, this function returns a pointer to the last occurrence of the character in the string. It returns a value of NULL otherwise.

Remarks

The **SzFindLastCh** function searches for an exact match only; it is sensitive to case and diacritical differences.

SzFindSz

This function may not be supported in future versions of MAPI.

The **SzFindSz** function searches for the first occurrence of a substring in a string.

At a Glance

Specified in header file:	MAPIUTIL.H
Implemented by:	MAPI
Called by:	Client applications and service providers

SzFindSz (LPCTSTR *sz*, **LPCTSTR** *szKey***)**

Parameters

sz
Input parameter specifying a pointer to the string to be searched.

szKey
Input parameter specifying the substring for which this function searches.

Remarks

This function searches for an exact match only; it is sensitive to case and diacritical differences.

UFromSz

This function may not be supported in future versions of MAPI.

The **UFromSz** function obtains an unsigned binary value from a string of ASCII digits.

At a Glance

Specified in header file:	MAPIUTIL.H
Implemented by:	MAPI
Called by:	Client applications and service providers

UFromSz (LPCTSTR *sz***)**

Parameters

sz

Input parameter specifying a pointer to an ASCII string.

Remarks

The **UFromSz** function stops interpreting digits when it reaches the first non numeric character.

This function returns an unsigned decimal integer value representing the ASCII string. For example, given the string 55. **UFromSz** returns the number 55. Given a string such as 5ab, the function returns the number five. Given a string that does not include a digit, such as ab, **UFromSz** returns zero.

UlFromSzHex

This function may not be supported in future versions of MAPI.

The **UlFromSzHex** function converts a string of ASCII hexadecimal values into an unsigned long integer.

At a Glance

Specified in header file:	MAPIUTIL.H
Implemented by:	MAPI
Called by:	Client applications and service providers

UlFromSzHex (LPCTSTR *sz***)**

Parameters

sz

Input parameter specifying a pointer to an ASCII string.

Return Values

This function returns an unsigned long integer representing the string of ASCII hexadecimal values.

Remarks

The **UlFromSzHex** function stops converting at the first character in the string that is not a hexadecimal digit.

UlAddRef

This function may not be supported in future versions of MAPI.

The **UlAddRef** function provides an alternative way to invoke the OLE method **IUnknown::AddRef**.

At a Glance

Specified in header file:	MAPIUTIL.H
Implemented by:	MAPI
Called by:	Client applications and service providers

ULONG UlAddRef(
  LPVOID *punk*
 );

Parameters

punk
  Input parameter pointing to an interface derived from the **IUnknown** interface, in other words any MAPI interface.

Remarks

The **UlAddRef** function generates less code than the **AddRef** method and can be used in situations requiring a minimum of code.

UlAddRef returns the value returned by the **IUnknown::AddRef** method, which is the new value of the reference count for the interface. The value is nonzero.

For more information on **IUnknown::AddRef**, see Implementing the IUnknown Interface.

UlPropSize

This function may not be supported in future versions of MAPI.

The **UlPropSize** function obtains the size of a single property value.

At a Glance

Specified in header file:	MAPIUTIL.H
Implemented by:	MAPI
Called by:	Client applications and service providers

ULONG UlPropSize(
  LPSPropValue *lpSPropValue*
 );

Parameters

lpSPropValue
  Input parameter pointing to an **SPropValue** structure defining the property to be measured.

Remarks

The **UlPropSize** function returns the size, in bytes, of the property value for the specified property. It disregards the size of the remainder of the **SPropValue** structure.

UlRelease

This function may not be supported in future versions of MAPI.

The **UlRelease** function provides an alternative way to invoke the OLE method **IUnknown::Release**.

At a Glance

Specified in header file:	MAPIUTIL.H
Implemented by:	MAPI
Called by:	Client applications and service providers

ULONG UlRelease(
  LPVOID *punk*
 );

Parameters

punk
  Input parameter pointing to an interface derived from the **IUnknown** interface, in other words any MAPI interface.

Remarks

The reference count is the number of existing pointers to the object to be released.

If the *punk* parameter is NULL, the function returns immediately without calling **IUnknown::Release**. This feature can save code if an application or provider declares an interface pointer and initializes it to NULL. Code is used in the method indication but saved in not testing for NULL.

UlRelease returns the value returned by the **IUnknown::Release** method, which can be equal to the reference count for the object to be released.

For more information on **IUnknown::Release**, see Implementing the IUnknown Interface.

UIValidateParameters

The **UIValidateParameters** macro calls an internal function to check the parameters client applications have passed to service providers and MAPI.

At a Glance

Specified in header file:	MAPIVAL.H
Implemented by:	MAPI
Called by:	Service providers

HRESULT UIValidateParameters(
  METHODS *eMethod***,**
  LPVOID *First*
 );

Parameters

eMethod
  (Input) Specifies, by enumeration, the method to validate.

First
  (Input) Pointer to the first argument on the stack.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

MAPI_E_CALL_FAILED
  An error of unexpected or unknown origin prevented the operation from completing.

Remarks

The **UIValidateParameters** macro has been superseded by the **UIValidateParms** macro. **UIValidateParameters** does not work correctly on RISC platforms and is now prevented from compiling on them. It still compiles and works correctly on Intel platforms, but **UIValidateParms** is recommended on all platforms.

UlValidateParms

The **UlValidateParms** macro calls an internal function to check the parameters client applications have passed to service providers and MAPI.

At a Glance

Specified in header file:	MAPIVAL.H
Implemented by:	MAPI
Called by:	Service providers

HRESULT UlValidateParms(
  METHODS *eMethod***,**
  LPVOID *First*
);

Parameters

eMethod
  (Input) Specifies, by enumeration, the method to validate.

First
  (Input) Pointer to the first argument on the stack.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

MAPI_E_CALL_FAILED
  An error of unexpected or unknown origin prevented the operation from completing.

Remarks

Parameters passed between MAPI and service providers are assumed to be correct and undergo only debug validation with the **CheckParms** macro. Providers should check all parameters passed in by client applications, but clients should assume that MAPI and provider parameters are correct. Use the **HR_FAILED** macro to test return values.

The **UlValidateParms** macro is called differently depending on whether the calling code is C or C++. This macro is used to validate parameters for the few **IUnknown** and MAPI methods that return ULONG rather than HRESULT values; the **ValidateParms** macro works for all others.

For more information on parameter validation, see Validating Parameters to Interface Methods.

ValidateParameters

The **ValidateParameters** macro calls an internal function to check the parameters client applications have passed to service providers.

At a Glance

Specified in header file:	MAPIVAL.H
Implemented by:	MAPI
Called by:	Service providers

HRESULT ValidateParameters(
  METHODS *eMethod***,**
  LPVOID *First*
 );

Parameters

eMethod
  (Input) Specifies, by enumeration, the method to validate.

First
  (Input) Pointer to the first argument on the stack.

Return Values

S_OK
  All of the parameters are valid.

MAPI_E_CALL_FAILED
  One or more of the parameters are not valid.

Remarks

The **ValidateParameters** macro has been superseded by the **ValidateParms** macro. **ValidateParameters** does not work correctly on RISC platforms and is now prevented from compiling on them. It still compiles and works correctly on Intel platforms, but **ValidateParms** is recommended on all platforms.

ValidateParms

The **ValidateParms** macro calls an internal function to check the parameters client applications have passed to service providers.

At a Glance

Specified in header file:	MAPIVAL.H
Implemented by:	MAPI
Called by:	Service providers

HRESULT ValidateParms(
  METHODS *eMethod***,**
  LPVOID *First*
 );

Parameters

eMethod
  (Input) Specifies, by enumeration, the method to validate.

First
  (Input) Pointer to the first argument on the stack.

Return Values

S_OK
  All of the parameters are valid.

MAPI_E_CALL_FAILED
  One or more of the parameters are not valid.

Remarks

Parameters passed between MAPI and service providers are assumed to be correct and undergo only debug validation with the **CheckParms** macro. Providers should check all parameters passed in by client applications, but clients should assume that MAPI and provider parameters are correct. Use the **HR_FAILED** macro to test return values.

ValidateParms is called differently depending on whether the calling code is C or C++. C++ passes an implicit parameter known as *this* to each method call, which becomes explicit in C and is the address of the object. The first parameter, *eMethod*, is an enumerator made from the interface and method being validated and tells what parameters to expect to find on the stack. The second parameter is different for C and C++. In C++ it is called *First*, and it is the first parameter to the method being validated. The second parameter for the C language, *ppThis*, is the address of the first parameter to the method which is always an object pointer. In both cases, the second parameter gives the address of the beginning of the method's parameter list, and based on *eMethod*, moves down the stack and validates the parameters.

Providers implementing common interfaces such as IMAPITable and IMAPIProp should always check parameters using the ValidateParms function in order to ensure consistency across all providers. Additional parameter validation functions have been defined for some complex parameter types to be used instead as appropriate. See the reference entries for the following functions: FBadColumnSet, FBadEntryList, FBadProp, FBadPropTag, FBadRestriction, FBadRglpNameID, FBadRglpszW, FBadRow, FBadRowSet, and FBadSortOrderSet.

Inherited methods use the same parameter validation as the interface from which they inherit. For example, the parameter checking for **IMessage** and **IMAPIProp** should be the same.

For more information on parameter validation, see Validating Parameters to Interface Methods.

See Also

UIValidateParms

WIZARDENTRY

The **WIZARDENTRY** function prototype defines a service provider entry point for the Profile Wizard. The Profile Wizard calls this entry point function to retrieve enough information to display the provider's configuration property sheets.

At a Glance

Specified in header file:	MAPIWZ.H
Implemented by:	Service providers
Called by:	MAPI

ULONG WIZARDENTRY(
  HINSTANCE *hProviderDLLInstance*,
  LPTSTR FAR * *lpcsResourceName*,
  DLGPROC FAR * *lppDlgProc*,
  LPMAPIPROP *lpMAPIProp*,
  LPVOID *lpMapiSupportObject*
 );

Parameters

hProviderDLLInstance
  Input parameter containing the instance handle of the provider's DLL.

lpcsResourceName
  Output parameter pointing to a string containing the full name of the dialog resource that should be displayed by the Profile Wizard during configuration. The maximum size of the string, including the NULL terminator, is 32 characters.

lppDlgProc
  Output parameter pointing to a standard Windows dialog box procedure that will be called by the Profile Wizard to notify the provider of various events.

lpMapiProp
  Input parameter pointing to a property interface implementation that provides access to the configuration properties. When the wizard is finished configuring all providers, it writes the properties to the profile by calling **IMsgServiceAdmin::ConfigureMsgService**.

lpMapiSupportObject
  Input parameter pointing to a MAPI support object.

Return Values

S_OK
  The provider's WIZARDENTRY function was called successfully.

MAPI_E_CALL_FAILED
  An error of unexpected or unknown origin prevented the operation from completing.

Remarks

The Profile Wizard calls the **WIZARDENTRY** function when it is ready to display the provider's configuration user interface. The pointer to the property interface implementation should be stored by the provider for future reference. This pointer allows the provider access to the profile. During configuration, providers should add their configuration properties to this object. After all providers have been configured, the Profile Wizard adds these properties to the profile.

The name of the **WIZARDENTRY** function must be placed in the PR_SERVICE_WIZARD_ENTRY_NAME property in MAPISVC.INF.

The resource name is the same as the dialog resource that will be rendered in the pane of the Profile Wizard. The resource that is passed back needs to contain all the pages (in a single dialog resource) that should be displayed by the Profile Wizard. When the Profile Wizard receives this resource, it ignores the dialog style (but not the control styles), and creates all the controls as children of the Profile Wizard page. All controls are initially hidden. Providers should ensure that the coordinates for their controls are zero or zero-based, and that they don't exceed a maximum width of 200 dialog units and a maximum height of 150 dialog units. Control identifiers below 400 are reserved for the Profile Wizard. The Profile Wizard displays the provider's title in bold text above the provider's user interface.

For more information on using this function, see About Profile Wizard Entry Point Functions.

WrapCompressedRTFStream

The **WrapCompressedRTFStream** function creates a text stream in uncompressed Rich Text Format (RTF) from the compressed format used in the PR_RTF_COMPRESSED property.

At a Glance

Specified in header file:	MAPIUTIL.H
Implemented by:	MAPI
Called by:	Client applications

HRESULT WrapCompressedRTFStream(
  LPSTREAM *lpCompressedRTFStream***,**
  ULONG *ulflags***,**
  LPSTREAM FAR * *lpUncompressedRTFStream*
 );

Parameters

lpCompressedRTFStream
  Input parameter pointing to a stream opened on the PR_RTF_COMPRESSED property of a message.

ulFlags
  Input parameter specifying a bitmask of option flags for the function. The following flag can be set:

  MAPI_MODIFY
    Indicates whether the client intends to read or write the wrapped stream interface that is returned.

  STORE_UNCOMPRESSED_RTF
    Indicates that uncompressed RTF should be written to the stream pointed to by *lpCompressedRTFStream*

lpUncompressedRTFStream
  Output parameter pointing to the location where **WrapCompressedRTFStream** returns a stream for the uncompressed RTF.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

Remarks

If the MAPI_MODIFY flag is passed in the *ulFlags* parameter, the *lpCompressedRTFStream* parameter must already be open for reading and writing. New, uncompressed RTF text should be written into the stream interface returned in *lpUncompressedRTFStream*. Because it is not possible to append the existing stream, the entire message text must be written.

If zero is passed in the *ulFlags* parameter, then *lpCompressedRTFStream* may be opened read-only. Only the entire message text can be read out of the stream interface returned in *lpUncompressedRTFStream;* it is not possible to search into the middle of the stream and begin reading.

The **WrapCompressedRTFStream** assumes that the compressed stream's pointer is set to the beginning of the stream. Certain OLE **IStream** methods are not supported by the returned uncompressed stream. These include **IStream::Clone**, **IStream::LockRegion**, **IStream::Revert**, **IStream::Seek**, **IStream::SetSize**, **IStream::Stat**, and **IStream::UnlockRegion**. In order to copy to the entire stream, a read/write loop is needed.

Since the client writes new RTF in uncompressed format, it should use **WrapCompressedRTFStream**, rather than directly writing to the stream. RTF-aware clients should search for the STORE_UNCOMPRESSED_RTF flag in the PR_STORE_SUPPORT_MASK property and pass it to **WrapCompressed RTFStream** if it is set.

See Also

RTFSync function

WrapStoreEntryID

The **WrapStoreEntryID** function converts a message store's internal entry identifier to an entry identifier more useful to the messaging system.

At a Glance

Specified in header file:	MAPIUTIL.H
Implemented by:	MAPI
Called by:	Client applications and service providers

WrapStoreEntryID(
  ULONG *ulFlags*,
  LPTSTR *szDLLName*,
  ULONG *cbOrigEntry*,
  LPENTRYID *lpOrigEntry*,
  ULONG * *lpcbWrappedEntry*,
  LPENTRYID * *lppWrappedEntry*
);

Parameters

ulFlags
Input parameter containing a bitmask of flags. The following flag can be set:

MAPI_UNICODE
Indicates the strings are in Unicode format. If the MAPI_UNICODE flag is not set, the strings are in ANSI format.

szDLLName
Input parameter containing the name of the message store provider DLL.

cbOrigEntry
Input parameter containing the size, in bytes, of the original entry identifier for the message store.

lpOrigEntry
Input parameter pointing to an **ENTRYID** structure defining the original entry identifier.

lpcbWrappedEntry
Output parameter pointing to the size, in bytes, of the new entry identifier.

lppWrappedEntry
Output parameter pointing to a pointer to an **ENTRYID** structure containing the new entry identifier.

Remarks

A message store object retains an internal entry identifier which is meaningful only to service providers. If a client application asks a provider for the PR_STORE_ENTRYID property on a message store, the provider must call **WrapStoreEntryID** to generate a form of the entry identifier that is usable by clients and by components in other messaging domains.

See Also

IMAPISession::OpenEntry interface

XPProviderInit

The **XPProviderInit** function initializes a transport provider for operation.

At a Glance

Specified in header file:	MAPISPI.H
Implemented by:	Transport providers
Called by:	MAPI

HRESULT XPProviderInit(
  HINSTANCE *hInstance*,
  LPMALLOC *lpMalloc*,
  LPALLOCATEBUFFER *lpAllocateBuffer*,
  LPALLOCATEMORE *lpAllocateMore*,
  LPFREEBUFFER *lpFreeBuffer*,
  ULONG *ulFlags*,
  ULONG *ulMAPIVer*,
  ULONG FAR * *lpulProviderVer*,
  LPXPPROVIDER FAR * *lppXPProvider*
 );

Parameters

hInstance
  Input parameter containing an instance of the transport provider's dynamic-link library (DLL) that MAPI used when it linked.

lpMalloc
  Input parameter pointing to a memory allocator object exposing the OLE **IMalloc** interface. The transport provider may need to use this allocation method when working with certain interfaces such as **IStream**.

lpAllocateBuffer
  Input parameter pointing to the **MAPIAllocateBuffer** function, to be used to allocate memory.

lpAllocateMore
  Input parameter pointing to the **MAPIAllocateMore** function, to be used to allocate additional memory where required.

lpFreeBuffer
  Input parameter pointing to the **MAPIFreeBuffer** function, to be used to free memory.

ulFlags
  Input parameter containing a bitmask of flags. The following flag can be set:

  MAPI_NT_SERVICE
    Indicates the provider is being loaded in the context of a Windows NT service, a special type of process without access to any user interface.

ulMAPIVer
  Input parameter containing the version number of the service provider interface that MAPI.DLL uses. For the current version number, see the MAPISPI.H header file.

lpulProviderVer
  Output parameter pointing to the version number of the service provider interface that this transport provider uses.

lppXPProvider
  Output parameter pointing to a pointer to the initialized transport provider object.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

Remarks

To initialize a transport provider, MAPI calls the function named **XPProviderInit**, based on the **XPPROVIDERINIT** function prototype defined in MAPISPI.H, from the transport provider's DLL. The transport provider must use its implementation of **XPProviderInit** to respond to the MAPI initialization call.

The transport provider must also define the **XPProviderInit** function using the CDECL calling convention. CDECL definition is required for each service provider initialization function to ensure the function can work with the current version of the service provider interface, even if the number of function parameters used is not the number set for that function in the current version of the interface. MAPI provides the **XPPROVIDERINIT** prototype to help define **XPProviderInit** as CDECL. The **XPPROVIDERINIT** prototype has a standard MAPI initialization call type, STDMAPIINITCALLTYPE.

The *lpAllocateBuffer*, *lpAllocateMore*, and *lpFreeBuffer* input parameters point to the **MAPIAllocateBuffer**, **MAPIAllocateMore**, and **MAPIFreeBuffer** functions, respectively, for use by the transport provider DLL. The provider should use these pointers for memory allocation or deallocation. In particular, the provider must use these pointers when allocating memory for use by client applications in calling object interfaces. For example, two provider interface methods that typically allocate memory are **IMAPIProp::GetProps** and **IMAPITable::QueryRows**.

The transport provider should retain information on the allocator pointers passed to it in *lpAllocateBuffer*, *lpAllocateMore*, and *lpFreeBuffer*. If the provider will use a memory allocator later, it should call the **IUnknown::AddRef** method for the allocation object pointed to by the *lpMalloc* parameter.

For more information, see Initializing the Transport Provider. For more information on entry point functions, see About Provider DLL Entry Point Functions.

See Also

ABProviderInit function, **HPProviderInit** function, **IXPProvider : IUnknown** interface, **MSProviderInit** function

MAPI Properties

For a general explanation of how MAPI works with properties and a list of the properties available on each object, see Properties. For information on the property-related macros, see the **SPropValue** structure.

Unlike members of OLE variant arrays, every member in a MAPI multivalued property array is of the same type.

For a comprehensive list of MAPI properties and their mappings to X.400 attributes, see Mapping of X.400 P2 Attributes to MAPI Properties.

PR_7BIT_DISPLAY_NAME

The PR_7BIT_DISPLAY_NAME property contains a 7-bit ASCII version of a messaging user's name.

Usage

Optional on messaging user objects.

Details

Identifier 0x39FF; property type PT_STRING8; property tag 0x39FF001E

Remarks

The PR_7BIT_DISPLAY_NAME property can be considered a mapping of the PR_DISPLAY_NAME property into a 7-bit character set. Some messaging systems, such as Internet and certain X.400 links, are limited to the 128-character 7-bit ASCII code set. Gateways to such messaging systems can improve their performance by calling the **IAddrBook::PrepareRecips** method directly to retrieve the PR_7BIT_DISPLAY_NAME property to avoid extra processing for code conversion.

PR_AB_DEFAULT_DIR

The PR_AB_DEFAULT_DIR property contains the entry identifier of the address book container that is to be shown to the user first.

Usage

Reserved.

Details

Identifier 0x3D06; property type PT_BINARY; property tag 0x3D060102

Remarks

Do not use this property.

PR_AB_DEFAULT_PAB

The PR_AB_DEFAULT_PAB property contains the entry identifier of the address book container that is to be used as the personal address book.

Usage

Reserved.

Details

Identifier 0x3D07; property type PT_BINARY; property tag 0x3D070102

Remarks

Do not use this property.

PR_AB_PROVIDER_ID

The PR_AB_PROVIDER_ID property contains an address book provider's unique identifier.

Usage

Required as a column entry in hierarchy tables.
Computed by address book providers on address book container objects.

Details

Identifier 0x3615; property type PT_BINARY; property tag 0x36150102

Remarks

The **MAPIUID** structure identifies which address book provider supplies this particular container in the container hierarchy. The value is unique to each provider.

An address book provider can provide more than one identifier. For example, a provider that supplies two very different containers can publish in PR_AB_PROVIDER_ID unique identifiers for each container.

PR_AB_PROVIDER_ID is analogous to the PR_MDB_PROVIDER property for message stores. Client applications can use PR_AB_PROVIDER_ID to find related rows in an address book hierarchy table.

PR_AB_PROVIDERS

The PR_AB_PROVIDERS property contains a list of identifiers for address book providers in the current profile.

Usage

Reserved.

Details

Identifier 0x3D01; property type PT_BINARY; property tag 0x3D010102

Remarks

Do not use this property.

PR_AB_SEARCH_PATH

The PR_AB_SEARCH_PATH property contains a list of entry identifiers for address book containers that are to be searched to resolve names.

Usage

Reserved.

Details

Identifier 0x3D05; property type PT_MV_BINARY; property tag 0x3D051102

Remarks

Do not use this property.

PR_AB_SEARCH_PATH_UPDATE

The PR_AB_SEARCH_PATH_UPDATE property contains a list of entry identifiers for address book containers explicitly configured by the user.

Usage

Reserved.

Details

Identifier 0x3D11; property type PT_BINARY; property tag 0x3D110102

Remarks

Do not use this property.

PR_ACCESS

The PR_ACCESS property contains a bitmask of flags indicating the operations a client application can perform on the open object.

Usage

Required on folder and message objects.

Details

Identifier 0x0FF4; property type PT_LONG; property tag 0x0FF40003

Remarks

Zero or more of the following flags can be set for the PR_ACCESS property:

MAPI_ACCESS_CREATE_ASSOCIATED
: The client can create an associated contents table.

MAPI_ACCESS_CREATE_CONTENTS
: The client can create a contents table.

MAPI_ACCESS_CREATE_HIERARCHY
: The client can create a hierarchy table.

MAPI_ACCESS_DELETE
: The client can delete the object.

MAPI_ACCESS_MODIFY
: The client can write to the object.

MAPI_ACCESS_READ
: The client can read the object.

The MAPI_ACCESS_DELETE, MAPI_ACCESS_MODIFY, and MAPI_ACCESS_READ flags are found on folder and message objects and in the PR_ACCESS column in contents tables and associated contents tables. The MAPI_ACCESS_CREATE_ASSOCIATED, MAPI_ACCESS_CREATE_CONTENTS, and MAPI_ACCESS_CREATE_HIERARCHY flags are found on folder objects only.

See Also

PR_ACCESS_LEVEL property

PR_ACCESS_LEVEL

The PR_ACCESS_LEVEL property contains a bitmask of flags indicating the level at which a client application can access the open object.

Usage

Required on address book container, distribution list, folder, messaging user, message, and message store objects.

Details

Identifier 0x0FF7; property type PT_LONG; property tag 0x0FF70003

Remarks

The PR_ACCESS_LEVEL property is used to determine whether or not read/write access was granted. Calls to the **IMsgStore::OpenEntry** method can request read/write access. By default, access is read-only, although the access granted is defined by the service provider. A provider can grant read/write access when the client requested read-only access.

The following flag can be set for PR_ACCESS_LEVEL:

MAPI_MODIFY
    Read/write access was granted.

See Also

PR_ACCESS property

PR_ACCOUNT

The PR_ACCOUNT property contains the recipient's account name.

Usage

Optional on messaging user objects.

Details

Identifier 0x3A00; property type PT_TSTRING; property tag 0x3A00001E (0x3A00001F for Unicode)

Remarks

The PR_ACCOUNT property is one of the properties that provide identification and access information for a recipient. These properties are defined by the recipient and the recipient's organization. For more information about this group of properties, see About Messaging User Objects.

PR_ACCOUNT commonly contains the recipient's e-mail name, that is, the final component of the e-mail address, which uniquely identifies the recipient within the local organization. The e-mail name corresponds to the X.400 distinguished name, which is meant to be a short name guaranteed to be unique within a certain messaging domain.

PR_ACKNOWLEDGEMENT_MODE

The PR_ACKNOWLEDGEMENT_MODE property contains the identifier of the mode for message acknowledgment.

Usage

Optional on message objects.

Details

Identifier 0x0001; property type PT_LONG; property tag 0x00010003

Remarks

The PR_ACKNOWLEDGEMENT_MODE property can have exactly one of the following values:

Value	Description
0	Manual acknowledgment.
1	Automatic acknowledgment.

PR_ACKNOWLEDGEMENT_MODE corresponds to the X.400 attribute IM_ACKNOWLEDGEMENT_MODE.

See Also

PR_MESSAGE_CLASS property

PR_ADDRTYPE

The PR_ADDRTYPE property contains the messaging user's e-mail address type, such as SMTP.

Usage

Required on distribution list and messaging user objects.

Details

Identifier 0x3002; property type PT_TSTRING; property tag 0x3002001E (0x3002001F for Unicode)

Remarks

The PR_ADDRTYPE property is one of the base address properties common to all messaging users. It specifies which messaging system MAPI uses to handle a given message.

PR_ADDRTYPE also determines the format of the address string in the PR_EMAIL_ADDRESS property. The string provided by PR_ADDRTYPE can contain only the uppercase alphabetic characters from A through Z and the numbers from 0 through 9.

Valid PR_ADDRTYPE examples include:

```
FAX
MHS
PROFS
SMTP
X400
```

For more information on the base address properties, see About Base Address Properties. For more information on address types, see Address Types.

PR_ALTERNATE_RECIPIENT

The PR_ALTERNATE_RECIPIENT property contains a list of entry identifiers for alternate recipients designated by the originally intended recipient.

Usage

Optional on recipient subobjects.

Details

Identifier 0x3A01; property type PT_BINARY; property tag 0x3A010102

Remarks

The PR_ALTERNATE_RECIPIENT property is used in autoforwarded messages. It contains a **FLATENTRYLIST** structure of alternate recipients. If autoforwarding is not permitted or if no alternate recipient has been designated, a nondelivery report should be generated.

PR_ALTERNATE_RECIPIENT corresponds to the X.400 attribute MH_T_ALTERNATE_RECIPIENT_NAME.

PR_ALTERNATE_RECIPIENT_ALLOWED

The PR_ALTERNATE_RECIPIENT_ALLOWED property contains TRUE if the sender permits autoforwarding of this message.

Usage

Optional on message objects.

Details

Identifier 0x0002; property type PT_BOOLEAN; property tag 0x0002000B

Remarks

If autoforwarding is not permitted or if no alternate recipient has been designated, a nondelivery report should be generated.

The PR_ALTERNATE_RECIPIENT_ALLOWED property corresponds to the X.400 attribute MH_T_ALTERNATE_RECIPIENT_ALLOWED.

PR_ANR

The PR_ANR property contains a string value for use in a property restriction on an address book container contents table.

Details

Identifier 0x360C; property type PT_TSTRING; property tag 0x360C001E (0x360C001F for Unicode)

Remarks

The PR_ANR property does not belong to any object; it is furnished by address book providers in **SPropertyRestriction** structures. This property contains an ambiguous name resolution (ANR) string that can be tested against an address book container's contents table to find corresponding message recipients.

Address book providers match the value of PR_ANR against every entry in the contents table, using a provider-defined matching algorithm. The column or columns used in this match are chosen by the provider as part of the algorithm. The PR_DISPLAY_NAME column is the most commonly used; the PR_ACCOUNT column is also useful when it contains the user's e-mail name.

For more information on ambiguous name resolution, see About Address Book Restrictions.

See Also

IAddrBook::ResolveName method, **IABContainer::ResolveNames** method

PR_ASSISTANT

The PR_ASSISTANT property contains the name of the recipient's administrative assistant.

Usage

Optional on messaging user objects.

Details

Identifier 0x3A30; property type PT_TSTRING; property tag 0x3A30001E (0x3A30001F for Unicode)

Remarks

The PR_ASSISTANT property is one of the properties that provide identification and access information for a recipient. These properties are defined by the recipient and the recipient's organization. For more information on this group of properties, see About Messaging User Objects.

See Also

PR_ASSISTANT_TELEPHONE_NUMBER property

PR_ASSISTANT_TELEPHONE_NUMBER

The PR_ASSISTANT_TELEPHONE_NUMBER property contains the telephone number of the recipient's administrative assistant.

Usage

Optional on messaging user objects.

Details

Identifier 0x3A2E; property type PT_TSTRING; property tag 0x3A2E001E (0x3A2E001F for Unicode)

Remarks

The PR_ASSISTANT_TELEPHONE_NUMBER property is one of the properties that provide identification and access information for a recipient. These properties are defined by the recipient and the recipient's organization. For more information on this group of properties, see About Messaging User Objects.

The telephone number is for the assistant specified in the PR_ASSISTANT property.

PR_ASSOC_CONTENT_COUNT

The PR_ASSOC_CONTENT_COUNT property contains the count of items in the associated contents table of the folder.

Usage

Required on folder objects.

Details

Identifier 0x3617; property type PT_LONG; property tag 0x36170003

Remarks

The PR_FOLDER_ASSOCIATED_CONTENTS property contains the associated contents table of the folder.

See Also

PR_ACCESS property

PR_ATTACH_DATA_BIN

The PR_ATTACH_DATA_BIN property contains binary attachment data typically accessed through the OLE **IStream** interface.

Usage

Optional on attachment subobjects.

Details

Identifier 0x3701; property type PT_BINARY; property tag 0x37010102

Remarks

The PR_ATTACH_DATA_BIN property holds the attachment when the value of the PR_ATTACH_METHOD property is ATTACH_BY_VALUE, which is the usual attachment method and the only one required to be supported. PR_ATTACH_DATA_BIN also holds an OLE 1.0 **OLESTREAM** attachment when the value of PR_ATTACH_METHOD is ATTACH_OLE.

OLESTREAM attachments can be copied into a file by calling the OLE **IStream::CopyTo** method. The OLE encoding type can be determined from the PR_ATTACH_TAG property.

For an OLE document file attachment, the message store provider must respond to an **IMAPIProp::OpenProperty** call on PR_ATTACH_DATA_OBJ and may optionally respond to a call on PR_ATTACH_DATA_BIN. Note that PR_ATTACH_DATA_BIN and PR_ATTACH_DATA_OBJ share the same property identifier and thus are two renditions of the same property.

For a storage object, such as a compound file in OLE 2.0 docfile format, some service providers allow it to be opened with the MAPI **IStreamDocfile** interface for improved performance. A provider supporting **IStreamDocfile** must expose it on PR_ATTACH_DATA_OBJ and may optionally expose it on PR_ATTACH_DATA_BIN.

For more information on OLE interfaces and formats, see the *OLE Programmer's Reference*.

PR_ATTACH_DATA_BIN corresponds to the X.400 attribute IM_EXTERNAL_DATA or IM_BILATERAL_DATA, depending on the object.

PR_ATTACH_DATA_OBJ

The PR_ATTACH_DATA_OBJ property contains an attachment object typically accessed through the OLE **IStorage** interface.

Usage

Optional on attachment subobjects.

Details

Identifier 0x3701; property type PT_OBJECT; property tag 0x3701000D

Remarks

The PR_ATTACH_DATA_OBJ property holds the attachment when the value of the PR_ATTACH_METHOD property is ATTACH_EMBEDDED_MSG or ATTACH_OLE. The OLE encoding type can be determined from PR_ATTACH_TAG.

For an attachment associated with the ATTACH_EMBEDDED_MSG value, the **IMessage:IMAPIProp** inferface can be used for faster access.

For an embedded dynamic OLE object, the PR_ATTACH_DATA_OBJ property contains its own rendering information, and the PR_ATTACH_RENDERING property should be either nonexistent or empty.

For an OLE document file attachment, the message store provider must respond to an **IMAPIProp::OpenProperty** call on PR_ATTACH_DATA_OBJ and may optionally respond to a call on PR_ATTACH_DATA_BIN. The PR_ATTACH_DATA_BIN and PR_ATTACH_DATA_OBJ properties share the same property identifier and thus are two renditions of the same property.

For a storage object, such as a compound file in OLE 2.0 docfile format, some service providers allow it to be opened with the MAPI **IStreamDocfile** interface, a subclass of **IStream** with no additional members, designed to optimize performance. The potential saving is enough to justify attempting to open PR_ATTACH_DATA_OBJ through **IStreamDocfile**. If MAPI_E_NOINTERFACE is returned, the client can then open PR_ATTACH_DATA_BIN with **IStream**.

If the client application or service provider cannot open an attachment subobject using PR_ATTACH_DATA_OBJ with the help of PR_ATTACH_METHOD, it should use PR_ATTACH_DATA_BIN.

For more information on OLE interfaces and formats, see the *OLE Programmer's Reference*.

PR_ATTACH_DATA_OBJ corresponds to the X.400 attribute IM_EXTERNAL_DATA.

PR_ATTACH_ENCODING

The PR_ATTACH_ENCODING property contains an ASN.1 object identifier specifying the encoding for an attachment.

Usage

Optional on attachment subobjects.

Details

Identifier 0x3702; property type PT_BINARY; property tag 0x37020102

Remarks

The PR_ATTACH_ENCODING property identifies the algorithm used to transform the data in an attachment.

Note The PR_ATTACH_ENCODING and PR_ATTACH_TAG properties should not be confused. They are not paired or related. PR_ATTACH_TAG identifies the application that originally generated the attachment. "Object" has a much more general meaning in the term object identifier, and in X.400, than in object-oriented programming.

The object identifier syntax and sample object identifiers are defined in the MAPIOID.H header file. Values for PR_ATTACH_ENCODING are not limited to those defined in MAPIOID.H.

For complete information on these object identifiers, see the documentation on ASN.1, X.208, and X.209. The object identifier is found in the application-reference element of the FTBP (File Transfer Body Part) environment.

PR_ATTACH_EXTENSION

The PR_ATTACH_EXTENSION property contains a filename extension that indicates the document type of an attachment.

Usage

Optional on attachment subobjects.

Details

Identifier 0x3703; property type PT_TSTRING; property tag 0x3703001E (0x3703001F for Unicode)

Remarks

The PR_ATTACH_EXTENSION property is set by the client application at submission time.

The messaging system uses PR_ATTACH_EXTENSION when converting message attachments (in-route conversion) or launching applications based on attachments in received messages. If the sending client does not provide a value for this property, the message store handling the attachment is not obligated to generate it. The receiving client should first check for PR_ATTACH_EXTENSION, and if it is not provided, should parse the filename extension from the attachment's PR_ATTACH_FILENAME or PR_ATTACH_LONG_FILENAME property.

PR_ATTACH_FILENAME

The PR_ATTACH_FILENAME property contains an attachment's base filename and extension, excluding path.

Usage

Optional but recommended on attachment subobjects.

Details

Identifier 0x3704; property type PT_TSTRING; property tag 0x3704001E (0x3704001F for Unicode)

Remarks

The PR_ATTACH_FILENAME property pertains to the ATTACH_BY_VALUE, ATTACH_BY_REFERENCE, ATTACH_BY_REF_RESOLVE, and ATTACH_BY_REF_ONLY values of the PR_ATTACH_METHOD property. PR_ATTACH_FILENAME is required when any of these values is used.

PR_ATTACH_FILENAME can be used as a suggested filename for saving the attachment and to supply the filename extension if the PR_ATTACH_EXTENSION property is not provided.

The filename is restricted to eight characters plus a three-character extension. For a platform that supports long filenames, set both this property and the PR_ATTACH_LONG_FILENAME property.

MAPI works only with filenames, and other strings passed to it, in the ANSI character set. Client applications that use filenames in an OEM character set must convert them to ANSI before calling MAPI.

PR_ATTACH_FILENAME corresponds to the X.400 attribute IM_EXTERNAL_PARAMETERS.

PR_ATTACH_LONG_FILENAME

The PR_ATTACH_LONG_FILENAME property contains an attachment's long filename and extension, excluding path.

Usage

Optional on attachment subobjects.

Details

Identifier 0x3707; property type PT_TSTRING; property tag 0x3707001E (0x3707001F for Unicode)

Remarks

The PR_ATTACH_LONG_FILENAME property pertains to the ATTACH_BY_VALUE, ATTACH_BY_REFERENCE, ATTACH_BY_REF_RESOLVE, and ATTACH_BY_REF_ONLY values of the PR_ATTACH_METHOD property. Platforms that support long filenames should set both the PR_ATTACH_LONG_FILENAME and PR_ATTACH_FILENAME properties when sending, and should check PR_ATTACH_LONG_FILENAME first when receiving.

The client application should set this property to a suggested long filename to be used if the host computer receiving a message supports long filenames. PR_ATTACH_LONG_FILENAME can be used as a filename for saving the attachment, and to supply the filename extension if the PR_ATTACH_EXTENSION property is not provided.

Unlike the filename provided by PR_ATTACH_FILENAME, this name is not restricted to an eight-character filename plus a three-character extension. Instead, it can be up to 256 characters long, including the filename, extension, and separator period.

MAPI works only with filenames, and other strings passed to it, in the ANSI character set. Client applications that use filenames in an OEM character set must convert them to ANSI before calling MAPI.

PR_ATTACH_LONG_PATHNAME

The PR_ATTACH_LONG_PATHNAME property contains an attachment's fully qualified long path and filename.

Usage

Optional on attachment subobjects.

Details

Identifier 0x370D; property type PT_TSTRING; property tag 0x370D001E (0x370D001F for Unicode)

Remarks

The PR_ATTACH_LONG_PATHNAME property is applicable when you use any of the PR_ATTACH_METHOD property's values that indicate attachment by reference: ATTACH_BY_REFERENCE, ATTACH_BY_REF_RESOLVE, or ATTACH_BY_REF_ONLY. Platforms that support long filenames should set both the PR_ATTACH_LONG_PATHNAME and PR_ATTACH_PATHNAME properties when sending, and should check PR_ATTACH_LONG_PATHNAME first when receiving.

The client application should set this property to a suggested long path and filename to be used if the host machine receiving a message supports long filenames. Setting PR_ATTACH_LONG_PATHNAME indicates that the attachment data is not included with the message but is available on a common file server.

Unlike the directories and filenames provided by PR_ATTACH_PATHNAME, these directories and filenames are not restricted to an eight-character directory or filename plus three-character extension. Instead, each directory or filename can be up to 256 characters long, including the name, extension, and separator period. However, the overall path is limited to 256 characters.

Clients should use a Universal Naming Convention (UNC) in most cases when the file is shared, and should use an absolute path when the file is local and in some shared cases, such as with NetWare®.

MAPI works only with paths, filenames, and other strings passed to it in the ANSI character set. Client applications that use paths and filenames in an OEM character set must convert them to ANSI before calling MAPI.

PR_ATTACH_METHOD

The PR_ATTACH_METHOD property contains a MAPI-defined constant representing the way the contents of an attachment can be accessed.

Usage

Required on attachment subobjects.

Details

Identifier 0x3705; property type PT_LONG; property tag 0x37050003

Remarks

The PR_ATTACH_METHOD property can have exactly one of the following values:

Value	Description
NO_ATTACHMENT	The attachment has just been created.
ATTACH_BY_VALUE	The PR_ATTACH_DATA_BIN property contains the attachment data.
ATTACH_BY_REFERENCE	The PR_ATTACH_PATHNAME or PR_ATTACH_LONG_PATHNAME property contains a fully qualified path identifying the attachment to recipients with access to a common file server.
ATTACH_BY_REF_RESOLVE	The PR_ATTACH_PATHNAME or PR_ATTACH_LONG_PATHNAME property contains a fully qualified path identifying the attachment.
ATTACH_BY_REF_ONLY	The PR_ATTACH_PATHNAME or PR_ATTACH_LONG_PATHNAME property contains a fully qualified path identifying the attachment.
ATTACH_EMBEDDED_MSG	The PR_ATTACH_DATA_OBJ property contains an embedded object that supports the **IMessage** interface.
ATTACH_OLE	The attachment is an embedded OLE object.

When created, all attachment objects have an initial PR_ATTACH_METHOD value of NO_ATTACHMENT.

Client applications and service providers are only required to support the attachment method represented by the ATTACH_BY_VALUE value. The other attachment methods are optional. The message store does not enforce any consistency between the value of PR_ATTACH_METHOD and the values of the other attachment properties.

Universal Naming Convention (UNC) names are recommended for fully qualified paths, which should be used with ATTACH_BY_REFERENCE and ATTACH_BY_REF_ONLY. With ATTACH_BY_REF_RESOLVE, an absolute path is faster, since the MAPI spooler converts the attachment to ATTACH_BY_VALUE.

If ATTACH_BY_REFERENCE is set, PR_ATTACH_DATA_BIN must be empty. An outbound gateway can turn an ATTACH_BY_REFERENCE attachment into an ATTACH_BY_VALUE attachment by copying the attachment data into the PR_ATTACH_DATA_BIN property.

If ATTACH_BY_REF_RESOLVE is set, PR_ATTACH_DATA_BIN must be empty. When the message that contains the ATTACH_BY_REF_RESOLVE attachment is sent, the MAPI spooler copies the attachment data into an ATTACH_BY_VALUE attachment. This resolution process places the attachment data in PR_ATTACH_DATA_BIN.

If ATTACH_BY_REF_ONLY is set, PR_ATTACH_DATA_BIN must be empty, and the messaging system never resolves the attachment reference. Use this value when you want to send the link but not the data.

When the OLE object is in OLE 2.0 **IStorage** format, the data are accessible through PR_ATTACH_DATA_OBJ. When the OLE object is in OLE 1.0 **OLESTREAM** format, the data are accessible through PR_ATTACH_DATA_BIN as an **IStream**. The type of the OLE encoding can be determined by the PR_ATTACH_TAG value.

For more information on OLE interfaces and formats, see the *OLE Programmer's Reference*.

See Also

PR_STORE_SUPPORT_MASK property

PR_ATTACH_MIME_TAG

The PR_ATTACH_MIME_TAG property contains formatting information about a Multipurpose Internet Mail Extensions (MIME) attachment.

Usage

Optional on attachment subobjects.

Details

Identifier 0x370E; property type PT_TSTRING; property tag 0x370E001E (0x370E001F for Unicode)

Remarks

If the PR_ATTACH_TAG property contains the value OID_MIMETAG, the transport provider should look at the PR_ATTACH_MIME_TAG property to determine how the attachment is formatted.

PR_ATTACH_MIME_TAG is copied from the Content-type parameter of the inbound MIME header. The composition of the string is defined in the RFC 1521 standard. The format is type/subtype, such as application/binary or text/plain.

PR_ATTACH_NUM

The PR_ATTACH_NUM property contains a number that uniquely identifies the attachment within its parent message.

Usage

Required on attachment subobjects.

Details

Identifier 0x3706; property type PT_LONG; property tag 0x37060003

Remarks

Message stores generate and maintain the PR_ATTACH_NUM property. The attachment number is the secondary sort key, after the rendering position, in the attachment table.

PR_ATTACH_NUM is used to open the attachment with the **IMessage::OpenAttach** method. Within a client application's session, the PR_ATTACH_NUM property of a message attachment remains constant as long as the attachment table is open.

The message store propagates changes to the table using the **IMessage::CreateAttach** and **IMessage::DeleteAttach** methods. At its option the message store can generate table notifications on open attachment tables so that clients can resynchronize to those changes.

PR_ATTACH_PATHNAME

The PR_ATTACH_PATHNAME property contains an attachment's fully qualified path and filename.

Usage

Optional but recommended on attachment subobjects.

Details

Identifier 0x3708; property type PT_TSTRING; property tag 0x3708001E (0x3708001F for Unicode)

Remarks

Setting the PR_ATTACH_PATHNAME property indicates that the attachment data is not included with the message but is available on a common file server. PR_ATTACH_PATHNAME is required in conjunction with any of the PR_ATTACH_METHOD flags that indicate attachment by reference: ATTACH_BY_REFERENCE, ATTACH_BY_REF_RESOLVE, or ATTACH_BY_REF_ONLY.

Each directory or filename is restricted to an eight-character name plus a three-character extension. The overall path is restricted to 256 characters. For a platform that supports long filenames, set both the PR_ATTACH_PATHNAME and PR_ATTACH_LONG_PATHNAME properties.

Client applications should use a Universal Naming Convention (UNC) in most cases when the file is shared, and should use an absolute path when the file is local and in some shared cases, such as with NetWare.

MAPI works only with paths, filenames, and other strings passed to it in the ANSI character set. Clients that use paths and filenames in an OEM character set must convert them to ANSI before calling MAPI.

See Also

ScLocalPathFromUNC function, **ScUNCFromLocalPath** function

PR_ATTACH_RENDERING

The PR_ATTACH_RENDERING property contains a Microsoft Windows metafile with rendering information for an attachment.

Usage

Optional but recommended on attachment subobjects.

Details

Identifier 0x3709; property type PT_BINARY; property tag 0x37090102

Remarks

The purpose of the PR_ATTACH_RENDERING property is to provide an icon or other pictorial representation that can be displayed within the parent message at the point of attachment. Such representation typically includes the name of the attachment, if any, and the nature of the attachment, such as a Microsoft Word document. A client application can use this representation in the display of the message.

For an attached file, PR_ATTACH_RENDERING usually portrays an icon for the file.

For an attached message, PR_ATTACH_RENDERING is typically not set. A client application needing to render an attached message should obtain its PR_MESSAGE_CLASS property, call **IMAPIFormMgr::ResolveMessageClass** for a pointer to the corresponding form information object, open the **IMAPIFormInfo** interface on that object, and use **GetProps** to retrieve the PR_ICON or PR_MINI_ICON property.

For an embedded static OLE object, PR_ATTACH_RENDERING contains a Microsoft Windows metafile that can be used to draw the attachment representation in a window.

For an embedded dynamic OLE object, the client should use the OLE data to generate the rendering information.

In all cases, the client application should be aware that PR_ATTACH_RENDERING is usually several hundred bytes in size and is subject to truncation in the attachment table. If a client wishes to render the attachment from PR_ATTACH_RENDERING without opening the attachment itself, it must work within the table truncation rule. For more information, see Working with Large Columns.

PR_ATTACH_SIZE

The PR_ATTACH_SIZE property contains the sum, in bytes, of the sizes of all properties on an attachment.

Usage

Optional but recommended on attachment subobjects.

Details

Identifier 0x0E20; property type PT_LONG; property tag 0x0E200003

Remarks

The sum contained in the PR_ATTACH_SIZE property includes the size of the PR_ATTACH_DATA_BIN or PR_ATTACH_DATA_OBJ property. Accordingly, PR_ATTACH_SIZE is usually larger than the contents of the attachment alone.

This property can be used to check the approximate size of the attachment before performing a remote transfer by modem and to display progress indicators when saving the attachment to disk. It is particularly useful with attached OLE objects.

See Also

PR_MESSAGE_SIZE property

PR_ATTACH_TAG

The PR_ATTACH_TAG property contains an ASN.1 object identifier specifying the application that supplied an attachment.

Usage

Optional on attachment subobjects.

Details

Identifier 0x370A; property type PT_BINARY; property tag 0x370A0102

Remarks

The PR_ATTACH_TAG property identifies the application that originally generated the attachment.

Note The PR_ATTACH_ENCODING and PR_ATTACH_TAG properties should not be confused. They are not paired or related. "Object" has a much more general meaning in the term "object identifier," and in X.400 usage, than in object-oriented programming.

The object identifier syntax and sample object identifiers are defined in the MAPIOID.H header file. Values for PR_ATTACH_TAG are not limited to those defined in MAPIOID.H.

For complete information on these object identifiers, see the documentation on ASN.1, X.208, and X.209. The object identifier is found in the application-reference element of the File Transfer Body Part (FTBP) environment.

PR_ATTACH_TAG corresponds to the X.400 attribute IM_EXTERNAL_PARAMETERS.

See Also

PR_ATTACH_MIME_TAG property

PR_ATTACH_TRANSPORT_NAME

The PR_ATTACH_TRANSPORT_NAME property contains the name of an attachment file modified so that it can be correlated with TNEF messages.

Usage

Optional on attachment subobjects.

Details

Identifier 0x370C; property type PT_TSTRING; property tag 0x370C001E (0x370C001F for Unicode)

Remarks

TNEF and the transport provider use the PR_ATTACH_TRANSPORT_NAME property. It is usually not available to client applications.

This property is commonly used by TNEF when the underlying messaging system doesn't support the supplied filenames. For example, PR_ATTACH_TRANSPORT_NAME is used when the user attaches multiple files with the same name, such as five files named CONFIG.SYS. The transport provider must modify the names to ensure uniqueness. Each modified name appears in its attachment's PR_ATTACH_TRANSPORT_NAME property.

MAPI works only with filenames, and other strings passed to it, in the ANSI character set. Applications that use filenames in an OEM character set must convert them to ANSI before calling MAPI.

PR_ATTACHMENT_X400_PARAMETERS

The PR_ATTACHMENT_X400_PARAMETERS property was originally meant to contain an ASN.1 object identifier specifying how the attachment should be handled during transmission.

Usage

Never used.

Details

Identifier 0x3700; property type PT_BINARY; property tag 0x37000102

Remarks

Do not use this property. It is not supported in MAPI 1.0.

PR_ATTACHMENT_X400_PARAMETERS corresponds to the X.400 attribute IM_EXTERNAL_PARAMETERS.

PR_AUTHORIZING_USERS

The PR_AUTHORIZING_USERS property contains a list of entry identifiers for users who have authorized the sending of a message.

Usage

Optional on message objects.

Details

Identifier 0x0003; property type PT_BINARY; property tag 0x00030102

Remarks

The message store does not maintain the PR_AUTHORIZING_USERS property.

PR_AUTHORIZING_USERS corresponds to the X.400 attribute IM_AUTHORIZING_USERS.

See Also

PR_ENTRYID property

PR_AUTO_FORWARD_COMMENT

The PR_AUTO_FORWARD_COMMENT property contains a comment added by the autoforwarding agent.

Usage

Optional on message objects.

Details

Identifier 0x0004; property type PT_TSTRING; property tag 0x0004001E (0x0004001F for Unicode)

Remarks

The PR_AUTO_FORWARD_COMMENT property corresponds to the X.400 attribute IM_AUTO_FORWARD_COMMENT.

PR_AUTO_FORWARDED

The PR_AUTO_FORWARDED property contains TRUE if an automatic agent has forwarded a message.

Usage

Optional on message objects.

Details

Identifier 0x0005; property type PT_BOOLEAN; property tag 0x0005000B

Remarks

The PR_AUTO_FORWARDED property corresponds to the X.400 attribute IM_AUTO_FORWARDED.

PR_BODY

The PR_BODY property contains the message text.

Usage

Optional on message objects.

Details

Identifier 0x1000; property type PT_TSTRING; property tag 0x1000001E (0x1000001F for Unicode)

Remarks

The PR_BODY property is typically used only in an interpersonal message (IPM).

Message stores that support Rich Text Format (RTF) ignore any changes to white space in the message text. When PR_BODY is stored for the first time, the message store also generates and stores the PR_RTF_COMPRESSED property, the RTF version of the message text. If the **IMAPIProp::SaveChanges** method is subsequently called and PR_BODY has been modified, the message store calls the **RTFSync** function to ensure synchronization with the RTF version. If only white space has been changed, the properties are left unchanged. This preserves any nontrivial RTF formatting when the message travels through non-RTF-aware clients and messaging systems.

The value for PR_BODY must be expressed in the code page of the operating system that MAPI is running on.

PR_BODY corresponds to the X.400 attributes IM_BODY and IM_TEXT.

See Also

PR_RTF_IN_SYNC property

PR_BODY_CRC

The PR_BODY_CRC property contains a circular redundancy check (CRC) value on the message text.

Usage

Computed by message store providers on message objects.

Details

Identifier 0x0E1C; property type PT_LONG; property tag 0x0E1C0003

Remarks

The message store can use any CRC algorithm that generates a PT_LONG value. It must compute PR_BODY_CRC as part of the **IMAPIProp::SaveChanges** method when the PR_BODY property has been set for the first time and whenever it has been subsequently modified.

A client application uses PR_BODY_CRC to aid in comparing message text strings contained in PR_BODY properties or their variants. Using this property, the client can quickly and easily detect when the message text has changed. It can realize significant performance gains by using PR_BODY_CRC instead of obtaining PR_BODY from the message store and comparing it with a local version.

PR_BUSINESS_FAX_NUMBER

The PR_BUSINESS_FAX_NUMBER property contains the telephone number of the recipient's business fax machine.

Usage

Optional on messaging user objects.

Details

Identifier 0x3A24; property type PT_TSTRING; property tag 0x3A24001E (0x3A24001F for Unicode)

Remarks

The PR_BUSINESS_FAX_NUMBER property is one of the properties that provide identification and access information about a recipient. These properties are defined by the recipient and the recipient's organization. For more information on this group of properties, see About Messaging User Objects.

PR_BUSINESS_TELEPHONE_NUMBER

The PR_BUSINESS_TELEPHONE_NUMBER property contains the primary telephone number of the recipient's place of business.

Usage

Optional on messaging user objects.

Details

Identifier 0x3A08; property type PT_TSTRING; property tag 0x3A08001E (0x3A08001F for Unicode)

Remarks

The PR_BUSINESS_TELEPHONE_NUMBER property is one of the properties that provide identification and access information about a recipient. These properties are defined by the recipient and the recipient's organization. For more information on this group of properties, see About Messaging User Objects.

MAPI also supports the PR_OFFICE_TELEPHONE_NUMBER property, which is synonymous with PR_BUSINESS_TELEPHONE_NUMBER.

PR_BUSINESS2_TELEPHONE_NUMBER

The PR_BUSINESS2_TELEPHONE_NUMBER property contains a secondary telephone number at the recipient's place of business.

Usage

Optional on messaging user objects.

Details

Identifier 0x3A1B; property type PT_TSTRING; property tag 3A1B001E (3A1B001F for Unicode)

Remarks

The PR_BUSINESS2_TELEPHONE_NUMBER property is one of the properties that provide identification and access information about a recipient. These properties are defined by the recipient and the recipient's organization. For more information on this group of properties, see About Messaging User Objects.

MAPI also supports the PR_OFFICE2_TELEPHONE_NUMBER property, which is synonymous with PR_BUSINESS2_TELEPHONE_NUMBER.

See Also

PR_BUSINESS_TELEPHONE_NUMBER property

PR_CALLBACK_TELEPHONE_NUMBER

The PR_CALLBACK_TELEPHONE_NUMBER property contains a telephone number that the message recipient can use to reach the sender.

Usage

Optional on messaging user objects.

Details

Identifier 0x3A02; property type PT_TSTRING; property tag 0x3A02001E (0x3A02001F for Unicode)

Remarks

The PR_CALLBACK_TELEPHONE_NUMBER property is one of the properties that provide identification and access information about a recipient. These properties are defined by the recipient and the recipient's organization. For more information on this group of properties, see About Messaging User Objects.

PR_CAPABILITIES_TABLE

The PR_CAPABILITIES_TABLE property was originally meant to contain an embedded table object to provide a summary of the address book capabilities.

Usage

Never used.

Details

Identifier 0x3903; property type PT_OBJECT; property tag 0x3903000D

Remarks

Do not use this property. It is not supported in MAPI 1.0.

PR_CAR_TELEPHONE_NUMBER

The PR_CAR_TELEPHONE_NUMBER property contains the recipient's car telephone number.

Usage

Optional on messaging user objects.

Details

Identifier 0x3A1E; property type PT_TSTRING; property tag 0x3A1E001E (0x3A1E001F for Unicode)

Remarks

The PR_CAR_TELEPHONE_NUMBER property is one of the properties that provide identification and access information about a recipient. These properties are defined by the recipient and the recipient's organization. For more information on this group of properties, see About Messaging User Objects.

PR_CLIENT_SUBMIT_TIME

The PR_CLIENT_SUBMIT_TIME property contains the date and time the message sender submitted a message.

Usage

Computed by store providers on message objects.

Details

Identifier 0x0039; property type PT_SYSTIME; property tag 0x00390040

Remarks

The store provider sets PR_CLIENT_SUBMIT_TIME to the time the client application calls **IMessage::SubmitMessage**.

PR_CLIENT_SUBMIT_TIME corresponds to the X.400 attribute MH_T_SUBMISSION_TIME.

PR_COMMENT

The PR_COMMENT property contains a comment about the purpose or content of an object.

Usage

Optional on all objects.

Details

Identifier 0x3004; property type PT_TSTRING; property tag 0x3004001E (0x3004001F for Unicode)

Remarks

The content of the string is defined by the messaging user.

PR_COMMON_VIEWS_ENTRYID

The PR_COMMON_VIEWS_ENTRYID property contains the entry identifier of the predefined common view folder.

Usage

Required on message store objects.

Details

Identifier 0x35E6; property type PT_BINARY; property tag 0x35E60102

Remarks

The common view folder contains a predefined set of standard view specifiers, while the view folder contains specifiers defined by a messaging user. These folders, which are not visible in the interpersonal message (IPM) hierarchy, can hold many view specifiers, each one stored as a message. A client application can choose to merge the two sets of specifiers and make them both available.

For more information on views, see About View Folders.

See Also

PR_DEFAULT_VIEW_ENTRYID property, PR_VIEWS_ENTRYID property

PR_COMPANY_NAME

The PR_COMPANY_NAME property contains the recipient's company name.

Usage

Optional on messaging user objects.

Details

Identifier 0x3A16; property type PT_TSTRING; property tag 0x3A16001E (0x3A16001F for Unicode)

Remarks

The PR_COMPANY_NAME property is one of the properties that provide identification and access information about a recipient. These properties are defined by the recipient and the recipient's organization.

PR_CONTAINER_CLASS

The PR_CONTAINER_CLASS property was originally meant to contain a text string describing the type of a folder.

Usage

Never used.

Details

Identifier 0x3613; property type PT_TSTRING; property tag 0x3613001E (0x3613001F for Unicode)

Remarks

Do not use this property. It is not supported in MAPI 1.0.

PR_CONTAINER_CONTENTS

The PR_CONTAINER_CONTENTS property contains an embedded contents table object that provides information about a container.

Usage

Required on address book container and folder objects.

Details

Identifier 0x360F; property type PT_OBJECT; property tag 0x360F000D

Remarks

The PR_CONTAINER_CONTENTS property can be excluded in **IMAPIProp::CopyTo** operations or included in **IMAPIProp::CopyProps** operations. As a property of type PT_OBJECT, it cannot be successfully retrieved by the **IMAPIProp::GetProps** method; its contents should be accessed by the **IMAPIProp::OpenProperty** method, requesting the IID_IMAPITable interface identifier. Service providers must report it to the **IMAPIProp::GetPropList** method if it is set, but can optionally report it or not if it is not set.

To retrieve table contents, a client application should call the **IMAPIContainer::GetContentsTable** method. For more information on contents tables, see About Contents Tables.

The PR_CONTAINER_CONTENTS, PR_CONTAINER_HIERARCHY, and PR_FOLDER_ASSOCIATED_CONTENTS properties are similar in usage. Several MAPI properties provide access to tables:

Property	**Table**
PR_CONTAINER_CONTENTS	Contents table
PR_CONTAINER_HIERARCHY	Hierarchy table
PR_FOLDER_ASSOCIATED_CONTENTS	Associated contents table
PR_MESSAGE_ATTACHMENTS	Attachment table
PR_MESSAGE_RECIPIENTS	Recipient table

PR_CONTAINER_FLAGS

The PR_CONTAINER_FLAGS property contains a bitmask of flags describing an address book container.

Usage

Required on address book container objects.

Details

Identifier 0x3600; property type PT_LONG; property tag 0x36000003

Remarks

One or more of the following flags can be set for the PR_CONTAINER_FLAGS bitmask:

AB_FIND_ON_OPEN
: Displays a dialog box to request a restriction before displaying any contents of the container.

AB_MODIFIABLE
: Entries can be added to and removed from the container. This setting does not indicate modification of the entries, but only whether they can be placed in the container.

AB_RECIPIENTS
: The contents table and the hierarchy table are both modifiable, and the container can contain recipients. This flag does not indicate whether any recipients are present in the container.

AB_SUBCONTAINERS
: The container holds child containers. This flag must be set for the container to support **IMAPIContainer::GetHierarchyTable**.

AB_UNMODIFIABLE
: Entries cannot be added to or removed from the container.

The flags refer to the container itself rather than to the entries within the container. For example, the AB_MODIFIABLE flag indicates that the container allows entries to be added or removed. The flag does not refer to whether the individual entries themselves can be modified.

The AB_FIND_ON_OPEN flag is highly recommended for containers, such as online services, where even a partial specification for messaging users can dramatically speed up a display of the contents.

Either the AB_MODIFIABLE or AB_UNMODIFIABLE flag must be set. Both flags can be set to indicate that the container does not know whether it can be modified or not. In this case, a client application must attempt a call and examine the return code to determine the container's capabilities. A clients typically starts by examining AB_MODIFIABLE. If it is set, the client makes a call that attempts to modify the container and check the return value.

PR_CONTAINER_HIERARCHY

The PR_CONTAINER_HIERARCHY property contains an embedded hierarchy table object that provides information about the child containers.

Usage

Required on address book container and folder objects.

Details

Identifier 0x360E; property type PT_OBJECT; property tag 0x360E000D

Remarks

The PR_CONTAINER_HIERARCHY property can be excluded in **IMAPIProp::CopyTo** operations or included in **IMAPIProp::CopyProps** operations. As a property of type PT_OBJECT, it cannot be successfully retrieved by the **IMAPIProp::GetProps** method; its contents should be accessed by the **IMAPIProp::OpenProperty** method, requesting the IID_IMAPITable interface identifier. Service providers must report it to the **IMAPIProp::GetPropList** method if it is set, but can optionally report it or not if it is not set.

To retrieve table contents, a client application should call the **IMAPIContainer::GetHierarchyTable** method. For more information on hierarchy tables, see About Hierarchy Tables.

The PR_CONTAINER_CONTENTS, PR_CONTAINER_HIERARCHY, and PR_FOLDER_ASSOCIATED_CONTENTS properties are similar in usage. Several MAPI properties provide access to tables:

Property	Table
PR_CONTAINER_CONTENTS	Contents table
PR_CONTAINER_HIERARCHY	Hierarchy table
PR_FOLDER_ASSOCIATED_CONTENTS	Associated contents table
PR_MESSAGE_ATTACHMENTS	Attachment table
PR_MESSAGE_RECIPIENTS	Recipient table

PR_CONTAINER_MODIFY_VERSION

The PR_CONTAINER_MODIFY_VERSION property was originally meant to contain the current modification version for a folder.

Usage

Never used.

Details

Identifier 0x3614; property type PT_I8; property tag 0x36140014

Remarks

Do not use this property. It is not supported in MAPI 1.0.

PR_CONTENT_CONFIDENTIALITY_ALGORITHM_ID

The PR_CONTENT_CONFIDENTIALITY_ALGORITHM_ID property contains an identifier for the algorithm used to confirm message content confidentiality.

Usage

Optional on message objects.

Details

Identifier 0x0006; property type PT_BINARY; property tag 0x00060102

Remarks

The PR_CONTENT_CONFIDENTIALITY_ALGORITHM_ID property corresponds to the X.400 attribute MH_T_ALGORITHM_ID or MH_T_CONFIDENTIALITY_ALGORITHM.

See Also

PR_SECURITY property

PR_CONTENT_CORRELATOR

The PR_CONTENT_CORRELATOR property contains a value the message sender can use to match a report with the original message.

Usage

Optional on message objects.

Details

Identifier 0x0007; property type PT_BINARY; property tag 0x00070102

Remarks

The contents of the binary string are defined by the message originator. If set on an outgoing message, the PR_CONTENT_CORRELATOR property should be copied onto any reports generated in response to the message.

PR_CONTENT_CORRELATOR corresponds to the X.400 attribute MH_T_CONTENT_CORRELATOR.

PR_CONTENT_COUNT

The PR_CONTENT_COUNT property contains the number of messages in a folder, as computed by the message store.

Usage

Required on folder objects and as a column entry in folder hierarchy tables. Optional as a column entry in folder contents tables.

Details

Identifier 0x3602; property type PT_LONG; property tag 0x36020003

Remarks

The number contained in the PR_CONTENT_COUNT property does not include associated entries in the folder. PR_CONTENT_UNREAD contains the count of unread messages for the folder. A client application can read but not change PR_CONTENT_COUNT and PR_CONTENT_UNREAD.

PR_CONTENT_IDENTIFIER

The PR_CONTENT_IDENTIFIER property contains a key value that enables the message recipient to identify its content.

Usage

Optional on message objects.

Details

Identifier 0x0008; property type PT_TSTRING; property tag 0x0008001E (0x0008001F for Unicode)

Remarks

The PR_CONTENT_IDENTIFIER property corresponds to the X.400 attribute MH_T_CONTENT_IDENTIFIER.

PR_CONTENT_INTEGRITY_CHECK

The PR_CONTENT_INTEGRITY_CHECK property contains an ASN.1 content integrity check value that allows a message sender to protect message content from disclosure to unauthorized recipients.

Usage

Optional on message objects.

Details

Identifier 0x0C00; property type PT_BINARY; property tag 0x0C000102

Remarks

The PR_CONTENT_INTEGRITY_CHECK property provides for non-repudiation of message content. In conjunction with PR_MESSAGE_TOKEN it insures that the content of a message arrives at its destination unchanged.

PR_CONTENT_INTEGRITY_CHECK corresponds to the X.400 attribute MH_T_INTEGRITY_CHECK.

See Also

PR_MESSAGE_SECURITY_LABEL property

PR_CONTENT_LENGTH

The PR_CONTENT_LENGTH property contains a message length, in bytes, passed to a client application or service provider to determine if a message of that length can be delivered.

Usage

Optional on message objects.

Details

Identifier 0x0009; property type PT_LONG; property tag 0x00090003

Remarks

The PR_CONTENT_LENGTH property corresponds to the X.400 attribute MH_T_CONTENT_LENGTH.

PR_CONTENT_RETURN_REQUESTED

The PR_CONTENT_RETURN_REQUESTED property contains TRUE if a message should be returned with a nondelivery report.

Usage

Optional on message objects.

Details

Identifier 0x000A; property type PT_BOOLEAN; property tag 0x000A000B

Remarks

If the PR_CONTENT_RETURN_REQUESTED property is not set, MAPI treats it as having a TRUE value.

PR_CONTENT_RETURN_REQUESTED corresponds to the X.400 attribute MH_T_CONTENT_RETURN_REQUESTED.

PR_CONTENT_UNREAD

The PR_CONTENT_UNREAD property contains the number of unread messages in a folder, as computed by the message store.

Usage

Required on folder objects and as a column entry in folder hierarchy tables. Optional as a column entry in folder contents tables.

Details

Identifier 0x3603; property type PT_LONG; property tag 0x36030003

Remarks

The PR_CONTENT_UNREAD property contains the number of messages in the folder contents table for which the MSGFLAG_READ flag is not set in the PR_MESSAGE_FLAGS property. The PR_CONTENT_COUNT property contains the total message count for the folder. PR_CONTENT_COUNT and PR_CONTENT_UNREAD are read-only to clients.

See Also

PR_CONTENT_IDENTIFIER property

PR_CONTENTS_SORT_ORDER

The PR_CONTENTS_SORT_ORDER property was originally meant to contain a value specifying the sort order for the columns of a container.

Usage

Never used.

Details

Identifier 0x360D; property type PT_MV_LONG; property tag 0x360D1003

Remarks

Do not use this property. It is not supported in MAPI 1.0.

See Also

SSortOrder structure

PR_CONTROL_FLAGS

The PR_CONTROL_FLAGS property contains a bitmask of flags governing the behavior of a control used in a dialog box.

Usage

Required as a column entry in display tables.

Details

Identifier 0x3F00; property type PT_LONG; property tag 0x3F000003

Remarks

One or more of the following flags can be set for PR_CONTROL_FLAGS:

DT_ACCEPT_DBCS
: The control can have Double-Byte Character Set (DBCS) characters in it. This flag is used with edit controls and combo box controls. It allows multiple-byte character sets.

DT_EDITABLE
: The control can be edited; the value associated with the control can be changed. When this flag is not set, the control is read-only. This value is ignored on label, group box, standard push button, multivalued drop down list box and list box controls.

DT_MULTILINE
: The edit control can contain multiple lines. This means a return character can be entered within the control.

DT_PASSWORD_EDIT
: Applies to edit controls. The edit control is treated like a password. The value is displayed using asterisks instead of echoing the actual characters entered.

DT_REQUIRED
: If the control allows changes (DT_EDITABLE), it must have a value before **IMAPIProp::SaveChanges** is called.

DT_SET_IMMEDIATE
: Enables immediate setting of a value; as soon as a value in the control changes, MAPI calls the **SetProps** method for the property associated with that control. When this flag is not set, the values are set when another control is selected.

DT_SET_SELECTION
: When a selection is made within the listbox, the index column of that listbox is set as a property. Always used with DT_SET_IMMEDIATE.

See Also

DTCTL structure

PR_CONTROL_ID

The PR_CONTROL_ID property contains a unique identifier for a control used in a dialog box.

Usage

Required as a column entry in display tables.

Details

Identifier 0x3F07; property type PT_BINARY; property tag 0x3F070102

Remarks

The PR_CONTROL_ID property contains a unique identifier for the control. This identifier should contain a **GUID** structure and a binary value of type LONG. All controls in the dialog box should use the same **GUID** to identify the service provider, and each control should use a unique LONG value to ensure that the controls do not collide.

PR_CONTROL_ID is used in notifications. For example, notifications sent on the display table must have PR_CONTROL_ID to uniquely identify the control to update.

See Also

DTCTL structure

PR_CONTROL_STRUCTURE

The PR_CONTROL_STRUCTURE property contains a pointer to a structure for a control used in a dialog box.

Usage

Required as a column entry in display tables.

Details

Identifier 0x3F01; property type PT_BINARY; property tag 0x3F010102

Remarks

The PR_CONTROL_STRUCTURE property represents a long pointer that is cast to one of the control structures. The control structures include:

- **DTBLBUTTON**
- **DTBLCHECKBOX**
- **DTBLCOMBOBOX**
- **DTBLDDLBX**
- **DTBLEDIT**
- **DTBLGROUPBOX**
- **DTBLLABEL**
- **DTBLLBX**
- **DTBLMVDDLBOX**
- **DTBLMVLISTBOX**
- **DTBLPAGE**
- **DTBLRADIOBUTTON**

See Also

DTCTL structure

PR_CONTROL_TYPE

The PR_CONTROL_TYPE property contains a value indicating a control type for a control used in a dialog box.

Usage

Required as a column entry in display tables.

Details

Identifier 0x3F02; property type PT_LONG; property tag 0x3F020003

Remarks

The PR_CONTROL_TYPE property can have exactly one of the following values:

Value	Description
DTCT_BUTTON	A dialog button control.
DTCT_CHECKBOX	A dialog check box.
DTCT_COMBOBOX	A dialog combo box.
DTCT_DDLBX	A dialog drop-down list box.
DTCT_EDIT	A dialog edit text box.
DTCT_GROUPBOX	A dialog group box.
DTCT_LABEL	A dialog label.
DTCT_LBX	A dialog list box.
DTCT_LISTBOX	A dialog list box.
DTCT_MVDDLBX	A multivalued list box populated by a multivalued property of type string.
DTCT_PAGE	A dialog tabbed page.
DTCT_RADIOBUTTON	A dialog radio button.

See Also

DTCTL structure

PR_CONVERSATION_INDEX

The PR_CONVERSATION_INDEX property contains an index that indicates the relative position of this message within a conversation thread.

Usage

Optional on message objects.

Details

Identifier 0x0071; property type PT_BINARY; property tag 0x00710102

Remarks

The conversation index is usually implemented using concatenated time stamp values. A message that represents a reply to another message concatenates a time stamp to the PR_CONVERSATION_INDEX value of the original message. All messages that have the same value for PR_CONVERSATION_TOPIC, can be sorted by PR_CONVERSATION_INDEX to indicate the hierarchical relationship of the messages.

PR_CONVERSATION_KEY

The PR_CONVERSATION_KEY property is the obsolete precursor of the PR_CONVERSATION_INDEX and PR_CONVERSATION_TOPIC properties.

Usage

No longer used.

Details

Identifier 0x000B; property type PT_BINARY; property tag 0x000B0102

Remarks

Do not use this property. Use PR_CONVERSATION_INDEX and PR_CONVERSATION_TOPIC instead.

PR_CONVERSATION_TOPIC

The PR_CONVERSATION_TOPIC property represents the topic of the first message in a conversation thread.

Usage

Optional on message objects.

Details

Identifier 0x0070; property type PT_TSTRING; property tag 0x0070001E (0x0070001F for Unicode)

Remarks

A conversation thread represents a series of messages and replies. The PR_CONVERSATION_TOPIC property is set for the first message in a thread, usually to the same value as the message subject, with any RE: and FW: strings removed. Subsequent messages in the conversation thread use the same PR_CONVERSATION_TOPIC. The PR_CONVERSATION_INDEX property indicates the order relationship between subsequent messages and replies.

PR_CONVERSION_EITS

The PR_CONVERSION_EITS property contains the encoded information types (EITs) that are applied to a message in transit to describe conversions.

Usage

Optional on message objects.

Details

Identifier 0x000C; property type PT_BINARY; property tag 0x000C0102

Remarks

X.400 environments use the PR_CONVERSION_EITS property for both nondelivery reports and delivery reports.

PR_CONVERSION_EITS corresponds to the X.400 attribute IM_CONVERSION_EITS.

PR_CONVERSION_PROHIBITED

The PR_CONVERSION_PROHIBITED property contains TRUE if message conversions are prohibited by default for the associated messaging user.

Usage

Optional on message objects.

Details

Identifier 0x3A03; property type PT_BOOLEAN; property tag 0x3A03000B

Remarks

The PR_CONVERSION_PROHIBITED property corresponds to the X.400 attribute MH_T_CONVERSION_PROHIBITED.

PR_CONVERSION_WITH_LOSS_PROHIBITED

The PR_CONVERSION_WITH_LOSS_PROHIBITED property contains TRUE if a message transfer agent (MTA) is prohibited from making message text conversions that lose information.

Usage

Optional on message objects.

Details

Identifier 0x000D; property type PT_BOOLEAN; property tag 0x000D000B

Remarks

An example of the type of conversion being prohibited is the "lossy" mapping from Unicode (two bytes per character) to a single-byte character set.

The PR_CONVERSION_WITH_LOSS_PROHIBITED property corresponds to the X.400 attribute MH_T_CONVERSION_LOSS_PROHIBITED.

PR_CONVERTED_EITS

The PR_CONVERTED_EITS property contains an identifier for the types of text in a message after conversion.

Usage

Optional on message objects.

Details

Identifier 0x000E; property type PT_BINARY; property tag 0x000E0102

Remarks

The PR_CONVERTED_EITS property indicates which encoded information types (EITs) were used to convert the text components of the message.

PR_CONVERTED_EITS corresponds to the X.400 attribute MH_T_CONVERTED_EITS.

PR_CORRELATE

The PR_CORRELATE property contains TRUE if the sender of a message requests the correlation feature of the messaging system.

Usage

Optional on message objects.

Details

Identifier 0x0E0C; property type PT_BOOLEAN; property tag 0x0E0C000B

Remarks

The PR_CORRELATE property is used to request the correlation of incoming reports with the original sent message. When a transport provider encounters a submitted message with PR_CORRELATE set to TRUE, it sets the PR_CORRELATE_MTSID property to the message transfer system (MTS) identifier for that message.

PR_CORRELATE should be used with messaging systems that support correlation by MTS identifier, such as X.400.

PR_CORRELATE_MTSID

The PR_CORRELATE_MTSID property contains the message transfer system (MTS) identifier used in correlating reports with sent messages.

Usage

Optional on message objects.

Details

Identifier 0x0E0D; property type PT_BINARY; property tag 0x0E0D0102

Remarks

When a transport provider encounters a submitted message with the PR_CORRELATE property set to TRUE, it sets the PR_CORRELATE_MTSID property to the MTS identifier for that message. Following transmission, PR_CORRELATE_MTSID is stored with the message in the interpersonal message (IPM) Sent Items folder.

Messaging systems that support correlation by MTS identifier, such as X.400, retain the identifier as part of the transport envelope of the original message and also of any reports generated in response to it. When a report is delivered from such a messaging system, the transport provider sets PR_CORRELATE_MTSID to the original MTS identifier from the report's transport envelope. PR_CORRELATE_MTSID will then be stored with the report.

A client application can maintain a search-results folder of all messages having a PR_CORRELATE_MTSID property. When a report comes in for such a message, the client can apply restrictions to the search-results folder, find the original version of the message, and correlate the original message information with the new information.

PR_COUNTRY

The PR_COUNTRY property contains the name of the recipient's country.

Usage

Optional on messaging user objects.

Details

Identifier 0x3A26; property type PT_TSTRING; property tag 0x3A26001E (0x3A26001F for Unicode)

Remarks

The PR_COUNTRY property is one of the properties that provide identification and access information about a recipient. These properties are defined by the recipient and the recipient's organization. For more information on this group of properties, see About Messaging User Objects.

PR_CREATE_TEMPLATES

The PR_CREATE_TEMPLATES property contains an embedded table object that contains dialog box template entry identifiers.

Usage

Required on address book container objects.

Details

Identifier 0x3604; property type PT_OBJECT; property tag 0x3604000D

Remarks

To learn what template objects can be created inside a container, call the **IMAPIProp::OpenProperty** method on the PR_CREATE_TEMPLATES property. The resulting object is the one-off table that gives the entry identifiers for all the templates you can create inside the container.

To subsequently create the template objects, call the container object's **CreateEntry** method on the PR_ENTRYID column values from the one-off table.

See Also

IABContainer::CreateEntry method, PR_DISPLAY_NAME property, PR_DISPLAY_TYPE property

PR_CREATION_TIME

The PR_CREATION_TIME property contains the creation date and time for a message.

Usage

Required on message objects.

Details

Identifier 0x3007; property type PT_SYSTIME; property tag 0x30070040

Remarks

A message store sets the PR_CREATION_TIME property for each message that it creates.

See Also

PR_CREATION_VERSION property

PR_CREATION_VERSION

The PR_CREATION_VERSION property was originally meant to contain the message store version current at the time a message was created.

Usage

Never used.

Details

Identifier 0x0E19; property type PT_I8; property tag 0x0E190014

Remarks

Do not use this property. It is not supported in MAPI 1.0.

See Also

PR_CURRENT_VERSION property, PR_MODIFY_VERSION property

PR_CURRENT_VERSION

The PR_CURRENT_VERSION property was originally meant to contain the current version of a message store.

Usage

Never used.

Details

Identifier 0x0E00; property type PT_I8; property tag 0x0E000014

Remarks

Do not use this property. It is not supported in MAPI 1.0.

See Also

PR_CREATION_VERSION property, PR_MODIFY_VERSION property

PR_DEF_CREATE_DL

The PR_DEF_CREATE_DL property contains the template entry identifier for a default distribution list object.

Usage

Required on address book container objects.

Details

Identifier 0x3611; property type PT_BINARY; property tag 0x36110102

Remarks

Client applications use the PR_DEF_CREATE_DL property to create a distribution list within a container. Support of entry creation is optional for address book containers; those that do not support it are not required to expose this property.

PR_DEF_CREATE_DL specifies an entry that can appear in the PR_CREATE_TEMPLATES property for distribution lists. After obtaining the identifier, the client uses it in a call to the **IABContainer::CreateEntry** method. The entry represents the template for the default distribution list.

See Also

IABLogon::CompareEntryIDs method

PR_DEF_CREATE_MAILUSER

The PR_DEF_CREATE_MAILUSER property contains the template entry identifier for a default messaging user object.

Usage

Required on address book container objects.

Details

Identifier 0x3612; property type PT_BINARY; property tag 0x36120102

Remarks

Client applications use the PR_DEF_CREATE_MAILUSER property to create a messaging user object within a container. Support of entry creation is optional for address book containers; those that do not support it are not required to expose this property.

PR_DEF_CREATE_MAILUSER specifies an entry that can appear in the PR_CREATE_TEMPLATES property for messaging users. After obtaining the identifier, the client uses it in a call to the **IABContainer::CreateEntry** method. The entry represents the template for the default messaging user.

See Also

IABLogon::CompareEntryIDs method

PR_DEFAULT_PROFILE

The PR_DEFAULT_PROFILE property contains TRUE if a messaging user profile is the MAPI default profile.

Usage

Optional on profile section objects.

Details

Identifier 0x3D04; property type PT_BOOLEAN; property tag 0x3D04000B

Remarks

The PR_DEFAULT_PROFILE property does not appear as a property of any object but only as a column in a profile table. A client application can use the **IProfAdmin::SetDefaultProfile** method to designate the default profile.

See Also

PR_DEFAULT_STORE property

PR_DEFAULT_STORE

The PR_DEFAULT_STORE property contains TRUE if a message store is the default message store in the message store table.

Usage

Optional on message store objects.

Details

Identifier 0x3400; property type PT_BOOLEAN; property tag 0x3400000B

Remarks

The PR_DEFAULT_STORE property appears as a column in the message store table. The value is based on PR_RESOURCE_FLAGS.

See Also

PR_RESOURCE_FLAGS property

PR_DEFAULT_VIEW_ENTRYID

The PR_DEFAULT_VIEW_ENTRYID property contains the entry identifier of a folder's default view.

Usage

Optional on folder objects.

Details

Identifier 0x3616; property type PT_BINARY; property tag 0x36160102

Remarks

The PR_DEFAULT_VIEW_ENTRYID property is the entry identifier of the folder view that should be set as the initial view. The property need not be set if the "Normal" view is to be used as the initial view.

A client application can obtain PR_DEFAULT_VIEW_ENTRYID at the time it opens the folder and realize significant performance gains. PR_DEFAULT_VIEW_ENTRYID can be used as a shortcut to obtain the default view, instead of opening the associated contents table and submitting a restriction.

A service provider implementation of the **IMAPIFolder::CopyFolder** method can copy this property when it copies folders.

See Also

PR_COMMON_VIEWS_ENTRYID property, PR_VIEWS_ENTRYID property

PR_DEFERRED_DELIVERY_TIME

The PR_DEFERRED_DELIVERY_TIME property contains the date and time at which a message sender wants a message delivered.

Usage

Optional on message objects.

Details

Identifier 0x000F; property type PT_SYSTIME; property tag 0x000F0040

Remarks

MAPI does not perform the deferred delivery; it is an option of the underlying messaging system to handle deferred delivery.

The PR_DEFERRED_DELIVERY_TIME property corresponds to the X.400 attribute MH_T_DEFERRED_DELIVERY_TIME.

PR_DELEGATION

The PR_DELEGATION property contains the converted value of the attDelegate workgroup property.

Usage

Optional on message objects used in scheduling.

Details

Identifier 0x007E; property type PT_BINARY; property tag 0x007E0102

Remarks

The PR_DELEGATION property is used for compatibility with earlier versions of Schedule+. TNEF computes it on messages for Schedule+ 7.0 and compatible versions. Client applications should not use PR_DELEGATION.

In inbound messages to Schedule+ 7.0, the workgroup property attDelegate is decoded into PR_DELEGATION. TNEF constructs PR_DELEGATION in current recipient table order. If a client calling TNEF has provided a recipient table for use in the **ITnef::EncodeRecips** method, that table is used to build the delegation properties.

PR_DELEGATION is not encoded into attDelegate in outbound messages. Instead, attDelegate is built from the PR_RCVD_REPRESENTING_EMAIL_ADDRESS property if the message is generated by Schedule+, and not built at all otherwise.

PR_DELETE_AFTER_SUBMIT

The PR_DELETE_AFTER_SUBMIT property contains TRUE if a client application wants MAPI to delete the associated message after submission.

Usage

Optional on message objects.

Details

Identifier 0x0E01; property type PT_BOOLEAN; property tag 0x0E01000B

Remarks

A client application uses the PR_DELETE_AFTER_SUBMIT property with the PR_SENTMAIL_ENTRYID property to control what happens to a message after it is submitted. Either one or the other should be set, but not both.

PR_DELIVER_TIME

The PR_DELIVER_TIME property contains the date and time at which the original message was delivered.

Usage

Required on message objects used for delivery reports.

Details

Identifier 0x0010; property type PT_SYSTIME; property tag 0x00100040

Remarks

The PR_DELIVER_TIME property is a per-recipient property on a delivery report that indicates the time the original message was delivered to the messaging user for which the delivery report is being generated.

The PR_DELIVER_TIME property corresponds to the X.400 attribute MH_T_DELIVERY_TIME.

See Also

IMAPISupport::StatusRecips method

PR_DELIVERY_POINT

The PR_DELIVERY_POINT property specifies the nature of the functional entity by means of which a message was or would have been delivered to the recipient.

Usage

Required on message objects used on reports.

Details

Identifier 0x0C07; property type PT_LONG; property tag 0x0C070003

Remarks

The PR_DELIVERY_POINT property can have exactly one of the following values:

Value	**Description**
MAPI_MH_DP_ML	Delivered to a distribution list, a delivery point which in turn may distribute the message to many recipients.
MAPI_MH_DP_MS	Delivered to a message store instead of directly to a recipient.
MAPI_MH_DP_OTHER_AU	Delivered to an access unit (AU) other than a physical delivery access unit (PDAU), such as a FAX system.
MAPI_MH_DP_PDAU	Delivered to a physical delivery access unit, such as a human postal carrier.
MAPI_MH_DP_PDS_PATRON	Delivered to a physical delivery system patron, such as a conventional postal mailbox.
MAPI_MH_DP_PRIVATE_UA	Delivered to a private user agent (UA), such as a client in an in-house messaging system.
MAPI_MH_DP_PUBLIC_UA	Delivered to a public user agent, or public service provider.

The default value is MAPI_MH_DP_PRIVATE_UA, that is, a MAPI client.

PR_DELIVERY_POINT corresponds to the X.400 attribute MH_T_DELIVERY_POINT.

PR_DELTAX

The PR_DELTAX property contains the width of a dialog box control in standard Windows dialog units.

Usage

Required as a column entry in display tables.

Details

Identifier 0x3F03; property type PT_LONG; property tag 0x3F030003

Remarks

The PR_XPOS, PR_YPOS, PR_DELTAX, and PR_DELTAY properties position and size the dialog box control.

PR_DELTAY

The PR_DELTAY property contains the height of a dialog box control in standard Windows dialog units.

Usage

Required as a column entry in display tables.

Details

Identifier 0x3F04; property type PT_LONG; property tag 0x3F040003

Remarks

The PR_XPOS, PR_YPOS, PR_DELTAX, and PR_DELTAY properties position and size the dialog box control.

PR_DEPARTMENT_NAME

The PR_DEPARTMENT_NAME property contains a name for the department in which the recipient works.

Usage

Optional on messaging user objects.

Details

Identifier 0x3A18; property type PT_TSTRING; property tag 0x3A18001E (0x3A18001F for Unicode)

Remarks

The PR_DEPARTMENT_NAME property is one of the properties that provide identification and access information about a recipient. These properties are defined by the recipient and the recipient's organization. For more information on this group of properties, see About Messaging User Objects.

PR_DEPTH

The PR_DEPTH property represents the relative level of indentation, or depth, of an object in a hierarchy table.

Usage

Required as a column entry in contents tables and hierarchy tables.

Details

Identifier 0x3005; property type PT_LONG; property tag 0x30050003

Remarks

The PR_DEPTH property can also specify the categorization level of a row in a contents table or the hierarchy depth in a hierarchy table. The depth is zero-based, where zero represents the leftmost category. In all cases, the property value represents a relative value rather than an absolute value. In the hierarchy table, for example, the depth value is relative to the container from which the hierarchy table was retrieved. The depth does not represent an absolute depth from the root container.

See Also

PR_OBJECT_TYPE property, PR_SELECTABLE property

PR_DETAILS_TABLE

The PR_DETAILS_TABLE property contains an embedded display table object.

Usage

Optional on address book container, distribution list, and messaging user objects.

Details

Identifier 0x3605; property type PT_OBJECT; property tag 0x3605000D

Remarks

Passing the PR_DETAILS_TABLE property to the **IMAPIProp::OpenProperty** method for the object returns an **IMAPITable** interface that allows creation of the display table. MAPI uses this table to display property sheets for an address book object in response to an **IAddrBook::Details** call.

For more information about display tables, see About Display Tables.

See Also

IAddrBook::RecipOptions method, **IMAPISession::MessageOptions** method, PR_CREATE_TEMPLATES property, PR_SEARCH property

PR_DISC_VAL

The PR_DISC_VAL property is the obsolete precursor of the PR_DISCRETE_VALUES property.

Usage

No longer used.

Details

Identifier 0x004A; property type PT_BOOLEAN; property tag 0x004A000B

Remarks

Do not use this property. Use PR_DISCRETE_VALUES.

See Also

PR_DISCRETE_VALUES property

PR_DISCARD_REASON

The PR_DISCARD_REASON property contains a reason why a message transfer agent (MTA) has discarded a message.

Usage

Optional on message objects.

Details

Identifier 0x0011; property type PT_LONG; property tag 0x00110003

Remarks

The reason contained in the PR_DISCARD_REASON property is used in a nondelivery report for the message.

PR_DISCARD_REASON corresponds to the X.400 attribute IM_DISCARD_REASON.

See Also

PR_NDR_REASON_CODE property

PR_DISCLOSURE_OF_RECIPIENTS

The PR_DISCLOSURE_OF_RECIPIENTS property contains TRUE if disclosure of recipients is allowed.

Usage

Optional on message objects.

Details

Identifier 0x0012; property type PT_BOOLEAN; property tag 0x0012000B

Remarks

The PR_DISCLOSURE_OF_RECIPIENTS property corresponds to the X.400 attribute MH_T_DISCLOSURE_ALLOWED.

PR_DISCRETE_VALUES

The PR_DISCRETE_VALUES property contains TRUE if a nondelivery report applies only to discrete members of a distribution list rather than the entire list.

Usage

Optional on message recipient subobjects used on nondelivery reports.

Details

Identifier 0x0E0E; property type PT_BOOLEAN; property tag 0x0E0E000B

Remarks

The PR_DISCRETE_VALUES property is used within a nondelivery report when the message could not be delivered to one or more members of a distribution list. Its purpose is to limit retransmission attempts to only those individual members and not the distribution list as a whole.

The recipient table of a nondelivery report contains entries for all recipients the message could not be delivered to, and also for the distribution lists, if any, to which they belong. The transport provider should set PR_DISCRETE_VALUES to TRUE for each distribution list entry, and it should copy PR_DISPLAY_NAME, PR_ENTRYID, and PR_SEARCH_KEY from the distribution list to PR_ORIGINAL_DISPLAY_NAME, PR_ORIGINAL_ENTRYID, and PR_ORIGINAL_SEARCH_KEY for each member of that distribution list.

PR_DISCRETE_VALUES should not be set at all for any nondelivery report recipient entry other than a distribution list.

PR_DISPLAY_BCC

The PR_DISPLAY_BCC property contains an ASCII list of the display names of any blind carbon copy (BCC) message recipients, separated by semicolons (;).

Usage

Computed by message stores on message objects.

Details

Identifier 0x0E02; property type PT_TSTRING; property tag 0x0E02001E (0x0E02001F for Unicode)

Remarks

The message store computes the PR_DISPLAY_BCC property on message objects using the **IMessage::ModifyRecipients** method. The message store also maintains the property so that it always reflects the last saved state of a message. The value is synchronized at the time of every call to the **IMAPIProp::SaveChanges** method.

If a message has no blind carbon copy recipients, the message store should respond to an **IMAPIProp::GetProps** call with a return value of S_OK and an empty string for PR_DISPLAY_BCC.

Note Semicolons cannot be used within recipient names in MAPI messaging.

PR_DISPLAY_BCC corresponds to the X.400 attribute IM_BLIND_COPY_RECIPIENTS.

PR_DISPLAY_CC

The PR_DISPLAY_CC property contains an ASCII list of the display names of any carbon copy (CC) message recipients, separated by semicolons (;).

Usage

Computed by message stores on message objects.

Details

Identifier 0x0E03; property type PT_TSTRING; property tag 0x0E03001E (0x0E03001F for Unicode)

Remarks

The message store computes the PR_DISPLAY_CC property on message objects using the **IMessage::ModifyRecipients** method. The message store also maintains the property so that it always reflects the last saved state of a message. The value is synchronized at the time of every call to **IMAPIProp::SaveChanges**.

If a message has no carbon copy recipients, the message store should respond to an **IMAPIProp::GetProps** call with a return value of S_OK and an empty string for PR_DISPLAY_CC.

Note Semicolons cannot be used within recipient names in MAPI messaging.

PR_DISPLAY_CC corresponds to the X.400 attribute IM_COPY_RECIPIENTS.

PR_DISPLAY_NAME

The PR_DISPLAY_NAME property contains the display name for a given MAPI object.

Usage

Required on folder, messaging user, provider, and status objects.

Details

Identifier 0x3001; property type PT_TSTRING; property tag 0x3001001E (0x3001001F for Unicode)

Remarks

Folders require sibling subfolders to have unique display names. For example, if a folder contains two subfolders, the two subfolders cannot use the same value for their PR_DISPLAY_NAME properties. This restriction does not apply to other containers, such as address books and distribution lists.

PR_DISPLAY_NAME is one of the base address properties for all messaging users. For more information on the base address properties, see About Base Address Properties.

Service providers should set the value of PR_DISPLAY_NAME so that it contains both the provider type and configuration information. The additional information helps to distinguish between instances of providers of the same type. Unconfigured providers should use a string that names the provider. Configured providers should use the same string followed by a distinguishing string in parentheses. For example, an unconfigured message store provider might set PR_DISPLAY_NAME to Personal Information Store, while the configured version could set PR_DISPLAY_NAME to Personal Information Store (April 16, 1996).

For status objects, this property contains the name of the component that can be displayed by the user interface.

Note Semicolons cannot be used within recipient names in MAPI messaging.

PR_DISPLAY_NAME corresponds to the X.400 attribute IM_FREE_FORM_NAME.

See Also

PR_TRANSMITTABLE_DISPLAY_NAME property

PR_DISPLAY_TO

The PR_DISPLAY_TO property contains an ASCII list of the display names of the primary (To) message recipients, separated by semicolons (;).

Usage

Computed by message stores on message objects.

Details

Identifier 0x0E04; property type PT_TSTRING; property tag 0x0E04001E (0x0E04001F for Unicode)

Remarks

The message store computes the PR_DISPLAY_TO property on message objects using the **IMessage::ModifyRecipients** method. The message store also maintains the property so that it always reflects the last saved state of a message. The value is synchronized at the time of every call to the **IMAPIProp::SaveChanges** method.

If a message has no primary recipients, the message store should respond to an **IMAPIProp::GetProps** call with a return value of S_OK and an empty string for PR_DISPLAY_TO.

Note Semicolons cannot be used within recipient names in MAPI messaging.

PR_DISPLAY_TO corresponds to the X.400 attribute IM_PRIMARY_RECIPIENTS.

PR_DISPLAY_TYPE

The PR_DISPLAY_TYPE property contains a value used to associate an icon with a particular row of a table.

Usage

Required as a column entry in address book contents tables, address book hierarchy tables, folder hierarchy tables, and one-off tables.

Details

Identifier 0x3900; property type PT_LONG; property tag 0x39000003

Remarks

The PR_DISPLAY_TYPE property contains a long integer that facilitates special treatment of the table entry based on its type. This special treatment typically consists of displaying an icon, or other display element, associated with the display type.

PR_DISPLAY_TYPE is not used in folder contents tables. Client applications should use a message's PR_MESSAGE_CLASS property and appropriate **IMAPIFormInfo** interface to get the PR_ICON and PR_MINI_ICON properties for that message.

PR_DISPLAY_TYPE can have exactly one of the following values:

Value	Description
DT_AGENT	An automated agent, such as Quote-Of-The-Day or a weather chart display.
DT_DISTLIST	A distribution list.
DT_FOLDER	Display default folder icon adjacent to folder.
DT_FOLDER_LINK	Display default folder link icon adjacent to folder rather than the default folder icon.
DT_FOLDER_SPECIAL	Display icon for a folder with an application-specific distinction, such as a special type of public folder.
DT_FORUM	A forum, such as a bulletin board service or a public or shared folder.
DT_GLOBAL	A global address book.
DT_LOCAL	A local address book that you share with a small workgroup.
DT_MAILUSER	A typical messaging user.
DT_MODIFIABLE	Modifiable; the container should be denoted as modifiable in the user interface.
DT_NOT_SPECIFIC	Does not match any of the other settings.
DT_ORGANIZATION	A special alias defined for a large group, such as helpdesk, accounting, or blood-drive

	coordinator.
DT_PRIVATE_DISTLIST	A private, personally administered distribution list.
DT_REMOTE_MAILUSER	A recipient known to be from a foreign or remote messaging system.
DT_WAN	A wide area network address book.

Address book contents tables use the DT_AGENT, DT_DISTLIST, DT_FORUM, DT_MAILUSER, DT_ORGANIZATION, DT_PRIVATE_DISTLIST, and DT_REMOTE_MAILUSER values. Address book hierarchy tables and one-off tables use the DT_GLOBAL, DT_LOCAL, DT_MODIFIABLE, DT_NOT_SPECIFIC, and DT_WAN values. Folder hierarchy tables use the DT_FOLDER, DT_FOLDER_LINK, and DT_FOLDER_SPECIAL values.

If PR_DISPLAY_TYPE is not set, the client should assume the default type appropriate for the table, typically DT_FOLDER, DT_LOCAL, or DT_MAILUSER.

Note All values not documented are reserved for MAPI. Client applications must not define any new values and must be prepared to deal with an undocumented value.

PR_DL_EXPANSION_HISTORY

The PR_DL_EXPANSION_HISTORY property contains a history showing how a distribution list has been expanded during message transmission.

Usage

Optional on message objects.

Details

Identifier 0x0013; property type PT_BINARY; property tag 0x00130102

Remarks

The PR_DL_EXPANSION_HISTORY property is available to receiving client applications if the transport provider has set it. It is also available to the sending client if the message content is returned with a report.

PR_DL_EXPANSION_HISTORY corresponds to the X.400 attribute MH_T_EXPANSION_HISTORY.

See Also

PR_DL_EXPANSION_PROHIBITED property

PR_DL_EXPANSION_PROHIBITED

The PR_DL_EXPANSION_PROHIBITED property contains TRUE if a message transfer agent (MTA) is prohibited from expanding distribution lists.

Usage

Optional on message objects.

Details

Identifier 0x0014; property type PT_BOOLEAN; property tag 0x0014000B

Remarks

The PR_DL_EXPANSION_PROHIBITED property corresponds to the X.400 attribute MH_T_EXPANSION_PROHIBITED.

See Also

PR_DL_EXPANSION_HISTORY property

PR_EMAIL_ADDRESS

The PR_EMAIL_ADDRESS property contains the messaging user's e-mail address.

Usage

Required on distribution list and messaging user objects.

Details

Identifier 0x3003; property type PT_TSTRING; property tag 0x3003001E (0x3003001F for Unicode)

Remarks

The PR_EMAIL_ADDRESS property is one of the base address properties for all messaging users. It is a null-terminated string whose format has meaning only for the underlying messaging system.

PR_EMAIL_ADDRESS is used in conjunction with the PR_ADDRTYPE and PR_MESSAGE_CLASS properties in addressing messages. The string format is qualified by PR_ADDRTYPE.

Valid PR_EMAIL_ADDRESS examples include:

```
network/postoffice/user
Bruce@XYZZY.COM
/c=US/a=att/p=Microsoft/o=Finance/ou=Purchasing/s=Furthur/g=Joe
```

For more information on the base address properties, see About Base Address Properties. For more information on address types, see Address Types.

PR_END_DATE

The PR_END_DATE property contains the ending date and time of an appointment as managed by a scheduling application.

Usage

Required on message objects used in scheduling.

Details

Identifier 0x0061; property type PT_SYSTIME; property tag 0x00610040

Remarks

Scheduling applications should set both the PR_START_DATE and PR_END_DATE properties when sending meeting requests.

PR_ENTRYID

The PR_ENTRYID property contains a MAPI entry identifier used to open and edit properties of a particular MAPI object.

Usage

Required on address book container, distribution list, folder, messaging user, message, message store, and status objects.

Details

Identifier 0x0FFF; property type PT_BINARY; property tag 0x0FFF0102

Remarks

The PR_ENTRYID property identifies an object for **OpenEntry** to instantiate and provides access to all of its properties through the appropriate derived interface of **IMAPIProp**.

PR_ENTRYID is one of the base address properties for all messaging users. For more information on the base address properties, see About Base Address Properties.

PR_ENTRYID can contain either a long-term or a short-term identifier. Short-term identifiers are easier and faster to construct, but are limited in their scope and duration, typically to the current session and workstation. They are commonly used for objects of a temporary nature, such as table rows or dialog box entries, and then abandoned. Long-term identifiers are used for objects of a more wide-ranging and long-lasting nature.

PR_ENTRYID is always available through the **IMAPIProp::GetProps** method following the first call to the **IMAPIProp::SaveChanges** method. Some service providers can make it available immediately after instantiation. The provider must always return a long-term entry identifier from **GetProps**. Therefore, to convert a short-term identifier to long-term, simply open the object and get its PR_ENTRYID through **GetProps**.

The following table summarizes important differences among PR_ENTRYID, PR_RECORD_KEY, and PR_SEARCH_KEY.

Characteristic	**PR_ENTRYID**	**PR_RECORD_KEY**	**PR_SEARCH_KEY**
Required on attachment objects	No	Yes	No
Required on folder objects	Yes	Yes	No
Required on message store objects	Yes	Yes	No
Required on status objects	Yes	No	No
Created by client	No	No	Yes
Available before call to **SaveChanges**	Maybe	Maybe	Messages – Yes Others – Maybe
Changed in a copy operation	Yes	Yes	No

Changeable by client after a copy	No	No	Yes
Unique within	Entire world	Provider instance	Entire world
Binary comparable (as with memcmp)	No – use **IMAPISupport::CompareEntryIDs**	Yes	Yes

See Also

ENTRYID structure, **NOTIFICATION** structure, PR_STORE_ENTRYID property

PR_EXPIRY_TIME

The PR_EXPIRY_TIME property contains the date and time at which the messaging system can invalidate the content of a message.

Usage

Optional on message objects.

Details

Identifier 0x0015; property type PT_SYSTIME; property tag 0x00150040

Remarks

The PR_EXPIRY_TIME property is used to direct the messaging system in handling delivered interpersonal messages.

PR_EXPIRY_TIME corresponds to the X.400 attribute IM_EXPIRY_TIME.

PR_EXPLICIT_CONVERSION

The PR_EXPLICIT_CONVERSION property indicates that a message sender has requested a message content conversion for a particular recipient.

Usage

Optional on recipient subobjects.

Details

Identifier 0x0C01; property type PT_LONG; property tag 0x0C010003

Remarks

The PR_EXPLICIT_CONVERSION property corresponds to the X.400 attribute MH_T_EXPLICIT_CONVERSION.

PR_FILTERING_HOOKS

The PR_FILTERING_HOOKS property was originally meant to contain hook identifiers.

Usage

Never used.

Details

Identifier 0x3D08; property type PT_BINARY; property tag 0x3D080102

Remarks

Do not use this property. It is not supported in MAPI 1.0.

PR_FINDER_ENTRYID

The PR_FINDER_ENTRYID property contains the entry identifier for the folder in which search results are typically created.

Usage

Required on message store objects.

Details

Identifier 0x35E7; property type PT_BINARY; property tag 0x35E70102

Remarks

The entry identifier contained in the PR_FINDER_ENTRYID property has the same format as the **ENTRYID** structure.

See Also

PR_ENTRYID property

PR_FOLDER_ASSOCIATED_CONTENTS

The PR_FOLDER_ASSOCIATED_CONTENTS property contains an embedded contents table object that provides information about the associated contents table.

Usage

Required on folder objects.

Details

Identifier 0x3610; property type PT_OBJECT; property tag 0x3610000D

Remarks

The associated contents table represents a subfolder that does not appear in the standard contents table. It contains the folder's associated, or hidden, messages.

The PR_FOLDER_ASSOCIATED_CONTENTS property can be excluded in **IMAPIProp::CopyTo** operations or included in **IMAPIProp::CopyProps** operations. As a property of type PT_OBJECT, it cannot be successfully retrieved by the **IMAPIProp::GetProps** method; its contents should be accessed by the **IMAPIProp::OpenProperty** method, requesting the IID_IMAPITable interface identifier. Service providers must report it to the **IMAPIProp::GetPropList** method if it is set, but may optionally report it or not if it is not set.

To retrieve table contents, client applications should call the **IMAPIContainer::GetContentsTable** method. For more information on folder contents tables, see About Folder Contents Tables.

The PR_CONTAINER_CONTENTS, PR_CONTAINER_HIERARCHY, and PR_FOLDER_ASSOCIATED_CONTENTS properties are similar in usage. Several MAPI properties provide access to tables:

Property	Table
PR_CONTAINER_CONTENTS	Contents table
PR_CONTAINER_HIERARCHY	Hierarchy table
PR_FOLDER_ASSOCIATED_CONTENTS	Associated contents table
PR_MESSAGE_ATTACHMENTS	Attachment table
PR_MESSAGE_RECIPIENTS	Recipient table

See Also

PR_ASSOC_CONTENT_COUNT property

PR_FOLDER_TYPE

The PR_FOLDER_TYPE property contains a constant that indicates the folder type.

Usage

Required on folder objects.

Details

Identifier 0x3601; property type PT_LONG; property tag 0x36010003

Remarks

The PR_FOLDER_TYPE property can have exactly one of the following values:

Value	Description
FOLDER_GENERIC	A generic folder that contains messages and other folders.
FOLDER_ROOT	The root folder of the folder hierarchy table, that is, a folder that has no parent folder.
FOLDER_SEARCH	A folder containing the results of a search, in the form of links to messages that meet search criteria.

The root of a message store should not be confused with the root of the interpersonal message (IPM) subtree in that store. The store's root folder, which has no parent, is obtained by calling the **IMsgStore::OpenEntry** method with a null entry identifier. The IPM subtree's root folder, which does have a parent, is obtained by using the value of the PR_IPM_SUBTREE_ENTRYID property for the **OpenEntry** call.

MAPI allows only one root folder per message store. This folder contains messages and other folders. The root folder's PR_PARENT_ENTRYID property contains the folder's own entry identifier.

The information in a search-results folder is mainly stored in its contents table, which contains the same columns as a typical contents table, as well as two extra columns identifying the folder in which each message was found: PR_PARENT_DISPLAY (display name, required) and PR_PARENT_ENTRYID (entry identifier, optional).

For more information on folder types, see About Types of Folders.

PR_FORM_CATEGORY

The PR_FORM_CATEGORY property contains the category of a form.

Usage

Optional on form objects.

Details

Identifier 0x3309; property type PT_TSTRING; property tag 0x3309001E (0x3309001F for Unicode)

Remarks

The category name is defined by a client application in a way that is appropriate to the application.

PR_FORM_CATEGORY_SUB

The PR_FORM_CATEGORY_SUB property contains the subcategory of a form, as defined by a client application.

Usage

Optional on form objects.

Details

Identifier 0x3310; property type PT_TSTRING; property tag 0x3310001E (0x3310001F for Unicode)

Remarks

The PR_FORM_CATEGORY_SUB property is subordinate to the main form category provided in the PR_FORM_CATEGORY property.

PR_FORM_CLSID

The PR_FORM_CLSID property contains the 128-bit OLE globally unique identifier (GUID) of a form.

Usage

Required on form objects.

Details

Identifier 0x3307; property type PT_CLSID; property tag 0x33070048

Remarks

The **MAPIUID** structure contains the definition of the unique identifier.

PR_FORM_CONTACT_NAME

The PR_FORM_CONTACT_NAME property contains the name of a contact for information concerning a form.

Usage

Optional on form objects.

Details

Identifier 0x3308; property type PT_TSTRING; property tag 0x3308001E (0x3308001F for Unicode)

Remarks

The contact typically contains the name of a person or an alias that is responsible for maintaining the form.

PR_FORM_DESIGNER_GUID

The PR_FORM_DESIGNER_GUID property contains the unique identifier for the object used to design a form.

Usage

Optional on form objects.

Details

Identifier 0x3314; property type PT_CLSID; property tag 0x33140048

Remarks

The PR_FORM_DESIGNER_GUID property usually contains the globally unique identifier of the design program used to create the form. This property can be empty.

The **MAPIUID** structure contains the definition of the unique identifier.

PR_FORM_DESIGNER_NAME

The PR_FORM_DESIGNER_NAME property contains the display name for the object used to design the form.

Usage

Optional on form objects.

Details

Identifier 0x3313; property type PT_TSTRING; property tag 0x3313001E (0x3313001F for Unicode)

Remarks

The PR_FORM_DESIGNER_GUID property contains the unique identifier for the form designer object.

PR_FORM_HIDDEN

The PR_FORM_HIDDEN property contains TRUE if a form is to be suppressed from display by compose menus and dialog boxes.

Usage

Optional on form objects.

Details

Identifier 0x3312; property type PT_BOOLEAN; property tag 0x3312000B

Remarks

Form-related properties are read-only.

PR_FORM_HOST_MAP

The PR_FORM_HOST_MAP property contains a host map of available forms.

Usage

Optional on address book container objects.

Details

Identifier 0x3311; property type PT_LONG; property tag 0x33110003

Remarks

A client application should update the PR_FORM_HOST_MAP property, along with the PR_DISPLAY_NAME property, when changing the underlying structure in the **IMAPIFormProp** interface.

PR_FORM_MESSAGE_BEHAVIOR

The PR_FORM_MESSAGE_BEHAVIOR property contains TRUE if a message should be composed in the current folder.

Usage

Optional on form objects.

Details

Identifier 0x3315; property type PT_BOOLEAN; property tag 0x3315000B

Remarks

A value of FALSE indicates that the message should be composed as any other interpersonal message, that is, in the Outbox folder.

PR_FORM_VERSION

The PR_FORM_VERSION property contains the version of a form.

Usage

Optional on form objects.

Details

Identifier 0x3306; property type PT_TSTRING; property tag 0x3306001E (0x3306001F for Unicode)

Remarks

The form version is a string indicating what design version is currently in effect for the form. The version is defined and maintained by the form's designer and is not necessarily related to any MAPI component version.

PR_GENERATION

The PR_GENERATION property contains a generational abbreviation that follows the full name of the recipient.

Usage

Optional on messaging user objects.

Details

Identifier 0x3A05; property type PT_TSTRING; property tag 0x3A05001E (0x3A05001F for Unicode)

Remarks

The PR_GENERATION property is one of the properties that provide identification and access information about a recipient. These properties are defined by the recipient and the recipient's organization. For more information on this group of properties, see About Messaging User Objects.

Common PR_GENERATION values include Jr., Sr., and III.

PR_GENERATION corresponds to the X.400 attribute MH_T_GENERATION.

PR_GIVEN_NAME

The PR_GIVEN_NAME property contains the first name of the recipient.

Usage

Optional on messaging user objects.

Details

Identifier 0x3A06; property type PT_TSTRING; property tag 0x3A06001E (0x3A06001F for Unicode)

Remarks

The PR_GIVEN_NAME property is one of the properties that provide identification and access information about a recipient. These properties are defined by the recipient and the recipient's organization. For more information on this group of properties, see About Messaging User Objects.

PR_GIVEN_NAME corresponds to the X.400 attribute MH_T_GIVEN_NAME.

PR_GOVERNMENT_ID_NUMBER

The PR_GOVERNMENT_ID_NUMBER property contains a government identifier for the recipient.

Usage

Optional on messaging user objects.

Details

Identifier 0x3A07; property type PT_TSTRING; property tag 0x3A07001E (0x3A07001F for Unicode)

Remarks

The PR_GOVERNMENT_ID_NUMBER property is one of the properties that provide identification and access information about a recipient. These properties are defined by the recipient and the recipient's organization. For more information on this group of properties, see About Messaging User Objects.

PR_GOVERNMENT_ID_NUMBER is commonly set to a passport number, citizen number, or taxpayer number such as a Social Security number.

PR_HASATTACH

The PR_HASATTACH property contains TRUE if a message contains at least one attachment.

Usage

Required on message objects.

Details

Identifier 0x0E1B; property type PT_BOOLEAN; property tag 0x0E1B000B

Remarks

The message store copies the PR_HASATTACH property from the MSGFLAG_HASATTACH flag of the PR_MESSAGE_FLAGS property. A client application can then use PR_HASATTACH to sort on message attachments in a message viewer.

The value of PR_HASATTACH is updated with the **IMAPIProp::SaveChanges** method.

PR_HEADER_FOLDER_ENTRYID

The PR_HEADER_FOLDER_ENTRYID property was originally meant to contain the entry identifier that a remote transport provider furnishes for its header folder.

Usage

Never used.

Details

Identifier 0x3E0A; property type PT_BINARY; property tag 0x3E0A0102

Remarks

Do not use this property. It is not supported in MAPI 1.0.

For information on obtaining the header folder for the purpose of viewing remote message headers, see Implementing the **IUnknown** Interface for Folder Objects for Remote Transports.

See Also

PR_ENTRYID property, PR_MSG_STATUS property

PR_HOME_FAX_NUMBER

The PR_HOME_FAX_NUMBER property contains the telephone number of the recipient's home fax machine.

Usage

Optional on messaging user objects.

Details

Identifier 0x3A25; property type PT_TSTRING; property tag 0x3A25001E (0x3A25001F for Unicode)

Remarks

The PR_HOME_FAX_NUMBER property is one of the properties that provide identification and access information about a recipient. These properties are defined by the recipient and the recipient's organization. For more information on this group of properties, see About Messaging User Objects.

PR_HOME_TELEPHONE_NUMBER

The PR_HOME_TELEPHONE_NUMBER property contains the primary telephone number of the recipient's home.

Usage

Optional on messaging user objects.

Details

Identifier 0x3A09; property type PT_TSTRING; property tag 0x3A09001E (0x3A09001F for Unicode)

Remarks

The PR_HOME_TELEPHONE_NUMBER property is one of the properties that provide identification and access for a recipient. These properties are defined by the recipient and the recipient's organization. For more information on this group of properties, see About Messaging User Objects.

See Also

PR_HOME2_TELEPHONE_NUMBER property

PR_HOME2_TELEPHONE_NUMBER

The PR_HOME2_TELEPHONE_NUMBER property contains a secondary telephone number at the recipient's home.

Usage

Optional on messaging user objects.

Details

Identifier 0x3A2F; property type PT_TSTRING; property tag 0x3A2F001E (0x3A2F001F for Unicode)

Remarks

The PR_HOME2_TELEPHONE_NUMBER property is one of the properties that provide identification and access information for a recipient. These properties are defined by the recipient and the recipient's organization. For more information on this group of properties, see About Messaging User Objects.

See Also

PR_HOME_TELEPHONE_NUMBER property

PR_ICON

The PR_ICON property contains a bitmap of a full size icon for a form.

Usage

Required on form objects.

Details

Identifier 0x0FFD; property type PT_BINARY; property tag 0x0FFD0102

Remarks

The PR_ICON property contains a 32 × 32 pixel image of an icon, the same as the contents of a .ICO file. This property is normally copied from the .ICO file specified in the LargeIcon line of the appropriate [Description] section of the form configuration file.

See Also

PR_MINI_ICON property

PR_IDENTITY_DISPLAY

The PR_IDENTITY_DISPLAY property contains the display name for a service provider's identity as defined within a messaging system.

Usage

Optional as a column entry in status tables.

Details

Identifier 0x3E00; property type PT_TSTRING; property tag 0x3E00001E (0x3E00001F for Unicode)

Remarks

The PR_IDENTITY_DISPLAY property does not appear as a property on any object but only as a column in a status table. It is part of the identity of the service provider exposing the status table row. The provider's identity typically refers to its account on the server, but can refer to any representation the provider defines within the messaging system.

A service provider furnishing any of the identity properties should furnish all of them. Providers that belong to the same message service should expose the same values for the identity properties.

See Also

IMAPISession::QueryIdentity method

PR_IDENTITY_ENTRYID

The PR_IDENTITY_ENTRYID property contains the entry identifier for a service provider's identity as defined within a messaging system.

Usage

Optional as a column entry in status tables.

Details

Identifier 0x3E01; property type PT_BINARY; property tag 0x3E010102

Remarks

The PR_IDENTITY_ENTRYID property does not appear as a property on any object but only as a column in a status table. It is part of the identity of the service provider exposing the status table row. The provider's identity typically refers to its account on the server, but can refer to any representation the provider defines within the messaging system.

PR_IDENTITY_ENTRYID is commonly set to the appropriate address book entry identifier.

A service provider furnishing any of the identity properties should furnish all of them. Providers that belong to the same message service should expose the same values for the identity properties.

See Also

IMAPISession::QueryIdentity method

PR_IDENTITY_SEARCH_KEY

The PR_IDENTITY_SEARCH_KEY property contains the search key for a service provider's identity as defined within a messaging system.

Usage

Optional as a column entry in status tables.

Details

Identifier 0x3E05; property type PT_BINARY; property tag 0x3E050102

Remarks

The PR_IDENTITY_SEARCH_KEY property does not appear as a property on any object but only as a column in a status table. It is part of the identity of the service provider exposing the status table row. The provider's identity typically refers to its account on the server, but can refer to any representation the provider defines within the messaging system.

A service provider furnishing any of the identity properties should furnish all of them. Providers that belong to the same message service should expose the same values for the identity properties.

See Also

IMAPISession::QueryIdentity method

PR_IMPLICIT_CONVERSION_PROHIBITED

The PR_IMPLICIT_CONVERSION_PROHIBITED property contains TRUE if a message transfer agent (MTA) is prohibited from making implicit message text conversions.

Usage

Optional on message objects.

Details

Identifier 0x0016; property type PT_BOOLEAN; property tag 0x0016000B

Remarks

If the PR_IMPLICIT_CONVERSION_PROHIBITED property is TRUE, the messaging system must not perform any content conversion on the message unless it is explicitly requested on a per-recipient basis with with PR_EXPLICIT_CONVERSION property.

PR_IMPORTANCE

The PR_IMPORTANCE property contains a value indicating the message sender's opinion of the importance of a message.

Usage

Optional on message objects.

Details

Identifier 0x0017; property type PT_LONG; property tag 0x00170003

Remarks

The PR_IMPORTANCE and PR_PRIORITY properties should not be confused. Importance indicates a value to users, while priority indicates the order or speed at which the message should be sent by the messaging system software. Higher priority usually indicates a higher cost. Higher importance usually is associated with a different display by the user interface.

PR_IMPORTANCE can have exactly one of the following values:

Value	Description
IMPORTANCE_LOW	The message has low importance.
IMPORTANCE_HIGH	The message has high importance.
IMPORTANCE_NORMAL	The message has normal importance.

PR_IMPORTANCE corresponds to the X.400 attribute IM_IMPORTANCE.

PR_INCOMPLETE_COPY

The PR_INCOMPLETE_COPY property contains TRUE if this message is an incomplete copy of another message.

Usage

Optional on message objects.

Details

Identifier 0x0035; property type PT_BOOLEAN; property tag 0x0035000B

Remarks

The PR_INCOMPLETE_COPY property corresponds to the X.400 attribute IM_INCOMPLETE_COPY.

PR_INITIAL_DETAILS_PANE

The PR_INITIAL_DETAILS_PANE property contains the property page in a property sheet to be displayed first.

Usage

Required as a column entry in display tables.

Details

Identifier 0x3F08; property type PT_LONG; property tag 0x3F080003

Remarks

The PR_INITIAL_DETAILS_PANE property contains the zero-based index of the initial foreground pane that appears when a client application displays a details table for an object that exposes property sheets. The value indicates which page should be displayed first. This property appears in an object from which a display table can be obtained.

See Also

IMAPISupport::DoConfigPropSheet method

PR_INITIALS

The PR_INITIALS property contains the initials for parts of the full name of the recipient.

Usage

Optional on messaging user objects.

Details

Identifier 0x3A0A; property type PT_TSTRING; property tag 0x3A0A001E (0x3A0A001F for Unicode)

Remarks

The PR_INITIALS property is one of the properties that provide identification and access information for a recipient. These properties are defined by the recipient and the recipient's organization. For more information on this group of properties, see About Messaging User Objects.

Examples of PR_INITIALS values are AJT, GMcD, JL, RCKJr, and WMSIII.

PR_INITIALS corresponds to the X.400 attribute MH_T_INITIALS.

PR_INSTANCE_KEY

The PR_INSTANCE_KEY property contains a value that uniquely identifies a row in a table.

Usage

Required as a column entry in all tables.

Details

Identifier 0x0FF6; property type PT_BINARY; property tag 0x0FF60102

Remarks

When a multivalued property is expanded in a table, a row is created for each instance of the expansion, that is, for each value of that property. Each row has a unique value for the PR_INSTANCE_KEY property, while all the other columns retain their original values throughout the expansion.

Use the PR_ENTRYID or PR_RECORD_KEY properties to correlate all the rows of an expansion. Use PR_INSTANCE_KEY to locate a particular instance within the expansion.

In a categorized sort of a table, rows not corresponding to actual data can be added to the result of the sort. Each such row, like all rows in all tables, has its own unique instance key.

PR_INSTANCE_KEY is also used in table event notifications. The **propIndex** and **propPrior** members of the **TABLE_NOTIFICATION** structure are **SPropValue** structures holding PR_INSTANCE_KEY values. The **propIndex** member indicates the row that was added or changed. The **propPrior** member indicates the row before the added or changed row (PR_NULL indicates a change to the first row).

This value is not copied as part of the display table.

PR_INSTANCE_KEY is a **MAPIUID** structure. All instance keys can be directly compared as binary values.

PR_IPM_ID

The PR_IPM_ID property was originally meant to contain an X.400 identifier of an interpersonal message.

Usage

Never used.

Details

Identifier 0x0018; property type PT_BINARY; property tag 0x00180102

Remarks

Do not use this property. It is not supported in MAPI 1.0.

PR_IPM_OUTBOX_ENTRYID

The PR_IPM_OUTBOX_ENTRYID property contains the entry identifier of the standard interpersonal message (IPM) Outbox folder.

Usage

Optional on message store objects.

Details

Identifier 0x35E2; property type PT_BINARY; property tag 0x35E20102

Remarks

Outbound messages are usually created in the Outbox folder. Interpersonal messages should be placed in this folder for submission.

See Also

PR_ENTRYID property

PR_IPM_OUTBOX_SEARCH_KEY

The PR_IPM_OUTBOX_SEARCH_KEY property was originally meant to contain the search key of the standard Outbox folder.

Usage

Never used.

Details

Identifier 0x3411; property type PT_BINARY; property tag 0x34110102

Remarks

Do not use this property. It is not supported in MAPI 1.0.

PR_IPM_RETURN_REQUESTED

The PR_IPM_RETURN_REQUESTED property contains TRUE if this message should be returned with a report.

Usage

Optional on message objects.

Details

Identifier 0x0C02; property type PT_BOOLEAN; property tag 0x0C02000B

Remarks

The PR_IPM_RETURN_REQUESTED property corresponds to the X.400 attribute IM_IPM_RETURN_REQUESTED.

PR_IPM_SENTMAIL_ENTRYID

The PR_IPM_SENTMAIL_ENTRYID property contains the entry identifier of the standard interpersonal message (IPM) Sent Items folder.

Usage

Optional on message store objects.

Details

Identifier 0x35E4; property type PT_BINARY; property tag 0x35E40102

Remarks

After being sent, interpersonal messages are usually placed in the Sent Items folder. A client can use the PR_IPM_SENTMAIL_ENTRYID property to set the PR_SENTMAIL_ENTRYID property on a submitted message.

See Also

PR_ENTRYID property

PR_IPM_SENTMAIL_SEARCH_KEY

The PR_IPM_SENTMAIL_SEARCH_KEY property was originally meant to contain the search key of the standard Sent Items folder.

Usage

Never used.

Details

Identifier 0x3413; property type PT_BINARY; property tag 0x34130102

Remarks

Do not use this property. It is not supported in MAPI 1.0.

PR_IPM_SUBTREE_ENTRYID

The PR_IPM_SUBTREE_ENTRYID property contains the entry identifier of the root of the interpersonal message (IPM) folder subtree in the message store's folder tree.

Usage

Required on message store objects.

Details

Identifier 0x35E0; property type PT_BINARY; property tag 0x35E00102

Remarks

The PR_IPM_SUBTREE_ENTRYID property represents the root of the IPM hierarchy. IPM clients should not display any folders that are not subfolders of the folder represented by PR_IPM_SUBTREE_ENTRYID.

See Also

PR_ENTRYID property

PR_IPM_SUBTREE_SEARCH_KEY

The PR_IPM_SUBTREE_SEARCH_KEY property was originally meant to contain the search key of the interpersonal message (IPM) root folder.

Usage

Never used.

Details

Identifier 0x3410; property type PT_BINARY; property tag 0x34100102

Remarks

Do not use this property. It is not supported in MAPI 1.0.

PR_IPM_WASTEBASKET_ENTRYID

The PR_IPM_WASTEBASKET_ENTRYID property contains the entry identifier of the standard interpersonal message (IPM) Deleted Items folder.

Usage

Required on message store objects.

Details

Identifier 0x35E3; property type PT_BINARY; property tag 0x35E30102

Remarks

A client application should move deleted interpersonal messages to the Deleted Items folder. If the message is already in this folder, or if the PR_IPM_WASTEBASKET_ENTRYID property is not supported, the client should delete the message.

See Also

PR_ENTRYID property

PR_IPM_WASTEBASKET_SEARCH_KEY

The PR_IPM_WASTEBASKET_SEARCH_KEY property was originally meant to contain the search key of the standard Deleted Items folder.

Usage

Never used.

Details

Identifier 0x3412; property type PT_BINARY; property tag 0x34120102

Remarks

Do not use this property. It is not supported in MAPI 1.0.

PR_ISDN_NUMBER

The PR_ISDN_NUMBER property contains the recipient's ISDN-capable telephone number.

Usage

Optional on messaging user objects.

Details

Identifier 0x3A2D; property type PT_STRING; property tag 0x3A2D001E

Remarks

The PR_ISDN_NUMBER property is one of the properties that provide identification and access information for a recipient. These properties are defined by the recipient and the recipient's organization. For more information on this group of properties, see About Messaging User Objects.

PR_KEYWORD

The PR_KEYWORD property contains a keyword identifying the recipient to the recipient's system administrator.

Usage

Optional on messaging user objects.

Details

Identifier 0x3A0B; property type PT_TSTRING; property tag 0x3A0B001E (0x3A0B001F for Unicode)

Remarks

The PR_KEYWORD property is one of the properties that provide identification and access information for a recipient. These properties are defined by the recipient and the recipient's organization. For more information on this group of properties, see About Messaging User Objects.

The contents of the keyword string property are defined based on the needs of the recipient's organization.

PR_LANGUAGE

The PR_LANGUAGE property contains a value indicating the language in which the messaging user is writing messages.

Usage

Optional on messaging user objects.

Details

Identifier 0x3A0C; property type PT_TSTRING; property tag 0x3A0C001E (0x3A0C001F for Unicode)

Remarks

The string contains a single two-character country code.

See Also

PR_LANGUAGES property

PR_LANGUAGES

The PR_LANGUAGES property contains an ASCII list of the languages incorporated in a message.

Usage

Optional on message objects.

Details

Identifier 0x002F; property type PT_TSTRING; property tag 0x002F001E (0x002F001F for Unicode)

Remarks

The underlying string is a sequence of two-character country codes separated by commas.

The PR_LANGUAGES property corresponds to the X.400 attribute IM_LANGUAGES.

See Also

PR_LANGUAGE property

PR_LAST_MODIFICATION_TIME

The PR_LAST_MODIFICATION_TIME property contains the date and time the object or subobject was last modified.

Usage

Optional on all objects including attachment subobjects.

Details

Identifier 0x3008; property type PT_SYSTIME; property tag 0x30080040

Remarks

The PR_LAST_MODIFICATION_TIME property is initially set to the same value as the PR_CREATION_TIME property. Attachment subobjects can update it as necessary by copying the last modification time maintained by the native file system. A client applicatin can set this property until the first call to the **IMAPIProp::SaveChanges** method. From then on the provider should update PR_LAST_MODIFICATION_TIME during every **SaveChanges** call.

PR_LATEST_DELIVERY_TIME

The PR_LATEST_DELIVERY_TIME property contains the latest date and time when a message transfer agent (MTA) should deliver a message.

Usage

Optional on message objects.

Details

Identifier 0x0019; property type PT_SYSTIME; property tag 0x00190040

Remarks

If an MTA cannot deliver a message by the time the PR_LATEST_DELIVERY_TIME property specifies, it cancels the message without delivery.

PR_LATEST_DELIVERY_TIME corresponds to the X.400 attribute MH_T_LATEST_DELIVERY_TIME.

PR_LOCALITY

The PR_LOCALITY property contains the name of the recipient's locality, such as the town or city.

Usage

Optional on messaging user objects.

Details

Identifier 0x3A27; property type PT_TSTRING; property tag 0x3A27001E (0x3A27001F for Unicode)

Remarks

The PR_LOCALITY property is one of the properties that provide identification and access information for a recipient. These properties are defined by the recipient and the recipient's organization. For more information on this group of properties, see About Messaging User Objects.

PR_LOCATION

The PR_LOCATION property contains the location of the recipient in a format that is useful to the recipient's organization.

Usage

Optional on messaging user objects.

Details

Identifier 0x3A0D; property type PT_TSTRING; property tag 0x3A0D001E (0x3A0D001F for Unicode)

Remarks

The PR_LOCATION property is one of the properties that provide identification and access information for a recipient. These properties are defined by the recipient and the recipient's organization. For more information on this group of properties, see About Messaging User Objects.

The contents of PR_LOCATION are defined by the needs of the recipient's organization. For example, some organizations might identify messaging users by specifying the building number and office number.

PR_MAIL_PERMISSION

The PR_MAIL_PERMISSION property contains TRUE if the messaging user is allowed to send and receive messages.

Usage

Optional on messaging user objects.

Details

Identifier 0x3A0E; property type PT_BOOLEAN; property tag 0x3A0E000B

Remarks

If the PR_MAIL_PERMISSION property is not set, MAPI treats it as having a TRUE value.

An example of PR_MAIL_PERMISSION being set to FALSE is a corporate directory where some of the entries are not e-mail-enabled.

PR_MAPPING_SIGNATURE

The PR_MAPPING_SIGNATURE property contains the mapping signature for named properties of a particular MAPI object.

Usage

Optional but recommended on objects having named properties.

Details

Identifier 0x0FF8; property type PT_BINARY; property tag 0x0FF80102

Remarks

A client application should check the PR_MAPPING_SIGNATURE properties of both objects when copying named properties from one object to another. Use of this property can minimize translating between copied properties' names and identifiers.

If PR_MAPPING_SIGNATURE does not exist for a given MAPI object, then the object has its own unique mapping of names and identifiers. In this case the client must call the **IMAPIProp::GetNamesFromIDs** method on the source object and then the **IMAPIProp::GetIDsFromNames** method on the destination object.

When two objects have the same PR_MAPPING_SIGNATURE value, the client does not need to translate name to identifier and identifier to name. The client can simply call the **IMAPIProp::GetProps** method on the source and then the **IMAPIProp::SetProps** method on the destination. This is convenient for clients that perform custom copying of named properties, and for providers implementing the **IMAPIProp::CopyTo** and **IMAPIProp::CopyProps** methods.

For more information on named properties and mapping of names and identifiers, see About Named Properties.

See Also

MAPINAMEID structure

PR_MDB_PROVIDER

The PR_MDB_PROVIDER property contains a provider-defined identifier that indicates the type of the message store.

Usage

Required as a column entry in message store tables.
Computed by message store providers on message store objects.

Details

Identifier 0x3414; property type PT_BINARY; property tag 0x34140102

Remarks

The **MAPIUID** structure identifies the type of message store. The value is computed by message store providers on message store objects and is unique to each provider. It is typically used for browsing through the message store table to find a store of the desired type, such as public folders.

See Also

PR_AB_PROVIDER_ID property

PR_MESSAGE_ATTACHMENTS

The PR_MESSAGE_ATTACHMENTS property contains a table of restrictions that can be applied to a contents table to find all messages that contain attachment subobjects meeting the restrictions.

Usage

Optional on message objects.

Details

Identifier 0x0E13; property type PT_OBJECT; property tag 0x0E13000D

Remarks

The PR_MESSAGE_ATTACHMENTS property can be excluded in **IMAPIProp::CopyTo** operations or included in **IMAPIProp::CopyProps** operations. As a property of type PT_OBJECT, it cannot be successfully retrieved by the **IMAPIProp::GetProps** method; its contents should be accessed by the **IMAPIProp::OpenProperty** method, requesting the IID_IMAPITable interface identifier. Service providers must report it to the **IMAPIProp::GetPropList** method if it is set, but may optionally report it or not if it is not set.

To retrieve table contents, a client application should call the d**IMessage::GetAttachmentTable** method. For more information on attachment tables, see About Attachment Tables.

PR_MESSAGE_ATTACHMENTS can be used for subobject restriction by specifying it in the **SSubRestriction** structure. This allows the client to limit the view of a container to messages with attachments meeting given criteria. A message qualifies for viewing if at least one row in its attachments table, that is, one attachment, satisfies the subobject restriction.

Note

Using subobject restriction results in the equivalent of an **IMAPISession::OpenEntry** call on every message in the table. Depending on the client application and the number of messages to be searched, it can affect performance.

The PR_MESSAGE_ATTACHMENTS and PR_MESSAGE_RECIPIENTS properties are similar in usage. Several MAPI properties provide access to tables:

Property	**Table**
PR_CONTAINER_CONTENTS	Contents table
PR_CONTAINER_HIERARCHY	Hierarchy table
PR_FOLDER_ASSOCIATED_CONTENTS	Associated contents table
PR_MESSAGE_ATTACHMENTS	Attachment table
PR_MESSAGE_RECIPIENTS	Recipient table

PR_MESSAGE_CC_ME

The PR_MESSAGE_CC_ME property contains TRUE if this messaging user is specifically named as a carbon copy (CC) recipient of this message and is not part of a group.

Usage

Optional on recipient subobjects.

Details

Identifier 0x0058; property type PT_BOOLEAN; property tag 0x0058000B

Remarks

The PR_MESSAGE_CC_ME property provides a convenient way to determine whether the user name explicitly appears in the recipient list, without examining all entries in the list.

This property also assists automated handling of received messages at the time of receipt. At the transport provider's option, PR_MESSAGE_CC_ME either contains FALSE or is not included if the messaging user is not listed directly in the recipient table.

Message delivery resulting from distribution list expansion or a blind carbon copy designation for the message sender does not cause this property to be set. The user must be explicitly named.

Unsent messages generally do not include PR_MESSAGE_CC_ME. If it is present in messages the user can access in public message stores, in other users' private stores, in files on disk, or embedded inside other received messages, it generally contains the value to which it was set the last time the transport provider delivered it.

PR_MESSAGE_CLASS

The PR_MESSAGE_CLASS property contains a text string that identifies the sender-defined message class, such as IPM.Note.

Usage

Required on message objects and as a column entry in folder contents tables.

Details

Identifier 0x001A; property type PT_TSTRING; property tag 0x001A001E (0x001A001F for Unicode)

Remarks

The message class specifies the type of the message. It determines the set of properties defined for the message, the kind of information the message conveys, and how to handle the message.

The PR_MESSAGE_CLASS property contains ASCII strings concatenated with periods. Each string represents a level of subclassing. For example, IPM.Note is a subclass of IPM and a superclass of IPM.Note.Private.

PR_MESSAGE_CLASS must consist of the ASCII characters 32 through 127 and must not end with a period (ASCII 46). Sort and compare operations must treat it as a case-insensitive string. The maximum possible length is 255 characters, but in order to allow MAPI room to append qualifiers it is recommended that the original length be kept under 128 characters. Note that these lengths are given in characters; on platforms such as Unicode and DBCS (Double-Byte Character Set) the actual byte count could be higher.

Every message is required to furnish the PR_MESSAGE_CLASS property. Normally the client application creating a new message sets PR_MESSAGE_CLASS as soon as **IMAPIFolder::CreateMessage** returns successfully. But if the property has not been set when the client calls **IMAPIProp::SaveChanges**, the message store should set it to IPM.

The PR_MESSAGE_CLASS values defined by MAPI are:

```
IPM.Note for a standard interpersonal message
REPORT.<subject message class>.DR for a delivery report
REPORT.<subject message class>.NDR for a nondelivery report
REPORT.<subject message class>.IPNRN for a read report
REPORT.<subject message class>.IPNNRN for a nonread report
```

IPM and IPC are intended to be superclasses only, and a message should have at least one subclass qualifier appended before being stored or submitted. For more information on message class usage, see About Message Classes.

A custom message class can define properties in a reserved range for use with that message class only. For more information, see About Property Identifiers.

See Also

IMAPIForm : IUnknown interface, **IMsgStore::GetReceiveFolderTable** method

PR_MESSAGE_DELIVERY_ID

The PR_MESSAGE_DELIVERY_ID property contains a message transfer system (MTS) identifier for a message delivered to a client application.

Usage

Optional on message objects.

Details

Identifier 0x001B; property type PT_BINARY; property tag 0x001B0102

Remarks

The PR_MESSAGE_DELIVERY_ID property corresponds to the X.400 attribute MH_T_MTS_IDENTIFIER.

See Also

PR_MESSAGE_SUBMISSION_ID property

PR_MESSAGE_DELIVERY_TIME

The PR_MESSAGE_DELIVERY_TIME property contains the date and time a message was delivered.

Usage

Required on message objects.

Details

Identifier 0x0E06; property type PT_SYSTIME; property tag 0x0E060040

Remarks

The PR_MESSAGE_DELIVERY_TIME property describes the time the message was stored at the server, rather than the download time when the transport provider copied the message from the server to the local store.

PR_MESSAGE_DELIVERY_TIME corresponds to the X.400 attribute MH_T_DELIVERY_TIME.

PR_MESSAGE_DOWNLOAD_TIME

The PR_MESSAGE_DOWNLOAD_TIME property contains the estimated time to download a message from a remote server to a local message store.

Usage

Optional as a column entry in header folder contents tables.

Details

Identifier 0x0E18; property type PT_LONG; property tag 0x0E180003

Remarks

The PR_MESSAGE_DOWNLOAD_TIME property is expressed in seconds and represents the best estimate of the time it would take a remote transport provider to download a given message from its current location to a message store local to the client viewing the header folder. The remote transport provider typically calculates PR_MESSAGE_DOWNLOAD_TIME by dividing the value of the PR_MESSAGE_SIZE property by the speed of the communications link in bytes per second. If the provider cannot calculate the download time, for example if it does not know the link speed, it should furnish a PT_ERROR value such as MAPI_E_NO_SUPPORT for this column in the header folder contents table.

PR_MESSAGE_FLAGS

The PR_MESSAGE_FLAGS property contains a bitmask of flags indicating the current state of a message object.

Usage

Required on message objects.

Details

Identifier 0x0E07; property type PT_LONG; property tag 0x0E070003

Remarks

The PR_MESSAGE_FLAGS property is a nontransmittable message property exposed at both the sending and receiving ends of a transmission, with different values depending upon the client application or store provider involved. This property exists on a message both before and after submission, and on all copies of the received message. Although it is not a recipient property, it is exposed differently to each recipient according to whether it has been read or modified by that recipient.

One or more of the following flags can be set for PR_MESSAGE_FLAGS:

MSGFLAG_ASSOCIATED
: The message is an associated message of a folder. The client or provider has read-only access to this flag.

MSGFLAG_FROMME
: The messaging user sending was the messaging user receiving the message. The client or provider has read/write access to this flag until the first **IMAPIProp::SaveChanges** call and read-only thereafter. This flag is meant to be set by the transport provider.

MSGFLAG_HASATTACH
: The message has at least one attachment. The client or provider has read-only access to this flag.

MSGFLAG_NRN_PENDING
: A nonread report needs to be sent for the message. The client or provider has read-only access to this flag.

MSGFLAG_READ
: The message has been read. The flag is also set when the client creates the message. The client or provider has read/write access to this flag until the first **IMAPIProp::SaveChanges** call and read-only thereafter, but can call **IMessage::SetReadFlag** to change it at any time.

MSGFLAG_RESEND
: The message includes a request for a resend operation with a nondelivery report. The client or provider has read/write access to this flag until the first **IMAPIProp::SaveChanges** call and read-only thereafter.

MSGFLAG_RN_PENDING
: A read report needs to be sent for the message. The client or provider has read-only access to this flag.

MSGFLAG_SUBMIT
: The message is marked for sending as a result of a call to **IMessage::SubmitMessage**. The client or provider has read-only access to this flag.

MSGFLAG_UNMODIFIED
: The message has not been modified since reception. The client or provider has read-only access to this flag.

MSGFLAG_UNSENT
: The message is still being composed. It is saved, but has not been sent. The client or provider has

read/write access to this flag until the first **IMAPIProp::SaveChanges** call and read-only thereafter. Typically, this flag is cleared after the message is sent.

A client or message store provider can check the current state of the message at any time by calling the **IMAPIProp::GetProps** method to read the flag values. The client or provider can also call the **IMAPIProp::SetProps** method to change any flags that currently have read/write access.

Several of the flags are always read-only. Some are read/write until the first call to the **IMAPIProp::SaveChanges** method and thereafter become read-only as far as **IMAPIProp::SetProps** is concerned. One of these, MSGFLAG_READ, can be changed later through the **IMessage::SetReadFlag** method.

See Also

IMsgStore::AbortSubmit method

PR_MESSAGE_RECIP_ME

The PR_MESSAGE_RECIP_ME property contains TRUE if this messaging user is specifically named as a primary (To), carbon copy (CC), or blind carbon copy (BCC) recipient of this message and is not part of a group.

Usage

Optional on recipient subobjects.

Details

Identifier 0x0059; property type PT_BOOLEAN; property tag 0x0059000B

Remarks

The PR_MESSAGE_RECIP_ME property provides a convenient way to determine whether the user name explicitly appears in the recipient list, without examining all entries in the list. The value represents the logical **OR** operation of the properties PR_MESSAGE_CC_ME and PR_MESSAGE_TO_ME, and the BCC information (which does not otherwise appear in a property).

PR_MESSAGE_RECIP_ME assists automated handling of received messages at the time of receipt. At the transport provider's option, this property either contains FALSE or is not included if the messaging user is not listed directly in the recipient table.

Message delivery resulting from distribution list expansion or a blind carbon copy designation for the message sender does not cause this property to be set. The user must be explicitly named.

Unsent messages generally do not include PR_MESSAGE_RECIP_ME. If it is present in messages the user can access in public message stores, in other users' private stores, in files on disk, or embedded inside other received messages, it generally contains the value to which it was set the last time the transport provider delivered it.

PR_MESSAGE_RECIPIENTS

The PR_MESSAGE_RECIPIENTS property contains a table of restrictions that can be applied to a contents table to find all messages that contain recipient subobjects meeting the restrictions.

Usage

Optional on message objects.

Details

Identifier 0x0E12; property type PT_OBJECT; property tag 0x0E12000D

Remarks

The PR_MESSAGE_RECIPIENTS property can be excluded in **IMAPIProp::CopyTo** operations or included in **IMAPIProp::CopyProps** operations. As a property of type PT_OBJECT, it cannot be successfully retrieved by the **IMAPIProp::GetProps** method; its contents should be accessed by the **IMAPIProp::OpenProperty** method, requesting the IID_IMAPITable interface identifier. Service providers must report it to the **IMAPIProp::GetPropList** method if it is set, but may optionally report it or not if it is not set.

To retrieve table contents, a client application should call the **IMessage::GetRecipientTable** method. For more information on recipient tables, see About Recipient Tables.

PR_MESSAGE_RECIPIENTS can be used for subobject restriction by specifying it in the **SSubRestriction** structure. This enables a client to limit the view of a container to messages with recipients meeting given criteria. A message qualifies for viewing if at least one row in its recipient table, that is, one recipient satisfies the subobject restriction.

Note Using subobject restriction results is the equivalent of an **IMAPISession::OpenEntry** call on every message in the table. Depending on the client application and the number of messages to be searched, it can affect performance.

The PR_MESSAGE_ATTACHMENTS and PR_MESSAGE_RECIPIENTS properties are similar in usage. Several MAPI properties provide access to tables:

Property	Table
PR_CONTAINER_CONTENTS	Contents table
PR_CONTAINER_HIERARCHY	Hierarchy table
PR_FOLDER_ASSOCIATED_CONTENTS	Associated contents table
PR_MESSAGE_ATTACHMENTS	Attachment table
PR_MESSAGE_RECIPIENTS	Recipient table

PR_MESSAGE_SECURITY_LABEL

The PR_MESSAGE_SECURITY_LABEL property contains a security label for a message.

Usage

Optional on message objects.

Details

Identifier 0x001E; property type PT_BINARY; property tag 0x001E0102

Remarks

The PR_MESSAGE_SECURITY_LABEL property provides the basis on which the PR_MESSAGE_TOKEN property protects a message. Its association with the message content is guaranteed by the token.

PR_MESSAGE_SECURITY_LABEL corresponds to the X.400 attribute MH_T_SECURITY_LABEL.

See Also

PR_CONTENT_INTEGRITY_CHECK property

PR_MESSAGE_SIZE

The PR_MESSAGE_SIZE property contains the sum, in bytes, of the sizes of all properties on a message object.

Usage

Optional but recommended on message objects.

Details

Identifier 0x0E08; property type PT_LONG; property tag 0x0E080003

Remarks

The message size indicates the approximate number of bytes transferred when the message is moved from one message store to another. Being the sum of the sizes of all properties on the message object, it is usually considerably greater than the message text alone.

Most message store providers compute the PR_MESSAGE_SIZE property for messages that they handle. However, some message store providers do not support this property.

See Also

PR_ATTACH_SIZE property

PR_MESSAGE_SUBMISSION_ID

The PR_MESSAGE_SUBMISSION_ID property contains a message transfer system (MTS) identifier for the message transfer agent (MTA).

Usage

Optional on message objects.

Details

Identifier 0x0047; property type PT_BINARY; property tag 0x00470102

Remarks

The PR_MESSAGE_SUBMISSION_ID property is returned by the MTA upon successful completion of message submission. Any future contact with the MTA regarding this message, such as requesting cancellation, uses the MTS identifier in the PR_MESSAGE_SUBMISSION_ID property.

PR_MESSAGE_SUBMISSION_ID corresponds to the X.400 attribute MH_T_MTS_IDENTIFIER.

See Also

PR_MESSAGE_DELIVERY_ID property

PR_MESSAGE_TO_ME

The PR_MESSAGE_TO_ME property contains TRUE if this messaging user is specifically named as a primary (To) recipient of this message and is not part of a group.

Usage

Optional on recipient subobjects.

Details

Identifier 0x0057; property type PT_BOOLEAN; property tag 0x0057000B

Remarks

The PR_MESSAGE_TO_ME property provides a convenient way to determine whether the user name explicitly appears in the recipient list, without examining all entries in the list.

This property also assists automated handling of received messages at the time of receipt. At the transport provider's option, PR_MESSAGE_TO_ME either contains FALSE or is not included if the messaging user is not listed directly in the recipient table.

Message delivery resulting from distribution list expansion or a blind carbon copy designation for the message sender does not cause this property to be set. The user must be explicitly named.

Unsent messages generally do not include PR_MESSAGE_TO_ME. If it is present in messages the user can access in public message stores, in other users' private stores, in files on disk, or embedded inside other received messages, it generally contains the value to which it was set the last time the transport provider delivered it.

PR_MESSAGE_TOKEN

The PR_MESSAGE_TOKEN property contains an ASN.1 security token for a message.

Usage

Optional on message objects.

Details

Identifier 0x0C03; property type PT_BINARY; property tag 0x0C030102

Remarks

The PR_MESSAGE_TOKEN property conveys protected security-related information from its originator to its recipient. In conjunction with PR_MESSAGE_SECURITY_LABEL it guarantees the label's association with the message content. In conjunction with PR_CONTENT_INTEGRITY_CHECK it verifies that the message content is unchanged.

PR_MESSAGE_TOKEN corresponds to the X.400 attribute MH_T_TOKEN.

PR_MHS_COMMON_NAME

The PR_MHS_COMMON_NAME property contains the common name of a messaging user for use in a message header.

Usage

Optional on message objects.

Details

Identifier 0x3A0F; property type PT_TSTRING; property tag 0x3A0F001E (0x3A0F001F for Unicode)

Remarks

The PR_MHS_COMMON_NAME property corresponds to the X.400 attribute MH_T_COMMON_NAME.

See Also

PR_DISPLAY_NAME property

PR_MINI_ICON

The PR_MINI_ICON property contains a bitmap of a half-size icon for a form.

Usage

Required on form objects.

Details

Identifier 0x0FFC; property type PT_BINARY; property tag 0x0FFC0102

Remarks

The PR_MINI_ICON property contains a 32 × 32 pixel image of an icon, the same as the contents of a .ICO file, but only the upper left 16 × 16 pixels are considered significant. This property is normally copied from the .ICO file specified in the SmallIcon line of the appropriate [Description] section of the form configuration file.

Note Some platforms do not support 16 × 16 pixel icons. The 32 × 32 format of PR_MINI_ICON is usable in such a case but client applications should be aware of display inconsistencies.

See Also

PR_ICON property

PR_MOBILE_TELEPHONE_NUMBER

The PR_MOBILE_TELEPHONE_NUMBER property contains the recipient's cellular telephone number.

Usage

Optional on messaging user objects.

Details

Identifier 0x3A1C; property type PT_TSTRING; property tag 0x3A1C001E (0x3A1C001F for Unicode)

Remarks

The PR_MOBILE_TELEPHONE_NUMBER property is one of the properties that provide identification and access information for a recipient. These properties are defined by the recipient and the recipient's organization. For more information on this group of properties, see About Messaging User Objects.

MAPI also supports the PR_CELLULAR_TELEPHONE_NUMBER property that is synonymous with PR_MOBILE_TELEPHONE_NUMBER.

PR_MODIFY_VERSION

The PR_MODIFY_VERSION property was originally meant to contain the message store version current at the time the message was last modified.

Usage

Never used.

Details

Identifier 0x0E1A; property type PT_I8; property tag 0x0E1A0014

Remarks

Do not use this property. It is not supported in MAPI 1.0.

See Also

PR_CREATION_VERSION property, PR_CURRENT_VERSION property

PR_MSG_STATUS

The PR_MSG_STATUS property contains a 32-bit bitmask of flags defining the status of a message in a contents table.

Usage

Required as a column entry in message store contents tables.

Details

Identifier 0x0E17; property type PT_LONG; property tag 0x0E170003

Remarks

A message can exist in a contents table and in one or more search-results tables, and each instance of the message can have a different status. The PR_MSG_STATUS property should not be considered a property on a message but a column in a contents table.

A client application can set one or more of the following flags in PR_MSG_STATUS:

MSGSTATUS_DELMARKED	The message has been marked for subsequent deletion.
MSGSTATUS_HIDDEN	The message is to be suppressed from recipients' folder displays.
MSGSTATUS_HIGHLIGHTED	The message is to be highlighted in recipients' folder displays.
MSGSTATUS_REMOTE_DELETE	The message has been marked for deletion at the remote message store without downloading to the local client.
MSGSTATUS_REMOTE_DOWNLOAD	The message has been marked for downloading from the remote message store to the local client.
MSGSTATUS_TAGGED	The message has been tagged for a client-defined purpose.

The MSGSTATUS_DELMARKED, MSGSTATUS_HIDDEN, MSGSTATUS_HIGHLIGHTED, and MSGSTATUS_TAGGED flags are defined by the client. Transport and store providers pass these bits without any action.

A remote viewer client can set MSGSTATUS_REMOTE_DELETE or MSGSTATUS_REMOTE_DOWNLOAD on messages in the header folder presented to it by the remote transport provider. The client application can examine each message header in this folder to determine whether the message should be downloaded or deleted at the remote message store. It then uses the **IMAPIFolder::SetMessageStatus** method to set the appropriate flag. **SetMessageStatus** is the only way to set any of the PR_MSG_STATUS flags. For more information on remote viewing and remote transport providers, see Remote Transport Architecture.

Bits 16 through 31 (0x10000 through 0x80000000) of PR_MSG_STATUS are available for use by the interpersonal message (IPM) client application. All other bits are reserved for use by MAPI; those not defined in the preceding table should be initially set to zero and not altered subsequently.

See Also

IMAPIFolder::GetMessageStatus method, **IMAPITable::QueryRows** method

PR_NDR_DIAG_CODE

The PR_NDR_DIAG_CODE property contains a diagnostic code that forms part of a nondelivery report.

Usage

Required on recipient subobjects within nondelivery report message objects.

Details

Identifier 0x0C05; property type PT_LONG; property tag 0x0C050003

Remarks

The PR_NDR_DIAG_CODE property can have exactly one of the following values:

Value
MAPI_DIAG_ALPHABETIC_CHARACTER_LOST
MAPI_DIAG_CONTENT_SYNTAX_IN_ERROR
MAPI_DIAG_CONTENT_TOO_LONG
MAPI_DIAG_CONTENT_TYPE_UNSUPPORTED
MAPI_DIAG_CONVERSION_LOSS_PROHIB
MAPI_DIAG_CONVERSION_UNSUBSCRIBED
MAPI_DIAG_CRITICAL_FUNC_UNSUPPORTED
MAPI_DIAG_EITS_UNSUPPORTED
MAPI_DIAG_EXPANSION_FAILED
MAPI_DIAG_EXPANSION_PROHIBITED
MAPI_DIAG_IMPRACTICAL_TO_CONVERT
MAPI_DIAG_LENGTH_CONSTRAINT_VIOLATD
MAPI_DIAG_LINE_TOO_LONG
MAPI_DIAG_LOOP_DETECTED
MAPI_DIAG_MAIL_ADDRESS_INCOMPLETE
MAPI_DIAG_MAIL_ADDRESS

_INCORRECT

MAPI_DIAG_MAIL_FORWARDING_PROHIB

MAPI_DIAG_MAIL_FORWARDING_UNWANTED

MAPI_DIAG_MAIL_NEW_ADDRESS_UNKNOWN

MAPI_DIAG_MAIL_OFFICE_INCOR_OR_INVD

MAPI_DIAG_MAIL_ORGANIZATION_EXPIRED

MAPI_DIAG_MAIL_RECIPIENT_DECEASED

MAPI_DIAG_MAIL_RECIPIENT_DEPARTED

MAPI_DIAG_MAIL_RECIPIENT_MOVED

MAPI_DIAG_MAIL_RECIPIENT_TRAVELLING

MAPI_DIAG_MAIL_RECIPIENT_UNKNOWN

MAPI_DIAG_MAIL_REFUSED

MAPI_DIAG_MAIL_UNCLAIMED

MAPI_DIAG_MAXIMUM_TIME_EXPIRED

MAPI_DIAG_MTS_CONGESTED

MAPI_DIAG_MULTIPLE_INFO_LOSSES

MAPI_DIAG_NO_BILATERAL_AGREEMENT

MAPI_DIAG_NO_DIAGNOSTIC

MAPI_DIAG_NUMBER_CONSTRAINT_VIOLATD

MAPI_DIAG_OR_NAME_AMBIGUOUS

MAPI_DIAG_OR_NAME_UNRECOGNIZED

MAPI_DIAG_PAGE_TOO_LONG

MAPI_DIAG_PARAMETERS_INVALID

MAPI_DIAG_PICTORIAL_SYMBOL_LOST

MAPI_DIAG_PROHIBITED_TO_CONVERT

MAPI_DIAG_PUNCTUATION_

SYMBOL_LOST

MAPI_DIAG_REASSIGNMENT_PROHIBITED

MAPI_DIAG_RECIPIENT_UNAVAILABLE

MAPI_DIAG_REDIRECTION_LOOP_DETECTED

MAPI_DIAG_RENDITION_UNSUPPORTED

MAPI_DIAG_SECURE_MESSAGING_ERROR

MAPI_DIAG_SUBMISSION_PROHIBITED

MAPI_DIAG_TOO_MANY_RECIPIENTS

PR_NDR_DIAG_CODE corresponds to the X.400 attribute MH_T_NON_DELIVERY_DIAGNOSTIC.

See Also

PR_NDR_REASON_CODE property

PR_NDR_REASON_CODE

The PR_NDR_REASON_CODE property contains an encoded reason for nondelivery that forms part of a nondelivery report.

Usage

Required on recipient subobjects within nondelivery report message objects.

Details

Identifier 0x0C04, property type PT_LONG; property tag 0x0C040003

Remarks

The PR_NDR_REASON_CODE property corresponds to the X.400 attribute MH_T_NON_DELIVERY_REASON.

See Also

PR_NDR_DIAG_CODE property

PR_NON_RECEIPT_NOTIFICATION_REQUESTED

The PR_NON_RECEIPT_NOTIFICATION_REQUESTED property contains TRUE if a message sender wants notification of nondelivery for a specified recipient.

Usage

Optional on message objects.

Details

Identifier 0x0C06; property type PT_BOOLEAN; property tag 0x0C06000B

Remarks

If the PR_NON_RECEIPT_NOTIFICATION_REQUESTED property contains FALSE and the PR_READ_RECEIPT_REQUESTED property contains TRUE, the service provider can override PR_NON_RECEIPT_NOTIFICATION_REQUESTED and generate a nondelivery report.

PR_NON_RECEIPT_NOTIFICATION_REQUESTED corresponds to the X.400 attribute IM_NOTIFICATION_REQUEST.

PR_NON_RECEIPT_REASON

The PR_NON_RECEIPT_REASON property contains reasons why a message was not received that forms part of a nondelivery report.

Usage

Required on nondelivery report message objects.

Details

Identifier 0x003E; property type PT_LONG; property tag 0x003E0003

Remarks

Reasons can include, for example, that the message was discarded.

The PR_NON_RECEIPT_REASON property corresponds to the X.400 attribute IM_NON_RECEIPT_REASON.

See Also

PR_NDR_DIAG_CODE property

PR_NORMALIZED_SUBJECT

The PR_NORMALIZED_SUBJECT property contains the message subject with any prefix removed.

Usage

Computed by a store or transport provider on message objects.

Details

Identifier 0x0E1D, property type PT_TSTRING; property tag 0x0E1D001E (0x0E1D001F for Unicode)

Remarks

The PR_NORMALIZED_SUBJECT property is computed from the PR_SUBJECT and PR_SUBJECT_PREFIX properties in the following manner. If PR_SUBJECT_PREFIX is set and is in fact an initial substring of PR_SUBJECT, then PR_NORMALIZED_SUBJECT becomes PR_SUBJECT with the prefix removed. If PR_SUBJECT_PREFIX is set but is not an initial substring of PR_SUBJECT, then PR_SUBJECT_PREFIX is deleted and an attempt is made to compute it from the beginning of PR_SUBJECT using the rule described under PR_SUBJECT_PREFIX. If PR_SUBJECT_PREFIX is not set, the same attempt is made to compute it from PR_SUBJECT.

Ultimately, PR_NORMALIZED_SUBJECT should be the part of PR_SUBJECT following the prefix. If there is no prefix, PR_NORMALIZED_SUBJECT becomes the same as PR_SUBJECT.

PR_SUBJECT_PREFIX and PR_NORMALIZED_SUBJECT should be computed as part of the **IMAPIProp::SaveChanges** implementation. A client application should not prompt the **IMAPIProp::GetProps** method for their values until they have been committed by an **IMAPIProp::SaveChanges** call.

The subject properties are typically small strings of fewer than 256 characters, and a message store provider is not obligated to support the OLE **IStream** interface on them. The client should always attempt access through the **IMAPIProp** interface first, and resort to **IStream** only if MAPI_E_NOT_ENOUGH_MEMORY is returned.

PR_NORMALIZED_SUBJECT corresponds to the X.400 attribute IM_SUBJECT.

PR_NULL

The PR_NULL property represents a null value or setting of a property or reserves array space.

Details

Identifier 0x0000; property type PT_NULL; property tag 0x00000001

Remarks

The PR_NULL property is used to reserve space in arrays of **SPropValue** structures. It is used in an array of **SPropTag** structures to tell the method to reserve space in the returned array of **SPropValue** structures. This allows for computed properties to be filled in an inexpensive way.

See Also

About Property Types

PR_OBJECT_TYPE

The PR_OBJECT_TYPE property contains the type of an object.

Usage

Required on address book container, distribution list, folder, messaging user, message, and message store objects.

Details

Identifier 0x0FFE; property type PT_LONG; property tag 0x0FFE0003

Remarks

The object type contained in the PR_OBJECT_TYPE property corresponds to the primary interface available for an object accessible through the **OpenEntry** interface. It is usually obtained by consulting the *lpulObjType* parameter returned by the appropriate **OpenEntry** method. When the interface is obtained in other ways, call **IMAPIProp::GetProps** to obtain the value for PR_OBJECT_TYPE.

PR_OBJECT_TYPE can have exactly one of the following values:

Value	Description
MAPI_ABCONT	Address book container object
MAPI_ADDRBOOK	Address book object
MAPI_ATTACH	Message attachment object
MAPI_DISTLIST	Distribution list object
MAPI_FOLDER	Folder object
MAPI_FORMINFO	Form object
MAPI_MAILUSER	Messaging user object
MAPI_MESSAGE	Message object
MAPI_PROFSECT	Profile section object
MAPI_STATUS	Status object
MAPI_STORE	Message store object

For more information on object types, see Objects and Interfaces.

PR_OBSOLETED_IPMS

The PR_OBSOLETED_IPMS property contains the identifiers of messages that this message supersedes.

Usage

Optional on message objects.

Details

Identifier 0x001F; property type PT_BINARY; property tag 0x001F0102

Remarks

The identifiers contained in the PR_OBSOLETED_IPMS property are standard search keys using the format of the PR_SEARCH_KEY property.

PR_OBSOLETED_IPMS corresponds to the X.400 attribute IM_OBSOLETED_IPMS.

See Also

PR_SEARCH_KEY property

PR_OFFICE_LOCATION

The PR_OFFICE_LOCATION property contains the recipient's office location.

Usage

Optional on messaging user objects.

Details

Identifier 0x3A19; property type PT_TSTRING; property tag 0x3A19001E (0x3A19001F for Unicode)

Remarks

The PR_OFFICE_LOCATION property is one of the properties that provide identification and access information for a recipient. These properties are defined by the recipient and the recipient's organization. For more information on this group of properties, see About Messaging User Objects.

PR_ORGANIZATIONAL_ID_NUMBER

The PR_ORGANIZATIONAL_ID_NUMBER property contains an identifier for the recipient used within the recipient's organization.

Usage

Optional on messaging user objects.

Details

Identifier 0x3A10; property type PT_TSTRING; property tag 0x3A10001E (0x3A10001F for Unicode)

Remarks

The PR_ORGANIZATIONAL_ID_NUMBER property is one of the properties that provide identification and access information for a recipient. These properties are defined by the recipient and the recipient's organization. For more information on this group of properties, see About Messaging User Objects.

PR_ORGANIZATIONAL_ID_NUMBER is commonly set to an employee number.

PR_ORIG_MESSAGE_CLASS

The PR_ORIG_MESSAGE_CLASS property contains the class of the original message for use in a report.

Usage

Optional on report message objects.

Details

Identifier 0x004B; property type PT_TSTRING; property tag 0x004B001E (0x004B001F for Unicode)

Remarks

The PR_ORIG_MESSAGE_CLASS property contains a copy of the PR_MESSAGE_CLASS property of the message for which the report is being generated.

PR_ORIGIN_CHECK

The PR_ORIGIN_CHECK property contains a binary verification value enabling a delivery report recipient to verify the origin of the original message.

Usage

Optional on message objects.

Details

Identifier 0x0027; property type PT_BINARY; property tag 0x00270102

Remarks

The PR_ORIGIN_CHECK property provides a means for a third party, such as a message transfer agent (MTA) or a messaging user receiving a delivery report, to verify the submitted message's origin. If present on a received message, this property should be copied onto any delivery report generated in response to the message.

PR_ORIGIN_CHECK replaces several obsolete properties supplied in pre-release versions of MAPI, including PR_MESSAGE_ORIGIN_AUTHENTICATION_CHECK, PR_PROBE_SUBMISSION_AUTHENTICATION_CHECK, and PR_REPORT_ORIGIN_AUTHENTICATION_CHECK.

PR_ORIGIN_CHECK corresponds to the X.400 attribute MH_T_ORIGIN_CHECK.

See Also

PR_ORIGINATOR_CERTIFICATE property

PR_ORIGINAL_AUTHOR_ADDRTYPE

The PR_ORIGINAL_AUTHOR_ADDRTYPE property contains the address type of the author of the first version of a message, that is, the message before being forwarded or replied to.

Usage

Optional on message objects.

Details

Identifier 0x0079; property type PT_TSTRING; property tag 0x0079001E (0x0079001F for Unicode)

Remarks

The PR_ORIGINAL_AUTHOR_ADDRTYPE property is one of the address properties for the author of a message. At first submission of the message, the client application should set this property to the value of the PR_SENDER_ADDRTYPE property. It is never changed when the message is forwarded or replied to.

The original author properties allow for preservation of information from outside the local messaging domain. When a message arrives from another messaging domain, such as from the Internet, these properties provide a way to ensure that original information is not lost.

For more information on the address properties, see About Base Address Properties.

See Also

PR_ADDRTYPE property

PR_ORIGINAL_AUTHOR_EMAIL_ADDRESS

The PR_ORIGINAL_AUTHOR_EMAIL_ADDRESS property contains the e-mail address of the author of the first version of a message, that is, the message before being forwarded or replied to.

Usage

Optional on message objects.

Details

Identifier 0x007A; property type PT_TSTRING; property tag 0x007A001E (0x007A001F for Unicode)

Remarks

The PR_ORIGINAL_AUTHOR_EMAIL_ADDRESS property is one of the address properties for the author of a message. At first submission of the message, the client application should set this property to the value of PR_SENDER_EMAIL_ADDRESS. It is never changed when the message is forwarded or replied to.

The original author properties allow for preservation of information from outside the local messaging domain. When a message arrives from another messaging domain, such as from the Internet, these properties provide a way to ensure that original information is not lost.

For more information on the address properties, see About Base Address Properties.

See Also

PR_EMAIL_ADDRESS property

PR_ORIGINAL_AUTHOR_ENTRYID

The PR_ORIGINAL_AUTHOR_ENTRYID property contains the entry identifier of the author of the first version of a message, that is, the message before being forwarded or replied to.

Usage

Optional on message objects.

Details

Identifier 0x004C; property type PT_BINARY; property tag 0x004C0102

Remarks

The PR_ORIGINAL_AUTHOR_ENTRYID property is one of the address properties for the author of a message. At first submission of the message, the client application should set this property to the value of PR_SENDER_ENTRYID. It is never changed when the message is forwarded or replied to.

The original author properties allow for preservation of information from outside the local messaging domain. When a message arrives from another messaging domain, such as from the Internet, these properties provide a way to ensure that original information is not lost.

For more information on the address properties, see About Base Address Properties.

See Also

PR_ENTRYID property

PR_ORIGINAL_AUTHOR_NAME

The PR_ORIGINAL_AUTHOR_NAME property contains the display name of the author of the first version of a message, that is, the message before being forwarded or replied to.

Usage

Optional on message objects.

Details

Identifier 0x004D; property type PT_TSTRING; property tag 0x004D001E (0x004D001F for Unicode)

Remarks

The PR_ORIGINAL_AUTHOR_NAME property is one of the address properties for the author of a message. At first submission of the message, the client application should set this property to the value of PR_SENDER_NAME. It is never changed when the message is forwarded or replied to.

The original author properties allow for preservation of information from outside the local messaging domain. When a message arrives from another messaging domain, such as from the Internet, these properties provide a way to ensure that original information is not lost.

For more information on the address properties, see About Base Address Properties.

PR_ORIGINAL_AUTHOR_NAME corresponds to the X.400 attribute MH_T_ORIGINATOR_NAME.

See Also

PR_DISPLAY_NAME property

PR_ORIGINAL_AUTHOR_SEARCH_KEY

The PR_ORIGINAL_AUTHOR_SEARCH_KEY property contains the search key of the author of the first version of a message, that is, the message before being forwarded or replied to.

Usage

Optional on message objects.

Details

Identifier 0x0056; property type PT_BINARY; property tag 0x00560102

Remarks

The PR_ORIGINAL_AUTHOR_SEARCH_KEY property is one of the address properties for the author of a message. At first submission of the message, the client application should set this property to the value of PR_SENDER_SEARCH_KEY. It is never changed when the message is forwarded or replied to.

The original author properties allow for preservation of information from outside the local messaging domain. When a message arrives from another messaging domain, such as from the Internet, these properties provide a way to ensure that original information is not lost.

For more information on the address properties, see About Base Address Properties.

See Also

PR_SEARCH_KEY property

PR_ORIGINAL_DELIVERY_TIME

The PR_ORIGINAL_DELIVERY_TIME property contains a copy of the original message's delivery date and time in a thread.

Usage

Required on read and nonread report objects.

Details

Identifier 0x0055; property type PT_SYSTIME; property tag 0x00550040

Remarks

The PR_ORIGINAL_DELIVERY_TIME property is copied from the original PR_MESSAGE_DELIVERY_TIME property in subsequent reply or forward operations and used in read and nonread reports. Delivery reports use the PR_DELIVER_TIME property instead.

PR_ORIGINAL_DISPLAY_BCC

The PR_ORIGINAL_DISPLAY_BCC property contains the display names of any blind carbon copy (BCC) recipients of the original message.

Usage

Furnished by MAPI on report message objects.

Details

Identifier 0x0072; property type PT_TSTRING; property tag 0x0072001E (0x0072001F for Unicode)

Remarks

The PR_ORIGINAL_DISPLAY_BCC property contains an ASCII list separated by semicolons. It is copied directly from PR_DISPLAY_BCC when a delivery or nondelivery report or a read or nonread report is generated. PR_ORIGINAL_DISPLAY_BCC may be present on other messages as defined by their message classes.

PR_ORIGINAL_DISPLAY_CC

The PR_ORIGINAL_DISPLAY_CC property contains the display names of any carbon copy (CC) recipients of the original message.

Usage

Furnished by MAPI on report message objects.

Details

Identifier 0x0073; property type PT_TSTRING; property tag 0x0073001E (0x0073001F for Unicode)

Remarks

The PR_ORIGINAL_DISPLAY_CC property contains an ASCII list separated by semicolons. It is copied directly from PR_DISPLAY_CC when a delivery or nondelivery report or a read or nonread report is generated. PR_ORIGINAL_DISPLAY_CC may be present on other messages as defined by their message classes.

PR_ORIGINAL_DISPLAY_NAME

The PR_ORIGINAL_DISPLAY_NAME property contains the original display name for an entry copied from an address book to a personal address book or other writable address book.

Usage

Optional on messaging user objects.

Details

Identifier 0x3A13; property type PT_TSTRING; property tag 0x3A13001E (0x3A13001F for Unicode)

Remarks

The PR_ORIGINAL_DISPLAY_NAME property is one of the properties that contain information about the original source of a copied entry.

For a nonread report, PR_ORIGINAL_DISPLAY_NAME contains a copy of the display name of the original message recipient for which the report is generated. When the original recipient is part of a distribution list, the display name of the distribution list is preserved for the report.

A client application can use PR_ORIGINAL_DISPLAY_NAME to prevent alteration or "spoofing" of entries. An example of spoofing is displaying John Doe as John (What a Guy) Doe.

See Also

PR_TRANSMITTABLE_DISPLAY_NAME property

PR_ORIGINAL_DISPLAY_TO

The PR_ORIGINAL_DISPLAY_TO property contains the display names of the primary (To) recipients of the original message.

Usage

Furnished by MAPI on report message objects.

Details

Identifier 0x0074; property type PT_TSTRING; property tag 0x0074001E (0x0074001F for Unicode)

Remarks

The PR_ORIGINAL_DISPLAY_TO property contains an ASCII list separated by semicolons. It is copied directly from PR_DISPLAY_TO when a delivery or nondelivery report or a read or nonread report is generated. PR_ORIGINAL_DISPLAY_TO may be present on other messages as defined by their message classes.

PR_ORIGINAL_EITS

The PR_ORIGINAL_EITS property contains a copy of the original encoded information types (EITs) for message text.

Usage

Optional on message objects.

Details

Identifier 0x0021; property type PT_BINARY; property tag 0x00210102

Remarks

The PR_ORIGINAL_EITS property corresponds to the X.400 attribute MH_T_ORIGINAL_EITS.

PR_ORIGINAL_ENTRYID

The PR_ORIGINAL_ENTRYID property contains the original entry identifier for an entry copied from an address book to a personal address book or other writeable address book.

Usage

Optional on messaging user objects.

Details

Identifier 0x3A12; property type PT_BINARY; property tag 0x3A120102

Remarks

The PR_ORIGINAL_ENTRYID property is one of the properties that contain information about the original source of a copied entry.

For a nonread report, PR_ORIGINAL_ENTRYID contains a copy of the entry identifier of the original message recipient for which the report is generated. When the original recipient is part of a distribution list, the entry identifier of the distribution list is preserved for the report.

PR_ORIGINAL_SEARCH_KEY

The PR_ORIGINAL_SEARCH_KEY property contains the original search key for an entry copied from an address book to a personal address book or other writeable address book.

Usage

Optional on messaging user objects.

Details

Identifier 0x3A14; property type PT_BINARY; property tag 0x3A140102

Remarks

The PR_ORIGINAL_SEARCH_KEY property is one of the properties that contain information about the original source of a copied entry.

For a nonread report, PR_ORIGINAL_SEARCH_KEY contains a copy of the search key of the original message recipient for which the report is generated. When the original recipient is part of a distribution list, the search key of the distribution list is preserved for the report.

PR_ORIGINAL_SENDER_ADDRTYPE

The PR_ORIGINAL_SENDER_ADDRTYPE property contains the address type of the sender of the first version of a message, that is, the message before being forwarded or replied to.

Usage

Optional on message objects.

Details

Identifier 0x0066; property type PT_TSTRING; property tag 0x0066001E (0x0066001F for Unicode)

Remarks

The PR_ORIGINAL_SENDER_ADDRTYPE property is one of the address properties for the original sender of a message. At first submission of the message, the client application should set this property to the value of PR_SENDER_ADDRTYPE. It is never changed when the message is forwarded or replied to.

For more information on the address properties, see About Base Address Properties.

PR_ORIGINAL_SENDER_EMAIL_ADDRESS

The PR_ORIGINAL_SENDER_EMAIL_ADDRESS property contains the e-mail address of the sender of the first version of a message, that is, the message before being forwarded or replied to.

Usage

Optional on message objects.

Details

Identifier 0x0067; property type PT_TSTRING; property tag 0x0067001E (0x0067001F for Unicode)

Remarks

The PR_ORIGINAL_SENDER_EMAIL_ADDRESS property is one of the address properties for the original sender of a message. At first submission of the message, the client application should set this property to the value of PR_SENDER_EMAIL_ADDRESS. It is never changed when the message is forwarded or replied to.

For more information on the address properties, see About Base Address Properties.

PR_ORIGINAL_SENDER_ENTRYID

The PR_ORIGINAL_SENDER_ENTRYID property contains the entry identifier of the sender of the first version of a message, that is, the message before being forwarded or replied to.

Usage

Optional on message objects.

Details

Identifier 0x005B; property type PT_BINARY; property tag 0x005B0102

Remarks

The PR_ORIGINAL_SENDER_ENTRYID property is one of the address properties for the original sender of a message. At first submission of the message, a client application should set this property to the value of the PR_SENDER_ENTRYID property. It is never changed when the message is forwarded or replied to.

For more information on the address properties, see About Base Address Properties.

PR_ORIGINAL_SENDER_NAME

The PR_ORIGINAL_SENDER_NAME property contains the display name of the sender of the first version of a message, that is, the message before being forwarded or replied to.

Usage

Optional on message objects.

Details

Identifier 0x005A; property type PT_TSTRING; property tag 0x005A001E (0x005A001F for Unicode)

Remarks

The PR_ORIGINAL_SENDER_NAME property is one of the address properties for the original sender of a message. At first submission of the message, a client application should set this property to the value of the PR_SENDER_NAME property. It is never changed when the message is forwarded or replied to.

For more information on the address properties, see About Base Address Properties.

PR_ORIGINAL_SENDER_NAME corresponds to the X.400 attribute MH_T_ORIGINATOR_NAME.

PR_ORIGINAL_SENDER_SEARCH_KEY

The PR_ORIGINAL_SENDER_SEARCH_KEY property contains the search key for the sender of the first version of a message, that is, the message before being forwarded or replied to.

Usage

Optional on message objects.

Details

Identifier 0x005C; property type PT_BINARY; property tag 0x005C0102

Remarks

The PR_ORIGINAL_SENDER_SEARCH_KEY property is one of the address properties for the original sender of a message. At first submission of the message, the client application should set this property to the value of the PR_SENDER_SEARCH_KEY property. It is never changed when the message is forwarded or replied to.

For more information on the address properties, see About Base Address Properties.

PR_ORIGINAL_SENSITIVITY

The PR_ORIGINAL_SENSITIVITY property contains the sensitivity value assigned by the sender of the first version of a message that is, the message before being forwarded or replied to.

Usage

Optional on message objects.

Details

Identifier 0x002E; property type PT_LONG; property tag 0x002E0003

Remarks

A client application should set the PR_ORIGINAL_SENSITIVITY property to the same value as the PR_SENSITIVITY property when the message is first submitted. It should never be changed subsequently.

PR_ORIGINAL_SENSITIVITY is used by the transport provider to protect the sensitivity on copied entries. It enables it, for example, to block modification of the original message text in a forward of or reply to a message that was originally marked SENSITIVITY_PRIVATE.

PR_ORIGINAL_SENT_REPRESENTING_ADDRTYPE

The PR_ORIGINAL_SENT_REPRESENTING_ADDRTYPE property contains the address type of the messaging user on whose behalf the original message was sent.

Usage

Optional on message objects.

Details

Identifier 0x0068; property type PT_TSTRING; property tag 0x0068001E (0x0068001F for Unicode)

Remarks

The PR_ORIGINAL_SENT_REPRESENTING_ADDRTYPE property is one of the address properties for the original represented sender of a message. It is used in a conversation thread.

A client application sending a message on behalf of another client should set this property to the value of the PR_SENT_REPRESENTING_ADDRTYPE property at the first submission of the message. Once set, it should never be changed.

For more information on the address properties, see About Base Address Properties.

PR_ORIGINAL_SENT_REPRESENTING_EMAIL_ADDRESS

The PR_ORIGINAL_SENT_REPRESENTING_EMAIL_ADDRESS property contains the e-mail address of the messaging user on whose behalf the original message was sent.

Usage

Optional on message objects.

Details

Identifier 0x0069; property type PT_TSTRING; property tag 0x0069001E (0x0069001F for Unicode)

Remarks

The PR_ORIGINAL_SENT_REPRESENTING_EMAIL_ADDRESS property is one of the address properties for the original represented sender of a message. It is used in a conversation thread.

A client application sending a message on behalf of another client should set this property to the value of the PR_SENT_REPRESENTING_EMAIL_ADDRESS property at the first submission of the message. Once set, it should never be changed.

For more information on the address properties, see About Base Address Properties.

PR_ORIGINAL_SENT_REPRESENTING_ENTRYID

The PR_ORIGINAL_SENT_REPRESENTING_ENTRYID property contains the entry identifier of the messaging user on whose behalf the original message was sent.

Usage

Optional on message objects.

Details

Identifier 0x005E; property type PT_BINARY; property tag 0x005E0102

Remarks

The PR_ORIGINAL_SENT_REPRESENTING_ENTRYID property is one of the address properties for the original represented sender of a message. It is used in a conversation thread.

A client application sending a message on behalf of another client should set this property to the value of the PR_SENT_REPRESENTING_ENTRYID property at the first submission of the message. Once set, it should never be changed.

For more information on the address properties, see About Base Address Properties.

PR_ORIGINAL_SENT_REPRESENTING_NAME

The PR_ORIGINAL_SENT_REPRESENTING_NAME property contains the display name of the messaging user on whose behalf the original message was sent.

Usage

Optional on message objects.

Details

Identifier 0x005D; property type PT_TSTRING; property tag 0x005D001E (0x005D001F for Unicode)

Remarks

The PR_ORIGINAL_SENT_REPRESENTING_NAME property is one of the address properties for the original represented sender of a message. It is used in a conversation thread.

A client application sending a message on behalf of another client should set this property to the value of the PR_SENT_REPRESENTING_NAME property at the first submission of the message. Once set, it should never be changed.

For more information on the address properties, see About Base Address Properties.

PR_ORIGINAL_SENT_REPRESENTING_SEARCH_KEY

The PR_ORIGINAL_SENT_REPRESENTING_SEARCH_KEY property contains the search key of the messaging user on whose behalf the original message was sent.

Usage

Optional on message objects.

Details

Identifier 0x005F; property type PT_BINARY; property tag 0x005F0102

Remarks

The PR_ORIGINAL_SENT_REPRESENTING_SEARCH_KEY property is one of the address properties for the original represented sender of a message. It is used in a conversation thread.

A client application sending a message on behalf of another client should set this property to the value of the PR_SENT_REPRESENTING_SEARCH_KEY property at the first submission of the message. Once set, it should never be changed.

For more information on the address properties, see About Base Address Properties.

PR_ORIGINAL_SUBJECT

The PR_ORIGINAL_SUBJECT property contains the subject of an original message for use in a report about the message.

Usage

Optional on message objects.

Details

Identifier 0x0049; property type PT_TSTRING; property tag 0x0049001E (0x0049001F for Unicode)

Remarks

The PR_ORIGINAL_SUBJECT property is originally set to the same value as the PR_SUBJECT property.

The subject properties are typically small strings of fewer than 256 characters, and a message store provider is not obligated to support the OLE **IStream** interface on them. The client should always attempt access through the **IMAPIProp** interface first, and resort to **IStream** only if MAPI_E_NOT_ENOUGH_MEMORY is returned.

PR_ORIGINAL_SUBJECT corresponds to the X.400 attribute IM_SUBJECT.

PR_ORIGINAL_SUBMIT_TIME

The PR_ORIGINAL_SUBMIT_TIME property contains the original submission date and time of the message in the report.

Usage

Optional on report message objects.

Details

Identifier 0x004E; property type PT_SYSTIME; property tag 0x004E0040

Remarks

At first submission of a message, a client application should set the PR_ORIGINAL_SUBMIT_TIME property to the value of the PR_CLIENT_SUBMIT_TIME property. It is not changed when the message is forwarded. This is used in reports only.

PR_ORIGINALLY_INTENDED_RECIP_ADDRTYPE

The PR_ORIGINALLY_INTENDED_RECIP_ADDRTYPE property contains the address type of the originally intended recipient of an autoforwarded message.

Usage

Optional on message objects.

Details

Identifier 0x007B; property type PT_TSTRING; property tag 0x007B001E (0x007B001F for Unicode)

Remarks

The PR_ORIGINALLY_INTENDED_RECIP_ADDRTYPE property is one of the address properties for the originally intended message recipient. It must be set by the automatic agent that has forwarded the message.

For more information on the address properties, see About Base Address Properties.

PR_ORIGINALLY_INTENDED_RECIP_ADDRTYPE corresponds to the X.400 attribute IM_IPM_INTENDED_RECIPIENT.

See Also

PR_ADDRTYPE property

PR_ORIGINALLY_INTENDED_RECIP_EMAIL_ADDRESS

The PR_ORIGINALLY_INTENDED_RECIP_EMAIL_ADDRESS property contains the e-mail address of the originally intended recipient of an autoforwarded message.

Usage

Optional on message objects.

Details

Identifier 0x007C; property type PT_TSTRING; property tag 0x007C001E (0x007C001F for Unicode)

Remarks

The PR_ORIGINALLY_INTENDED_RECIP_EMAIL_ADDRESS property is one of the address properties for the originally intended message recipient. It must be set by the automatic agent that has forwarded the message.

For more information on the address properties, see About Base Address Properties.

PR_ORIGINALLY_INTENDED_RECIP_EMAIL_ADDRESS corresponds to the X.400 report per-recipient attribute.

See Also

PR_EMAIL_ADDRESS property

PR_ORIGINALLY_INTENDED_RECIP_ENTRYID

The PR_ORIGINALLY_INTENDED_RECIP_ENTRYID property contains the entry identifier of the originally intended recipient of an autoforwarded message.

Usage

Optional on message objects.

Details

Identifier 0x1012; property type PT_BINARY; property tag 0x10120102

Remarks

The PR_ORIGINALLY_INTENDED_RECIP_ENTRYID property is one of the address properties for the originally intended message recipient. It must be set by the automatic agent that has forwarded the message.

For more information on the address properties, see About Base Address Properties.

PR_ORIGINALLY_INTENDED_RECIP_ENTRYID corresponds to the X.400 report per-recipient attribute.

See Also

PR_ENTRYID property

PR_ORIGINALLY_INTENDED_RECIPIENT_NAME

The PR_ORIGINALLY_INTENDED_RECIPIENT_NAME property contains the encoded name of the originally intended recipient of an autoforwarded message.

Usage

Optional on message objects.

Details

Identifier 0x0020; property type PT_BINARY; property tag 0x00200102

Remarks

The PR_ORIGINALLY_INTENDED_RECIPIENT_NAME property must be set by the automatic agent that has forwarded the message.

This property is not one of the address properties.

PR_ORIGINALLY_INTENDED_RECIPIENT_NAME corresponds to the X.400 attribute MH_T_ORIGINALLY_INTENDED_RECIP.

PR_ORIGINATING_MTA_CERTIFICATE

The PR_ORIGINATING_MTA_CERTIFICATE property contains an identifier for the message transfer agent (MTA) that originated the message.

Usage

Optional on message objects.

Details

Identifier 0x0E25; property type PT_BINARY; property tag 0x0E250102

Remarks

The PR_ORIGINATING_MTA_CERTIFICATE property, if set, is available on sent messages in the Sent Items folder.

PR_ORIGINATING_MTA_CERTIFICATE corresponds to the X.400 report per-message attribute.

See Also

PR_REPORTING_MTA_CERTIFICATE property

PR_ORIGINATOR_AND_DL_EXPANSION_HISTORY

The PR_ORIGINATOR_AND_DL_EXPANSION_HISTORY property contains information about a message originator and a distribution list expansion history.

Usage

Optional on message objects.

Details

Identifier 0x1002; property type PT_BINARY; property tag 0x10020102

Remarks

The PR_ORIGINATOR_AND_DL_EXPANSION_HISTORY property is used in report generation.

PR_ORIGINATOR_AND_DL_EXPANSION_HISTORY corresponds to the X.400 attribute MH_T_ORIG_AND_EXPANSION_HISTORY.

PR_ORIGINATOR_CERTIFICATE

The PR_ORIGINATOR_CERTIFICATE property contains an ASN.1 certificate for the message originator.

Usage

Optional on message objects.

Details

Identifier 0x0022; property type PT_BINARY; property tag 0x00220102

Remarks

The PR_ORIGINATOR_CERTIFICATE property is a copy of the originator's PR_USER_CERTIFICATE property.

PR_ORIGINATOR_CERTIFICATE corresponds to the X.400 attribute MH_T_ORIGINATOR_CERTIFICATE.

PR_ORIGINATOR_DELIVERY_REPORT_REQUESTED

The PR_ORIGINATOR_DELIVERY_REPORT_REQUESTED property contains TRUE if a message sender requests a delivery report for a particular recipient from the messaging system before the message is placed in the message store.

Usage

Optional on recipient subobjects.

Details

Identifier 0x0023; property type PT_BOOLEAN; property tag 0x0023000B

Remarks

The PR_ORIGINATOR_DELIVERY_REPORT_REQUESTED property is used to direct the messaging system in handling delivered messages. In this case, the message must also furnish the PR_ORIGINATOR_NON_DELIVERY_REPORT_REQUESTED property set to FALSE.

PR_ORIGINATOR_DELIVERY_REPORT_REQUESTED corresponds to the X.400 attribute MH_T_ORIGINATOR_REPORT_REQUEST.

PR_ORIGINATOR_NON_DELIVERY_REPORT_REQUESTED

The PR_ORIGINATOR_NON_DELIVERY_REPORT_REQUESTED property contains TRUE if a message sender requests a nondelivery report for a particular recipient.

Usage

Optional on recipient subobjects.

Details

Identifier 0x0C08; property type PT_BOOLEAN; property tag 0x0C08000B

Remarks

The PR_ORIGINATOR_NON_DELIVERY_REPORT_REQUESTED property is used to direct the messaging system in handling undelivered messages. In this case, the message must also furnish the PR_ORIGINATOR_DELIVERY_REPORT_REQUESTED property set to FALSE.

PR_ORIGINATOR_NON_DELIVERY_REPORT_REQUESTED corresponds to the X.400 attribute MH_T_ORIGINATOR_REPORT_REQUEST.

PR_ORIGINATOR_REQUESTED_ALTERNATE_RECIPIENT

The PR_ORIGINATOR_REQUESTED_ALTERNATE_RECIPIENT property contains an entry identifier for an alternate recipient designated by the sender.

Usage

Optional on recipient subobjects.

Details

Identifier 0x0C09; property type PT_BINARY; property tag 0x0C090102

Remarks

The PR_ORIGINATOR_REQUESTED_ALTERNATE_RECIPIENT property is used in autoforwarded messages. If autoforwarding is not permitted or if no alternate recipient has been designated, a nondelivery report should be generated.

PR_ORIGINATOR_RETURN_ADDRESS

The PR_ORIGINATOR_RETURN_ADDRESS property contains the binary-encoded return address of the message originator.

Usage

Optional on message objects.

Details

Identifier 0x0024; property type PT_BINARY; property tag 0x00240102

Remarks

The PR_ORIGINATOR_RETURN_ADDRESS property corresponds to the X.400 attribute MH_T_ORIGINATOR_RETURN_ADDRESS.

PR_OTHER_TELEPHONE_NUMBER

The PR_OTHER_TELEPHONE_NUMBER property contains an alternate telephone number for the recipient.

Usage

Optional on messaging user objects.

Details

Identifier 0x3A1F; property type PT_TSTRING; property tag 0x3A1F001E (0x3A1F001F for Unicode)

Remarks

The PR_OTHER_TELEPHONE_NUMBER property is one of the properties that provide identification and access information for a recipient. These properties are defined by the recipient and the recipient's organization. For more information on these properties, see About Messaging User Objects.

PR_OTHER_TELEPHONE_NUMBER is used for a telephone number other than at the recipient's place of business, home, or office.

PR_OWN_STORE_ENTRYID

The PR_OWN_STORE_ENTRYID property contains the entry identifier of a transport's tightly coupled message store.

Usage

Optional on status objects and as a column entry in status tables.

Details

Identifier 0x3E06; property type PT_BINARY; property tag 0x3E060102

Remarks

The PR_OWN_STORE_ENTRYID property specifies the entry identifier for the tightly coupled store, if one exists. For example, a transport provider can specify the private folder store entry identifier so that the MAPI spooler can connect the transport provider to the store.

See Also

PR_STORE_ENTRYID property

PR_OWNER_APPT_ID

The PR_OWNER_APPT_ID property contains an identifier for an appointment in the owner's schedule.

Usage

Required on message objects used for scheduling.

Details

Identifier 0x0062; property type PT_LONG; property tag 0x00620003

Remarks

The PR_OWNER_APPT_ID property is used in meeting requests. It does not represent an entry identifier, but a long integer that uniquely identifies the appointment within the sender's schedule.

See Also

PR_ORIGINAL_AUTHOR_SEARCH_KEY property

PR_PAGER_TELEPHONE_NUMBER

The PR_PAGER_TELEPHONE_NUMBER property contains the recipient's pager telephone number.

Usage

Optional on messaging user objects.

Details

Identifier 0x3A21; property type PT_TSTRING; property tag 0x3A21001E (0x3A21001F for Unicode)

Remarks

The PR_PAGER_TELEPHONE_NUMBER property is one of the properties that provide identification and access information for a recipient. These properties are defined by the recipient and the recipient's organization. For more information on these properties, see About Messaging User Objects.

MAPI also supports the PR_BEEPER_TELEPHONE_NUMBER property that is synonymous with PR_PAGER_TELEPHONE_NUMBER.

PR_PARENT_DISPLAY

The PR_PARENT_DISPLAY property contains the display name of the folder in which a message was found during a search.

Usage

Required as a column entry in contents tables for search-results folders.

Details

Identifier 0x0E05; property type PT_TSTRING; property tag 0x0E05001E (0x0E05001F for Unicode)

Remarks

The PR_PARENT_DISPLAY property is not a property of any object. It can only appear in the contents table of a search-results folder.

The PR_PARENT_DISPLAY, PR_PARENT_ENTRYID, and PR_PARENT_KEY properties are not related to each other. They belong to entirely different contexts.

PR_PARENT_ENTRYID

The PR_PARENT_ENTRYID property contains the entry identifier of the folder containing a folder or message.

Usage

Required on folder and message objects.
Optional as a column entry in folder contents tables.
Computed by message stores for all folders and messages.

Details

Identifier 0x0E09; property type PT_BINARY; property tag 0x0E090102

Remarks

For a message store root folder, the PR_PARENT_ENTRYID property contains the folder's own entry identifier.

The PR_PARENT_DISPLAY, PR_PARENT_ENTRYID, and PR_PARENT_KEY properties are not related to each other. They belong to entirely different contexts.

See Also

PR_FOLDER_TYPE property

PR_PARENT_KEY

The PR_PARENT_KEY property was originally meant to contain a value used in correlating conversation threads.

Usage

Never used.

Details

Identifier 0x0025; property type PT_BINARY; property tag 0x00250102

Remarks

Do not use this property. It is not supported in MAPI 1.0.

PR_PARENT_KEY corresponds to the X.400 attribute IM_REPLIED_TO_IPM.

PR_PHYSICAL_DELIVERY_BUREAU_FAX_DELIVERY

The PR_PHYSICAL_DELIVERY_BUREAU_FAX_DELIVERY property contains TRUE if the messaging system should use a fax bureau for physical delivery of this message.

Usage

Optional on message objects.

Details

Identifier 0x0C0A; property type PT_BOOLEAN; property tag 0x0C0A000B

Remarks

The PR_PHYSICAL_DELIVERY_BUREAU_FAX_DELIVERY property corresponds to the X.400 attribute MH_T_BUREAU_FAX_DELIVERY.

PR_PHYSICAL_DELIVERY_MODE

The PR_PHYSICAL_DELIVERY_MODE property contains a bitmask of flags defining the physical delivery mode (for example, special delivery) for a message designated for a specific recipient.

Usage

Optional on recipient subobjects.

Details

Identifier 0x0C0B; property type PT_LONG; property tag 0x0C0B0003

Remarks

The PR_PHYSICAL_DELIVERY_MODE property corresponds to the X.400 attribute MH_T_POSTAL_MODE.

PR_PHYSICAL_DELIVERY_REPORT_REQUEST

The PR_PHYSICAL_DELIVERY_REPORT_REQUEST property contains the mode of a report to be delivered to a particular message recipient upon completion of physical message delivery or delivery by the message handling system.

Usage

Optional on recipient subobjects.

Details

Identifier 0x0C0C; property type PT_LONG; property tag 0x0C0C0003

Remarks

The PR_PHYSICAL_DELIVERY_REPORT_REQUEST property corresponds to the X.400 attribute MH_T_POSTAL_REPORT.

PR_PHYSICAL_FORWARDING_ADDRESS

The PR_PHYSICAL_FORWARDING_ADDRESS property contains the physical forwarding address of a message recipient and is used only with message reports.

Usage

Optional on recipient subobjects.

Details

Identifier 0x0C0D; property type PT_BINARY; property tag 0x0C0D0102

Remarks

The PR_PHYSICAL_FORWARDING_ADDRESS property corresponds to the X.400 attribute MH_T_FORWARDING_ADDRESS.

PR_PHYSICAL_FORWARDING_ADDRESS_REQUESTED

The PR_PHYSICAL_FORWARDING_ADDRESS_REQUESTED property contains TRUE if a message sender requests the message transfer agent to attach a physical forwarding address for a message recipient.

Usage

Optional on recipient subobjects.

Details

Identifier 0x0C0E; property type PT_BOOLEAN; property tag 0x0C0E000B

Remarks

The PR_PHYSICAL_FORWARDING_ADDRESS_REQUESTED property corresponds to the X.400 attribute MH_T_FORWARDING_ADR_REQUESTED.

PR_PHYSICAL_FORWARDING_PROHIBITED

The PR_PHYSICAL_FORWARDING_PROHIBITED property contains TRUE if a message sender prohibits physical message forwarding for a specific recipient.

Usage

Optional on recipient subobjects.

Details

Identifier 0x0C0F; property type PT_BOOLEAN; property tag 0x0C0F000B

Remarks

The PR_PHYSICAL_FORWARDING_PROHIBITED property corresponds to the X.400 attribute MH_T_FORWARDING_PROHIBITED.

PR_PHYSICAL_RENDITION_ATTRIBUTES

The PR_PHYSICAL_RENDITION_ATTRIBUTES property contains an ASN.1 object identifier used for rendering message attachments.

Usage

Optional on attachment subobjects.

Details

Identifier 0x0C10; property type PT_BINARY; property tag 0x0C100102

Remarks

The PR_PHYSICAL_RENDITION_ATTRIBUTES property corresponds to the X.400 attribute MH_T_RENDITION_ATTRIBUTES.

PR_POST_OFFICE_BOX

The PR_POST_OFFICE_BOX property contains the number or identifier of the recipient's post office box.

Usage

Optional on messaging user objects.

Details

Identifier 0x3A2B; property type PT_TSTRING; property tag 0x3A2B001E (0x3A2B001F for Unicode)

Remarks

The PR_POST_OFFICE_BOX property is one of the properties that provide identification and access information for a recipient. These properties are defined by the recipient and the recipient's organization. For more information on this group of properties, see About Messaging User Objects.

PR_POSTAL_ADDRESS

The PR_POSTAL_ADDRESS property contains the recipient's postal address.

Usage

Optional on messaging user objects.

Details

Identifier 0x3A15; property type PT_TSTRING; property tag 0x3A15001E (0x3A15001F for Unicode)

Remarks

The PR_POSTAL_ADDRESS property is one of the properties that provide identification and access information for a recipient. These properties are defined by the recipient and the recipient's organization. For more information on this group of properties, see About Messaging User Objects.

PR_POSTAL_CODE

The PR_POSTAL_CODE property contains the postal code for the recipient's postal address.

Usage

Optional on messaging user objects.

Details

Identifier 0x3A2A; property type PT_TSTRING; property tag 0x3A2A001E (0x3A2A001F for Unicode)

Remarks

The PR_POSTAL_CODE property is one of the properties that provide identification and access information for a recipient. These properties are defined by the recipient and the recipient's organization. For more information on this group of properties, see About Messaging User Objects.

The postal code is specific to the recipient's country. In the United States of America, this property contains the ZIP code.

PR_PREPROCESS

The PR_PREPROCESS property contains TRUE if the message requires preprocessing.

Usage

Reserved.

Details

Identifier 0x0E22; property type PT_BOOLEAN; property tag 0x0E22000B

Remarks

Do not use this property.

See Also

PR_SUBMIT_FLAGS property

PR_PRIMARY_CAPABILITY

Obsolete address book property.

Usage

No longer used.

Details

Identifier 0x3904; property type PT_BINARY; property tag 0x39040102

Remarks

Do not use this property.

PR_PRIMARY_FAX_NUMBER

The PR_PRIMARY_FAX_NUMBER property contains the telephone number of the recipient's primary fax machine.

Usage

Optional on messaging user objects.

Details

Identifier 0x3A23; property type PT_TSTRING; property tag 0x3A23001E (0x3A23001F for Unicode)

Remarks

The PR_PRIMARY_FAX_NUMBER property is one of the properties that provide identification and access information for a recipient. These properties are defined by the recipient and the recipient's organization. For more information on this group of properties, see About Messaging User Objects.

PR_PRIMARY_TELEPHONE_NUMBER

The PR_PRIMARY_TELEPHONE_NUMBER property contains the recipient's primary telephone number.

Usage

Optional on messaging user objects.

Details

Identifier 0x3A1A; property type PT_TSTRING; property tag 0x3A1A001E (0x3A1A001F for Unicode)

Remarks

The PR_PRIMARY_TELEPHONE_NUMBER property is one of the properties that provide identification and access information for a recipient. These properties are defined by the recipient and the recipient's organization. For more information on this group of properties, see About Messaging User Objects.

PR_PRIMARY_TELEPHONE_NUMBER corresponds to the X.400 attribute IM_TELEPHONE_NUMBER.

PR_PRIORITY

The PR_PRIORITY property contains the relative priority of a message.

Usage

Optional on message objects.

Details

Identifier 0x0026; property type PT_LONG; property tag 0x00260003

Remarks

The PR_PRIORITY and PR_IMPORTANCE properties should not be confused. Importance indicates a value to users, while priority indicates the order or speed at which the message should be sent by the messaging system software. Higher priority usually indicates a higher cost. Higher importance usually is associated with a different display by the user interface.

The priority of a report message should be the same as the priority of the original message being reported.

PR_PRIORITY can have exactly one of the following values:

Value	Description
PRIO_NONURGENT	The message is not urgent.
PRIO_NORMAL	The message has normal priority.
PRIO_URGENT	The message is urgent.

PR_PRIORITY corresponds to the X.400 attribute MH_T_PRIORITY.

PR_PROFILE_NAME

The PR_PROFILE_NAME property contains the name of the profile.

Usage

Computed by providers on profile section objects

Details

Identifier 0x3D12; property type PT_TSTRING, property tag 0x3D12001E (0x3D12001F for Unicode)

Remarks

A provider's implementation of the **ServiceEntry** function can use the PR_PROFILE_NAME property to discover the profile name.

Client applications can use PR_PROFILE_NAME as a convenient alternative to obtaining the profile name by examining the PR_DISPLAY_NAME property in the MAPI subsystem's status table row.

PR_PROFILE_NAME may not be unique across time, for example where a profile is deleted and later recreated with the same name. MAPI furnishes a totally unique PR_SEARCH_KEY property in a hard-coded profile section called MUID_PROFILE_INSTANCE.

PR_PROOF_OF_DELIVERY

The PR_PROOF_OF_DELIVERY property contains an ASN.1 proof of delivery value.

Usage

Optional on recipient subobjects.

Details

Identifier 0x0C11; property type PT_BINARY; property tag 0x0C110102

Remarks

The PR_PROOF_OF_DELIVERY property corresponds to the X.400 attribute MH_T_PROOF_OF_DELIVERY.

PR_PROOF_OF_DELIVERY_REQUESTED

The PR_PROOF_OF_DELIVERY_REQUESTED property contains TRUE if a message sender requests proof of delivery for a particular recipient.

Usage

Optional on recipient subobjects.

Details

Identifier 0x0C12; property type PT_BOOLEAN; property tag 0x0C12000B

Remarks

The PR_PROOF_OF_DELIVERY_REQUESTED property corresponds to the X.400 attribute MH_T_PROOF_OF_DELIV_REQUESTED.

PR_PROOF_OF_SUBMISSION

The PR_PROOF_OF_SUBMISSION property contains an ASN.1 proof of submission value.

Usage

Optional on message objects.

Details

Identifier 0x0E26; property type PT_BINARY; property tag 0x0E260102

Remarks

The PR_PROOF_OF_SUBMISSION property corresponds to an X.400 submission envelope per-message attribute.

PR_PROOF_OF_SUBMISSION_REQUESTED

The PR_PROOF_OF_SUBMISSION_REQUESTED property contains TRUE if a message sender requests proof that the message transfer system has submitted a message for delivery to the originally intended recipient.

Usage

Optional on message objects.

Details

Identifier 0x0028; property type PT_BOOLEAN; property tag 0x0028000B

Remarks

The PR_PROOF_OF_SUBMISSION_REQUESTED property corresponds to the X.400 attribute MH_T_PROOF_OF_SUBMSN_REQUESTED.

PR_PROVIDER_DISPLAY

The PR_PROVIDER_DISPLAY property contains the vendor-defined display name for a service provider.

Usage

Required on profile section objects.

Details

Identifier 0x3006; property type PT_TSTRING; property tag 0x3006001E (0x3006001F for Unicode)

Remarks

The PR_PROVIDER_DISPLAY and PR_PROVIDER_DLL_NAME properties are defined only on profile sections belonging to service providers. They must be present in MAPISVC.INF.

PR_PROVIDER_DLL_NAME

The PR_PROVIDER_DLL_NAME property contains the base filename of the MAPI service provider DLL.

Usage

Required on profile section objects.

Details

Identifier 0x300A; property type PT_TSTRING; property tag 0x300A001E (0x300A001F for Unicode)

Remarks

MAPI uses a DLL file naming convention. The base filename contains up to six characters that uniquely identify the DLL. MAPI appends the string 32 to the base DLL name to identify the version that runs on 32-bit platforms. For example, when the name MAPI.DLL is specified, MAPI constructs the name MAPI32.DLL to represent the corresponding 32-bit version of the DLL.

The PR_PROVIDER_DLL_NAME property should specify the base name. MAPI appends the string 32 as appropriate. Including the string 32 as part of the PR_PROVIDER_DLL_NAME property results in an error.

See Also

PR_SERVICE_DLL_NAME property

PR_PROVIDER_ORDINAL

The PR_PROVIDER_ORDINAL property contains the zero-based index of a service provider's position in the provider table.

Usage

Computed by MAPI on provider table objects.

Details

Identifier 0x300D; property type PT_LONG; property tag 0x300D0003

Remarks

Obtain the provider table by calling the **IMsgServiceAdmin::GetProviderTable** method. Sort the provider table on the PR_PROVIDER_ORDINAL column to display the transport order.

PR_PROVIDER_SUBMIT_TIME

The PR_PROVIDER_SUBMIT_TIME property contains the date and time a transport provider passed a message to its underlying messaging system.

Usage

Optional on message objects.

Details

Identifier 0x0048; property type PT_SYSTIME; property tag 0x00480040

Remarks

The PR_PROVIDER_SUBMIT_TIME property is set by the outgoing transport provider at the time a message is sent.

PR_PROVIDER_SUBMIT_TIME corresponds to an X.400 submission envelope per-message attribute.

PR_PROVIDER_UID

The PR_PROVIDER_UID property contains a **MAPIUID** structure of the service provider that is handling a message.

Usage

Computed by all service providers on provider table objects.

Details

Identifier 0x300C; property type PT_BINARY; property tag 0x300C0102

Remarks

The PR_PROVIDER_UID property contains a **MAPIUID** structure associated with, and usually hard-coded by, the provider. It is typically used by a client application that is interested in only the address book containers supplied by a particular provider.

PR_PROVIDER_UID appears only as a column entry in the provider table.

PR_RADIO_TELEPHONE_NUMBER

The PR_RADIO_TELEPHONE_NUMBER property contains the recipient's radio telephone number.

Usage

Optional on messaging user objects.

Details

Identifier 0x3A1D; property type PT_TSTRING; property tag 0x3A1D001E (0x3A1D001F for Unicode)

Remarks

The PR_RADIO_TELEPHONE_NUMBER property is one of the properties that provide identification and access information for a recipient. These properties are defined by the recipient and the recipient's organization. For more information on this group of properties, see About Messaging User Objects.

PR_RCVD_REPRESENTING_ADDRTYPE

The PR_RCVD_REPRESENTING_ADDRTYPE property contains the address type for the messaging user represented by the user actually receiving the message.

Usage

Required on message objects.

Details

Identifier 0x0077; property type PT_TSTRING; property tag 0x0077001E (0x0077001F for Unicode)

Remarks

The PR_RCVD_REPRESENTING_ADDRTYPE property is one of the address properties for the messaging user being represented by the receiving user. It must be set by the incoming transport provider, which is also responsible for authorization or verification of the delegate. If no messaging user is being represented, PR_RCVD_REPRESENTING_ADDRTYPE should be set to the address type contained in the PR_RECEIVED_BY_ADDRTYPE property.

A client application replying to a message received on behalf of another client should copy PR_RCVD_REPRESENTING_ADDRTYPE from the received message into the PR_SENT_REPRESENTING_ADDRTYPE property for the reply.

For more information on the address properties, see About Base Address Properties.

See Also

PR_ADDRTYPE property

PR_RCVD_REPRESENTING_EMAIL_ADDRESS

The PR_RCVD_REPRESENTING_EMAIL_ADDRESS property contains the e-mail address for the messaging user represented by the receiving user.

Usage

Required on message objects.

Details

Identifier 0x0078; property type PT_TSTRING; property tag 0x0078001E (0x0078001F for Unicode)

Remarks

The PR_RCVD_REPRESENTING_EMAIL_ADDRESS property is one of the address properties for the messaging user being represented by the receiving user. It must be set by the incoming transport provider, which is also responsible for authorization or verification of the delegate. If no messaging user is being represented, PR_RCVD_REPRESENTING_EMAIL_ADDRESS should be set to the e-mail address contained in the PR_RECEIVED_BY_EMAIL_ADDRESS property.

A client application replying to a message received on behalf of another client should copy PR_RCVD_REPRESENTING_EMAIL_ADDRESS from the received message into the PR_SENT_REPRESENTING_EMAIL_ADDRESS property for the reply.

For more information on the address properties, see About Base Address Properties.

See Also

PR_EMAIL_ADDRESS property

PR_RCVD_REPRESENTING_ENTRYID

The PR_RCVD_REPRESENTING_ENTRYID property contains the entry identifier for the messaging user represented by the receiving user.

Usage

Required on message objects.

Details

Identifier 0x0043; property type PT_BINARY; property tag 0x00430102

Remarks

The PR_RCVD_REPRESENTING_ENTRYID property is one of the address properties for the messaging user being represented by the receiving user. It must be set by the incoming transport provider, which is also responsible for authorization or verification of the delegate. If no messaging user is being represented, PR_RCVD_REPRESENTING_ENTRYID should be set to the entry identifier contained in the PR_RECEIVED_BY_ENTRYID property.

A client application replying to a message received on behalf of another client should copy PR_RCVD_REPRESENTING_ENTRYID from the received message into the PR_SENT_REPRESENTING_ENTRYID property for the reply.

For more information on the address properties, see About Base Address Properties.

See Also

PR_ENTRYID property

PR_RCVD_REPRESENTING_NAME

The PR_RCVD_REPRESENTING_NAME property contains the display name for the messaging user represented by the receiving user.

Usage

Required on message objects.

Details

Identifier 0x0044; property type PT_TSTRING; property tag 0x0044001E (0x0044001F for Unicode)

Remarks

The PR_RCVD_REPRESENTING_NAME property is one of the address properties for the messaging user being represented by the receiving user. It must be set by the incoming transport provider, which is also responsible for authorization or verification of the delegate. If no messaging user is being represented, PR_RCVD_REPRESENTING_NAME should be set to the display name contained in the PR_RECEIVED_BY_NAME property.

A client application replying to a message received on behalf of another client should copy PR_RCVD_REPRESENTING_NAME from the received message into the PR_SENT_REPRESENTING_NAME property for the reply.

For more information on the address properties, see About Base Address Properties.

See Also

PR_DISPLAY_NAME property

PR_RCVD_REPRESENTING_SEARCH_KEY

The PR_RCVD_REPRESENTING_SEARCH_KEY property contains the search key for the messaging user represented by the receiving user.

Usage

Required on message objects.

Details

Identifier 0x0052; property type PT_BINARY; property tag 0x00520102

Remarks

The PR_RCVD_REPRESENTING_SEARCH_KEY is one of the address properties for the messaging user being represented by the receiving user. It must be set by the incoming transport provider, which is also responsible for authorization or verification of the delegate. If no messaging user is being represented, PR_RCVD_REPRESENTING_SEARCH_KEY should be set to the search key contained in the PR_RECEIVED_BY_SEARCH_KEY property.

A client application replying to a message received on behalf of another client should copy PR_RCVD_REPRESENTING_SEARCH_KEY from the received message into the PR_SENT_REPRESENTING_SEARCH_KEY property for the reply.

For more information on the address properties, see About Base Address Properties.

See Also

PR_SEARCH_KEY property

PR_READ_RECEIPT_ENTRYID

The PR_READ_RECEIPT_ENTRYID property contains an entry identifier for the messaging user to which the messaging system should direct a read report for this message.

Usage

Optional on message objects.

Details

Identifier 0x0046; property type PT_BINARY; property tag 0x00460102

Remarks

The PR_READ_RECEIPT_ENTRYID property is ignored unless the PR_READ_RECEIPT_REQUESTED property is set to TRUE.

If a client application wants to receive read reports itself, it can leave PR_READ_RECEIPT_ENTRYID unset or set it to the entry identifier contained in the PR_SENDER_ENTRYID property at message submission time.

PR_READ_RECEIPT_REQUESTED

The PR_READ_RECEIPT_REQUESTED property contains TRUE if a message sender wants the messaging system to generate a read report when the recipient has read a message.

Usage

Optional on recipient subobjects.

Details

Identifier 0x0029; property type PT_BOOLEAN; property tag 0x0029000B

Remarks

The PR_READ_RECEIPT_REQUESTED property must be set to TRUE to validate the values in the PR_READ_RECEIPT_ENTRYID and PR_READ_RECEIPT_SEARCH_KEY properties.

If a message with PR_READ_RECEIPT_REQUESTED set is deleted or expires before the messaging system can generate a read report, a nonread report is generated.

PR_READ_RECEIPT_REQUESTED corresponds to the X.400 attribute IM_NOTIFICATION_REQUEST.

PR_READ_RECEIPT_SEARCH_KEY

The PR_READ_RECEIPT_SEARCH_KEY property contains a search key for the messaging user to which the messaging system should direct a read report for a message.

Usage

Optional on message objects.

Details

Identifier 0x0053; property type PT_BINARY; property tag 0x00530102

Remarks

The PR_READ_RECEIPT_SEARCH_KEY property is ignored unless the PR_READ_RECEIPT_REQUESTED property is set to TRUE.

If a client application wants to receive read reports itself, it can leave PR_READ_RECEIPT_SEARCH_KEY unset or set it to the search key contained in the PR_SENDER_SEARCH_KEY property at message submission time.

PR_RECEIPT_TIME

The PR_RECEIPT_TIME property contains the date and time a delivery report is generated.

Usage

Required on report message objects.

Details

Identifier 0x002A; property type PT_SYSTIME; property tag 0x002A0040

Remarks

The PR_RECEIPT_TIME property must be set by the message store provider receiving the original message and generating the report. The report is usually generated with the **IMAPISupport::ReadReceipt** method.

PR_RECEIPT_TIME corresponds to the X.400 attribute IM_RECEIPT_TIME.

See Also

PR_REPORT_ENTRYID property

PR_RECEIVE_FOLDER_SETTINGS

The PR_RECEIVE_FOLDER_SETTINGS property contains a table of a message store's receive folder settings.

Usage

Required on message store objects.

Details

Identifier 0x3415; property type PT_OBJECT; property tag 0x3415000D

Remarks

The PR_RECEIVE_FOLDER_SETTINGS property can be excluded in **IMAPIProp::CopyTo** operations or included in **IMAPIProp::CopyProps** operations. As a property of type PT_OBJECT, it cannot be successfully retrieved by the **IMAPIProp::GetProps** method; its contents should be accessed by the **IMAPIProp::OpenProperty** method, requesting the interface with identifier IID_IMAPITable. Service providers must report it to the **IMAPIProp::GetPropList** method if it is set, but can optionally report it or not if it is not set.

To retrieve table contents, a client application should call the **IMsgStore::GetReceiveFolderTable** method. For more information on receive folder tables, see About Receive Folder Tables.

PR_RECEIVE_FOLDER_SETTINGS contains a table of mappings of the receive folders for the message store. Calling **OpenProperty** on this property is equivalent to calling **GetReceiveFolderTable** on the message store.

PR_RECEIVED_BY_ADDRTYPE

The PR_RECEIVED_BY_ADDRTYPE property contains the e-mail address type, such as SMTP, for the messaging user that actually receives the message.

Usage

Required on message objects.

Details

Identifier 0x0075; property type PT_TSTRING; property tag 0x0075001E (0x0075001F for Unicode)

Remarks

The PR_RECEIVED_BY_ADDRTYPE property is one of the address properties for the messaging user that actually receives the message. It must be set by the incoming transport provider.

The address type string can contain only the uppercase alphabetic characters A through Z and the numbers 0 through 9. PR_RECEIVED_BY_ADDRTYPE qualifies the PR_RECEIVED_BY_EMAIL_ADDRESS property by specifying an address type, such as SMTP, thereby indicating how the address should be constructed.

For more information on the address properties, see About Base Address Properties.

See Also

PR_ADDRTYPE property

PR_RECEIVED_BY_EMAIL_ADDRESS

The PR_RECEIVED_BY_EMAIL_ADDRESS property contains the e-mail address for the messaging user that actually receives the message.

Usage

Required on message objects.

Details

Identifier 0x0076; property type PT_TSTRING; property tag 0x0076001E (0x0076001F for Unicode)

Remarks

The PR_RECEIVED_BY_EMAIL_ADDRESS property is one of the address properties for the messaging user that actually receives the message. It must be set by the incoming transport provider.

For more information on the address properties, see About Base Address Properties.

See Also

PR_EMAIL_ADDRESS property

PR_RECEIVED_BY_ENTRYID

The PR_RECEIVED_BY_ENTRYID property contains the entry identifier of the messaging user that actually receives the message.

Usage

Required on message objects.

Details

Identifier 0x003F; property type PT_BINARY; property tag 0x003F0102

Remarks

The PR_RECEIVED_BY_ENTRYID property is one of the address properties for the messaging user that actually receives the message. It must be set by the incoming transport provider.

For more information on the address properties, see About Base Address Properties.

PR_RECEIVED_BY_ENTRYID corresponds to an X.400 delivery envelope per-message MH_T_ attribute.

See Also

PR_ENTRYID property

PR_RECEIVED_BY_NAME

The PR_RECEIVED_BY_NAME property contains the display name of the messaging user that actually receives the message.

Usage

Required on message objects.

Details

Identifier 0x0040; property type PT_TSTRING; property tag 0x0040001E (0x0040001F for Unicode)

Remarks

The PR_RECEIVED_BY_NAME property is one of the address properties for the messaging user that actually receives the message. It must be set by the incoming transport provider.

For more information on the address properties, see About Base Address Properties.

PR_RECEIVED_BY_NAME corresponds to the X.400 attribute MH_T_ACTUAL_RECIPIENT_NAME.

See Also

PR_DISPLAY_NAME property

PR_RECEIVED_BY_SEARCH_KEY

The PR_RECEIVED_BY_SEARCH_KEY property contains the search key of the messaging user that actually receives the message.

Usage

Required on message objects.

Details

Identifier 0x0051; property type PT_BINARY; property tag 0x00510102

Remarks

The PR_RECEIVED_BY_SEARCH_KEY property is one of the address properties for the messaging user that actually receives the message. It must be set by the incoming transport provider.

For more information on the address properties, see About Base Address Properties.

See Also

PR_SEARCH_KEY property

PR_RECIPIENT_CERTIFICATE

The PR_RECIPIENT_CERTIFICATE property contains a message recipient's ASN.1 certificate for use in a report.

Usage

Optional on recipient subobjects.

Details

Identifier 0x0C13; property type PT_BINARY; property tag 0x0C130102

Remarks

The PR_RECIPIENT_CERTIFICATE property is a copy of the recipient's PR_USER_CERTIFICATE property for use in a report. It can be used to prove to the originator that the recipient actually received the message, which a delivery report does not necessarily indicate.

PR_RECIPIENT_CERTIFICATE corresponds to the X.400 attribute MH_T_RECEIPT_CERTIFICATE.

See Also

PR_RECIPIENT_NUMBER_FOR_ADVICE property

PR_RECIPIENT_NUMBER_FOR_ADVICE

The PR_RECIPIENT_NUMBER_FOR_ADVICE property contains a message recipient's voice telephone number to call to advise of the physical delivery of a message.

Usage

Optional on recipient subobjects.

Details

Identifier 0x0C14; property type PT_TSTRING; property tag 0x0C14001E (0x0C14001F for Unicode)

Remarks

The PR_RECIPIENT_NUMBER_FOR_ADVICE property is meant to be used in conjunction with delivery to a physical destination, rather than an electronic mailbox, when the human recipient is not expected to be present at delivery. An example is the telephone number usually included on a fax cover sheet.

The PR_RECIPIENT_NUMBER_FOR_ADVICE property corresponds to the X.400 attribute MH_T_RECIP_NUMBER_FOR_ADVICE.

See Also

PR_RECIPIENT_REASSIGNMENT_PROHIBITED property

PR_RECIPIENT_REASSIGNMENT_PROHIBITED

The PR_RECIPIENT_REASSIGNMENT_PROHIBITED property contains TRUE if recipient reassignment is prohibited.

Usage

Optional on message objects.

Details

Identifier 0x002B; property type PT_BOOLEAN; property tag 0x002B000B

Remarks

The PR_RECIPIENT_REASSIGNMENT_PROHIBITED property corresponds to the X.400 attribute MH_T_REASSIGNMENT_PROHIBITED.

PR_RECIPIENT_STATUS

The PR_RECIPIENT_STATUS property contains a value used by the MAPI spooler in assigning delivery responsibility among transport providers.

Usage

Reserved.

Details

Identifier 0x0E15; property type PT_LONG; property tag 0x0E150003

Remarks

Do not use this property.

PR_RECIPIENT_TYPE

The PR_RECIPIENT_TYPE property contains the recipient type for a message recipient.

Usage

Required on recipient subobjects.

Details

Identifier 0x0C15; property type PT_LONG; property tag 0x0C150003

Remarks

The recipient type contained in the PR_RECIPIENT_TYPE property consists of one required value and one optional flag.

PR_RECIPIENT_TYPE must contain exactly one of the following values:

Value	Description
MAPI_TO	The recipient is a primary (To) recipient.
MAPI_CC	The recipient is a carbon copy (CC) recipient.
MAPI_BCC	The recipient is a blind carbon copy (BCC) recipient.
MAPI_P1	This is a resend of an earlier transmission. The recipient did not successfully receive the message on the previous attempt.

In addition, the following flag can be set:

MAPI_SUBMITTED
: This is a resend of an earlier transmission. The recipient has already received the message and does not need it to be sent again.

The MAPI_P1 value and the MAPI_SUBMITTED flag are used when a message is being retransmitted due to nondelivery to one or more of the intended recipients. For this retransmission, the client sets MAPI_SUBMITTED on every recipient that does not need the message again but should be displayed in the recipient list. For every recipient that did not receive the message previously, the client retains the original recipient with its PR_RECIPIENT_TYPE value unchanged, but additionally submits a copy of the recipient with MAPI_P1 in place of the original value. This copy, which is discarded before actual delivery, forces the recipient into the P1 envelope and guarantees physical retransmission to that recipient.

In X.400, the P1, or delivery envelope, is the information needed to deliver a message, including the recipient's address properties and any option flags controlling delivery and replies. The P2 or display envelope is the information usually displayed to each recipient other than the message text itself. It typically includes the subject, importance, priority, sensitivity, and submission time, as well as the primary and copied recipient names.

See Also

PR_RECIPIENT_STATUS property

PR_RECORD_KEY

The PR_RECORD_KEY property contains a unique binary-comparable identifier for a specific object.

Usage

Required on address book container, attachment, distribution list, folder, messaging user, message, and message store objects.

Details

Identifier 0x0FF9; property type PT_BINARY; property tag 0x0FF90102

Remarks

The PR_RECORD_KEY property facilitates locating references to an object, such as finding its row in a contents table. PR_RECORD_KEY cannot be used to open an object; use the entry identifier for that purpose.

An attachment subobject should be uniquely identified within a message by PR_RECORD_KEY. This identifier is the only attachment characteristic guaranteed to stay the same after the message is closed and reopened. The store provider must preserve PR_RECORD_KEY across sessions to ensure this guarantee.

For folders, this property contains a key used in the folder hierarchy table. Typically this is the same value as that provided by the PR_ENTRYID property.

For message stores, this property is identical to the PR_STORE_RECORD_KEY property.

In a message store object, PR_RECORD_KEY should be unique across all store providers. One way to do this is to combine the value of the PR_MDB_PROVIDER property for the store (unique to that provider type) with a **GUID** structure or other value unique to the specific message store.

PR_RECORD_KEY is always available through the **IMAPIProp::GetProps** method following the first call to the **IMAPIProp::SaveChanges** method. Some providers can make it available immediately after instantiation.

A client or service provider can compare PR_RECORD_KEY values using memcmp. This is not possible for entry identifier values. However, PR_RECORD_KEY is guaranteed to be unique only within the same message store or address book container; two objects from different containers can have the same PR_RECORD_KEY value.

One distinction between the record and search keys is that the record key is specific to the object, whereas the search key can be copied to other objects. For example, two copies of the object can have the same PR_SEARCH_KEY value but must have different PR_RECORD_KEY values.

The following table summarizes important differences among PR_ENTRYID, PR_RECORD_KEY, and PR_SEARCH_KEY.

Characteristic	PR_ENTRYID	PR_RECORD_KEY	PR_SEARCH_KEY
Required on attachment objects	No	Yes	No
Required on folder objects	Yes	Yes	No
Required on message store objects	Yes	Yes	No
Required on status objects	Yes	No	No

Creatable by client	No	No	Yes
Available before a call to **SaveChanges**	Maybe	Maybe	Messages – Yes Others – Maybe
Changed in a copy operation	Yes	Yes	No
Changeable by a client after a copy	No	No	Yes
Unique within ...	Entire world	Provider instance	Entire world
Binary comparable (as with memcmp)	No -- use **IMAPISupport::CompareEntryIDs**	Yes	Yes

PR_REDIRECTION_HISTORY

The PR_REDIRECTION_HISTORY property contains information about the route covered by a delivered message.

Usage

Optional on message objects.

Details

Identifier 0x002C; property type PT_BINARY; property tag 0x002C0102

Remarks

The PR_REDIRECTION_HISTORY property is used when a message is autoforwarded by an automatic agent.

PR_REDIRECTION_HISTORY corresponds to the X.400 attribute MH_T_REDIRECTION_HISTORY.

PR_REGISTERED_MAIL_TYPE

The PR_REGISTERED_MAIL_TYPE property contains the type of registration used for physical delivery of a message.

Usage

Optional on message objects.

Details

Identifier 0x0C16; property type PT_LONG; property tag 0x0C160003

Remarks

The PR_REGISTERED_MAIL_TYPE property corresponds to the X.400 attribute MH_T_REGISTRATION.

See Also

PR_X400_CONTENT_TYPE property

PR_RELATED_IPMS

The PR_RELATED_IPMS property contains a list of identifiers for messages to which a message is related.

Usage

Optional on message objects.

Details

Identifier 0x002D; property type PT_BINARY; property tag 0x002D0102

Remarks

The identifiers use the same specific construction rules as are used for the PR_SEARCH_KEY property.

PR_RELATED_IPMS corresponds to the X.400 attribute IM_RELATED_IPMS.

PR_REMOTE_PROGRESS

The PR_REMOTE_PROGRESS property contains a number indicating the status of a remote transfer.

Usage

Required on message objects.

Details

Identifier 0x3E0B; property type PT_LONG; property tag 0x3E0B0003

Remarks

If no transfer is in progress, the PR_REMOTE_PROGRESS property should be set to -1. If a transfer is in progress, it should be set to a value from 0 to 100 indicating the transfer's percent of completion.

The text associated with the numeric status code appears in the PR_REMOTE_PROGRESS_TEXT property.

The following flags can be set for PR_REMOTE_PROGRESS:

MSGSTATUS_REMOTE_DELETE
    The message transfer is deleted.
MSGSTATUS_REMOTE_DOWNLOAD
    The message transfer is in progress.

PR_REMOTE_PROGRESS_TEXT

The PR_REMOTE_PROGRESS_TEXT property contains an ASCII string indicating the status of a remote transfer.

Usage

Optional on message objects.

Details

Identifier 0x3E0C; property type PT_TSTRING; property tag 0x3E0C001E (0x3E0C001F for Unicode)

Remarks

A numeric code associated with this text is passed in the PR_REMOTE_PROGRESS property.

PR_REMOTE_VALIDATE_OK

The PR_REMOTE_VALIDATE_OK property contains TRUE if the remote viewer is allowed to call the **IMAPIStatus::ValidateState** method.

Usage

Optional as a column entry in status tables.

Details

Identifier 0x3E0D; property type PT_BOOLEAN; property tag 0x3E0D000B

Remarks

The PR_REMOTE_VALIDATE_OK property appears in the status table and offers some control over transport performance. It can be considered as another way of directing the remote viewer to idle. When it is set to TRUE, the remote viewer can call **IMAPIStatus::ValidateState** as often as desired. A value of FALSE indicates that the remote viewer cannot make any more calls.

The transport provider usually sets this property dynamically, setting the value to FALSE to disable additional calls when the transport provider has a sufficient amount of processing to perform. When the transport provider is done, it then sets the value to TRUE to allow the client application to make further **IMAPIStatus::ValidateState** calls.

PR_RENDERING_POSITION

The PR_RENDERING_POSITION property contains an offset, in characters, to use in rendering an attachment within the main message text.

Usage

Required on attachment subobjects.

Details

Identifier 0x370B; property type PT_LONG; property tag 0x370B0003

Remarks

When the supplied offset is -1 (0xFFFFFFFF), the attachment is not rendered using the PR_RENDERING_POSITION property. All values other than -1 indicate the position within PR_BODY at which the attachment is to be rendered.

Note The character at PR_RENDERING_POSITION in PR_BODY is replaced by the attachment. Typically this character is a blank, although a special placeholder character can also be used.

PR_RENDERING_POSITION is expressed in characters. In some character sets this is not equivalent to bytes. Unicode applications can compute the position based on two-byte characters. Double-Byte Character Set (DBCS) applications must scan the text up to the PR_RENDERING_POSITION value, since their character representation varies between one and two bytes per character.

This property should not be used with Rich Text Format (RTF) text. The rendering position is indicated in RTF by an escape sequence called the object attachment placeholder. This sequence consists of the string `\objattph` followed by a single character, normally a space, that will be replaced by the attachment rendering.

PR_REPLY_RECIPIENT_ENTRIES

The PR_REPLY_RECIPIENT_ENTRIES property contains a sized array of entry identifiers for recipients that are to get a reply.

Usage

Optional on message objects.

Details

Identifier 0x004F; property type PT_BINARY; property tag 0x004F0102

Remarks

The PR_REPLY_RECIPIENT_ENTRIES property contains a **FLATENTRYLIST** structure and is not a multivalued property.

When this property is not present, a reply is sent only to the user identified by PR_SENDER_ENTRYID. When PR_REPLY_RECIPIENT_ENTRIES and PR_REPLY_RECIPIENT_NAMES are defined, the reply is sent to all of the recipients identified by these two properties. A transport provider uses these properties to override the usual reply logic.

If either PR_REPLY_RECIPIENT_ENTRIES or PR_REPLY_RECIPIENT_NAMES is set, the other property must be set also. These properties must contain the same number of recipients, and they must contain them in the same order. Failure to observe these requirements can cause unpredictable results.

PR_REPLY_RECIPIENT_ENTRIES corresponds to the X.400 attribute IM_REPLY_RECIPIENTS.

PR_REPLY_RECIPIENT_NAMES

The PR_REPLY_RECIPIENT_NAMES property contains a list of display names for recipients that are to get a reply.

Usage

Optional on message objects.

Details

Identifier 0x0050; property type PT_TSTRING; property tag 0x0050001E (0x0050001F for Unicode)

Remarks

The PR_REPLY_RECIPIENT_NAMES property contains one string, with the display names separated by semicolons.

When this property is not present, a reply is sent only to the user identified by PR_SENDER_NAME. When PR_REPLY_RECIPIENT_ENTRIES and PR_REPLY_RECIPIENT_NAMES are defined, the reply is sent to all of the recipients identified by these two properties. A transport provider uses these properties to override the usual reply logic.

If either PR_REPLY_RECIPIENT_ENTRIES or PR_REPLY_RECIPIENT_NAMES is set, the other property must be set also. These properties must contain the same number of recipients, and they must contain them in the same order. Failure to observe these requirements can cause unpredictable results.

PR_REPLY_REQUESTED

The PR_REPLY_REQUESTED property contains TRUE if a message sender requests a reply from a recipient.

Usage

Optional on message objects.

Details

Identifier 0x0C17; property type PT_BOOLEAN; property tag 0x0C17000B

Remarks

The PR_REPLY_REQUESTED property corresponds to the X.400 attribute IM_REPLY_REQUESTED.

See Also

PR_REPLY_TIME property

PR_REPLY_TIME

The PR_REPLY_TIME property contains the date and time by which a reply is expected for a message.

Usage

Optional on message objects.

Details

Identifier 0x0030; property type PT_SYSTIME; property tag 0x00300040

Remarks

The PR_REPLY_TIME property corresponds to the X.400 attribute IM_REPLY_TIME.

See Also

PR_REPLY_REQUESTED property

PR_REPORT_ENTRYID

The PR_REPORT_ENTRYID property contains the entry identifier for the recipient that should get reports for this message.

Usage

Optional on message objects.

Details

Identifier 0x0045; property type PT_BINARY; property tag 0x00450102

Remarks

The PR_REPORT_ENTRYID property is one of the address properties for the recipient that the sender has delegated to receive any reports generated for this message.

A client application that needs to route reports to another user should set this property at message submission time. If it is not set, the reports are sent to the message sender.

For more information on the address properties, see About Base Address Properties.

See Also

PR_SENDER_ENTRYID property

PR_REPORT_NAME

The PR_REPORT_NAME property contains the display name for the recipient that should get reports for this message.

Usage

Optional on message objects.

Details

Identifier 0x003A; property type PT_TSTRING; property tag 0x003A001E (0x003A001F for Unicode)

Remarks

The PR_REPORT_NAME property is one of the address properties for the recipient that the sender has delegated to receive any reports generated for this message.

A client application that needs to route reports to another user should set this property at message submission time. If it is not set, the reports are sent to the message sender.

For more information on the address properties, see About Base Address Properties.

See Also

PR_SENDER_NAME property

PR_REPORT_SEARCH_KEY

The PR_REPORT_SEARCH_KEY property contains the search key for the recipient that should get reports for this message.

Usage

Optional on message objects.

Details

Identifier 0x0054; property type PT_BINARY; property tag 0x00540102

Remarks

The PR_REPORT_SEARCH_KEY property is one of the address properties for the recipient that the sender has delegated to receive any reports generated for this message.

A client application that needs to route reports to another user should set this property at message submission time. If it is not set, the reports are sent to the message sender.

For more information on the address properties, see About Base Address Properties.

See Also

PR_SENDER_SEARCH_KEY property

PR_REPORT_TAG

The PR_REPORT_TAG property contains a binary tag value that the messaging system should copy to any report generated for the message.

Usage

Required on message objects.

Details

Identifier 0x0031; property type PT_BINARY; property tag 0x00310102

Remarks

The PR_REPORT_TAG property is used to correlate a message and its corresponding report.

See Also

PR_REPORT_NAME property

PR_REPORT_TEXT

The PR_REPORT_TEXT property contains optional text for a report generated by the messaging system.

Usage

Required on report message objects.

Details

Identifier 0x1001; property type PT_TSTRING; property tag 0x1001001E (0x1001001F for Unicode)

Remarks

Usually, the text contained in the PR_REPORT_TEXT property is generated in response to a delivery or nondelivery report or a read or nonread report received from the underlying messaging system, but is not itself text that was transferred through that system.

PR_REPORT_TIME

The PR_REPORT_TIME property contains the date and time when the messaging system generated a report.

Usage

Required on read and nonread report message objects.
Required on recipient subobjects within delivery and nondelivery report message objects.

Details

Identifier 0x0032; property type PT_SYSTIME; property tag 0x00320040

Remarks

The PR_REPORT_TIME property represents a per-recipient property on delivery and nondelivery reports and a per-message property on read and nonread reports.

PR_REPORTING_DL_NAME

The PR_REPORTING_DL_NAME property contains the display name of a distribution list for which the messaging system is delivering a report.

Usage

Optional on report message objects.

Details

Identifier 0x1003; property type PT_BINARY; property tag 0x10030102

Remarks

The PR_REPORTING_DL_NAME property corresponds to the X.400 attribute MH_T_REPORTING_DL_NAME.

PR_REPORTING_MTA_CERTIFICATE

The PR_REPORTING_MTA_CERTIFICATE property contains an identifier for the message transfer agent that generated a report.

Usage

Optional on report message objects.

Details

Identifier 0x1004; property type PT_BINARY; property tag 0x10040102

Remarks

The PR_REPORTING_MTA_CERTIFICATE property corresponds to an X.400 report per-message attribute.

See Also

PR_ORIGINATING_MTA_CERTIFICATE property

PR_REQUESTED_DELIVERY_METHOD

The PR_REQUESTED_DELIVERY_METHOD property contains a binary array of delivery methods (service providers), in order of a message sender's preference.

Usage

Optional on message objects.

Details

Identifier 0x0C18; property type PT_BINARY; property tag 0x0C180102

Remarks

The array contained in the PR_REQUESTED_DELIVERY_METHOD property consists of 32-bit ASN.1 identifiers for each of the service providers. Typical identifier values are MH_DM_TELEX and MH_DM_G3FAX.

PR_REQUESTED_DELIVERY_METHOD corresponds to the X.400 attribute MH_T_PREFERRED_DELIVERY_MODE.

See Also

PR_PROOF_OF_DELIVERY_REQUESTED property

PR_RESOURCE_FLAGS

The PR_RESOURCE_FLAGS property contains a bitmask of flags for message services and providers.

Usage

Required as a column entry in message service, message store, provider, and status tables. Required for message services and all service providers in MAPISVC.INF.

Details

Identifier 0x3009; property type PT_LONG; property tag 0x30090003

Remarks

The flags are of three classes: static, modifiable, and dynamic. Static flags are set by MAPI from data in MAPISVC.INF and never altered. Modifiable flags are set by MAPI from MAPISVC.INF but can be subsequently changed. Dynamic flags can be set and reset by MAPI methods.

For a message service, one or more of the following flags can be set in PR_RESOURCE_FLAGS:

SERVICE_CREATE_WITH_STORE
: Reserved. Do not use.

SERVICE_DEFAULT_STORE
: Dynamic. The message service contains the default store. A user interface should be displayed prompting the user for confirmation before deleting or moving this service out of the profile.

SERVICE_NO_PRIMARY_IDENTITY
: Static. The service level flag that should be set to indicate that none of the providers in the message service can be used to supply an identity. Either this flag or SERVICE_PRIMARY_IDENTITY should be set, but not both.

SERVICE_PRIMARY_IDENTITY
: Modifiable. The corresponding message service contains the provider used for the primary identity for this session. Use **IMsgServiceAdmin::SetPrimaryIdentity** to set this flag. Either this flag or SERVICE_NO_PRIMARY_IDENTITY should be set, but not both.

SERVICE_SINGLE_COPY
: Static. Any attempt to create or copy this message service into a profile where the service already exists will fail. To create a single copy message service add the PR_RESOURCE_FLAGS property to the service's section in MAPISVC.INF and set this flag.

For a service provider, one or more of the following flags can be set in PR_RESOURCE_FLAGS:

HOOK_INBOUND
: Static. The spooler hook needs to process inbound messages.

HOOK_OUTBOUND
: Static. The spooler hook needs to process outbound messages.

STATUS_DEFAULT_OUTBOUND
: Modifiable. This identity should be applied to outbound messages if the profile contains multiple instances of this transport provider.

STATUS_DEFAULT_STORE
: Modifiable. This message store is the default store for the profile.

STATUS_NEED_IPM_TREE
: Dynamic. The standard folders in this message store, including the interpersonal message (IPM) root folder, have not yet been verified. MAPI sets and clears this flag.

STATUS_NO_DEFAULT_STORE
: Static. This message store is incapable of becoming the default message store for the profile.

STATUS_NO_PRIMARY_IDENTITY
Static. This provider does not furnish an identity in its status row.

STATUS_OWN_STORE
Static. This transport provider is tightly coupled with a message store and furnishes the PR_OWN_STORE_ENTRYID property in its status row.

STATUS_PRIMARY_IDENTITY
Modifiable. This provider furnishes an identity in its status row that will be returned from **IMAPISession::QueryIdentity**.

STATUS_PRIMARY_STORE
Modifiable. This message store is to be used when a client application logs on. Once opened, this store should be set as the default store for the profile.

STATUS_SECONDARY_STORE
Modifiable. This message store is to be used if the primary store is not available when a client application logs on. Once opened, this store should be set as the default store for the profile.

STATUS_SIMPLE_STORE
Dynamic. This message store will be used by Simple MAPI or CMC as its default message store.

STATUS_TEMP_SECTION
Dynamic. This message store should not be published in the message store table and will be deleted from the profile after logoff.

STATUS_XP_PREFER_LAST
Static. This transport expects to be tried last of all for outbound messages.

The STATUS_PRIMARY_STORE and STATUS_SECONDARY_STORE flags are available for communication between a client and a message store. Neither implementation should use these flags without ascertaining that the other implementation supports them.

See Also

IMsgServiceAdmin::MsgServiceTransportOrder method, PR_IDENTITY_ENTRYID property

PR_RESOURCE_METHODS

The PR_RESOURCE_METHODS property contains a bitmask of flags indicating the status object methods that are supported.

Usage

Required as a column entry in status tables.

Details

Identifier 0x3E02; property type PT_LONG; property tag 0x3E020003

Remarks

All system resources contain a status table for the **IMAPIStatus** interface and set flags in the PR_RESOURCE_METHODS property. If a flag is set in the status table, the corresponding **IMAPIStatus** method exists and can be called. If that flag is clear in the status table, the method should not be called.

One or more of the following flags can be set in PR_RESOURCE_METHODS:

STATUS_CHANGE_PASSWORD
: The **IMAPIStatus::ChangePassword** method is supported.

STATUS_FLUSH_QUEUES
: The **IMAPIStatus::FlushQueues** method is supported.

STATUS_SETTINGS_DIALOG
: The **IMAPIStatus::SettingsDialog** method is supported.

STATUS_VALIDATE_STATE
: The **IMAPIStatus::ValidateState** method is supported.

PR_RESOURCE_PATH

The PR_RESOURCE_PATH property contains a path to the service provider's server.

Usage

Optional on status objects and as a column entry in status tables.

Details

Identifier 0x3E07; property type PT_TSTRING; property tag 0x3E07001E (0x3E07001F for Unicode)

Remarks

The path contained in the PR_RESOURCE_PATH property represents the suggested path where the user can find resources. The definition of the property is provider specific. For example, a scheduling application uses this property to specify the suggested location for its scheduling application files.

The messaging user profile furnishes PR_RESOURCE_PATH as a convenience so that a client application does not have to prompt the messaging user for a network path or network drive letter.

MAPI works only with filenames, and other strings passed to it, in the ANSI character set. Applications that use filenames in an OEM character set must convert them to ANSI before calling MAPI.

PR_RESOURCE_TYPE

The PR_RESOURCE_TYPE property contains a value indicating the service provider type.

Usage

Required on provider objects.

Details

Identifier 0x3E03; property type PT_LONG; property tag 0x3E030003

Remarks

The PR_RESOURCE_TYPE property can have exactly one of the following values:

Value	Description
MAPI_AB	Address book
MAPI_AB_PROVIDER	Address book provider
MAPI_HOOK_PROVIDER	Spooler hook provider
MAPI_PROFILE_PROVIDER	Profile provider
MAPI_SPOOLER	Message spooler
MAPI_STORE_PROVIDER	Message store provider
MAPI_SUBSYSTEM	Internal MAPI subsystem
MAPI_TRANSPORT_PROVIDER	Transport provider

PR_RESPONSE_REQUESTED

The PR_RESPONSE_REQUESTED property contains TRUE if the message sender wants a response to a meeting request.

Usage

Optional on message objects used for scheduling.

Details

Identifier 0x0063; property type PT_BOOLEAN; property tag 0x0063000B

Remarks

The PR_RESPONSE_REQUESTED property is used for meeting requests. The receiving client application should prompt the user to accept or decline the request and then send this response back to the sender.

PR_RESPONSIBILITY

The PR_RESPONSIBILITY property contains TRUE if some transport provider has already accepted responsibility for delivering the message to this recipient, and FALSE if the MAPI spooler considers that this transport provider should accept responsibility.

Usage

Required on recipient subobjects.

Details

Identifier 0x0E0F; property type PT_BOOLEAN; property tag 0x0E0F000B

Remarks

When the MAPI spooler presents an outbound message to a transport provider, through **IXPLogon::SubmitMessage**, it sets the PR_RESPONSIBILITY property to FALSE for all recipients for which the MAPI spooler considers that transport provider responsible, and TRUE for all other recipients. The transport provider should attempt to handle all recipients with PR_RESPONSIBILITY set to FALSE. After successfully sending, or conclusively failing to send, to a recipient, the transport provider should set PR_RESPONSIBILITY to TRUE in the source message to indicate that it has accepted responsibility for that recipient.

If, after examining a recipient, a transport provider decides that it cannot or should not handle it, the transport provider should leave PR_RESPONSIBILITY set to FALSE. The MAPI spooler will then look for another transport provider that can handle that recipient. The MAPI spooler ultimately creates a nondelivery report for any recipients for which no transport provider accepts responsibility.

If the transport provider attempts and fails to deliver the message, it should call the **IMAPISupport::StatusRecips** method to indicate to MAPI the reasons for the failure, so that MAPI can generate a nondelivery report.

PR_RESPONSIBILITY corresponds to the X.400 attribute MH_T_MTA_RESPONSIBILITY.

See Also

PR_DELETE_AFTER_SUBMIT property

PR_RETURNED_IPM

The PR_RETURNED_IPM property contains TRUE if the original message is being returned with a nonread report.

Usage

Optional on report message objects.

Details

Identifier 0x0033; property type PT_BOOLEAN; property tag 0x0033000B

Remarks

An X.400 transport provider sets the PR_RETURNED_IPM property in the nonread report.

PR_ROW_TYPE

The PR_ROW_TYPE property contains a value that indicates the type of a row in a table.

Usage

Required as a column entry in contents tables.

Details

Identifier 0x0FF5; property type PT_LONG; property tag 0x0FF50003

Remarks

The PR_ROW_TYPE property appears only on contents tables. A category only exists when it has items.

PR_ROW_TYPE can have exactly one of the following values:

Value	Description
TBL_LEAF_ROW	Represents actual data, rather than a category row.
TBL_EMPTY_CATEGORY	Not currently used.
TBL_EXPANDED_CATEGORY	The category is expanded; the user interface usually displays this with the minus sign ' - ' next to it.
TBL_COLLAPSED_CATEGORY	The category is collapsed; the user interface usually displays this with the plus sign '+' next to it.

See Also

PR_ROWID property

PR_ROWID

The PR_ROWID property contains a unique identifier for a recipient in a recipient table or status table.

Usage

Required as a column entry in recipient tables and status tables.

Details

Identifier 0x3000; property type PT_LONG; property tag 0x30000003

Remarks

The PR_ROWID property is a temporary value that is valid only for the life of the object that owns the table. It appears as a column of the table but is not stored.

See Also

IMessage::GetRecipientTable method, **IMessage::ModifyRecipients** method

PR_RTF_COMPRESSED

The PR_RTF_COMPRESSED property contains the Rich Text Format version of the message text, usually in compressed form.

Usage

Optional on message objects.

Details

Identifier 0x1009; property type PT_BINARY; property tag 0x10090102

Remarks

The PR_RTF_COMPRESSED property contains the same message text as the PR_BODY property but in Rich Text Format (RTF).

Message text in RTF is normally stored in compressed form. However, some systems do not compress formatted text. To accommodate them, MAPI provides the dwMagicUncompressedRTF value for a stream header to identify uncompressed RTF, and the STORE_UNCOMPRESSED_RTF in PR_STORE_SUPPORT_MASK for the message store to indicate it can store uncompressed RTF.

To obtain the contents of PR_RTF_COMPRESSED, call **OpenProperty**, then call **WrapCompressedRTFStream** with the MAPI_READ flag. To write into PR_RTF_COMPRESSED, open it with the MAPI_MODIFY and MAPI_CREATE flags. This ensures that the new data completely replace any old data and that the writes are performed using the minimum number of store updates.

Message stores that support RTF ignore any changes to white space in the message text. When PR_BODY is stored for the first time, the message store also generates and stores PR_RTF_COMPRESSED. If the **IMAPIProp::SaveChanges** method is subsequently called and PR_BODY has been modified, the message store calls the **RTFSync** function to ensure synchronization with the RTF version. If only white space has been changed, the properties are left unchanged. This preserves any nontrivial RTF formatting when the message travels through non-RTF-aware clients and messaging systems.

PR_RTF_IN_SYNC

The PR_RTF_IN_SYNC property contains TRUE if PR_RTF_COMPRESSED has the same text content as PR_BODY for this message.

Usage

Optional on message objects.

Details

Identifier 0x0E1F; property type PT_BOOLEAN; property tag 0x0E1F000B

Remarks

A value of TRUE means that the PR_BODY property, the plain text version of this message, and the PR_RTF_COMPRESSED property, the Rich Text Format version, are identical except for white space in PR_BODY and formatting in PR_RTF_COMPRESSED. The text in the two versions consists of the same characters in the same sequence.

A value of FALSE means that the two versions are not synchronized for text content but are capable of being synchronized by the **RTFSync** function. One version has been altered and the other version has not.

No value, PR_RTF_IN_SYNC is not set at all, means that the two versions, if both exist or ever existed, cannot be synchronized. One version has been deleted or altered so radically that synchronization is no longer possible.

A client application that has modified PR_RTF_COMPRESSED should set a value of FALSE in PR_RTF_IN_SYNC to force synchronization. RTF-aware message stores should perform the synchronization using **RTFSync** during an **IMAPIProp::SaveChanges** call. RTF-aware clients should check the setting of PR_RTF_IN_SYNC before reading PR_RTF_COMPRESSED, and call **RTFSync** first if necessary.

If PR_BODY has had modifications to anything other than its white space, the message store must delete PR_RTF_IN_SYNC to terminate synchronization.

PR_RTF_SYNC_BODY_COUNT

The PR_RTF_SYNC_BODY_COUNT property contains a count of the significant characters of the message text.

Usage

Reserved.

Details

Identifier 0x1007; property type PT_LONG; property tag 0x10070003

Remarks

The **RTFSync** function computes the count of characters in the text using only those that it considers to be significant to the message. For example, some white space and other ignorable characters are omitted from the count.

The PR_RTF_SYNC_BODY_COUNT property is an RTF auxiliary property. These properties are used by the **RTFSync** function and are not intended to be used directly by client applications.

PR_RTF_SYNC_BODY_CRC

The PR_RTF_SYNC_BODY_CRC property contains the cyclical redundancy check (CRC) computed for the message text.

Usage

Reserved.

Details

Identifier 0x1006; property type PT_LONG; property tag 0x10060003

Remarks

The **RTFSync** function computes the CRC using only the characters that it considers to be significant to the message. For example, some white space and other ignorable characters are omitted from the CRC.

The PR_RTF_SYNC_BODY_CRC property is a Rich Text Format auxiliary property. These properties are used by the **RTFSync** function and are not intended to be used directly by client applications.

PR_RTF_SYNC_BODY_TAG

The PR_RTF_SYNC_BODY_TAG property contains significant characters that appear at the beginning of the message text.

Usage

Reserved.

Details

Identifier 0x1008; property type PT_TSTRING; property tag 0x1008001E (0x1008001F for Unicode)

Remarks

The **RTFSync** function uses the text tag to indicate the beginning of the message text. When the text is modified, the tag is used to find the beginning of the previous text.

The PR_RTF_SYNC_BODY_TAG property is a Rich Text Format auxiliary property. These properties are used by the **RTFSync** function and are not intended to be used directly by client applications.

PR_RTF_SYNC_PREFIX_COUNT

The PR_RTF_SYNC_PREFIX_COUNT property contains a count of the ignorable characters that appear before the significant characters of the message.

Usage

Reserved.

Details

Identifier 0x1010; property type PT_LONG; property tag 0x10100003

Remarks

The count of prefix characters does not include white space.

The PR_RTF_SYNC_PREFIX_COUNT property is a Rich Text Format auxiliary property. These properties are used by the **RTFSync** function and are not intended to be used directly by client applications.

PR_RTF_SYNC_TRAILING_COUNT

The PR_RTF_SYNC_TRAILING_COUNT property contains a count of the ignorable characters that appear after the significant characters of the message.

Usage

Reserved.

Details

Identifier 0x1011; property type PT_LONG; property tag 0x10110003

Remarks

The PR_RTF_SYNC_TRAILING_COUNT property is a Rich Text Format auxiliary property. These properties are used by the **RTFSync** function and are not intended to be used directly by client applications.

PR_SEARCH

The PR_SEARCH property contains a container object that is used for advanced searches.

Usage

Optional on address book container and message store objects.

Details

Identifier 0x3607; property type PT_OBJECT; property tag 0x3607000D

Remarks

A container that does not support advanced search capabilities, does not have to supply the PR_SEARCH property.

See Also

IMAPIContainer : IMAPIProp interface

PR_SEARCH_KEY

The PR_SEARCH_KEY property contains a binary-comparable key that identifies correlated objects for a search.

Usage

Required on address book container, distribution list, messaging user, and message objects.

Details

Identifier 0x300B; property type PT_BINARY; property tag 0x300B0102

Remarks

The PR_SEARCH_KEY property provides a trace for related objects, such as message copies, and facilitates finding unwanted occurrences, such as duplicate recipients.

PR_SEARCH_KEY is one of the base address properties for all messaging users. For more information on the base address properties, see About Base Address Properties.

MAPI uses specific rules for constructing search keys for message recipients. The search key is formed by concatenating the address type (in uppercase characters), the colon character ':', the e-mail address in canonical form, and the terminating null character. Canonical form here means that case-sensitive addresses appear in the correct case, and addresses that are not case-sensitive are converted to uppercase. This is important in preserving correlations among messages.

For message objects, PR_SEARCH_KEY is available through the **IMAPIProp::GetProps** method immediately following message creation. For other objects, it is available following the first call to the **IMAPIProp::SaveChanges** method. Since PR_SEARCH_KEY is changeable, it is unreliable to obtain it through **GetProps** until a **SaveChanges** call has committed any values set or changed by the **IMAPIProp::SetProps** method.

For profiles, MAPI also furnishes a hard-coded profile section named MUID_PROFILE_INSTANCE, with PR_SEARCH_KEY as its single property. This key is guaranteed to be unique among all profiles ever created, and can be more reliable than the PR_PROFILE_NAME property, which can be, for example, deleted and recreated with the same name.

The following table summarizes important differences among the PR_ENTRYID, PR_RECORD_KEY, and PR_SEARCH_KEY properties.

Characteristic	PR_ENTRYID	PR_RECORD_KEY	PR_SEARCH_KEY
Required on attachment objects	No	Yes	No
Required on folder objects	Yes	Yes	No
Required on message store objects	Yes	Yes	No
Required on status objects	Yes	No	No
Creatable by client	No	No	Yes
Available before **SaveChanges**	Maybe	Maybe	Messages – Yes Others – Maybe
Changed in a	Yes	Yes	NO

copy operation			
Changeable by client after a copy	No	No	Yes
Unique within ...	Entire world	Provider instance	Entire world
Binary comparable (as with memcmp)	No -- use **IMAPISupport::CompareEntryIDs**	Yes	Yes

PR_SEARCH_KEY corresponds to the X.400 attribute IM_THIS_IPM.

See Also

PR_RESPONSIBILITY property, PR_STORE_RECORD_KEY property

PR_SECURITY

The PR_SECURITY property contains a flag that indicates the security level of a message.

Usage

Optional on message objects.

Details

Identifier 0x0034; property type PT_LONG; property tag 0x00340003

Remarks

The underlying messaging system chooses whether to use the flag contained in the PR_SECURITY property. The PR_MESSAGE_CLASS property can determine whether the messaging system honors the security settings.

The following flags can be set:

SECURITY_SIGNED
: The message has been digitally signed. The recipient can read the digital signature to see if it is expired or suspended, or if the contents have been altered since the digital signing.

SECURITY_ENCRYPTED
: The message has been encrypted. The recipient must present a security password to read the message text.

PR_SELECTABLE

The PR_SELECTABLE property contains TRUE if the entry in the one-off table can be selected.

Usage

Optional as a column entry in one-off tables.

Details

Identifier 0x3609, property type PT_BOOLEAN; property tag 0x3609000B

Remarks

The PR_SELECTABLE property is used primarily for visual formatting of a one-off table. X.400 or gateway templates can be grouped by creating an entry that indicates the heading for the group. Setting PR_SELECTABLE to FALSE for the heading ensures that the user can select only the actual templates in the group and not this heading entry.

This property applies only to a one-off table, not to an address book hierarchy table.

MAPI allows an address book provider to group items visually by two means. First, certain rows can function as headings by being unselectable. Second, the selectable items can be indented relative to their headings using the PR_DEPTH property. PR_SELECTABLE is used in such grouping to indicate whether or not this item can be selected from a list box to create a one-off address. For example, if a client has several templates for building FAX addresses, it can display them as follows:

```
FAX templates (depth 0, not selectable)
     Local (depth 1, selectable)
     Long-distance (depth 1, selectable)
```

See Also

IABLogon::GetOneOffTable method, PR_FOLDER_TYPE property

PR_SEND_RICH_INFO

The PR_SEND_RICH_INFO property contains TRUE if the recipient can receive all message content, including Rich Text Format and OLE objects.

Usage

Optional but recommended on distribution list and messaging user objects.

Details

Identifier 0x3A40; property type PT_BOOLEAN; property tag 0x3A40000B

Remarks

The PR_SEND_RICH_INFO property indicates whether the sender considers the recipient to be MAPI-enabled.

When this property is set to TRUE, the transport and gateway can transmit the full content of the message, including Rich Text Format and OLE objects. The transport provider and gateway should use TNEF to encapsulate any properties that are not native to all the messaging systems involved.

When this property is set to FALSE, the transport provider and gateway are free to discard message content that their native clients cannot use. For example, when the clients do not support Rich Text Format, the transport provider can send only plain text.

When PR_SEND_RICH_INFO is not set, default behavior is determined by the implementation of the transport provider, MTA, or gateway. Address book providers are not required to support this property. For example, a tightly coupled address book and transport provider can choose to send TNEF but never use Rich Text Format.

The client should not assume the transport provider and gateway will use TNEF on their own initiative. Some transport providers and gateways that support TNEF transmit it without regard to the value of PR_SEND_RICH_INFO, but others decline to construct or send TNEF if PR_SEND_RICH_INFO is not set to TRUE.

Note The setting of this property, and the decisions based on its value, are on a per-recipient basis.

By default, MAPI sets the value of PR_SEND_RICH_INFO to TRUE. A client calling **IAddrBook::CreateOneOff** or a provider calling **IMAPISupport::CreateOneOff** can set the MAPI_SEND_NO_RICH_INFO bit in the *ulFlags* parameter, which causes MAPI to set PR_SEND_RICH_INFO to FALSE. One-offs created by the user interface use the value specified by the creating template.

On calls to the **IAddrBook::ResolveName** method when the name cannot be resolved but can be interpreted as an Internet address (SMTP), PR_SEND_RICH_INFO is set to FALSE. To be construed as an Internet address, the display name of the unresolved entry must be in the format *X@Y.Z*, such as "pete@pinecone.com".

See Also

PR_ATTACH_DATA_OBJ property

PR_SENDER_ADDRTYPE

The PR_SENDER_ADDRTYPE property contains the message sender's e-mail address type.

Usage

Required on message objects.

Details

Identifier 0x0C1E; property type PT_TSTRING; property tag 0x0C1E001E (0x0C1E001F for Unicode)

Remarks

The PR_SENDER_ADDRTYPE property is one of the address properties for the message sender. It must be set by the outgoing transport provider, which should never propagate any previously existing values.

If no transport provider has supplied any sender address properties, the MAPI spooler attempts to fill them in by calling the **IMAPISession::QueryIdentity** method for an entry identifier. If no entry identifiers have been provided, the MAPI spooler stores "Unknown" in all the sender address properties of type PT_TSTRING.

For more information on the address properties, see About Base Address Properties.

See Also

PR_ADDRTYPE property, PR_SENT_REPRESENTING_ADDRTYPE property

PR_SENDER_EMAIL_ADDRESS

The PR_SENDER_EMAIL_ADDRESS property contains the message sender's e-mail address.

Usage

Required on message objects.

Details

Identifier 0x0C1F; property type PT_TSTRING; property tag 0x0C1F001E (0x0C1F001F for Unicode)

Remarks

The PR_SENDER_EMAIL_ADDRESS property is one of the address properties for the message sender. It must be set by the outgoing transport provider, which should never propagate any previously existing values.

If no transport provider has supplied any sender address properties, the MAPI spooler attempts to fill them in by calling the **IMAPISession::QueryIdentity** method for an entry identifier. If no entry identifiers have been provided, the MAPI spooler stores "Unknown" in all the sender address properties of type PT_TSTRING.

For more information on the address properties, see About Base Address Properties.

See Also

PR_EMAIL_ADDRESS property, PR_SENT_REPRESENTING_EMAIL_ADDRESS property

PR_SENDER_ENTRYID

The PR_SENDER_ENTRYID property contains the message sender's entry identifier.

Usage

Required on message objects.

Details

Identifier 0x0C19; property type PT_BINARY; property tag 0x0C190102

Remarks

The PR_SENDER_ENTRYID property is one of the address properties for the message sender. It must be set by the outgoing transport provider, which should never propagate any previously existing values.

If no transport provider has supplied any sender address properties, the MAPI spooler attempts to fill them in by calling the **IMAPISession::QueryIdentity** method for an entry identifier. If no entry identifiers have been provided, the MAPI spooler stores in PR_SENDER_ENTRYID an identifier corresponding to the string "Unknown."

For more information on the address properties, see About Base Address Properties.

PR_SENDER_ENTRYID corresponds to the X.400 attribute IM_ORIGINATOR.

See Also

PR_ENTRYID property, PR_SENT_REPRESENTING_ENTRYID property

PR_SENDER_NAME

The PR_SENDER_NAME property contains the message sender's display name.

Usage

Required on message objects.

Details

Identifier 0x0C1A; property type PT_TSTRING; property tag 0x0C1A001E (0x0C1A001F for Unicode)

Remarks

The PR_SENDER_NAME property is one of the address properties for the message sender. It must be set by the outgoing transport provider, which should never propagate any previously existing values.

If no transport provider has supplied any sender address properties, the MAPI spooler attempts to fill them in by calling the **IMAPISession::QueryIdentity** method for an entry identifier. If no entry identifiers have been provided, the MAPI spooler stores "Unknown" in all the sender address properties of type PT_TSTRING.

For more information on the address properties, see About Base Address Properties.

PR_SENDER_NAME corresponds to the X.400 attributes IM_FREE_FORM_NAME and MH_T_ORIGINATOR_NAME.

See Also

PR_DISPLAY_NAME property, PR_SENT_REPRESENTING_NAME property

PR_SENDER_SEARCH_KEY

The PR_SENDER_SEARCH_KEY property contains the message sender's search key.

Usage

Required on message objects.

Details

Identifier 0x0C1D; property type PT_BINARY; property tag 0x0C1D0102

Remarks

The PR_SENDER_SEARCH_KEY property is one of the address properties for the message sender. It must be set by the outgoing transport provider, which should never propagate any previously existing values.

If no transport provider has supplied any sender address properties, the MAPI spooler attempts to fill them in by calling the **IMAPISession::QueryIdentity** method for an entry identifier. If no entry identifiers have been provided, the MAPI spooler stores in PR_SENDER_SEARCH_KEY a key corresponding to the string "Unknown."

For more information on the address properties, see About Base Address Properties.

See Also

PR_SEARCH_KEY property, PR_SENT_REPRESENTING_SEARCH_KEY property

PR_SENSITIVITY

The PR_SENSITIVITY property contains a value indicating the message sender's opinion of the sensitivity of a message.

Usage

Optional but recommended on message objects.

Details

Identifier 0x0036; property type PT_LONG; property tag 0x00360003

Remarks

The PR_SENSITIVITY property can have exactly one of the following values:

Value	Description
SENSITIVITY_NONE	The message has no special sensitivity.
SENSITIVITY_PERSONAL	The message is personal.
SENSITIVITY_PRIVATE	The message is private.
SENSITIVITY_COMPANY_CONFIDENTIAL	The message is designated Company Confidential.

PR_SENSITIVITY corresponds to the X.400 attribute IM_SENSITIVITY.

PR_SENT_REPRESENTING_ADDRTYPE

The PR_SENT_REPRESENTING_ADDRTYPE property contains the address type for the messaging user represented by the sender.

Usage

Required on message objects.

Details

Identifier 0x0068; property type PT_TSTRING; property tag 0x0068001E (0x0068001F for Unicode)

Remarks

The PR_SENT_REPRESENTING_ADDRTYPE property is one of the address properties for the messaging user being represented by the sender. When a client application sends a message on behalf of another client, it should set all the represented sender properties to the values for that client. A messaging user sending on its own behalf typically leaves the represented sender properties unset.

The outgoing transport provider must always leave PR_SENT_REPRESENTING_ADDRTYPE unchanged if it has been set by the sending client. If it is unset, the transport provider should set it to PR_SENDER_ADDRTYPE on the outbound copy of the message, and leave it unset on the local copy.

For more information on the address properties, see About Base Address Properties.

See Also

PR_ADDRTYPE property

PR_SENT_REPRESENTING_EMAIL_ADDRESS

The PR_SENT_REPRESENTING_EMAIL_ADDRESS property contains the e-mail address for the messaging user represented by the sender.

Usage

Required on message objects.

Details

Identifier 0x0069; property type PT_TSTRING; property tag 0x0069001E (0x0069001F for Unicode)

Remarks

The PR_SENT_REPRESENTING_EMAIL_ADDRESS property is one of the address properties for the messaging user being represented by the sender. When a client application sends a message on behalf of another client, it should set all the represented sender properties to the values for that client. A messaging user sending on its own behalf typically leaves the represented sender properties unset.

The outgoing transport provider must always leave PR_SENT_REPRESENTING_EMAIL_ADDRESS unchanged if it has been set by the sending client. If it is unset, the transport provider should set it to PR_SENDER_EMAIL_ADDRESS on the outbound copy of the message, and leave it unset on the local copy.

For more information on the address properties, see About Base Address Properties.

See Also

PR_EMAIL_ADDRESS property

PR_SENT_REPRESENTING_ENTRYID

The PR_SENT_REPRESENTING_ENTRYID property contains the entry identifier for the messaging user represented by the sender.

Usage

Required on message objects.

Details

Identifier 0x0041; property type PT_BINARY; property tag 0x00410102

Remarks

The PR_SENT_REPRESENTING_ENTRYID property is one of the address properties for the messaging user being represented by the sender. When a client application sends a message on behalf of another client, it should set all the represented sender properties to the values for that client. A messaging user sending on its own behalf typically leaves the represented sender properties unset.

The outgoing transport provider must always leave PR_SENT_REPRESENTING_ENTRYID unchanged if it has been set by the sending client. If it is unset, the transport provider should set it to PR_SENDER_ENTRYID on the outbound copy of the message, and leave it unset on the local copy.

For more information on the address properties, see About Base Address Properties.

See Also

PR_ENTRYID property

PR_SENT_REPRESENTING_NAME

The PR_SENT_REPRESENTING_NAME property contains the display name for the messaging user represented by the sender.

Usage

Required on message objects.

Details

Identifier 0x0042; property type PT_TSTRING; property tag 0x0042001E (0x0042001F for Unicode)

Remarks

The PR_SENT_REPRESENTING_NAME property is one of the address properties for the messaging user being represented by the sender. When a client application sends a message on behalf of another client, it should set all the represented sender properties to the values for that client. A messaging user sending on its own behalf typically leaves the represented sender properties unset.

The outgoing transport provider must always leave PR_SENT_REPRESENTING_NAME unchanged if it has been set by the sending client. If it is unset, the transport provider should set it to PR_SENDER_NAME on the outbound copy of the message, and leave it unset on the local copy.

For more information on the address properties, see About Base Address Properties.

See Also

PR_DISPLAY_NAME property

PR_SENT_REPRESENTING_SEARCH_KEY

The PR_SENT_REPRESENTING_SEARCH_KEY property contains the search key for the messaging user represented by the sender.

Usage

Required on message objects.

Details

Identifier 0x003B; property type PT_BINARY; property tag 0x003B0102

Remarks

The PR_SENT_REPRESENTING_SEARCH_KEY property is one of the address properties for the messaging user being represented by the sender. When a client application sends a message on behalf of another client, it should set all the represented sender properties to the values for that client. A messaging user sending on its own behalf typically leaves the represented sender properties unset.

The outgoing transport provider must always leave PR_SENT_REPRESENTING_SEARCH_KEY unchanged if it has been set by the sending client. If it is unset, the transport provider should set it to PR_SENDER_SEARCH_KEY on the outbound copy of the message, and leave it unset on the local copy.

For more information on the address properties, see About Base Address Properties.

See Also

PR_SEARCH_KEY property

PR_SENTMAIL_ENTRYID

The PR_SENTMAIL_ENTRYID property contains the entry identifier of the folder where the message should be moved after submission.

Usage

Optional on message objects.

Details

Identifier 0x0E0A; property type PT_BINARY; property tag 0x0E0A0102

Remarks

The PR_SENTMAIL_ENTRYID property is often copied from the PR_IPM_SENTMAIL_ENTRYID property, the client application's standard Sent Items folder.

The client application uses PR_SENTMAIL_ENTRYID with the PR_DELETE_AFTER_SUBMIT property to control what happens to a message after it is submitted. Either one or the other should be set, but not both.

PR_SERVICE_DELETE_FILES

The PR_SERVICE_DELETE_FILES property contains a list of filenames that are to be deleted when the message service is uninstalled.

Usage

Optional on profile section objects.

Details

Identifier 0x3D10; property type PT_MV_TSTRING; property tag 0x3D10101E

Remarks

The filenames in the list contained in the PR_SERVICE_DELETE_FILES property are deleted from the computer when using the control panel to uninstall the message service. Do not include in the list any DLL that supports multiple message services, or additional message services could be inadvertently removed.

MAPI works only with filenames, and other strings passed to it, in the ANSI character set. Applications that use filenames in an OEM character set must convert them to ANSI before calling MAPI.

PR_SERVICE_DLL_NAME

The PR_SERVICE_DLL_NAME property contains the filename of the DLL containing the message service provider entry point function to call for configuration.

Usage

Required on profile section objects and as a column in message service tables.

Details

Identifier 0x3D0A; property type PT_TSTRING; property tag 0x3D0A001E (0x3D0A001F for Unicode)

Remarks

When the entry point function name appears in the PR_SERVICE_ENTRY_NAME method, it indicates that the entry point exists.

MAPI uses a DLL file naming convention. The base filename contains up to six characters that uniquely identify the DLL. MAPI appends the string 32 to the base DLL name to identify the version that runs on 32-bit platforms. For example, when the name MAPI.DLL is specified, MAPI constructs the name MAPI32.DLL to represent the corresponding 32-bit version of the DLL.

The PR_SERVICE_DLL_NAME property should specify the base name. MAPI appends the string 32 as appropriate. Including the string 32 as part of the PR_SERVICE_DLL_NAME property results in an error.

See Also

PR_PROVIDER_DLL_NAME property

PR_SERVICE_ENTRY_NAME

The PR_SERVICE_ENTRY_NAME property contains the name of the entry point function for configuration of a message service.

Usage

Required as a column in message service tables.
Optional but recommended on profile section objects.

Details

Identifier 0x3D0B; property type PT_STRING8; property tag 0x3D0B001E

Remarks

It is recommended that message service implementors provide a message service entry point, but the entry point is not required. However, the entry point should be supplied only if the related configuration properties exist. If these properties do not exist, MAPI assumes that no entry point is provided.

The DLL in which the entry point function appears is named by the PR_SERVICE_DLL_NAME property.

For more information on message service entry points, see About Message Service Entry Point Functions.

PR_SERVICE_EXTRA_UIDS

The PR_SERVICE_EXTRA_UIDS property contains a list of **MAPIUID** structures that identify additional profile sections for the message service.

Usage

Optional on profile section objects.

Details

Identifier 0x3D0D; property type PT_BINARY; property tag 0x3D0D0102

Remarks

New profile sections can be created for each message filter. When the information about the message service is to be copied to another profile, it is important to copy the additional profile sections for the filters as well. A service provider that uses additional profile sections can store the **MAPIUID** structures of those profile sections in PR_SERVICE_EXTRA_UIDS, allowing MAPI to copy the additional message service information.

PR_SERVICE_NAME

The PR_SERVICE_NAME property contains the name of a message service as set by the user in the MAPISVC.INF file.

Usage

Required on profile section objects and as a column in message service tables.

Details

Identifier 0x3D09; property type PT_TSTRING; property tag 0x3D09001E (0x3D09001F for Unicode)

Remarks

The name contained in the PR_SERVICE_NAME property is specific to the message service. It comes from the [Services] section in MAPISVC.INF.

PR_SERVICE_NAME appears as a column in the message service table and can be used to filter services. Because it is used to identify and filter services, the value should not be localized.

PR_SERVICE_SUPPORT_FILES

The PR_SERVICE_SUPPORT_FILES property contains a list of the files that belong to the message service.

Usage

Required as a column in message service tables.
Optional on profile section objects.

Details

Identifier 0x3D0F, property type PT_MV_TSTRING; property tag 0x3D0F101E

Remarks

Using a dialog box in the control panel applet, a user can obtain the list of files that belong to the message service. For example, the user can obtain the names of all DLLs that belong to the service. The user can then seek additional details about the specified files, such as the names and version numbers of all the DLLs. MAPI uses the PR_SERVICE_SUPPORT_FILES property to create a support file list in a dialog box for messaging user selection.

MAPI works only with filenames, and other strings passed to it, in the ANSI character set. Client applications that use filenames in an OEM character set must convert them to ANSI before calling MAPI.

PR_SERVICE_UID

The PR_SERVICE_UID property contains the **MAPIUID** structure for a message service.

Usage

Required as a column in message service tables.
Computed by MAPI on profile section objects.

Details

Identifier 0x3D0C; property type PT_BINARY; property tag 0x3D0C0102

Remarks

MAPI uses the PR_SERVICE_UID property to group all the providers that belong to the same message service. PR_SERVICE_UID is supplied as a parameter to most of the **IMsgServiceAdmin** methods. It must not appear in MAPISVC.INF.

See Also

IMsgServiceAdmin::IUnknown method

PR_SERVICES

The PR_SERVICES property contains a list of identifiers of message services in the current profile.

Usage

Reserved.

Details

Identifier 0x3D0E: property type PT_BINARY; property tag 0x3D0E0102

Remarks

Do not use this property.

See Also

MAPIUID structure

PR_SPOOLER_STATUS

The PR_SPOOLER_STATUS property contains the status of the message based on information available to the MAPI spooler.

Usage

Computed by MAPI on message objects.

Details

Identifier 0x0E10; property type PT_LONG; property tag 0x0E100003

Remarks

The PR_SPOOLER_STATUS property appears on inbound messages only and is reserved in all other cases. It indicates whether or not a message has been delivered to its final location or whether a messaging hook provider potentially deleted the message while rerouting it.

Client applications should never set this property. For an inbound message, a client or service provider can call the **IMAPIProp::GetProps** property on PR_SPOOLER_STATUS to determine the message status. The value S_OK indicates that the message was successfully delivered to the message store. The value MAPI_E_OBJECT_DELETED indicates that the message was deleted and was never committed to the store.

Message store providers should support PR_SPOOLER_STATUS on messages, recipient tables, and the outgoing queue table. Clients and providers should be able to set columns on the outgoing queue table and restrict based on this property.

PR_START_DATE

The PR_START_DATE property contains the starting date and time of an appointment as managed by a scheduling application.

Usage

Required on message objects used for scheduling.

Details

Identifier 0x0060; property type PT_SYSTIME; property tag 0x00600040

Remarks

Scheduling applications should set both the PR_START_DATE and PR_END_DATE properties when sending meeting requests.

PR_STATE_OR_PROVINCE

The PR_STATE_OR_PROVINCE property contains the name of the recipient's state or province.

Usage

Optional on messaging user objects.

Details

Identifier 0x3A28; property type PT_TSTRING; property tag 0x3A28001E (0x3A28001F for Unicode)

Remarks

The PR_STATE_OR_PROVINCE property is one of the properties that provide identification and access information for a recipient. These properties are defined by the recipient and the recipient's organization. For more information on this group of properties, see About Messaging User Objects.

PR_STATUS

The PR_STATUS property contains a 32-bit bitmask of flags defining folder status.

Usage

Required on folder objects and as a column entry in hierarchy tables.

Details

Identifier 0x360B; property type PT_LONG; property tag 0x360B0003

Remarks

The PR_STATUS property for folders is analogous to the PR_MSG_STATUS property for messages. Its flags are provided for the client application only and do not affect the message store. Clients can use or ignore these settings. The client can also define its own values for the client-definable bits of the PR_STATUS property.

One or more of the following flags can be set for the PR_STATUS bitmask:

FLDSTATUS_DELMARKED
: The folder is marked for deletion. The client application sets this flag.

FLDSTATUS_HIDDEN
: The folder is hidden.

FLDSTATUS_HIGHLIGHTED
: The folder is highlighted, for example, shown in reverse video.

FLDSTATUS_TAGGED
: The folder is tagged.

Bits 16 through 31 (0x10000 through 0x80000000) of PR_STATUS are available for use by the IPM client application. All other bits are reserved for use by MAPI; those not defined in the preceding list should be initially set to zero and not altered subsequently.

PR_STATUS_CODE

The PR_STATUS_CODE property contains a bitmask of flags indicating the current status of a service provider.

Usage

Required on profile section and provider objects and as a column entry in status tables.

Details

Identifier 0x3E04; property type PT_LONG; property tag 0x3E040003

Remarks

The status code must appear in the MAPISVC.INF file for all providers.

One or more of the following flags can be set for the PR_STATUS_CODE bitmask:

STATUS_AVAILABLE
: The message service is available for use by MAPI and client applications.

STATUS_FAILURE
: The service provider is experiencing severe unexpected problems and the provider session may soon end if the problems are not resolved.

STATUS_INBOUND_ACTIVE
: The transport provider is receiving an inbound message.

STATUS_INBOUND_ENABLED
: The transport provider is enabled to receive inbound messages.

STATUS_INBOUND_FLUSH
: The transport provider is flushing its inbound message queue.

STATUS_OFFLINE
: The message services available from the service provider are limited to those that use locally available data.

STATUS_OUTBOUND_ACTIVE
: The transport provider is receiving an outbound message.

STATUS_OUTBOUND_ENABLED
: The transport provider is enabled to handle outbound messages.

STATUS_OUTBOUND_FLUSH
: The transport provider is flushing its outbound message queue.

STATUS_REMOTE_ACCESS
: The transport provider supports remote access.

See Also

PR_STATUS_STRING property

PR_STATUS_STRING

The PR_STATUS_STRING property contains an ASCII message indicating the current status of a service provider.

Usage

Optional on provider objects.

Details

Identifier 0x3E08; property type PT_TSTRING; property tag 0x3E08001E (0x3E08001F for Unicode)

Remarks

The PR_STATUS_STRING property gives service providers the opportunity to supply specific information about the provider status, for example, whether or not it is processing a message. PR_STATUS_STRING is optional, based on the value of the PR_STATUS_CODE property. When the transport provider does not supply a value, the MAPI spooler supplies a default value.

The string is generated on the same side of the remote procedure call as the MAPI spooler; it travels through shared memory rather than being marshaled across a process boundary.

See Also

PR_STATUS_CODE property

PR_STORE_ENTRYID

The PR_STORE_ENTRYID property contains the unique entry identifier of the message store in which an object resides.

Usage

Required on folder, message, and message store objects.

Details

Identifier 0x0FFB; property type PT_BINARY; property tag 0x0FFB0102

Remarks

The PR_STORE_ENTRYID property is used to open a message store with the **IMAPISession::OpenMsgStore** method. It is also used to open any object owned by the message store.

For a message store, this property is identical to the store's own PR_ENTRYID property. A client application can compare the two properties using the **IMAPISession::CompareEntryIDs** method.

PR_STORE_PROVIDERS

The PR_STORE_PROVIDERS property contains a list of identifiers of message store providers in the current profile.

Usage

Reserved.

Details

Identifier 0x3D00; property type PT_BINARY; property tag 0x3D000102

Remarks

Do not use this property.

See Also

MAPIUID structure

PR_STORE_RECORD_KEY

The PR_STORE_RECORD_KEY property contains the unique binary-comparable identifier (record key) of the message store in which an object resides.

Usage

Required on folder, message, and message store objects.

Details

Identifier 0x0FFA; property type PT_BINARY; property tag 0x0FFA0102

Remarks

For a message store, the PR_STORE_RECORD_KEY property is identical to the store's own PR_RECORD_KEY property.

The relationship between PR_STORE_RECORD_KEY and PR_RECORD_KEY is the same as the relationship between PR_STORE_ENTRYID and PR_ENTRYID.

PR_STORE_STATE

The PR_STORE_STATE property contains a flag that describes the state of the message store.

Usage

Required on message store objects.

Details

Identifier 0x340E; property type PT_LONG; property tag 0x340E0003

Remarks

The PR_STORE_STATE property is dynamic and can change based on user actions, unlike the PR_STORE_SUPPORT_MASK property.

The following value can be set:

Value	Description
STORE_HAS_SEARCHES	The user has created one or more active searches in the store.

See Also

PR_STORE_ENTRYID property

PR_STORE_SUPPORT_MASK

The PR_STORE_SUPPORT_MASK property contains a bitmask of flags that client applications should query to determine the characteristics of a message store.

Usage

Required on folder, message, and message store objects.

Details

Identifier 0x3A0D; property type PT_LONG, Property tag:0x3A0D0003

Remarks

The PR_STORE_SUPPORT_MASK property discloses the capabilities of a message store to client applications planning to send it a message. The flags can facilitate decisions by a client or another store, such as whether to send PR_BODY or only PR_RTF_COMPRESSED. A client should never set PR_STORE_SUPPORT_MASK; an attempt returns MAPI_E_COMPUTED.

One or more of the following flags can be set for the PR_STORE_SUPPORT_MASK bitmask:

STORE_ATTACH_OK
: The message store supports attachments (OLE or non-OLE) to messages.

STORE_CATEGORIZE_OK
: The message store supports categorized views of tables.

STORE_CREATE_OK
: The message store supports creation of new messages.

STORE_ENTRYID_UNIQUE
: Entry identifiers for the objects in the message store are unique, that is, never reused during the life of the store.

STORE_MODIFY_OK
: The message store supports modification of its existing messages.

STORE_MV_PROPS_OK
: The message store supports multivalued properties, guarantees the stability of value order in a multivalued property throughout a save operation, and supports instantiation of multivalued properties in tables.

STORE_NOTIFY_OK
: The message store supports notifications.

STORE_OLE_OK
: The message store supports OLE attachments. The OLE data is accessible through an **IStorage** interface, such as that available through the PR_ATTACH_DATA_OBJ property.

STORE_PUBLIC_FOLDERS
: The folders in this store are public (multi-user), not private (possibly multi-instance but not multi-user).

STORE_READONLY
: All interfaces for the message store have a read-only access level.

STORE_RESTRICTION_OK
: The message store supports restrictions.

STORE_RTF_OK
: The message store supports Rich Text Format (RTF) messages, usually stored compressed, and the store itself keeps PR_BODY and PR_RTF_COMPRESSED synchronized.

STORE_SEARCH_OK
: The message store supports search-results folders.

STORE_SORT_OK
  The message store supports sorting views of tables.

STORE_SUBMIT_OK
  The message store supports marking a message for submission.

STORE_UNCOMPRESSED_RTF
  The message store supports storage of Rich Text Format (RTF) messages in uncompressed form. An uncompressed RTF stream is identified by the value dwMagicUncompressedRTF in the stream header. The dwMagicUncompressedRTF value is defined in the RTFLIB.H file.

An RTF version of a message can always be stored, even if the message store is non-RTF-aware. If the STORE_RTF_OK bit is not set for a particular store, a client maintaining RTF versions must itself call the **RTFSync** function to keep the PR_BODY and PR_RTF_COMPRESSED versions synchronized for text content. RTF is always stored in PR_RTF_COMPRESSED, whether it is actually compressed or not.

PR_STREET_ADDRESS

The PR_STREET_ADDRESS property contains the recipient's street address.

Usage

Optional on messaging user objects.

Details

Identifier 0x3A29; property type PT_TSTRING; property tag 0x3A29001E (0x3A29001F for Unicode)

Remarks

The PR_STREET_ADDRESS property is one of the properties that provide identification and access information for a recipient. These properties are defined by the recipient and the recipient's organization. For more information on this group of properties, see About Messaging User Objects.

PR_SUBFOLDERS

The PR_SUBFOLDERS property contains TRUE if a folder contains subfolders.

Details

Identifier 0x360a; property type PT_BOOLEAN; property tag 0x360A000B

Remarks

Message stores must compute the PR_SUBFOLDERS property for all folders.

See Also

PR_FOLDER_TYPE property

PR_SUBJECT

The PR_SUBJECT property contains the full subject of a message.

Usage

Optional but recommended on message objects.

Details

Identifier 0x0037; property type PT_TSTRING; property tag 0x0037001E (0x0037001F for Unicode)

Remarks

The PR_SUBJECT property is always the full subject text, that is, the concatenation of the prefix and the normalized subject. If there is no prefix, the normalized subject should be the same as the subject. A message store or transport provider uses both the PR_SUBJECT and PR_SUBJECT_PREFIX properties to compute the normalized subject using the rule described under PR_NORMALIZED_SUBJECT.

The subject properties are typically small strings of fewer than 256 characters, and a message store provider is not obligated to support the OLE **IStream** interface on them. The client should always attempt access through the **IMAPIProp** interface first, and resort to **IStream** only if MAPI_E_NOT_ENOUGH_MEMORY is returned.

For a report, PR_SUBJECT contains the original message's subject preceded by a string indicating what has happened to the message.

PR_SUBJECT corresponds to the X.400 attribute IM_SUBJECT.

PR_SUBJECT_PREFIX

The PR_SUBJECT_PREFIX property contains a subject prefix that typically indicates some action on a message, such as "FW: " for forwarding.

Usage

Optional but recommended on message objects.

Details

Identifier 0x003D; property type PT_TSTRING; property tag 0x003D001E (0x003D001F for Unicode)

Remarks

The subject prefix consists of one or more alphanumeric characters, followed by a colon and a space (which are part of the prefix). It must not contain any nonalphanumeric characters before the colon. Absence of a prefix can be represented by an empty string or by PR_SUBJECT_PREFIX not being set.

If PR_SUBJECT_PREFIX is set explicitly, it can be of any length and use any alphanumeric characters, but it must match a substring at the beginning of the PR_SUBJECT property. If PR_SUBJECT_PREFIX is not set by the client and is to be computed, its contents are more restricted. The rule for computing the prefix is that PR_SUBJECT must begin with one, two, or three letters (alphabetic only) followed by a colon and a space. If such a substring is found at the beginning of PR_SUBJECT, it then becomes PR_SUBJECT_PREFIX (and also stays at the beginning of PR_SUBJECT). Otherwise PR_SUBJECT_PREFIX remains unset.

PR_SUBJECT_PREFIX and PR_NORMALIZED_SUBJECT should be computed as part of the **IMAPIProp::SaveChanges** implementation. A client should not prompt **IMAPIProp::GetProps** for their values until they have been committed by an **IMAPIProp::SaveChanges** call.

The subject properties are typically small strings of fewer than 256 characters, and a message store provider is not obligated to support the OLE **IStream** interface on them. A client should always attempt access through the **IMAPIProp** interface first, and resort to **IStream** only if MAPI_E_NOT_ENOUGH_MEMORY is returned.

PR_SUBMIT_FLAGS

The PR_SUBMIT_FLAGS property contains a bitmask of flags indicating details about a message submission.

Usage

Required as a column entry in outgoing queue tables.

Details

Identifier 0x0E14; property type PT_LONG; property tag 0x0E140003

Remarks

One or more of the following flags can be set for the PR_SUBMIT_FLAGS bitmask:

SUBMITFLAG_LOCKED
: The MAPI spooler currently has the message locked.

SUBMITFLAG_PREPROCESS\
: The message needs preprocessing. When the MAPI spooler is done preprocessing this message, it should call the **IMessage::SubmitMessage** method. The message store provider recognizes that the spooler, rather than the client application, has called **SubmitMessage**, clears the flag, and continues message submission.

See Also

IMsgStore::SetLockState method

PR_SUPPLEMENTARY_INFO

The PR_SUPPLEMENTARY_INFO property contains additional information for use in a report.

Usage

Optional on message objects used for reports.

Details

Identifier 0x0C1B; property type PT_TSTRING; property tag 0x0C1B001E (0x0C1B001F for Unicode)

Remarks

The PR_SUPPLEMENTARY_INFO property contains information generated by the message transfer agent or transport provider related to the report. It is typically used for delivery or nondelivery report text that originated with the underlying messaging system.

PR_SUPPLEMENTARY_INFO corresponds to the X.400 attribute IM_SUPPLEMENTARY_INFO.

PR_SURNAME

The PR_SURNAME property contains the recipient's family name.

Usage

Optional on messaging user objects.

Details

Identifier 0x3A11; property type PT_TSTRING; property tag 0x3A11001E (0x3A11001F for Unicode)

Remarks

The PR_SURNAME property is one of the properties that provide identification and access information for a recipient. These properties are defined by the recipient and the recipient's organization. For more information on this group of properties, see About Messaging User Objects.

PR_TELEX_NUMBER

The PR_TELEX_NUMBER property contains the recipient's telex number.

Usage

Optional on messaging user objects.

Details

Identifier 0x3A2C; property type PT_TSTRING; property tag 0x3A2C001E (0x3A2C001F for Unicode)

Remarks

The PR_TELEX_NUMBER property is one of the properties that provide identification and access information for a recipient. These properties are defined by the recipient and the recipient's organization. For more information on this group of properties, see About Messaging User Objects.

PR_TEMPLATEID

The PR_TEMPLATEID property contains an entry identifier that can find the code associated with a service provider.

Usage

Optional on address book provider objects.

Details

Identifier 0x3902; property type PT_BINARY; property tag 0x39020102

Remarks

The entry identifier can bind the code to the data.

See Also

IABLogon::OpenTemplateID method, **IMAPISupport::OpenTemplateID** method

PR_TITLE

The PR_TITLE property contains the recipient's job title.

Usage

Optional on messaging user objects.

Details

Identifier 0x3A17; property type PT_TSTRING; property tag 0x3A17001E (0x3A17001F for Unicode)

Remarks

The PR_TITLE property is one of the properties that provide identification and access information for a recipient. These properties are defined by the recipient and the recipient's organization. For more information on this group of properties, see About Messaging User Objects.

PR_TITLE is commonly used to indicate the recipient's formal job title, such as Senior Programmer, rather than occupational class, such as programmer. It is not typically used for "suffix" titles such as Esq. or DDS.

Common PR_TITLE values include: Managing Director, Programmer II, Associate Professor, and Development Lead

PR_TNEF_CORRELATION_KEY

The PR_TNEF_CORRELATION_KEY property contains a value used to correlate a TNEF attachment with a message.

Usage

Optional but recommended in TNEF attachment subobjects.

Details

Identifier 0x007F; property type PT_BINARY; property tag 0x007F0102

Remarks

The PR_TNEF_CORRELATION_KEY property determines whether or not an inbound TNEF file belongs to the message it is attached to. It is used primarily by transport providers and gateways.

On an outbound message, the transport provider should compute a binary value unique to that message, or use an existing value that satisfies the uniqueness requirement, such as a message identifier. The transport provider should store this value in PR_TNEF_CORRELATION_KEY and then call the **ITnef::AddProps** method to encapsulate it. The same value should also be stored in the transport envelope in a place defined by the provider, such as the message header.

On an inbound message, the transport provider should call the **ITnef::ExtractProps** method to decapsulate the TNEF attachment and then compare PR_TNEF_CORRELATION_KEY with the value stored in the transport envelope. If the values match, TNEF should be processed normally, that is, all the properties extracted from the TNEF attachment should be used. If the values do not match, all the properties from the TNEF attachment should be ignored. If PR_TNEF_CORRELATION_KEY is not set, the TNEF file should be considered to belong to this message, and the other properties extracted from it should be used.

PR_TRANSMITTABLE_DISPLAY_NAME

The PR_TRANSMITTABLE_DISPLAY_NAME property contains a recipient's display name in a secure form that cannot be changed.

Usage

Required on messaging user objects.

Details

Identifier 0x3a20; property type PT_TSTRING; property tag 0x3A20001E (0x3A20001F for Unicode)

Remarks

The PR_TRANSMITTABLE_DISPLAY_NAME property should be furnished by all address book providers. It contains the version of the recipient's display name that is transmitted with the message. For most address book providers this property has the same value as the PR_DISPLAY_NAME property. Providers that do not have a secure display name return PT_ERROR and MAPI changes the display name by adding quotation marks around the name.

A client application can use PR_TRANSMITTABLE_DISPLAY_NAME to prevent alteration or "spoofing" of entries. An example of spoofing is transmitting John Doe as John (What a Guy) Doe.

This property was formerly defined as PR_TRANSMITABLE_DISPLAY_NAME.

PR_TRANSPORT_KEY

The PR_TRANSPORT_KEY property contains a value used by the MAPI spooler to track the progress of an outbound message through the outgoing transport providers.

Usage

Reserved.

Details

Identifier 0x0E16; property type PT_LONG; property tag 0x0E160003

Remarks

Do not use this property.

See Also

PR_TRANSPORT_PROVIDERS property

PR_TRANSPORT_MESSAGE_HEADERS

The PR_TRANSPORT_MESSAGE_HEADERS property contains transport-specific message envelope information.

Usage

Optional on message objects.

Details

Identifier 0x007D; property type PT_TSTRING; property tag 0x007D001E (0x007D001F for Unicode)

Remarks

The transport provider can generate the message header information for inbound messages.

The PR_TRANSPORT_MESSAGE_HEADERS property offers an alternative to either discarding the transport message header information or prepending it to the message text. The client can choose whether or not to display the information.

See Also

PR_BODY property

PR_TRANSPORT_PROVIDERS

The PR_TRANSPORT_PROVIDERS property contains a list of identifiers of transport providers in the current profile.

Usage

Reserved.

Details

Identifier 0x3D02; property type PT_BINARY; property tag 0x3D020102

Remarks

Do not use this property.

See Also

MAPIUID structure

PR_TRANSPORT_STATUS

Obsolete MAPI spooler property.

Usage

No longer used.

Details

Identifier 0x0E11; property type PT_LONG; property tag 0x0E110003

Remarks

Do not use this property.

PR_TYPE_OF_MTS_USER

The PR_TYPE_OF_MTS_USER property contains the type of a message recipient, for use in a report.

Usage

Optional on recipient subobjects within report message objects.

Details

Identifier 0x0C1C; property type PT_LONG; property tag 0x0C1C0003

Remarks

The PR_TYPE_OF_MTS_USER property corresponds to the X.400 attribute MH_T_DELIVERY_POINT.

See Also

PR_CORRELATE_MTSID property

PR_USER_CERTIFICATE

The PR_USER_CERTIFICATE property contains an ASN.1 authentication certificate for a messaging user.

Usage

Optional on messaging user objects.

Details

Identifier 0x3A22; property type PT_BINARY; property tag 0x3A220102

Remarks

An authentication certificate is similar to a digital signature. Several MAPI properties supply ASN.1 certificates.

See Also

PR_ORIGINATOR_CERTIFICATE property, PR_RECIPIENT_CERTIFICATE property

PR_VALID_FOLDER_MASK

The PR_VALID_FOLDER_MASK property contains a bitmask of flags that indicate the validity of the entry identifiers of the folders in a message store.

Usage

Required on message store objects.

Details

Identifier 0x35DF; property type PT_LONG; property tag 0x35DF0003

Remarks

A folder's entry identifier can become invalid if a user deletes the folder or if the message store becomes corrupted.

One or more of the following flags can be set for the PR_VALID_FOLDER_MASK bitmask:

FOLDER_COMMON_VIEWS_VALID
: The common views folder has a valid entry identifier. See PR_COMMON_VIEWS_ENTRYID.

FOLDER_FINDER_VALID
: The finder folder has a valid entry identifier. See PR_FINDER_ENTRYID.

FOLDER_IPM_INBOX_VALID
: The interpersonal message (IPM) receive folder has a valid entry identifier. See **IMsgStore::GetReceiveFolder**.

FOLDER_IPM_OUTBOX_VALID
: The IPM Outbox folder has a valid entry identifier. See PR_IPM_OUTBOX_ENTRYID.

FOLDER_IPM_SENTMAIL_VALID
: The IPM Sent Items folder has a valid entry identifier. See PR_IPM_SENTMAIL_ENTRYID.

FOLDER_IPM_SUBTREE_VALID
: The IPM folder subtree has a valid entry identifier. See PR_IPM_SUBTREE_ENTRYID.

FOLDER_IPM_WASTEBASKET_VALID
: The IPM Deleted Items folder has a valid entry identifier. See PR_IPM_WASTEBASKET_ENTRYID.

FOLDER_VIEWS_VALID
: The views folder has a valid entry identifier. See PR_VIEWS_ENTRYID.

For more information on folder entry identifiers, see About Folder Identifiers.

See Also

PR_FOLDER_TYPE property

PR_VIEWS_ENTRYID

The PR_VIEWS_ENTRYID property contains the entry identifier of the user-defined Views folder.

Usage

Required on message store objects.

Details

Identifier 0x35E5; property type PT_BINARY; property tag 0x35E50102

Remarks

The common view folder contains a predefined set of standard view specifiers, while the view folder contains specifiers defined by a messaging user. These folders, which are not visible in the interpersonal message (IPM) hierarchy, can hold many view specifiers, each one stored as a message. The client application can choose to merge the two sets of specifiers and make them both available.

For more information on views, see About View Folders.

See Also

PR_COMMON_VIEWS_ENTRYID property, PR_DEFAULT_VIEW_ENTRYID property

PR_X400_CONTENT_TYPE

The PR_X400_CONTENT_TYPE property contains the content type for a submitted message.

Usage

Optional on message objects.

Details

Identifier 0x003C; property type PT_BINARY; property tag 0x003C0102

Remarks

The PR_X400_CONTENT_TYPE property corresponds to the X.400 attribute MH_T_CONTENT_TYPE.

See Also

PR_X400_DEFERRED_DELIVERY_CANCEL property

PR_X400_DEFERRED_DELIVERY_CANCEL

The PR_X400_DEFERRED_DELIVERY_CANCEL property was originally meant to contain TRUE if the message transfer system (MTS) allows X.400 deferred delivery cancellation.

Usage

Never used.

Details

Identifier 0x3E09; property type PT_BOOLEAN; property tag 0x3E09000B

Remarks

Do not use this property. It is not supported in MAPI 1.0.

See Also

PR_X400_CONTENT_TYPE property

PR_XPOS

The PR_XPOS property contains the x coordinate of the starting position (the upper-left corner) of a dialog box control, in standard Windows dialog units.

Usage

Required as a column entry in display tables.

Details

Identifier 0x3F05; property type PT_LONG; property tag 0x3F050003

Remarks

The PR_XPOS, PR_YPOS, PR_DELTAX, and PR_DELTAY properties position and size the dialog box control.

PR_YPOS

The PR_YPOS property contains the y coordinate of the starting position (the upper-left corner) of a dialog box control, in standard Windows dialog units.

Usage

Required as a column entry in display tables.

Details

Identifier 0x3F06; property type PT_LONG; property tag 0x3F060003

Remarks

The PR_XPOS, PR_YPOS, PR_DELTAX, and PR_DELTAY properties position and size the control.

MAPI Structures and Related Macros

The following alphabetized entries describe MAPI structures and related macros. For more detailed information, cross references to other publications or to other sections of this documentation are also included.

ADRENTRY

The **ADRENTRY** structure contains properties related to a recipient.

```
typedef struct _ADRENTRY
{
     ULONG        ulReserved1;
     ULONG        cValues;
     LPSPropValue rgPropVals;
} ADRENTRY, FAR *LPADRENTRY;
```

Members

ulReserved1
Reserved; must be zero.

cValues
Specifies the number of properties in the property value array pointed to by the **rgPropVals** member.

rgPropVals
Points to an array of **SPropValue** structures containing values for the properties of the recipient.

Remarks

An **ADRENTRY** structure describes recipient properties. The properties that are typically used to describe a recipient include:

- The display name
- The type of messaging system
- The messaging system address
- The entry identifier

ADRENTRY structures typically exist as components of **ADRLIST** structures. **ADRLIST** structures are used to represent collections of recipients.

Each **rgPropVals** member of an **ADRENTRY** structure must be allocated in a separate allocation from the **ADRLIST** structure. This insures the validity of any pointers to recipients when the **ADRLIST** structure containing them is reallocated or resized. The **rgPropVals** member must be deallocated prior to deallocation of the containing **ADRLIST** structure so that pointers to allocated **SPropValue** structures are not lost.

Use **FreePAdrList** to free the **ADRLIST** structure. In addition to resolved and unresolved recipient entries, **ADRENTRY** structures can be NULL, that is, the **cValues** member is zero and there are no property values. This is the case, for example, when the dialog box presented by **IAddrBook::Address** is used to remove a recipient from the list.

The structure member types and allocation rules for the **ADRENTRY** structures and **SRow** structures are identical. This means a row set retrieved from an address book contents table or a message recipient table can be cast to an **ADRLIST** and used as is.

For more information on **ADRENTRY** allocation and deallocation issues, see **IMessage::ModifyRecipients** and **IAddrBook::Address**.

The **ADRENTRY** structure is defined in MAPIDEFS.H.

See Also

IAddrBook::Address method, **IMessage::ModifyRecipients** method, **MAPIAllocateBuffer** function, **SRow** structure

ADRLIST

The **ADRLIST** structure describes one or more recipients. There is one one **ADRENTRY** structure for each recipient.

```
typedef struct _ADRLIST
{
     ULONG      cEntries;
     ADRENTRY   aEntries[MAPI_DIM];
} ADRLIST, FAR *LPADRLIST;
```

Members

cEntries
Indicates the number of entries in the array specified by the **aEntries** member.

aEntries
Indicates an array of **ADRENTRY** structures, each describing a recipient.

Remarks

An **ADRLIST** structure has the same format as an **SRowSet** structure. This means a row set retrieved from an address book contents table or a message recipient table can be cast to an **ADRLIST** structure and used as is.

An **ADRLIST** structure contains several **ADRENTRY** structures. These structures can contain both unresolved and resolved entries. An unresolved entry is lacking an entry identifier, or PR_ENTRYID, in its property value array. The e-mail address, or PR_EMAIL_ADDRESS, is typical but not required for resolved entries.

In addition to resolved and unresolved recipient entries, **ADRENTRY** structures can be NULL, that is, the **cValues** member is zero and there are no property values. This is the case, for example, when the dialog box presented by **IAddrBook::Address** is used to remove a recipient from the list.

If the client deals with a recipient list that is too large to fit in memory, the client can use an **ADRLIST** structure along with the **IMessage::ModifyRecipients** method to work with a subset of the list. The client should not use the address book common dialog boxes in such a situation.

For information about allocation rules for **ADRLIST** structures, see the entry for **ADRENTRY**.

Use the **CbADRLIST** macro to compute the number of bytes of memory occupied by an existing **ADRLIST** structure. The syntax for this macro is:

int CbADRLIST (**LPADRLIST** *_lpadrlist*)

The *_lpadrlist* parameter points to an **ADRLIST** structure. This macro returns the number of bytes occupied by the **ADRLIST** structure pointed to by *_lpadrlist*.

Use the **CbNewADRLIST** macro to determine the memory allocation requirements of an **ADRLIST** structure containing a specified number of **ADRENTRY** structures (recipients). The syntax is:

int CbNewADRLIST (**int** *_centries*)

The *_centries* parameter specifies the number of recipients. This macro returns the number of bytes of memory that an **ADRLIST** structure containing the number of recipients specified by *_centries* would occupy.

Use the **SizedADRLIST** macro to define a structure with the specified name that contains the specified number of **ADRENTRY** structures. The syntax is:

SizedADRLIST (**int** *_centries*, *_name*)

The *_centries* parameter specifies the number of **ADRENTRY** structures. The structure type is defined with the tag _ADRLIST_ _*name* and type name _*name*.

SizedADRLIST provides a way to define a recipient list with explicit bounds when array length requirements are known.

To use a sized recipient list pointer *lpSizedADRList* in any function call or structure that expects a **LPADRLIST** pointer, perform the following cast:

```
lpADRList = (LPADRLIST) & SizedADRList;
```

The **ADRLIST** structure is defined in MAPIDEFS.H.

See Also

ADRENTRY structure, **CbNewADRLIST** macro , **IMessage::ModifyRecipients** method, **SRowSet** structure

ADRPARM

The **ADRPARM** structure contains data used to control the display and behavior of an address dialog box.

```
typedef struct _ADRPARM
{
     ULONG            cbABContEntryID;
     LPENTRYID        lpABContEntryID;
     ULONG            ulFlags;
     LPVOID           lpReserved;
     ULONG            ulHelpContext;
     LPTSTR           lpszHelpFileName;
     LPFNABSDI        lpfnABSDI;
     LPFNDISMISS      lpfnDismiss;
     LPVOID           lpvDismissContext;
     LPTSTR           lpszCaption;
     LPTSTR           lpszNewEntryTitle;
     LPTSTR           lpszDestWellsTitle;
     ULONG            cDestFields;
     ULONG            nDestFieldFocus;
     LPTSTR FAR      *lppszDestTitles;
     ULONG FAR       *lpulDestComps;
     LPSRestriction   lpContRestriction;
     LPSRestriction   lpHierRestriction;
} ADRPARM, FAR *LPADRPARM;
```

Members

cbABContEntryID
Specifies the number of bytes in the entry identifier pointed to by **lpABContEntryID**.

lpABContEntryID
Points to the entry identifier of the container that initially supplies the list of recipient addresses that are displayed in the the address dialog box.

ulFlags
Bitmask of flags associated with various address dialog box options. The most significant four bits of the **ulFlags** member contain a version number identifying the version of the **ADRPARM** structure. The current version is 0 (zero), or ADRPARM_HELP_CTX. Future versions of the structure may be completely different. The current implementation of MAPI will fail for any version of the structure other than zero . Future versions of MAPI may or may not support the version-zero structure. The following macros are provided for extracting the version number from the **ulFlags** member and for combining it with the defined flags:

GET_ADRPARM_VERSION(*ulFlags*)

SET_ADRPARM_VERSION(*ulFlags*, *ulVersion*)

ADRPARM_HELP_CTX

The following flags can be set:

AB_RESOLVE
Causes all names to be resolved after the address dialog box is closed. The **Resolve Name** dialog box will be displayed if there are ambiguous entries in the recipient list. This guarantees that all the names returned by Address() are resolved.

AB_SELECTONLY
Disables the creation of one-off addresses and direct type-in entries for a recipient list. This flag is used only if the dialog box is modal.

ADDRESS_ONE
Indicates that the user can select exactly one message recipient, instead of a number of recipients from a recipient list. This flag is valid only when **cDestFields** is zero. This flag is used only if the dialog box is modal.

DIALOG_MODAL
Causes a modal dialog box to be displayed. The client must set either this flag or DIALOG_SDI, but not both.

DIALOG_OPTIONS
Causes the **Send Options** button to be displayed on the dialog box. This flag is used only if the dialog box is modal.

DIALOG_SDI
Causes a modeless dialog box to be displayed. This call returns immediately and hence does not modify the **ADRLIST** structure passed in.

This flag causes the **lpfnABSDI**, **lpfnDismiss**, and **lpvDismissContext** members of the **ADRPARM** structure to be used in the call to the **IAddrBook::Address** method. The client must set either this flag or DIALOG_MODAL, but not both.

lpReserved
Reserved, must be zero.

ulHelpContext
Specifies the context within Help that will first be shown when the user clicks the **Help** button in the address dialog box.

lpszHelpFileName
Points to the name of a Help file that will be associated with the address dialog box. The **lpszHelpFileName** member is used in conjunction with **ulHelpContext** to call the Windows **WinHelp** function.

lpfnABSDI
Points to a MAPI function based on the **ACCELERATEABSDI** function prototype.

lpfnDismiss
Points to a client application's dismiss function, a function based on the **DISMISSMODELESS** function prototype. MAPI calls a client's dismiss function when the address dialog box is modeless and the user has dismissed it. The **lpfnDismiss** member is used only if the DIALOG_SDI flag is set in **ulFlags**. The value of **lpfnDismiss** is NULL if the DIALOG_MODAL flag is set in **ulFlags**.

lpvDismissContext
Points to context information to be passed to the dismiss function specified by the *lpfnDismiss* parameter. The **lpvDismissContext** member is used only if the DIALOG_SDI flag is set in **ulFlags**, indicating that the address dialog box is modeless.

lpszCaption
Points to text to be used as a caption for the address dialog box.

lpszNewEntryTitle
Points to text to be used as a new entry prompt in the **New Entry** dialog box.

lpszDestWellsTitle
Points to text to be used as a title for the set of recipient edit controls that appear in the dialog box. This member is used only if the address dialog box is modal.

cDestFields
Indicates the number of recipient edit boxes in the address dialog box. A number from zero through three is typical. If the **cDestFields** member is zero and the ADDRESS_ONE flag is not set in **ulFlags**, the address book is open for browsing only. 0XFFFFFFFF implies use of the default number of wells; in this case **lppszDestTitles** and **lpulDestComps** must be NULL.

nDestFieldFocus
Indicates the field, specifying the button that adds to a well, in the address dialog box that should have the initial focus when the dialog box appears. This value must be between 0 and the value of

cDestFields minus 1.

lppszDestTitles
Points to an array of button captions to be displayed in the recipient edit controls of the address dialog box. The size of the array is the value of **cDestFields**. If the **lppszDestTitles** member is NULL, the **Address** method chooses default titles.

lpulDestComps
Points to an array of recipient types, such as MAPI_TO, MAPI_CC, and MAPI_BCC, associated with each recipient edit control. The size of the array is the value of **cDestFields**. If the **lpulDestComps** member is NULL, the **Address** method chooses default recipient types.

lpContRestriction
Points to an **SRestriction** structure containing restrictions that the client or service provider set on any address book container to be viewed. The **Address** method combines using the logical AND operator with any restrictions the user specifies while using the address dialog box.

lpHierRestriction
Points to an **SRestriction** structure containing restrictions on the hierarchy table used in the address dialog box.

Remarks

This structure is used with the **IAddrBook::Address** and **IMAPISupport::Address** methods.

If the address book is open for browsing only, **IAddrBook::Address** ignores its *lppAdrList* parameter. If the **cDestFields** member has the value 0xFFFFFFFF, the **Address** method displays the address dialog box in its default configuration, ignoring the **lppszDestTitles** and **lpulDestComps** members of **ADRPARM**.

When a client calls **IAddrBook::Address** or a service provider calls **IMAPISupport::Address**, it should always set the **lpfnABSDI** member to NULL. If the DIALOG_SDI flag is set, MAPI will fill in this member before it returns from the **Address** method. The client or service provider must then call the function pointed to by **lpfnABSDI** from its message loop in order for the address dialog box's accelerators to work. When the dialog box is dismissed and MAPI calls the function pointed to by the **lpfnDismiss** member, then the client or the service provider should disconnect the **lpfnABSDI** from its message loop.

The **ADRPARM** structure is defined in MAPIDEFS.H.

See Also

ACCELERATEABSDI function prototype, **DISMISSMODELESS** function prototype, **ENTRYID** structure, **IAddrBook::Address** method, **IMAPISupport::Address**, **SRestriction** structure

DTBLBUTTON

The **DTBLBUTTON** structure contains information about a button control for a dialog box built from a display table.

```
typedef struct _DTBLBUTTON
{
     ULONG ulbLpszLabel;
     ULONG ulFlags;
     ULONG ulPRControl;
} DTBLBUTTON, FAR *LPDTBLBUTTON;
```

Members

ulbLpszLabel
Indicates the offset from the beginning of the structure, in bytes, to a null-terminated string to be the label of the button.

ulFlags
Bitmask of flags used to designate the format of the label pointed to by the **ulbLpszLabel** member. The following flag can be set:

MAPI_UNICODE
Indicates the passed-in strings are in Unicode format. If the MAPI_UNICODE flag is not set, the strings are in ANSI format.

ulPRControl
Specifies a property of type PT_OBJECT on which MAPI can open an **IMAPIControl** interface using an **OpenProperty** call. The client supplies the **IMAPIControl** interface. It is opened when the button is clicked.

Remarks

For more information on how this structure is used, see About Display Tables.

Use the **SizedDtblButton** macro to create a structure similar to **DTBLBUTTON** but containing a label of specified character length. The syntax is:

SizedDtblButton (**int** *n*, *u*)

The *n* parameter specifies the character length of the button's label. The *u* parameter specifies the structure type is defined with the tag _DTBLBUTTON_ *u* and type name *u*.

A **DTBLBUTTON** structure does not explicitly give the length of the *lpszLabel* string, it only provides an offset to the first character in it.

The **DTBLBUTTON** structure is defined in MAPIDEFS.H.

See Also

DTCTL structure, PR_CONTROL_TYPE property

DTBLCHECKBOX ▸

The **DTBLCHECKBOX** structure contains information about a check box to be used in a dialog box built from a display table.

```
typedef struct _DTBLCHECKBOX
{
     ULONG ulbLpszLabel;
     ULONG ulFlags;
     ULONG ulPRPropertyName;
} DTBLCHECKBOX, FAR *LPDTBLCHECKBOX;
```

Members

ulbLpszLabel
Indicates the offset from the beginning of the structure, in bytes, to the label for the check box.

ulFlags
Bitmask of flags used to designate the format of the label pointed to by the **ulbLpszLabel** member. The following flag can be set:

MAPI_UNICODE
Indicates the passed-in strings are in Unicode format. If the MAPI_UNICODE flag is not set, the strings are in ANSI format.

ulPRPropertyName
Specifies a property tag of type PT_BOOLEAN whose value is manipulated by the user changing the state of the check box. The **GetProps** method is used to initalize the check box. The **SetProps** method is used to manipulate this property's value as the user makes changes to the check box.

Remarks

A **DTBLCHECKBOX** structure does not specify the character length of the label, it only contains an offset to the beginning of the label in memory allocated for the structure.

The **SizedDtblCheckBox** macro defines a structure similar to **DTBLCHECKBOX** but containing a label of specified character length. The syntax is:

SizedDtblCheckBox (**int** *n*, *u*)

The *n* parameter specifies the character length of the check box's label. The structure type is defined with the tag _DTBLCHECKBOX_ *u* and type name *u*.

SizedDtblCheckBox provides a way to define a check box when the number of label characters is known. A **DTBLCHECKBOX** structure does not explicitly give the length of the *lpszLabel* string, it only provides an offset to the first character in it.

To use a sized display table check box pointer *lpSizedDtblCheckBox* in any function call or structure that expects a **LPDTBLCHECKBOX** pointer, perform the following cast:

```
lpDtblCheckBox = (LPDTBLCHECKBOX) lpSizedDtblCheckBox
```

The **DTBLCHECKBOX** structure is defined in MAPIDEFS.H.

For more information on how this structure is used, see About Display Tables.

See Also

DTCTL structure, PR_CONTROL_TYPE property

DTBLCOMBOBOX

The **DTBLCOMBOBOX** structure contains information about a combo box that is to be part of a dialog box.

```
typedef struct _DTBLCOMBOBOX
{
     ULONG ulbLpszCharsAllowed;
     ULONG ulFlags;
     ULONG ulNumCharsAllowed;
     ULONG ulPRPropertyName;
     ULONG ulPRTableName;
} DTBLCOMBOBOX, FAR *LPDTBLCOMBOBOX;
```

Members

ulbLpszCharsAllowed
Indicates the offset from the beginning of the structure, in bytes, to a string that lists the characters allowed in the combo box's edit box. The same filter is applied to every character entered, it is not interrupted as a regular expression.

The format of the string is as follows:

*****	Any character is allowed (for example, "*").
[ ]	Defines a set of characters (for example, "**[**0123456789**]**").
-	Indicates a range of characters. Typically used like "[a-z]".
~	Indicates that these characters are not allowed. (for example, "[~0-9]").
\	Used to quote any of the above symbols (for example, "[\-\\\[\]]" means -, \, [, and] characters are allowed).

ulFlags
Bitmask of flags used to designate the format of the text pointed to by the **ulbLpszCharsAllowed** member. The following flag can be set:

MAPI_UNICODE
Indicates the passed-in strings are in Unicode format. If the MAPI_UNICODE flag is not set, the strings are in ANSI format.

ulNumCharsAllowed
Specifies the maximum number of characters that the user can type into the combo box's edit box.

ulPRPropertyName
Specifies a property tag whose values uniquely identify rows in the table (typically of type PT_TSTRING) used to populate the list box. The value of this property identifies the row selected via a choice in the list box.

ulPRTableName
Specifies a property tag of type PT_OBJECT on which an **IMAPITable** interface can be opened using an **OpenProperty** call. The rows of the table are to be used as items in the list box. The table must have a single column of the same type as **ulPRPropName**.

Remarks

A **DTBLCOMBOBOX** structure does not specify the character length of the allowed character string, it only contains an offset to the beginning of it in memory allocated for the structure.

The **ulPRPropertyName** and **ulPRTableName** members work together. The **ulPRTableName** member supplies the choices; **ulPRPropertyName** is set to the chosen item when the dialog is dismissed (or when the combo box is exited, if the DT_SETIMMEDIATE flag is set.)

Use the **SizedDtblComboBox** macro to define a structure similar to **DTBLCOMBOBOX**, but with the added explicit number of characters in the combo box's text edit, or static text, control. **SizedDtblComboBox** provides a way to define a combo box when the length of the allowed character string is known. The syntax is:

SizedDtblComboBox (**int** *n*, *u*)

> The *n* parameter specifies the length of the allowed character string. The structure type is defined with the tag _DTBLCOMBOBOX_ *u* and type name *u*. See the **ulbLpszCharsAllowed** member of this structure for the formatting rules of this character string.

To use a sized display table combo box pointer *lpSizedDtblComboBox* in any function call or structure that expects a **LPDTBLCOMBOBOX** pointer, perform the following cast:

```
lpDtblComboBox = (LPDTBLCOMBOBOX) lpSizedDtblComboBox
```

The **DTBLCOMBOBOX** structure is defined in MAPIDEFS.H.

For more information on how this structure is used, see About Display Tables.

See Also

DTCTL structure, PR_CONTROL_TYPE property

DTBLDDLBX

The **DTBLDDLBX** structure contains information about a drop-down list box that will be part of a dialog box.

```
typedef struct _DTBLDDLBX
{
     ULONG ulFlags;
     ULONG ulPRDisplayProperty;
     ULONG ulPRSetProperty;
     ULONG ulPRTableName;
} DTBLDDLBX, FAR *LPDTBLDDLBX;
```

Members

ulFlags
Reserved, must be zero.

ulPRDisplayProperty
Specifies a property name of type PT_TSTRING. The text value of this property in each row is displayed as an item in the list box.

ulPRSetProperty
Specifies a property name whose values uniquely identify rows in the table – for example, PR_ENTRYID. The value of this property identifies the row selected from the choices in the list box. If the property name is PR_NULL, the list box is not a single selection type.

ulPRTableName
Specifies a property of type PT_OBJECT on which an **IMAPITable** interface can be opened using an **OpenProperty** call. The rows of the table should correspond to items in the list box. The table should have two columns, **ulPRDisplayProperty** and **ulPrSetProperty**.

Remarks

The DTBLDDLBX structure is defined in MAPIDEFS.H.

For more information on how this structure is used, see About Display Tables.

See Also

DTCTL structure

DTBLEDIT

The **DTBLEDIT** structure contains information about an edit box that will be part of a dialog box.

```
typedef struct _DTBLEDIT
{
     ULONG ulbLpszCharsAllowed;
     ULONG ulFlags;
     ULONG ulNumCharsAllowed;
     ULONG ulPropTag;
} DTBLEDIT, FAR *LPDTBLEDIT;
```

Members

ulbLpszCharsAllowed

Offset from the beginning of the structure, in bytes, to a string that lists the characters allowed in the combo box's edit box. The same filter is applied to every character.

The allowed character string follows the formatting rules:

*****	Any character is allowed (for example, "*****").
[ ]	Defines a set of characters (for example, "**[**0123456789**]**").
-	Indicates a range of characters. Typically used like "[a-z]".
~	Indicates that these characters are not allowed (for example, "[~0-9]").

\ Used to quote any of the above symbols (for example, "[\-\\\[\]]" means -, \, [, and] characters are allowed).

ulFlags

Bitmask of flags used to designate the format of the text pointed to by the **ulbLpszCharsAllowed** member. The following flag can be set:

MAPI_UNICODE

Indicates the passed-in strings are in Unicode format. If the MAPI_UNICODE flag is not set, the strings are in ANSI format.

ulNumCharsAllowed

Specifies the maximum number of characters that the user can type into the edit box.

ulPropTag

Specifies a property name of type PT_TSTRING. The text in this property is displayed and edited in the edit box.

Remarks

A **DTBLEDIT** structure does not specify the character length of the allowed character string, it only contains an offset to it in memory allocated for the structure.

Use the **SizedDtblEdit** macro to define a structure similar to **DTBLEDIT** but with the added explicit number of characters contained in the allowed characters member. The syntax is:

SizedDtblEdit (**int** *n*, *u*)

The *n* parameter specifies the length of the allowed characters string. See **ulbLpszCharsAllowed** for the formatting rules of the allowed character string.

SizedDtblEdit provides a way to define a text edit box when the number of allowed characters is known. A **DTBLEDIT** structure does not explicitly give the length of the *lpszCharsAllowed* string, it only provides an offset to the first character in the string.

To use *lpSizedDtblEdit*, which is a sized, display table edit pointer, in any function call or structure that expects a **LPDTBLEDIT** pointer, perform the following cast:

```
lpDtblEdit = (LPDTBLEDIT) lpSizedDtblEdit
```

The **DBTLEDIT** stucture is defined in MAPIDEFS.H.

For more information on how this structure is used, see About Display Tables.

See Also

DTCTL structure, **IMAPIProp::GetProps** method, PR_CONTROL_TYPE property

DTBLGROUPBOX

The **DTBLGROUPBOX** structure contains information about a group box that will be part of a dialog box.

```
typedef struct _DTBLGROUPBOX
{
     ULONG ulbLpszLabel;
     ULONG ulFlags;
} DTBLGROUPBOX, FAR *LPDTBLGROUPBOX;
```

Members

ulbLpszLabel
Offset from the beginning of the structure, in bytes, to be the label of the group box.

ulFlags
Bitmask of flags used to designate the format of the text pointed to by the **ulbLpszLabel** member. The following flag can be set:

MAPI_UNICODE
Indicates the passed-in strings are in Unicode format. If the MAPI_UNICODE flag is not set, the strings are in ANSI format.

Remarks

A **DTBLGROUPBOX** structure does not specify the character length of the label, it only contains an offset to the beginning of the label in memory allocated for the structure.

Use the **SizedDtblGroupBox** macro to define a structure similar to **DTBLGROUPBOX** but containing a label of specified character length. The syntax is:

SizedDtblGroupBox (**int** *n*, *u*)

The *n* parameter specifies the character length of the group box's label. The structure type is defined with the tag _DTBLGROUPBOX_ *u* and type name *u*.

SizedDtblGroupBox provides a way to define a group box when the number of label characters is known. A **DTBLGROUPBOX** structure does not explicitly give the length of the *lpszLabel* string, it only provides an offset to the first character in it.

To use a sized display table group box pointer *lpSizedDtblGroupBox* in any function call or structure that expects a **LPDTBLGROUPBOX** pointer, perform the following cast:

```
lpDtblGroupBox = (LPDTBLGROUPBOX) lpSizedDtblGroupBox
```

For more information on how this structure is used, see About Display Tables.

The **DTBLGROUPBOX** structure is defined in MAPIDEFS.H.

See Also

DTCTL structure

DTBLLABEL

The **DTBLLABEL** structure contains information about a label that will be part of a dialog box.

```
typedef struct _DTBLLABEL
{
     ULONG ulbLpszLabelName;
     ULONG ulFlags;
} DTBLLABEL, FAR *LPDTBLLABEL;
```

Members

ulbLpszLabelName
Specifies the offset from the beginning of the structure, in bytes, to be the label text string.

ulFlags
Bitmask of flags used to designate the format of the label text string. The following flag can be set:

MAPI_UNICODE
Indicates the passed-in strings are in Unicode format. If the MAPI_UNICODE flag is not set, the strings are in ANSI format.

Remarks

Use the **SizedDtblLabel** macro to create a structure definition similar to **DTBLLABEL** but containing a specified character length for the label. The syntax is: **SizedDtblLabel** (**int** *n*, *u*).

The *n* parameter specifies the character length of the label, including the terminating NULL character. The structure type is defined with the tag _DTBLLABEL_ *u* and type name *u*.

SizedDtblLabel provides a way to define a display table label when the number of characters in the label is known. A **DTBLLABEL** structure does not explicitly give the length of the *lpszLabelName* string, it only provides an offset to the first character of the string.

To use a pointer to a sized label control such as *lpSizedDtblLabel* in any function call or structure that expects a **LPDTBLLABEL** pointer, perform the following cast:

```
lpDtblLabel = (LPDtblLabel) lpSizedDtblLabel
```

For more information on how this structure is used, see About Display Tables.

The **DTBLLABEL** structure is defined in MAPIDEFS.H.

See Also

DTCTL structure

DTBLLBX

The **DTBLLBX** structure contains information about a list box that will be part of a dialog box.

```
typedef struct _DTBLLBX
{
     ULONG ulFlags;
     ULONG ulPRSetProperty;
     ULONG ulPRTableName;
} DTBLLBX, FAR *LPDTBLLBX
```

Members

ulFlags
A bitmask of flags used to eliminate a horizontal or vertical scroll bar for a listbox. The following flags can be set:

MAPI_NO_HBAR
Indicates that no horizontal scroll bar will be shown for the list box.

MAPI_NO_VBAR
Indicates that no vertical scroll bar will be shown for the list box.

ulPRSetProperty
Specifies a property name whose values uniquely identify rows in the table – for example, PR_ENRTYID. The value of this property identifies the row selected from the choices in the list box. Initially, the **ulPRPropertyName** member contains the value of a default selection. If the property name is PR_NULL, the list box is not a single selection type.

ulPRTableName
Specifies a property of type PT_OBJECT on which an **IMAPITable** interface can be opened using an **OpenProperty** call. The rows of the table will correspond to items in the list box.

Remarks

The **ulPRSetProperty** member and **ulPRTableName** member work together; when one value is chosen from the table, it is written back to **ulPRSetProperty** when the dialog is dismissed.

The **DTBLLX** structure is defined in MAPIDEFS.H.

See Also

DTCTL structure

DTBLMVDDLBOX

The **DTBLMVDDLBOX** structure contains information about a multivalued property to be used in a drop-down list box.

```
typedef struct _DTBLMVDDLBX
{
     ULONG ulFlags;
     ULONG ulMVPropTag;
} DTBLMVDDLBX, FAR * LPDTBLMVDDLBX;
```

Members

ulFlags
Reserved; must be zero.

ulMVPropTag
Specifies a property name that is a multivalued type, in this case PT_MV_TSTRING. The different values of this property are displayed as distinct entries in the drop-down list box.

Remarks

This structure defines a list box that is browse only; the user cannot make a selection.

The **DTBLMVDDLBOX** structure is defined in MAPIDEFS.H.

See Also

DTCTL structure

DTBLMVLISTBOX

The **DTBLMVLISTBOX** structure contains a multivalued property to be displayed in a list box.

```
typedef struct _DTBLMVLISTBOX
{
     ULONG ulFlags;
     ULONG ulMVPropTag;
} DTBLMVLISTBOX, FAR * LPDTBLMVLISTBOX;
```

Members

ulFlags
Reserved; must be zero.

ulMVPropTag
Specifies a property name that is a multivalued type, PT_MV_TSTRING.

Remarks

The list box defined by this structure is browse-only; the user cannot make a selection.

Only multivalued string properties are supported for the multivalued list box and drop-down list box; other multivalued property types are not supported.

The **DTBLMVLISTBOX** structure is defined in MAPIDEFS.H.

See Also

DTCTL structure

DTBLPAGE

The **DTBLPAGE** structure contains information about a tabbed page that will be part of a dialog box.

```
typedef struct _DTBLPAGE
{
     ULONG ulbLpszLabel;
     ULONG ulFlags;
     ULONG ulbLpszComponent;
     ULONG ulContext;
} DTBLPAGE, FAR *LPDTBLPAGE;
```

Members

ulbLpszLabel
Specifies the offset from the beginning of the structure, in bytes, to the label text for the page tab.

ulFlags
Bitmask of flags used to designate the format of the text pointed to by the **ulbLpszLabel** member. The following flag can be set:

MAPI_UNICODE
Indicates the passed-in strings are in Unicode format. If the MAPI_UNICODE flag is not set, the strings are in ANSI format.

ulbLpszComponent
Specifies the offset, in number of bytes, to a Help file component string that appears in a MAPISVC.INF entry mapping Help files to user interface items. If this string is not NULL, a user can access extended Help for the tabbed page by clicking the **Help** button on the property sheet.

ulContext
Specifies a unique identifier for the tabbed page within the string defined by the **ulbLpszComponent** member. If this identifier is zero and the component string is NULL, there is no Help associated with the page.

Remarks

A **DTBLPAGE** structure does not specify the character length of the label and Help component strings, it only contains offsets to the beginnings of these strings in memory allocated for the structure.

The **ulbLpszComponent** member and the **ulContext** member must both be nonzero in order for the **Help** button to work.

Use the **SizedDtblPage** macro to define a structure similar to **DTBLPAGE** but whose page label and Help file component string are of specified character lengths. The syntax is:

SizedDtblPage (**int** *n*, **int** *n1*, *u*)

The *n* parameter specifies the character length of the page's label. The *n1* parameter specifies the character length of a Help file component string that appears in a MAPISVC.INF entry mapping Help files to user interface items. If this string is not NULL, a user can access extended Help for the tabbed page by clicking the **Help** button on the property sheet. The structure type is defined with the tag _DTBLPAGE_ *u* and type name *u*.

SizedDtblPage provides a means of defining a display table page when the number of label and component string characters is known. A **DTBLPAGE** structure does not explicitly give the length of these strings, only offsets to their locations in memory allocated for the structure.

To use *lpSizedDtblPage*, a sized display-table page pointer in any function call or structure that expects a **LPDTBLPAGE** pointer, perform the following cast:

```
lpDtblPage = (LPDTBLPAGE) lpSizedDtblPage
```

The **DTBLPAGE** stucture is defined in MAPIDEFS.H.

See Also

DTCTL structure

DTBLRADIOBUTTON

The **DTBLRADIOBUTTON** structure contains information about one radio button that will be part of a radio button group in a display table dialog box.

```
typedef struct _DTBLRADIOBUTTON
{
     ULONG ulbLpszLabel;
     ULONG ulFlags;
     ULONG ulcButtons;
     ULONG ulPropTag;
     long lReturnValue;
} DTBLRADIOBUTTON, FAR *LPDTBLRADIOBUTTON;
```

Members

ulbLpszLabel
Specifies the offset from the beginning of the structure, in number of bytes, to the label text for the radio button.

ulFlags
Bitmask of flags used to designate the format of the text pointed to by the **ulbLpszLabel** member. The following flag can be set:

MAPI_UNICODE
Indicates the passed-in strings are in Unicode format. If the MAPI_UNICODE flag is not set, the strings are in ANSI format.

ulcButtons
Specifies the number of buttons in the radio button group. The remaining   button structures must appear in successive rows of the display table. Each of these rows should contain the same value for the **ulcButtons** member.

ulPropTag
Specifies the name of the property associated with the group of radio buttons. The property tag must be of type PT_LONG. The initial selection within the radio button group is based on the initial value of this property. Each button in the group must have **ulPropTag** set to the same property.

lReturnValue
Specifies a unique number that represents the radio button within the group when the button is selected at run time. Unless the flag DT_SET_IMMEDIATE is set in the display table, the client application sets the value of **lReturnValue** to **ulPropTag** when the dialog box is closed. If this flag is set, the client application writes the value of **lReturnValue** to **ulPropTag** at the time the radio button is selected.

Remarks

A **DTBLRADIOBUTTON** structure does not specify the character length of the label, it only contains an offset to the beginning of the label in memory allocated for the structure.

The **DTBLRADIOBUTTON** structure is defined in MAPIDEFS.H.

See Also

BuildDisplayTable function, **DTCTL** structure, **SizedDtblButton** macro

DTCTL

The **DTCTL** structure contains information about one or more dialog box controls for inclusion in a display table. The **BuildDisplayTable** function uses this structure for building the display table from control resources.

```
typedef struct
{
     ULONG      ulCtlType;
     ULONG      ulCtlFlags;
     LPBYTE     lpbNotif;
     ULONG      cbNotif;
     LPTSTR     lpszFilter;
     ULONG      ulItemID;
     union
     {
          LPVOID                 lpv;
          LPDTBLLABEL            lplabel;
          LPDTBLEDIT             lpedit;
          LPDTBLLBX              lplbx;
          LPDTBLCOMBOBOX         lpcombobox;
          LPDTBLDDLBX            lpddlbx;
          LPDTBLCHECKBOX         lpcheckbox;
          LPDTBLGROUPBOX         lpgroupbox;
          LPDTBLBUTTON           lpbutton;
          LPDTBLRADIOBUTTON      lpradiobutton;
          LPDTBLMVLISTBOX        lpmvlbx;
          LPDTBLMVDDLBX          lpmvddlbx;
          LPDTBLPAGE             lppage;
     } ctl;
} DTCTL, FAR *LPDTCTL;
```

Members

ulCtlType
Specifies a value indicating the control represented by the **DTCTL** structure. The possible values are:

DTCT_LABEL
Indicates a label.

DTCT_EDIT
Indicates a text box.

DTCT_LBX
Indicates a list box.

DTCT_COMBOBOX
Indicates a combo box.

DTCT_DDLBX
Indicates a drop-down list box.

DTCT_CHECKBOX
Indicates a check box.

DTCT_GROUPBOX
Indicates a group box.

DTCT_BUTTON
Indicates a button control.

DTCT_PAGE
: Indicates a tabbed page.

DTCT_RADIOBUTTON
: Indicates a radio button.

DTCT_INKEDIT
: Indicates an ink-aware edit box.

DTCT_MVLISTBOX
: Indicates a multivalued list box.

DTCT_MVDDLBX
: Indicates a multivalued drop-down list box.

ulCtlFlags

Bitmask of dialog box control flags. The following flags can be set:

DT_ACCEPT_DBCS
: Indicates ANSI or DBCS format is accepted.

DT_EDITABLE
: Indicates the text in the dialog box control can be edited.

DT_MULTILINE
: Indicates the dialog box control can contain multiple lines.

DT_PASSWORD_EDIT
: Indicates that the edit control contains a password, and therefore the contents of the edit control should not be displayed to the user.

DT_REQUIRED
: Indicates the dialog box control is required.

DT_SET_IMMEDIATE
: Enables immediate output of a value upon a change in value of the control. This includes checking a check box or radio button, entering text in an edit control, or changing the selection in a list box.

lpbNotif

Points to notification data.

cbNotif

Indicates the size, in bytes, of the notification data.

lpszFilter

Points to a character filter for an edit box.

ulItemID

Item identifier validating parallel dialog box template entries.

lpv

Points to a value that the client application should initialize to avoid warnings when calling **BuildDisplayTable**.

lplabeln

Points to a **DTBLLABEL** structure.

lpedit

Points to a **DTBLEDIT** structure.

lplbx

Points to a **DTBLLBX** structure.

lpcombobox

Points to a **DTBLCOMBOBOX** structure.

lpddlbx

Points to a **DTBLDDLBX** structure.

lpcheckbox

Points to a **DTBLCHECKBOX** structure.

lpgroupbox
Points to a **DTBLGROUPBOX** structure.

lpbutton
Points to a **DTBLBUTTON** structure.

lpradiobutton
Points to a **DTBLRADIOBUTTON** structure.

lpmvlbx
Points to a **DTBLMVLISTBOX** structure.

lpmvddlbx
Points to a **DTBLMVDDLBOX** structure.

lppage
Points to a **DTBLPAGE** structure.

Remarks

Although the the **BuildDisplayTable** function uses this structure for building the display table from control resources, the DTCTL structure never appears in the display table itself. This structure simply supplies information to **BuildDisplayTable**.

In the **ulCtlFlags** member, four flags – DT_ACCEPT_DBCS, DT_EDITABLE, DT_MULTILINE_and DT_PASSWORD_EDIT – affect edit controls only. Two others – DT_REQUIRED and DT_SET_IMMEDIATE – affect any editable control.

The controls available for a dialog box are label, edit box, ink-aware edit box, list box, drop-down list box, combo box, check box, group box, button, radio button, and tabbed page.

The **DTCTL** structure is defined in MAPIUTIL.H.

See Also

BuildDisplayTable function, **DTBLBUTTON** structure, **DTBLCHECKBOX** structure, **DTBLCOMBOBOX** structure, **DTBLDDLBX** structure, **DTBLEDIT** structure, **DTBLGROUPBOX** structure, **DTBLLABEL** structure, **DTBLLBX** structure, **DTBLMVDDLBOX** structure, **DTBLMVLISTBOX** structure, **DTBLPAGE** structure, **DTBLRADIOBUTTON** structure

DTPAGE

The **DTPAGE** structure contains information about a tabbed dialog box page. The **BuildDisplayTable** function uses this structure to build the display table from control resources.

```
typedef struct DTPAGE
{
     ULONG        cctl;
     LPTSTR       lpszResourceName;
     union
     {
          LPTSTR              lpszComponent;
          ULONG               ulItemID;
      }
     LPDTCTL      lpctl;
}    DTPAGE, FAR *LPDTPAGE;
```

Members

cctl
Contains the count of controls in the **lpctl** array member of the same **DTPAGE** structure.

lpszResourceName
Points to the name of the tabbed page resource or to an integer identifier for the resource. The **MAKEINTRESOURCE** macro should be used with an integer identifer to guarantee correctness.

lpszComponent
Points to a component string that appears in a Help file mapping entry in MAPISVC.INF. The **ulItemID** member and this member are alternatives.

ulItemID
Contains a string resource identifier from which the component name can be read.

lpctl
Points to a array of **DTCTL** structures, one for each control on the page.

Remarks

To identify the tabbed page, a caller can use either a hard-coded string, which should be assigned to the **lpszComponent** member or an integer resource identifier, which should be assigned to the **ulItemID** member. When an integer identifier is used, **BuildDisplayTable** obtains the component name from the corresponding string resource.

Although the **BuildDisplayTable** function uses this structure to build the display table from control resources, the **DTPAGE** structure never appears in the display table itself.

Each help file mapping entry in MAPISVC.INF consists of a component string, no longer than 30 characters, on the left side and a Help file path on the right. Both **ulItemID** and **lpszResourceName** are found in the *hInstance* parameter of **BuildDisplayTable**.

The **DTPAGE** structure is defined in MAPIUTIL.H.

See Also

BuildDisplayTable function, **DTBLPAGE** structure, **DTCTL** structure

ENTRYID

The **ENTRYID** structure contains an entry identifier for a MAPI object.

```
typedef struct
{
     BYTE        abFlags[4];
     BYTE        ab[MAPI_DIM];
} ENTRYID, FAR *LPENTRYID;
```

Members

abFlags
Bitmask of flags that provide information describing the object. Only the first byte of the flags, **abFlags**[0], may be set by the provider; the other three are reserved. The client should never change anything in the structure.

The following flags can be set in **abFlags**[0]:

MAPI_NOTRECIP
Indicates the entry identifier cannot be used as a recipient on a message.

MAPI_NOTRESERVED
Indicates that other users cannot access the entry identifier.

MAPI_NOW
Indicates the entry identifier cannot be used at other times.

MAPI_SHORTTERM
Indicates the entry identifier is short-term. All other values in this byte must be set unless other uses of the entry identifier are allowed.

MAPI_THISSESSION
Indicates the entry identifier cannot be used on other sessions.

The flags must not be set for permanent entry identifiers and set for short-term entry identifiers.

ab
Array of binary data used by service providers. The client application cannot use this array.

Remarks

A message store or address book provider fills an entry identifier with information that makes sense for that provider. For a message store, entry identifiers identify folders or messages but not attachments. For an address book, entry identifiers identify address book containers, individual messaging users, and distribution lists. Entry identifiers also identify message store, status, profile, and session objects. An entry identifier is a unique number that distinguishes one object from all other objects of the same type. For example, a message entry identifier uniquely identifies a message in the message store.

Directly comparing the binary data in two entry identifiers does not provide much information to the application because MAPI can use different binary values in **ENTRYID** structures to refer to the same object. To determine if two entry identifiers refer to the same object, the client application can use the **IMAPISession::CompareEntryIDs** method for the appropriate object.

The following rules apply for using the **abFlags[0]** byte in entry identifiers.

- Each object, if queried for its entry identifier through its **IMAPIProp** interface, generates and provides the most permanent form of its entry identifier. To indicate an entry identifier is the most permanent one available for a given object, clear all the bits in the **abFlags** array.
- Entry identifiers found in most MAPI tables can be short-term instead of permanent. The use of short-term entry identifiers is usually restricted. In general, a client can use a short-term entry identifier to open an object only in the current MAPI session. Use the short-term entry identifier with the message store only for the current session and only while the address book remains open. To

indicate a short-term entry identifier, set all the values in the **abFlags** array. The next rule slightly modifies these limitations.

- In some cases, your short-term entry identifier might be just as good as a permanent entry identifier. In such cases, clear individual bits in the **abFlags** array as appropriate.

Although providers should handle arbitrarily aligned entry identifiers, clients cannot expect providers to handle arbitrarily aligned entry identifiers. Failure to pass in a suitable aligned entry identifier can result in an alignment fault on RISC CPUs. To allow for those providers that do not handle arbitrarily aligned identifiers, clients should always pass in naturally aligned entry identifiers. The natural alignment factor, typically 8 bytes, is the largest data type supported by the CPU, and usually the same alignment factor used by the system memory allocator. A naturally aligned memory address allows the CPU to access any data type it supports at that address without generating an alignment fault. For RISC CPUs, a data type of size N bytes must usually be aligned on an even multiple of N bytes, with the address being an even multiple of N.

The **CbNewENTRYID** macro determines the memory allocation requirements of an **ENTRYID** structure with an entry identifier of a specified byte size. The syntax is:

int CbNewENTRYID (**int** _*cb*)

The *cb* parameter specifies the number of bytes used for the entry identifier. This macro returns the size, in bytes, of an **ENTRYID** containing a _*cb* byte entry identifier.

The **SizedENTRYID** macro creates a structure type definition identical to that of **ENTRYID** but containing an **ab** member that is a sized array. Use this macro to create an entry identifier array with explicit bounds. The syntax is:

SizedENTRYID (**int** _*cb*, _*name*)

The _*cb* parameter specifies the number of bytes used for the entry identifier. The structure type is defined with the tag _ENTRYID_ _*name* and type name _*name*.

The **SizedENTRYID** macro provides a way to define an entry identifier after array length requirements are known.

To use a sized recipient list pointer such as *lpSizedENTRYID* in any function call or structure that expects an **LPENTRYID** pointer, perform the following cast:

```
lpENTRYID = (LPENTRYID) lpSizedENTRYID
```

The **ENTRYID** structure is defined in MAPIDEFS.H.

See Also

IMAPISupport::CompareEntryIDs method, PR_RECORD_KEY property

ENTRYLIST

The **ENTRYLIST** structure is an array of entry identifiers representing MAPI objects.

```
typedef SBinaryArray ENTRYLIST, FAR *LPENTRYLIST;
```

Remarks

The **ENTRYLIST** structure is defined in MAPIDEFS.H.

See Also

ENTRYID structure, **SBinaryArray** structure

ERROR_NOTIFICATION

The **ERROR_NOTIFICATION** structure describes an error notification event. When a critical error occurs in an object, the object calls the **IMAPIAdviseSink::OnNotify** method of all registered advise sinks, passing an **ERROR_NOTIFICATION** structure for the *lpNotifications* parameter.

```
typedef struct _ERROR_NOTIFICATION
{
     ULONG          cbEntryID;
     LPENTRYID      lpEntryID;
     SCODE          scode;
     ULONG          ulFlags;
     LPMAPIERROR    lpMAPIError;
} ERROR_NOTIFICATION;
```

Members

cbEntryID
Indicates the size, in bytes, of the entry identifier of the object causing the error.

lpEntryID
Points to the entry identifier of the object causing the error.

scode
Specifies the error value for the critical error.

ulFlags
Bitmask of flags used to designate the format of the text pointed to by the **lpszError** member in the structure pointed to by **lpMAPIError**. The following flag can be set:

MAPI_UNICODE
Indicates the passed-in strings are in Unicode format. If the MAPI_UNICODE flag is not set, the strings are in ANSI format.

lpMAPIError
Points to a **MAPIERROR** structure describing the error.

Remarks

The value of the **cbEntryID** member and the **lpEntryID** member can be NULL.

The **ERROR_NOTIFICATION** structure is defined in MAPIDEFS.H.

See Also

NOTIFICATION structure

EXTENDED_NOTIFICATION

The **EXTENDED_NOTIFICATION** structure describes a notification for an event that is specific to a provider. When this provider-specific event occurs, the provider calls the **IMAPIAdviseSink::OnNotify** method of all registered advise sinks, passing an **EXTENDED_NOTIFICATION** structure for the *lpNotifications* parameter.

```
typedef struct _EXTENDED_NOTIFICATION
{
     ULONG    ulEvent;
     ULONG    cb;
     LPBYTE   pbEventParameters;
} EXTENDED_NOTIFICATION;
```

Members

ulEvent
Indicates an extended event code that is defined by the provider.

cb
Indicates the size, in bytes, of the event-specific parameters defined in the **pbEventParameters** member.

pbEventParameters
Points to event-specific parameters. What type of parameters are used depends on the value of the **ulEvent** member; these parameters are documented by the provider that issued the event.

Remarks

The provider defines and issues the event for which the **EXTENDED_NOTIFICATION** structure contains information.

The **EXTENDED_NOTIFICATION** structure is defined in MAPIDEFS.H.

See Also

NOTIFICATION structure

FILETIME

The **FILETIME** structure, implemented for Windows, holds a 64-bit time value for a file. This value represents the number of 100-nanosecond intervals since January 1, 1601.

```
typedef struct _FILETIME
{
     DWORD dwLowDateTime;
     DWORD dwHighDateTime;
} FILETIME, FAR *LPFILETIME;
```

Members

dwLowDateTime
Indicates low-order 32 bits of the file time value.

dwHighDateTime
Indicates high-order 32 bits of the file time value.

Remarks

Each MAPI property includes a **FILETIME** structure in its definition in an **SPropValue** structure. A property of type PT_SYSTIME has a FILETIME structure for its value.

The full definition of the FILETIME structure is in the 32-bit Windows reference. MAPI defines this structure in MAPIDEFS.H.

See Also

SPropValue structure

FLAGLIST

The **FLAGLIST** structure contains a list of flags used to indicate the status of address entries during the name resolution process.

```
typedef struct
{
     ULONG cFlags;
     ULONG ulFlags[MAPI_DIM];
} FlagList, FAR * LPFlagList;
```

Members

cFlags
Indicates the number of MAPI-defined flags in the list.

ulFlags
Indicates the first flag in the list.

Remarks

See **IABContainer::ResolveNames** and **IDistList::ResolveNames** for descriptions of the flags that can be part of this list.

The **FLAGLIST** structure is defined in MAPIDEFS.H.

See Also

IABContainer::ResolveNames method

FLATENTRY

The **FLATENTRY** structure contains a "flattened" entry identifier. That is, it holds the actual data of an entry identifier, in contrast to an **ENTRYID** structure, as retrieved from **IMAPIProp**.

```
typedef struct
{
     ULONG   cb;
     BYTE    abEntry[MAPI_DIM];
} FLATENTRY, FAR *LPFLATENTRY;
```

Members

cb
Indicates the size, in bytes, of the data in the **abEntry** member.

abEntry
Indicates the byte array that contains the data of an entry identifier.

Remarks

FLATENTRY is useful because a transport provider can store it in a file or pass it in a stream of bytes. The MAPI SDK sample transport provider contains code that can represent a standard entry identifier in a **FLATENTRY** structure.

Use the **cbFLATENTRY** macro to determine the number of bytes of memory occupied by an existing **FLATENTRY** structure. The syntax is:

int CbFLATENTRY (**LPFLATENTRY** _*lpentry*)

The _*lpentry* parameter points to a **FLATENTRY** structure. This macro returns the number of bytes occupied by the **FLATENTRY** structure pointed to by _*lpentry*.

Use the **CbNewFLATENTRY** macro to determine the memory allocation requirements of a **FLATENTRY** structure containing an entry identifier of a specified byte size. The syntax is:

int CbNewFLATENTRY (**int** _*cb*)

The _*cb* parameter specifies the entry identifier byte size. This macro returns the number of bytes of memory that a **FLATENTRY** with a _*cb* byte entry identifier would occupy.

The **FLATENTRY** structure is defined in MAPIDEFS.H.

See Also

ENTRYID structure

FLATENTRYLIST

The **FLATENTRYLIST** structure contains an array of **FLATENTRY** structures.

```
typedef struct
{
     ULONG     cEntries;
     ULONG     cbEntries;
     BYTE      abEntries[MAPI_DIM];
} FLATENTRYLIST, FAR *LPFLATENTRYLIST;
```

Members

cEntries
Indicates the number of **FLATENTRY** structures in the array described by the **abEntries** member.

cbEntries
Indicates the number of bytes occupied by the entire array described by **abEntries**.

abEntries
Indicates the byte array containing the number of **FLATENTRY** structures designated by the **cEntries** member, arranged end to end.

Remarks

In the **abEntries** array, each **FLATENTRY** structure is aligned on a naturally aligned boundary. Extra bytes are included as padding to ensure natural alignment between any two **FLATENTRY** structures. For example, the second **FLATENTRY** structure starts at an offset made up of the first **FLATENTRY** structure, plus extra bytes as padding to round the space taken up to the next multiple of the natural alignment, plus the offset of the second **FLATENTRY** structure.

The **CbFLATENTRYLIST** macro determines the number of bytes of memory occupied by an existing **FLATENTRYLIST** structure. The syntax is:

int CbFLATENTRYLIST (LPFLATENTRYLIST _*lplist*).

The _*lplist* parameter points to a **FLATENTRYLIST** structure. This macro returns the number of bytes occupied by the **FLATENTRY** structure pointed to by _*lplist*.

The **CbNewFLATENTRYLIST** macro determines the memory allocation requirements of a **FLATENTRYLIST** whose list of **FLATENTRY** structures is of a specified byte size. The syntax is:

int CbNewFLATENTRY (**int** _*cb*)

The _*cb* parameter specifies the byte size of the list of **FLATENTRY** structures. This macro returns the number of bytes in memory that a **FLATENTRYLIST** with a _*cb* byte **FLATENTRY** list would occupy.

The **FLATENTRYLIST** structure is defined in MAPIDEFS.H.

See Also

FLATENTRY structure, PR_REPLY_RECIPIENT_ENTRIES property

FLATMTSIDLIST

The **FLATMTSIDLIST** structure contains an array of **MTSID** structures, each of which contains an X.400 message transport system (MTS) entry identifier.

```
typedef struct
{
     ULONG     cMTSIDs;
     ULONG     cbMTSIDs;
     BYTE      abMTSIDs[MAPI_DIM];
} FLATMTSIDLIST, FAR *LPFLATMTSIDLIST;
```

Members

cMTSIDs
Indicates the number of **MTSID** structures in the array described by the **abMTSIDs** member.

cbMTSIDs
Indicates the number of bytes in the array described by **abMTSIDs**.

abMTSIDs
Indicates a byte array of **MTSID** structures.

Remarks

The **FLATMTSIDLIST** structure's use in X.400 messaging corresponds to the **FLATENTRYLIST** structure's use in MAPI messaging. MAPI uses **FLATMTSIDLIST** structures to maintain X.400 properties during message handling. Service providers use **FLATMTSIDLIST** structures when handling inbound and outbound X.400 messages.

In the **abMTSIDs** array, each **MTSID** structure is aligned on a naturally aligned boundary. Extra bytes are included as padding to ensure natural alignment between any two **MTSID** structures.

The **CbFLATMTSIDLIST** macro determines the number of bytes of memory occupied by a **FLATMTSIDLIST** structure. The syntax is:

int CbFLATMTSIDLIST (**LPFLATMTSIDLIST** *_lplist*)

The *_lplist* parameter is a pointer to a **FLATMTSIDLIST** structure. This macro returns the number of bytes occupied by the **FLATMTSIDLIST** structure pointed to by *_lpentry*.

Use the **CbNewFLATMTSIDLIST** macro to determine the memory allocation requirements of a **FLATMTSIDLIST** structure whose list of **MTSID** structures is of a specified byte size. The syntax is:

int CbNewMTSID (**int** *_cb*)

The *_cb* parameter specifies the byte size of the **MTSID** list. This macro returns the number of bytes of memory occupied by a **FLATMTSIDLIST** with a *_cb* byte **MTSID** list.

The **FLATMTSIDLIST** stucture is defined in MAPIDEFS.H.

See Also

CbNewFLATMTSIDLISTmacro **FLATENTRYLIST** structure, **MTSID** structure

FORMPRINTSETUP

The **FORMPRINTSETUP** structure describes the print setup information for the form object.

```
typedef struct
{
    ULONG         ulFlags;
    HDEVMODE      hDevMode;
    HDEVNAMES     hDevNames;
    ULONG         ulFirstPageNumber;
    ULONG         ulFPrintAttachments;
} FORMPRINTSETUP, FAR * LPFORMPRINTSETUP;
```

Members

ulFlags
Controls the type of the strings. The following flag can be used:

MAPI_UNICODE
Indicates the passed-in strings are in Unicode format. If the MAPI_UNICODE flag is not set, the strings are in ANSI format.

hDevmode
Specifies the mode of the printer.

hDevnames
Specifies the path of the printer.

ulFirstPageNumber
Contains the page number of the first page to be printed.

ulFPrintAttachments
Indicates whether there are attachments to be printed.

Remarks

FORMPRINTSETUP is used to describe the print setup information for the form object. Implementations of **IMAPIViewContext::GetPrintSetup** fill in the **FORMPRINTSETUP** structure and return it in the *lppformprintsetup* output parameter of **GetPrintSetup**.

If the MAPI_UNICODE flag is passed in the *ulflags* parameter of **GetPrintSetup**, the strings in the **hDevmode** and **hDevnames** members should be in Unicode format. Otherwise, the strings should be ANSI format.

The **hDevMode** and **hDevNames** members must be allocated using the Windows function **GlobalAlloc** and must be freed using the Windows function **GlobalFree**. The **FORMPRINTSETUP** structure must be freed by the calling form object using the **MAPIFreeBuffer** function. The application must free the **FORMPRINTSETUP** structure using **MAPIFreeBuffer** and the other two items using **GlobalFree**. Implementations of **IMAPIViewContext** must allocate to match.

The **FORMPRINTSETUP** structure is defined in MAPIFORM.H.

See Also

IMAPIViewContext::GetPrintSetup **MAPIFreeBuffer**

GUID

The **GUID** structure describes a globally unique identifier. **GUID** structures are used, for example, for the **MAPIUID** structures that service providers register when they call **IMAPISupport::SetProviderUID** and for the property set names of named properties.

```
typedef struct _GUID
{
    unsigned long         Data1;
    unsigned short        Data2;
    unsigned short        Data3;
    unsigned char         Data4[8];
} GUID;
```

Members

Data1
Specifies an unsigned long integer data value.

Data2
Specifies an unsigned short integer data value.

Data3
Specifies an unsigned short integer data value.

Data4
Specifies an array of unsigned characters.

Remarks

The **GUID** structure is defined in MAPIGUID.H. For more information about **GUID** structures, see the Remote Procedure Call (RPC) documentation, *OLE Programmer's Reference*, and *Inside OLE, Second Edition.*

See Also

MAPIUID structure

IID

The **IID** structure is a specialized **GUID** structure that identifies a MAPI interface. For more information about IIDs, see the Remote Procedure Call (RPC) documentation, *OLE Programmer's Reference*, and *Inside OLE*, *Second Edition.*

Remarks

The **IID** structure is defined in MAPIGUID.H.

See Also

GUID structure

LARGE_INTEGER

The **LARGE_INTEGER** structure, implemented for all platforms, contains a 64-bit signed integer value.

```
typedef struct _LARGE_INTEGER
{
     DWORD LowPart;
     DWORD HighPart;
} LARGE_INTEGER, FAR *LARGE_INTEGER;
```

Members

LowPart
  Indicates low-order 32 bits of the value.

HighPart
  Indicates high-order 32 bits of the value.

Remarks

A MAPI property includes a **LARGE_INTEGER** structure in its definition in an **SPropValue** structure.

The **Large_Integer** structure is defined in MAPIDEFS.H.

See Also

SPropValue structure

MAPIERROR

The **MAPIERROR** structure provides detailed information about an error, typically generated by the operating system or a service provider.

```
typedef struct _MAPIERROR
{
     ULONG      ulVersion;
     LPTSTR     lpszError;
     LPTSTR     lpszComponent;
     ULONG      ulLowLevelError;
     ULONG      ulContext;

} MAPIERROR, FAR * LPMAPIERROR;
```

Members

ulVersion
Contains the version number of the structure, which must be zero.

lpszError
Points to an error message string.

lpszComponent
Indicates the name of the component that generated the error. Also, this member is used to map the component's Help file. For example:

```
[Help File Mappings]
MAPI 1.0 = mapi.hlp
```

ulLowLevelError
Specifies the low-level error value used only when the error to be returned is low-level.

ulContext
Identifies the error location within the **lpszComponent** member. If there is a Help file for this component, **ulContext** is a Help file context identifier; otherwise, it specifies a unique numeric identifier.

Remarks

The **MAPIERROR** structure is used to provide error information to calling processes. MAPI methods and functions that return error information typically include an *lppMAPIError* parameter in which the **MAPIERROR** structure is placed. To get information on the error, clients call the **IMAPIProp::GetLastError** method of the object that caused the error. Free the memory for the **MAPIERROR** structure call using the **MAPIFreeBuffer** function.

The **ulVersion** member provides for future expansion of the structure; its value must be zero.

The **ulContext** member primarily identifies the point in the code where the error occurred. This assists support personnel and developers in tracking and isolating problems. It can also be used to refer to an online Help topic for common errors.

The **MAPIERROR** structure is defined in MAPIDEFS.H.

See Also

IABLogon::GetLastError, **IMAPISession::GetLastError**, **IProfAdmin::GetLastError**, **IProviderAdmin::GetLastError**, **IMAPISupport::GetLastError**, **IMAPIControl::GetLastError**, **IMAPITable::GetLastError**, **IMAPIProp::GetLastError**, **IMsgServiceAdmin::GetLastError**, **IMSLogon::GetLastError**, **IMSProvider::Logon**, **IABProvider::Logon**, **IMAPISession::OpenAddressBook**, **IMAPISupport::OpenAddressBook**, IMAPIProp::GetLastError

MAPIINIT_0

The **MAPIINIT_0** structure conveys options to **MAPIInitialize**.

```
typedef struct
{
     ULONG ulVersion;
     ULONG ulFlags;
} MAPIINIT_0, FAR *LPMAPIINIT_0;
```

Members

ulVersion
Indicates the integer value of the **MAPIINIT_0** structure version. The following flag can be set:

MAPI_INIT_VERSION
This number should be set to ensure the client uses the correct version of the structure.

ulFlags
Bitmask of flags used to control the format of the data. The following flags can be set:

MAPI_MULTITHREAD_NOTIFICATIONS
When running under Windows NT or Windows 95, the client should set this flag if MAPI notification callbacks should happen on a separate thread. If the flag is not set, notification callbacks will happen on the first thread on which MAPI was initialized, and that thread must have a message loop.

MAPI_NT_SERVICE
When running under Windows NT, the client must set this flag if it is running as a service. Clients that are capable of running either as a service or an application must set the flag when running as a service, and should not set it when running as an application.

Remarks

MAPIInitialize uses this structure in its processing. The **ulVersion** member should be set to MAPI_INIT_VERSION.

The **MAPIINIT_0** structure is defined in MAPIX.H.

See Also

MapiInitialize function

MAPINAMEID

The **MAPINAMEID** structure is used to describe a property name.

```
typedef struct _MAPINAMEID
{
     LPGUID lpguid;
     ULONG  ulKind;
     union
     {
          LONG    lID;
          LPWSTR  lpwstrName;
     } Kind;
} MAPINAMEID, FAR *LPMAPINAMEID;
```

Members

lpguid
Points to a **GUID** structure defining a particular property set; this member cannot be NULL.

ulKind
Specifies a value describing an object described in the **Kind** union.

lID
Specifies a numeric value representing the property name.

lpwstrName
Points to a wide (Unicode) character string representing the property name.

Remarks

The **lpguid** member cannot be NULL; may be PS_PUBLIC_STRINGS, PS_MAPI, or a client-defined value. The **ulkind** member should be set to either MNID_ID if the value of **Kind** is numeric, or set to MNID_STRING if it is a character string.

MAPINAMEID is used to describe the names of named properties, that is, properties that have identifiers over 0x8000. A property set is an integral part of a named property's property name. For example PS_PUBLIC_STRINGS or PS_ROUTING_ADDRTYPE are property sets defined by MAPI.

Property names enable users to define custom properties in a larger name space than the MAPI property identifier. Property names cannot be used to obtain property values directly, they must first be mapped to property identifiers through the **IMAPIProp::GetIDsFromNames** method.

The MAPINAMEID **structure is defined in MAPIDEFS.H.See Also**

GUID structure, **IMAPIProp::GetIDsFromNames** interface

MAPIUID

The **MAPIUID** structure contains a unique identifier (UID) used to identify a particular message conversation or string-to-identifier mapping.

```
typedef struct _MAPIUID
{
     BYTE ab[16];
} MAPIUID, FAR *LPMAPIUID;
```

Members

ab
Specifies an array containing a 16-byte UID.

Remarks

A MAPI unique identifier is a globally unique identifier (GUID) put into Intel® processor byte order. That is, a MAPI unique identifier and a GUID have the same byte order when used on an Intel-processor computer, but on a computer that uses a different byte order – for example, a Motorola®-processor computer – the MAPI UID has the same byte order as on the Intel machine and the GUID uses the byte order specific to the computer.

MAPI creates **MAPIUID** structures in a way that makes it extremely rare for two different items to have the same UID. The client application can store **MAPIUID** structures as binary object properties or as files, without regard for the byte ordering of the computer storing or accessing the information. When the client transmits across a network, it should use a protocol or transmission format that does not change the byte order of **MAPIUID** data.

The most common use of a **MAPIUID** structure by MAPI applications is to define the unique identifier of a profile section. You can define a MAPI UID to identify the profile section your application uses to store configuration information specific to a user's selected profile. The UID can then be formed from the GUID by placing the bytes of the GUID into Intel byte order. For more information on generating a UID, see the *OLE Programmer's Reference.*

The **IsEqualMAPIUID** macro is used by a client application or a service provider to compare two MAPI unique identifiers (UIDs). It evaluates to TRUE if the two UIDs are equal. The syntax is:

BOOL IsEqualMAPIUID(*lpuid1*, *lpuid2***)**

The *lpuid1* and *lpuid2* input parameters specify pointers to a **MAPIUID** structure identifying the first UID and second UID, respectively.

If the client or provider uses this macro, be sure to include MEMORY.H in code. This macro returns TRUE, if the UIDs are equal, and FALSE otherwise.

The **MAPIUID** structure is defined in MAPIDEFS.H.

See Also

GUID structure, **IMAPISession::OpenProfileSection** method **IMAPISupport::NewUID** method

MTSID

The **MTSID** structure contains the actual data of an X.400 message transport system (MTS) entry identifier. Its format contrasts with that of an **ENTRYID** structure, which contains only a pointer to entry identifier data.

```
typedef struct
{
     ULONG          cb;
     BYTE           ab[MAPI_DIM];
} MTSID, FAR *LPMTSID;
```

Members

cb
  Indicates the size, in bytes, of the data in the **abEntry** member.

abEntry
  Indicates a byte array containing the MTS entry identifier data.

Remarks

The **MTSID** structure is used only for X.400 mappings of MAPI entry identifiers. It corresponds to the MAPI **FLATENTRY** structure.

An MTS identifier has the same format as a MAPI entry identifier or a binary property value. MTS identifiers can be particularly useful for canceling deferred messages.

Use the **CbMTSID** macro to determine the number of bytes of memory occupied by a **MTSID** structure. The syntax is:

int CbFLATENTRYLIST (**LPMTSID** *_lpentry*)

The *_lpentry* parameter points to a **MTSID** structure. This macro returns the number of bytes occupied by the **MTSID** structure pointed to by *_lpentry*.

The **CbNewMTSID** macro determines the memory allocation requirements of an **MTSID** structure using a specified number of bytes for its message transfer agent identifier. The syntax is:

int CbNewMTSID (**int** *_cb*)

The value in the *_cb* parameter is the byte size of the identifier, a message transfer agent. This macro returns the number of bytes of memory occupied by an **MTSID** structure with a *_cb* byte message transfer agent identifier.

The **MTSID** structure is defined in MAPIDEFS.H.

See Also

FLATENTRY structure, **FLATMTSIDLIST** structure

NEWMAIL_NOTIFICATION

The **NEWMAIL_NOTIFICATION** structure contains information about a notification event that indicates to an application a new message has arrived.

```
typedef struct _NEWMAIL_NOTIFICATION
{
     ULONG      cbEntryID;
     LPENTRYID  lpEntryID;
     ULONG      cbParentID;
     LPENTRYID  lpParentID;
     ULONG      ulFlags;
     LPTSTR     lpszMessageClass;
     ULONG      ulMessageFlags;
} NEWMAIL_NOTIFICATION;
```

Members

cbEntryID
Indicates the size, in bytes, of the entry identifier of the arriving message.

lpEntryID
Points to the entry identifier of the arriving message.

cbParentID
Indicates the size, in bytes, of the data pointed to by the **lpParentID** member.

lpParentID
Points to the entry identifier of the folder where the arriving message was placed.

ulFlags
Bitmask of flags used to describe the format of the message. The following flag can be set:

MAPI_UNICODE
Indicates the passed-in strings are in Unicode format. If the MAPI_UNICODE flag is not set, the strings are in ANSI format.

lpszMessageClass
Points to the message class of the new message, indicating whether the messaging system is using a Unicode or an ANSI platform.

ulMessageFlags
Specifies a copy of the PR_MESSAGE_FLAGS property for the message, which contains a bitmask of flags used to control the current state of the message object.

Remarks

MAPI uses this structure only as a member of the **NOTIFICATION** structure, which holds information about a notification event for the advise sink.

The **NEWMAIL_NOTIFICATION** structure is defined in MAPIDEFS.H.

See Also

NOTIFICATION structure, PR_MESSAGE_FLAGS property

NOTIFICATION

The **NOTIFICATION** structure contains information about one of several possible notification events for an advise sink.

```
typedef struct
{
     ULONG     ulEventType;
     union
     {
          ERROR_NOTIFICATION          err;
          NEWMAIL_NOTIFICATION        newmail;
          OBJECT_NOTIFICATION         obj;
          TABLE_NOTIFICATION          tab;
          EXTENDED_NOTIFICATION       ext;
          STATUS_OBJECT_NOTIFICATION statobj;
     } info;
} NOTIFICATION, FAR *LPNOTIFICATION;
```

Members

ulEventType

Indicates the type of notification event that occurred. The client application uses this value to locate and read the data of the actual notification in the corresponding data structure within a **NOTIFICATION** structure member. The following table lists the possible types of notification events along with the data structures that hold information for each type of event and the **NOTIFICATION** members in which these data structures can be found.

Notification event type	Corresponding data structure	In member
fnevCriticalError	**ERROR_NOTIFICATION**	**err**
fnevNewMail	**NEWMAIL_NOTIFICATION**	**newmail**
fnevObjectCreated	**OBJECT_NOTIFICATION**	**obj**
fnevObjectDeleted	**OBJECT_NOTIFICATION**	**obj**
fnevObjectModified	**OBJECT_NOTIFICATION**	**obj**
fnevObjectCopied	**OBJECT_NOTIFICATION**	**obj**
fnevObjectMoved	**OBJECT_NOTIFICATION**	**obj**
fnevSearchComplete	**OBJECT_NOTIFICATION**	**obj**
fnevTableModified	**TABLE_NOTIFICATION**	**tab**
fnevStatusObjectModified	**STATUS_OBJECT_NOTIFICATION**	**statobj**

err

ERROR_NOTIFICATION structure containing information about a critical error, if one has occurred.

newmail

NEWMAIL_NOTIFICATION structure containing information about a new message's arrival, if one has arrived.

obj

OBJECT_NOTIFICATION structure containing information about an object's having been created, deleted, modified, copied, or searched, if this has occurred.

tab

TABLE_NOTIFICATION structure containing information about modifications, if any, to a table.

ext

EXTENDED_NOTIFICATION structure containing information for a provider-defined event, if one has occurred.

statobj

STATUS_OBJECT_NOTIFICATION structure containing information about a change in a row of the status table, if a change has occurred.

Remarks

One possible use of the **NOTIFICATION** structure is shown in the following code example. In this example, a test is first performed to determine whether a new mail notification event has occurred. If it has, the message class of the new message is displayed.

```
if (pNotif -> ulEventType == fnevNewMail)
{
     printf("%s\n", pNotif -> newmail.lpszMessageClass)
}
```

The **NOTIFICATION** structure is defined in MAPIDEFS.H.

See Also

ERROR_NOTIFICATION structure, **EXTENDED_NOTIFICATION** structure, **NEWMAIL_NOTIFICATION** structure, **OBJECT_NOTIFICATION** structure, **STATUS_OBJECT_NOTIFICATION** structure, **TABLE_NOTIFICATION** structure

NOTIFKEY

The **NOTIFKEY** structure contains the notification key for the MAPI support object notification methods; service providers create this key and make it available. MAPI uses this key to identify an object globally so that notifications can reach it. A key works across multiple processes.

```
typedef struct
{
     ULONG     cb;
     BYTE      ab[MAPI_DIM];
} NOTIFKEY, FAR *LPNOTIFKEY;
```

Members

cb
Indicates the size, in bytes, of the notification key in the **ab** member.

ab
Specifies the byte array containing the notification key.

Remarks

The interpretation of a notification key depends on the object that it represents. An object's notification key is frequently its entry identifier. However, the key can be a constant, a name, or another such identifying item. In the sample message store, for example, the notification key for a folder directory is its path.

The **NOTIFKEY** structure is defined in MAPISPI.H.

OBJECT_NOTIFICATION

The **OBJECT_NOTIFICATION** structure contains information about a notification event indicating that an object was created, deleted, modified, copied, or searched. MAPI uses this structure only as a member of the **NOTIFICATION** structure for the advise sink.

```
typedef struct _OBJECT_NOTIFICATION
{
     ULONG              cbEntryID;
     LPENTRYID          lpEntryID;
     ULONG              ulObjType;
     ULONG              cbParentID;
     LPENTRYID          lpParentID;
     ULONG              cbOldID;
     LPENTRYID          lpOldID;
     ULONG              cbOldParentID;
     LPENTRYID          lpOldParentID;
     LPSPropTagArray    lpPropTagArray;
} OBJECT_NOTIFICATION;
```

Members

cbEntryID
Indicates the size, in bytes, of the entry identifier of the object that was created or modified.

lpEntryID
Points to the entry identifier of the object.

ulObjType
Indicates the type of object affected. Possible types are:

MAPI_STORE
Indicates a message store.

MAPI_ADDRBOOK
Indicates an address book.

MAPI_FOLDER
Indicates a folder.

MAPI_ABCONT
Indicates an address book container.

MAPI_MESSAGE
Indicates a message.

MAPI_MAILUSER
Indicates a messaging user.

MAPI_ATTACH
Indicates a message attachment.

MAPI_DISTLIST
Indicates a distribution list.

MAPI_PROFSECT
Indicates a profile section.

MAPI_STATUS
Indicates status.

MAPI_SESSION
Specifes which session is in use.

cbParentID
Indicates the size, in bytes, of the data pointed to by the **lpParentID** member.

lpParentID
Points to the entry identifier of the parent of the object.

cbOldID
Indicates the size, in bytes, of the entry identifier for the original object.

lpOldID
Points to the entry identifier of the original object. This pointer is NULL if the object has not changed.

cbOldParentID
Indicates the size, in bytes, of the data pointed to by the **lpOldParentID** member.

lpOldParentID
Points to the entry identifier of the parent of the original object.

lpPropTagArray
Points to an **SPropTagArray** structure containing the tags of object properties affected by the notification event. If the notification is for the creation of a new object, for example, the creation of a new message in a message store, then **lpPropTagArray** should point to an array of properties tags for the properties on the newly created object. If this is unfeasable, MAPI service providers can return NULL instead.

Remarks

The **OBJECT_NOTIFICATION** structure is defined in MAPIDEFS.H.

See Also

NOTIFICATION structure, **SPropTagArray** structure

OPTIONDATA

The **OPTIONDATA** structure contains information specific to an e-mail address type supported by the transport provider. **OPTIONDATA** address information can apply to a message or to a message recipient.

```
typedef struct _OPTIONDATA
{
     ULONG            ulFlags;
     LPGUID           lpRecipGUID;
     LPTSTR           lpszAdrType;
     LPTSTR           lpszDLLName;
     ULONG            ulOrdinal;
     ULONG            cbOptionsData;
     LPBYTE           lpbOptionsData;
     ULONG            cOptionsProps;
     LPSPropValue     lpOptionsProps;
} OPTIONDATA, FAR *LPOPTIONDATA;
```

Members

ulFlags
Bitmask of flags used to specify recipient or message options. The following flags can be set:

MAPI_MESSAGE
Indicates message-specific options.

MAPI_RECIPIENT
Indicates recipient-specific options.

lpRecipGUID
Points to a **GUID** structure identifying the message recipient, if applicable.

lpszAdrType
Points to the e-mail address type that must be NULL.

lpszDLLName
Points to the name of the DLL to be loaded.

ulOrdinal
Specifies which ordinal the transport provider DLL uses to find the client's option callback function.

cbOptionsData
Indicates the size, in bytes, of the data pointed to by the **lpbOptionsData** member.

lpbOptionsData
Points to transport-provider option data that is specific to the recipient or message.

cOptionsProps
Indicates the number of properties in the array pointed to by the **lpOptionsProps** member.

lpOptionsProps
Points to an array of **SPropValue** structures, each containing information about a default option property.

Remarks

The MAPI spooler logs onto a transport provider with the information that this provider can support a number of options specific to different e-mail address types. An option, for example, might indicate the use of a specific message cover page or direct a phone number be redialed a specific number of times. To register options, the MAPI spooler calls the **IXPLogon::RegisterOptions** method provided by the transport provider.

RegisterOptions writes one or two **OPTIONDATA** structures for each supported address type, depending on whether the provider is registered for both recipient and message options, recipient options only, or message options only. If a provider is registered for both option types, **RegisterOptions** writes one structure containing address information for message recipients and one containing address information for messages. For each structure, the **ulFlags** member indicates whether the options apply to a recipient or a message.

For an example of the use of **OPTIONDATA**, consider a transport provider that handles recipients for both Microsoft Mail and Microsoft Mail for the Macintosh. If the provider is registered for both recipient and message options, it provides two pairs of **OPTIONDATA** structures, one pair for each platform. The MAPI spooler can use this information to determine what options are valid for each platform. Once it has this option information, the MAPI spooler prompts the user with a dialog box to retrieve the actual settings for the options.

The DLL name in the *lpszDLLname* parameter should not indicate the operating system platform when **OPTIONDATA** is passed in the *lppOptions* parameter of the **LXPLogon::Register** method.

The **OPTIONDATA** structure is defined in MAPISPI.H.

See Also

IXPLogon::AddressTypes method, **IXPLogon::RegisterOptions** method, **SPropValue** structure

PFNIDLE

PFNIDLE is a pointer type used to declare a MAPI idle function – that is, a function with the prototype **FNIDLE**.

```
typedef FNIDLE     *PFNIDLE;
```

Remarks

Functions based on **FNIDLE** are client application or service provider idle functions that the MAPI idle engine calls periodically according to priority; the specific functionality of such idle functions is defined by the application or provider.

The **PFNIDLE** type is defined in MAPIUTIL.H.

See Also

FNIDLE function prototype

SAndRestriction

The **SAndRestriction** structure contains a group of search restrictions that are combined using a logical AND operation.

```
typedef struct _SAndRestriction
{
     ULONG          cRes;
     LPSRestriction lpRes;
} SAndRestriction;
```

Members

cRes
Indicates the number of search restrictions in the array pointed to by the **lpRes** member.

lpRes
Points to an array of **SRestriction** structures to be combined with a logical AND operation.

Remarks

The **SAndRestriction** structure is defined in MAPIDEFS.H.

See Also

SRestriction structure

SAppTimeArray

The **SAppTimeArray** structure contains a pointer to an array of application time values for use in an **SPropValue** structure containing information about a property.

```
typedef struct _SAppTimeArray
{
     ULONG      cValues;
     double     FAR *lpat;
} SAppTimeArray;
```

Members

cValues
Indicates the number of values in the array pointed to by the **lpat** member.

lpat
Points to an array of application time values.

Remarks

The **SAppTimeArray** structure is defined in MAPIDEFS.H.

SBinary

The **SBinary** structure contains a pointer to a property value of type PT_BINARY for use in an **SPropValue** structure containing information about a property.

```
typedef struct _SBinary
{
     ULONG       cb;
     LPBYTE      lpb;
} SBinary, FAR *LPSBinary;
```

Members

cb
Indicates the size, in bytes, of the data pointed to by the **lpb** member.

lpb
Points to the PT_BINARY property value.

Remarks

The **SBinary** structure is defined in MAPIDEFS.H.

See Also

SPropValue structure

SBinaryArray

The **SBinaryArray** structure contains a property value of type PT_MV_BINARY for use in an **SPropValue** structure containing information about a multivalued property.

```
typedef struct _SBinaryArray
{
     ULONG        cValues;
     SBinary      FAR *lpbin;
} SBinaryArray;
```

Members

cValues
Indicates the number of values in the array pointed to by the **lpbin** member.

lpbin
Points to an array of **SBinary** structures holding the multiple values for the property.

Remarks

The **SBinaryArray** structure is defined in MAPIDEFS.H.

See Also

PT_MV_BINARY property type, **SBinary** structure, **SPropValue** structure

SBitMaskRestriction

The **SBitMaskRestriction** structure contains a bitmask search restriction for objects such as tables and message stores.

```
typedef struct _SBitMaskRestriction
{
     ULONG relBMR;
     PT_LONG ulPropTag;
     ULONG ulMask;
} SBitMaskRestriction;
```

Members

relBMR
Relational operator describing how the mask specified in the ulMask member should be applied to the property tag. Possible values are:

BMR_EQZ
Indicates the value is equal to zero.

BMR_NEZ
Indicates the value is not equal to zero.

ulPropTag
Property tag of the property being masked in the search.

ulMask
Bitmask of flags to compare with the specified property value.

Remarks

This restriction operates by taking the logical AND of **ulMask** with the value of the property indicated by the **ulPropTag** member. If the result is zero, then BMR_EQZ is satisfied. If it's nonzero, that is, if the property value has at least one of the same bits set as **ulMask**, then BMR_NEZ is satisfied.

The result of a property value restriction is undefined when the property does not exist. When a client requires well-defined behavior for such a restriction and is not sure whether the property exists – for example, it is not a required column of a table – it should combine the property restriction with an **SExistsRestriction** in an **SAndRestriction**.

The **SBitmaskRestriction** structure is defined in MAPIDEFS.H.

See Also

SRestriction structure

SCommentRestriction

The **SCommentRestriction** structure contains a comment search restriction for objects such as tables and message stores.

```
typedef struct _SCommentRestriction
{
     ULONG          cValues;
     LPSRestriction lpRes;
     LPSPropValue   lpProp;
} SCommentRestriction;
```

Members

cValues
Indicates the number of values in the array pointed to by the **lpProp** member.

lpRes
Points to an **SRestriction** structure.

lpProp
Points to an array of **SPropValue** structures, each containing information about a property. The method using the **SCommentRestriction** structure ignores the values of the properties when computing the restriction result.

Remarks

The **SCommentRestriction** structure associates an object with a set of named properties. This structure is used by clients that save restrictions on disk to keep application-specific information with the restriction. For example, a client saving the name of a named property used in a property restriction can do so in an **SCommentRestriction** structure. Saving the name is not possible in a property restriction; the **SPropertyRestriction** structure holds only the property tag. Comment restrictions are ignored by the **IMAPITable::Restrict** method. There is no effect on the rows returned by the **IMAPITable::QueryRows** method after an **IMAPITable::Restrict** call has been made.

The **SCommentsRestriction** structure is defined in MAPIDEFS.H.

See Also

SPropValue structure, **SRestriction** structure

SComparePropsRestriction

The **SComparePropsRestriction** structure contains a search restriction that compares properties for objects such as tables and message stores.

```
typedef struct _SComparePropsRestriction
{
     ULONG relop;
     ULONG ulPropTag1;
     ULONG ulPropTag2;
} SComparePropsRestriction;
```

Members

relop
Relational operator to use in the property comparison restriction. Possible values are:

RELOP_GE
Indicates the comparison is made based on a greater or equal first value.

RELOP_GT
Indicates the comparison is made based on a greater first value.

RELOP_LE
Indicates the comparison is made based on a lesser or equal first value.

RELOP_LT
Indicates the comparison is made based on a lesser first value.

RELOP_NE
Indicates the comparison is made based on unequal values.

RELOP_RE
Indicates the comparison is made based on LIKE (regular expression) values.

RELOP_EQ
Indicates the comparison is made based on equal values.

ulPropTag1
Specifies the property tag of the first property to be compared.

ulPropTag2
Specifies the property tag of the second property to be compared.

Remarks

The comparison order is *(property tag 1) (relational operator) (property tag 2)*. The two properties compared should be of the same type. Attempting to compare properties of different types commonly returns the error value MAPI_E_TOO_COMPLEX.

The result of a property value restriction is undefined when the property does not exist. When a client requires well-defined behavior for such a restriction and is not sure whether the property exists (for example, it is not a required column of a table), it should combine the property restriction with an **SExistsRestriction** in an **SAndRestriction**.

The MV_FLAG is combined, using the OR operation, with the type portion of the property tags to make the property accommodate an array of the base type. MV_FLAG is not actually a flag; it is a combination of the MV_FLAG and MV_INSTANCE.

If supported, property tags with MV_FLAG can be used anywhere single valued property tags can be used. They can be used in **IMAPIProp::SetProps**, **IMAPIProp::GetProps**, **IMAPITable::SetColumns**, **IMAPITable::SortTable**, and **IMAPITable::Restrict**.

Property tags with MVI_FLAG are only used on tables and have special semantics.They can be used as input to **IMAPITable::SetColumns** in the **SPropTagArray** structure, **IMAPITable::SortTable** in the **SSortOrderSet** structure and **IMAPITable::Restrict** in the **SRestriction** structure's **ulPropTag** member. They are never found in **SPropValue** structures or in the output parameters of methods other than **IMAPITable::QueryColumns** and **IMAPITable::QuerySortOrder**.

Columns with MVI_FLAG request the provider to return that column as single value properties, with one row per instance of the MV_FLAG property that is stored in the object. The property tags in **SPropValues** returned by **IMAPITable::QueryRows** are single valued for that column. For example, if you ask for PR_FOO you will get PR_FOO and ~MVI_FLAG in the **ulPropTag** member of the **SPropValue** structure.

SortOrderSet works identically to the **IMAPITable::SetColumns** behavior. Sorts are done based on the single values in the instances, and rows are added based on the expansion of each object's MVI_FLAG columns in both the **SSortOrder** and **SPropTagArray** structures.

Unlike the MVI_FLAG in **SSortOrder** and **SPropTagArray**, MVI_FLAG in **SRestriction** does not expand the objects into computed rows. Rather, given that a column has been set by **IMAPITable::SetColumns** or **IMAPITable::SortTable** to be instances of the underlying multivalued property, putting a property tag with the MVI_FLAG in **SRestriction** tells the provider to use that column in restricting the table. The **SPropValue** structure (if any) to restrict against must be a single valued property tag identical to the one that would be returned by **IMAPITable::QueryRows** for the column.

The **SComparePropsRestriction** structure is defined in MAPIDEFS.H.

See Also

SBitMaskRestriction structure, **SRestriction** structure

SContentRestriction

The **SContentRestriction** structure contains a search restriction to limit a view of the messages in a table based on checking for a search string.

```
typedef struct _SContentRestriction
{
     ULONG        ulFuzzyLevel;
     ULONG        ulPropTag;
     LPSPropValue lpProp;
} SContentRestriction;
```

Members

ulFuzzyLevel
Option settings defining the fuzzy level for a message contents search. The lower 16 bits apply to property types PT_STRING8 and PT_BINARY. They contain the substring alignment code, which must be set to exactly one of the following values:

FL_FULLSTRING
Indicates that the **lpProp** search string must be contained as a full string in the message property string – that is, the two strings must be identical.

FL_PREFIX
Indicates that the **lpProp** search string must be contained as a substring at the beginning of the message property string. The two strings should be compared only up to the length of the search string indicated by **lpProp**.

FL_SUBSTRING
Indicates that the **lpProp** search string must be contained as a substring anywhere within the message property string.

The upper 16 bits of the fuzzy level apply only to property type PT_STRING8. They are a bitmask of different options for comparing character strings within messages. The following flags can be set in any combination:

FL_IGNORECASE
Indicates that the function should make the comparison in case-insensitive fashion.

FL_IGNORENONSPACE
Indicates that the function should make the comparison so as to ignore Unicode-defined "nonspacing characters," for example, diacritical marks.

FL_LOOSE
Indicates that the service provider should perform as many fuzzy level heuristics of types FL_IGNORECASE and FL_IGNORENONSPACE as it has been designed to handle.

ulPropTag
Property tag identifying the property (string) in each message to be checked for occurrence of the search string.

lpProp
Pointer to an **SPropValue** structure containing the search string to be checked for in each message.

Remarks

The **SPropValue** structure pointed to by **lpProp** also contains a **ulPropTag** member. In both tags, MAPI requires only the property type field and ignores the property identifier field. However, the two property types must match, or else the error value MAPI_E_TOO_COMPLEX is returned.

The codes FL_FULLSTRING, FL_PREFIX, and FL_SUBSTRING are mutually exclusive. Only one of them can be set, and one of them must be set. Their meanings are fixed, and the provider must implement them exactly as defined. The provider should return MAPI_E_TOO_COMPLEX if it is unable to implement a specified code.

The flags FL_IGNORECASE, FL_IGNORENONSPACE, and FL_LOOSE are independent. Anywhere from zero to all three of them can be set. Their definitions are provided as a guideline only, and the provider is free to implement its own specific meaning of each flag. The provider should not return any error indication if it has no implementation of a specified flag.

The result of a property value restriction is undefined when the property does not exist. When a client requires well-defined behavior for such a restriction and is not sure whether the property exists (for example, it is not a required column of a table), it should combine the property restriction with an **SExistsRestriction** in an **SAndRestriction**.

The MV_FLAG is combined using the OR operation to the type portion of the property tags to make the property accommodate an array of the base type. MVI_FLAG is not actually a flag, but instead a combination of the MV_FLAG and MV_INSTANCE.

If supported, property tags with MV_FLAG can be used anywhere single valued property tags can be used. They can be used in **IMAPIProp::SetProps**, **IMAPIProp::GetProps**, **IMAPITable::SetColumns**, **IMAPITable::SortTable**, and **IMAPITable::Restrict**.

Property tags with MVI_FLAG are only used on tables and have special semantics.They can be used as input to **IMAPITable::SetColumns** in the **SPropTagArray** structure, **IMAPITable::SortTable** in the **SSortOrderSet** structure and **IMAPITable::Restrict** in the **SRestriction** structure's **ulPropTag** member. They are never found in **SPropValue** or in the output parameters of methods other than **IMAPITable::QueryColumns** and **IMAPITable::QuerySortOrder**.

Columns with MVI_FLAG request the provider to return that column as single value properties, with one row per instance of the MV_FLAG property that is stored in the object. The property tags in **SPropValues** returned by **IMAPITable::QueryRows** are single valued for that column. For example, if you ask for PR_FOO you will get PR_FOO and ~MVI_FLAG in the **ulPropTag** field of the **SPropValue** structure.

SortOrderSet works identically to the **IMAPITable::SetColumns** behavior. Sorts are done based on the single values in the instances, and rows are added based on the expansion of each object's MVI_FLAG columns in both the **SSortOrder** and **SPropTagArray** structures.

Unlike the MVI_FLAG in **SSortOrder** and **SPropTagArray**, MVI_FLAG in **SRestriction** does not expand the objects into computed rows. Rather, given that a column has been set by **IMAPITable::SetColumns** or **IMAPITable::SortTable** to be instances of the underlying multivalued property, putting a property tag with the MVI_FLAG in the **SRestriction** structure tells the provider to use that column in restricting the table. The **SPropValue** structure (if any) to restrict against must be a single valued property tag identical to the one that would be returned by **IMAPITable::QueryRows** for the column.

The **SContentRestriction** structure is defined in MAPIDEFS.H.

See Also

SPropValue structure, **SRestriction** structure

SCurrencyArray

The **SCurrencyArray** structure contains a property value of type PT_MV_CURRENCY for use in an **SPropValue** structure containing information about a multivalued property.

```
typedef struct _SCurrencyArray
{
     ULONG         cValues;
     CURRENCY FAR *lpcur;
} SCurrencyArray;
```

Members

cValues
Indicates the number of values in the array pointed to by the **lpcur** member.

lpcur
Points to an array of **CURRENCY** structures containing the multiple values for the property.

Remarks

The **SCurrencyArray** structure is defined in MAPIDEFS.H.

See Also

PT_MV_CURRENCY property type, **SPropValue** structure

SDateTimeArray

The **SDateTimeArray** structure contains a property value of type PT_MV_SYSTIME for use in an **SPropValue** structure containing information about a multivalued property.

```
typedef struct _SDateTimeArray
{
     ULONG          cValues;
     FILETIME FAR *lpft;
} SDateTimeArray;
```

Members

cValues
Indicates the number of values in the array pointed to by the **lpft** member.

lpft
Points to an array of **FILETIME** structures containing the multiple values for the property.

Remarks

The **SDateTimeArray** structure is defined in MAPIDEFS.H.

See Also

FILETIME structure, PT_MV_SYSTIME property type, **SPropValue** structure

SDoubleArray

The **SDoubleArray** structure contains a property value of type PT_MV_DOUBLE for use in an **SPropValue** structure containing information about a multivalued property.

```
typedef struct _SDoubleArray
{
     ULONG           cValues;
     double     FAR *lpdbl;
} SDoubleArray;
```

Members

cValues
Indicates the number of values in the array pointed to by the **lpdbl** member.

lpdbl
Points to the array of double values making up the property.

Remarks

The **SDoubleArray** structure is defined in MAPIDEFS.H.

See Also

PT_MV_DOUBLE property type, **SPropValue** structure

SExistRestriction

The **SExistRestriction** structure contains a search restriction to limit a view of the messages in a table based on the existence of a particular property.

```
typedef struct _SExistRestriction
{
     ULONG ulReserved1;
     ULONG ulPropTag;
     ULONG ulReserved2;
} SExistRestriction;
```

Members

ulReserved1
Reserved; must be zero.

ulPropTag
Property tag identifying the property in each message to be tested for existence.

ulReserved2
Reserved; must be zero.

Remarks

For meaningful results, restrictions on properties that are not required columns in a table should be ANDed with **SExistRestriction** structures on those properties. The provider is not obligated to return consistent or predictable results when a non-existent property is tested in a restriction.

Since PR_MESSAGE_ATTACHMENTS and PR_MESSAGE_RECIPIENTS are subobject properties, a restriction on them using **SExistRestriction** does not produce reliable results.

The **SExistRestriction** structure is defined in MAPIDEFS.H.

See Also

SRestriction structure

SGuidArray

The **SGuidArray** structure contains a property value of type PT_MV_CLSID for use in an **SPropValue** structure containing information about a multivalued property.

```
typedef struct _SGuidArray
{
     ULONG     cValues;
     GUID      FAR *lpguid;
} SGuidArray;
```

Members

cValues
Indicates the number of values in the array pointed to by the **lpguid** member.

lpguid
Points to an array of property values for use in the **SPropValue** structure.

Remarks

The **SGUIDArray** structure is defined in MAPIDEFS.H.

See Also

GUID structure, PT_MV_CLSID property type, **SPropValue** structure

SLargeIntegerArray

The **SLargeIntegerArray** structure contains a property value of type PT_MV_I8 for use in an **SPropValue** structure containing information about a multivalued property.

```
typedef struct _SLargeIntegerArray
{
     ULONG     cValues;
     LARGE_INTEGER      FAR *lpli;
} SLargeIntegerArray;
```

Members

cValues
Indicates the number of values in the array pointed to by the **lpli** member.

lpli
Points to an array of **LARGE_INTEGER** structures holding the multiple values for the property.

Remarks

The **SLargeIntegerArray** structure is defined in MAPIDEFS.H.

See Also

LARGE_INTEGER structure, PT_MV_I8 property type, **SPropValue** structure

SLongArray

The **SLongArray** structure contains a property value of type PT_MV_LONG for use in an **SPropValue** structure containing information about a multivalued property.

```
typedef struct _SLongArray
{
     ULONG     cValues;
     LONG      FAR *lpl;
} SLongArray;
```

Members

cValues
Indicates the number of values in the array pointed to by the **lpl** member.

lpl
Points to the array of long values making up the property.

Remarks

The **SLongArray** structure is defined in MAPIDEFS.H.

See Also

PT_MV_LONG property type, **SPropValue** structure

SLPSTRArray

The **SLPSTRArray** structure contains a property value of type PT_MV_STRING8 for use in an **SPropValue** structure containing information about a multivalued property.

```
typedef struct _SLPSTRArray
{
     ULONG      cValues;
     LPSTR FAR *lppszA;z
} SLPSTRArray;
```

Members

cValues
Indicates the number of values in the array pointed to by the **lppszA** member.

lppszA
Points to the array of null-terminated 8-bit character strings making up the property.

Remarks

The **SLPSTRArray** structure is defined in MAPIDEFS.H.

See Also

PT_MV_STRING8 property type, **SPropValue** structure

SMAPIFormInfoArray

The **SMAPIFormInfoArray** structure contains a list of pointers to **IMAPIFormInfo** interfaces.

```
typedef struct
{
     ULONG          cForms;
     LPMAPIFORMINFO aFormInfo[MAPI_DIM];
} SMAPIFormInfoArray, FAR * LPSMAPIFORMINFOARRAY;
```

Members

cForms
Indicates the number of **IMAPIFormInfo** interface pointers in the array in the **aFormInfo** member.

aFormInfo
Holds an array of pointers to **IMAPIFormInfo** interface implementations.

Remarks

This structure is passed as a parameter in the **IMAPIFormMgr** methods **ResolveMultipleMessageClasses**, **CalcFormPropSet**, **SelectMultipleForms** and the **IMAPIFormContainer** method **ResolveMultipleMessageClasses**.

Use the **CbMAPIFormInfoArray** macro to determine the minimum number of bytes required to hold an **SMAPIFormInfoArray** containing a specified number of **IMAPIFormInfo** interface pointers. The syntax is:

int CbMAPIFormInfoArray (**int** _*c*)

The _*c* parameter specifies the number of **IMAPIFormInfo** interface pointers. This macro returns the number of bytes of memory needed to hold a **SMAPIFormInfoArray** containing the number of interface pointers specified by _*c*.

The **SMAPIFormInfoArray** structure is defined in MAPIFORM.H.

See Also

IMAPIFormContainer::ResolveMultipleMessageClasses method, **IMAPIFormMgr::CalcFormPropSet** method, **IMAPIFormMgr::ResolveMultipleMessageClasses** method, **IMAPIFormMgr::SelectMultipleForms** method

SMAPIFormProp

The **SMAPIFormProp** structure contains a form property used with form interfaces.

```
typedef struct _SMAPIFormProp
{
     ULONG                    ulFlags;
     ULONG                    nPropType;
     MAPINAMEID               nmid;
     LPTSTR                   pszDisplayName;
     FORMPROPSPECIALTYPE      nSpecialType;
     union
     {
          struct
          {
          MAPINAMEID                  nmidIdx;
          ULONG                       cfpevAvailable;
          LPMAPIFormPropEnumVal       pfpevAvailable;
          } s1;
     } u;
} SMAPIFormProp;
```

Members

ulFlags
Contains MAPI_UNICODE if the strings in the structure are Unicode; zero if they are not.

nPropType
Property type of the form property, with the most significant word equal to zero.

nmid
MAPINAMEID structure containing the property's globally unique identifier (**GUID**) and a form kind, made up of an interface identifier and the form's name.

pszDisplayName
Points to the display name of the property.

nSpecialType
Indicates the special type tag defined in the **FORMPROPSPECIALTYPE** enumeration. This tag applies to the **u** union.

nmidIdx
Indicates the **MAPINAMEID** structure containing the identifier for the interface implementing the property.

cfpevAvailable
Indicates the number of **SMAPIFormPropEnumVal** structures in the array pointed to by the **pfpevAvailable** member.

pfpevAvailable
Points to an array of **SMAPIFormPropEnumVal** structures, each of which holds a value for a form property.

Remarks

SMAPIFormProp contains information about a form property used as part of the definitions of the **IMAPIFormInfo** interface; **nSpecialType** contains a tag that applies to the **u** union that is part of **SMAPIFormProp**.

The **FORMPROPSPECIALTYPE** enumeration lists possible special type tags for the **nSpecialType** member of the **SMAPIFormProp** structure. In the **FORMPROPSPECIALTYPE** enumeration, the

FPST_VANILLA member indicates that no enumeration is used, and the **FPST_ENUM_PROP** member indicates the enumeration contains a MAPI property.

The **SMAPIFormProp** structure is defined in MAPIDEFS.H.

See Also

MAPINAMEID structure, **SMAPIFormPropEnumVal** structure

SMAPIFormPropArray

The **SMAPIFormPropArray** structure contains a list of form properties, that is, **SMAPIFormProp** structures.

```
typedef struct
{
     ULONG         cProps;
     ULONG         ulPad;
     SMAPIFormProp aFormProp[MAPI_DIM];
} SMAPIFormPropArray, FAR * LPMAPIFORMPROPARRAY;
```

Members

cProps
Indicates the number of verbs in the list.

ulPad
Indicates an eight-byte pad used for alignment of the **SMAPIFormPropArray**.

aFormProp
Indicates the name of the first form property in the list.

Remarks

This structure is passed as a parameter in the methods **IMAPIFormInfo::CalcFormPropSet**, **IMAPIFormMgr::CalcFormPropSet**, **IMAPIFormContainer::CalcFormPropSet**.

Use the **CbMAPIFormPropArray** macro to determine the memory allocation requirements for an **SMAPIFormPropArray** containing a specified number of form properties, that is a specified number of **SMAPIFormProp** structures. The syntax is:

int CbMAPIFormPropArray (**int** *_c*)

The *_c* parameter specifies the number of form properties. This macro returns the number of bytes of memory needed to hold an **SMAPIFormPropArray** containing the number of form properties specified by *_c*.

The **SMAPIFormPropArray** structure is defined in MAPIFORM.H.

See Also

CbMAPIFormPropArray macro, **IMAPIFormContainer::CalcFormPropSet** method, **IMAPIFormInfo::CalcFormPropSet** method, **IMAPIFormMgr::CalcFormPropSet** method, **SMAPIFormProp** structure

SMAPIFormPropEnumVal

The **SMAPIFormPropEnumVal** structure contains a value of a form property that will be used as part of the **SMAPIFormProp** structure.

```
typedef struct _SMAPIFormPropEnumVal
{
     SPropValue     val;
     ULONG          nVal;
} SMAPIFormPropEnumVal;
```

Members

val
Specifies an **SPropValue** structure containing information about the form property.

nVal
Specifies an enumeration value for the structure in the **val** member.

Remarks

The **SMAPIFormProp** structure contains form property information used in forms definitions.

The **SMAPIFormPropEnumVal** structure is defined in MAPIDEFS.H.

See Also

SMAPIFormProp structure, **SPropValue** structure

SMAPIVerb

The **SMAPIVerb** structure contains arrays of MAPI verbs.

```
typedef struct
{
     LONG    lVerb;
     LPTSTR  szVerbname;
     DWORD   fuFlags;
     DWORD   grfAttribs;
     ULONG   ulFlags;                 /* Either 0 or MAPI_UNICODE */
} SMAPIVerb, FAR * LPMAPIVERB;
```

Members

lVerb
Numeric value for the verb.

szVerbname
Character string containing the name of the verb.

fuFlags
Flags for the verb.

grfAttribs
Attributes of the verb.

ulFlags
Bitmask of flags. If the **szVerbname** member has Unicode format, the following flag can be set:

MAPI_UNICODE
Indicates the passed-in strings are in Unicode format. If the MAPI_UNICODE flag is not set, the strings are in ANSI format.

Remarks

This structure is passed as a parameter in the **IMAPIFormMgr** and **IMAPIFormContainer** method **ResolveMultipleMessageClasses**.

The **SMAPIVerb** structure is defined in MAPIFORM.H.

See Also

CbMessageClassArray macro, **IMAPIFormContainer::ResolveMultipleMessageClasses** method, **IMAPIFormMgr::ResolveMultipleMessageClasses** method

SMAPIVerbArray

The **SMAPIVerbArray** structure contains a list of MAPI verbs, that is **SMAPIVerb** structures.

```
typedef struct
{
     ULONG     cMAPIVerb;
     SMAPIVerb aMAPIVerb[MAPI_DIM];
} SMAPIVerbArray, FAR * LPMAPIVERBARRAY;
```

Members

cForms
Indicates the number of verbs in the list.

aFormInfo
Names the first verb in the list.

Remarks

This structure is passed as a parameter in the **IMAPIFormInfo::CalcVerbSet** method.

Use the **cbMAPIVerbArray** macro to determine the memory allocation requirements of an **SMAPIVerbArray** containing a specified number of verbs, that is a specified number of **SMAPIVerb** structures. The syntax is:

int CbMAPIVerbArray (**int** _*c*)

The _*c* parameter specifies the number of verbs. This macro returns the number of bytes of memory needed to hold an **SMAPIVerbArray** containing the number of verbs specified by _*c*.

The **SMAPIVerbArray** structure is defined in MAPIFORM.H.

See Also

IMAPIFormInfo::CalcVerbSet method, **SMAPIVerb** structure

SMessageClassArray

The **SMessageClassArray** structure contains a list of message class string pointers.

```
typedef struct
{
     ULONG   cValues;
     LPCSTR  aMessageClass[MAPI_DIM];
} SMessageClassArray, FAR * LPSMESSAGECLASSARRAY;
```

Members

cValues
Indicates the number of message class string pointers in this structure.

aMessageClass
Names the first pointer in the list of message class string pointers.

Remarks

This structure is passed as a parameter in the **IMAPIFormMgr** and **IMAPIFormContainer** method **ResolveMultipleMessageClasses**.

Use the **cbMessageClassArray** macro to determine the minimum number of bytes required to hold an **SMessageClassArray** containing a specified number of message class string pointers. The syntax is:

int CbMessageClassArray (**int** _*c*)

The _*c* parameter specifies the number of message class string pointers. This macro returns the number of bytes of memory needed to hold a **SMessageClassArray** containing the number of pointers specified by _*c*.

The **SMessageClassArray** structure is defined in MAPIDEFS.H.

See Also

IMAPIFormContainer::ResolveMultipleMessageClasses method,
IMAPIFormMgr::ResolveMultipleMessageClasses method

SNotRestriction

The **SNotRestriction** structure contains a group of search restrictions to which a logical NOT operation has been applied.

```
typedef struct _SNotRestriction
{
     ULONG           ulReserved;
     LPSRestriction  lpRes;
} SNotRestriction;
```

Members

ulReserved
Reserved; must be zero.

lpRes
Points to an **SRestriction** structure containing the restrictions to be included in the logical NOT operation.

Remarks

The **SNotRestriction** structure is defined in MAPIDEFS.H.

See Also

SRestriction structure

SOrRestriction

The **SOrRestriction** structure contains a group of search restrictions combined in a logical OR operation.

```
typedef struct _SOrRestriction
{
     ULONG           cRes;
     LPSRestriction  lpRes;
} SOrRestriction;
```

Members

cRes
Indicates the number of structures in the array pointed to by the **lpRes** member.

lpRes
Points to an array of **SRestriction** structures defining the restrictions to be combined using the logical OR operation.

Remarks

The **SOrRestriction** structure is defined in MAPIDEFS.H.

See Also

SRestriction structure

SPropAttrArray

The **SPropAttrArray** structure contains a list of attributes for a MAPI property.

```
typedef struct
{
     ULONG     cValues;
     ULONG     aPropAttr[MAPI_DIM];
} SPropAttrArray, FAR *LPSPropAttrArray;
```

Members

cValues
Indicates the number of property attributes in the array in the **aPropAttr** member.

aPropAttr
Contains an array of property attributes. Attributes that can be placed in this list are:

PROPATTR_MANDATORY
PROPATTR_READABLE
PROPATTR_WRITEABLE
PROPATTR_NOT_PRESENT

Remarks

Use the **CbSPropAttrArray** macro to determine the number of bytes occupied by an existing **SPropAttrArray** structure. The syntax is:

int CbSPropAttrArray (**LPSPROPATTRARRAY** _*lparray*)

The _*lparray* parameter points to an **SPropAttrArray** structure. This macro returns the number of bytes of memory occupied by the **SPropAttrArray** structure.

Use the **CbNewSPropAttrArray** macro to determine the memory allocation requirement of an **SPropAttrArray** structure containing a specified number of property attributes. The syntax is:

int CbNewSPropAttrArray (**int** _*c*)

The _*c* parameter specifies the number of property attributes. This macro returns the number of bytes of memory needed to hold an **SPropAttrArray** containing the number of property attributes specified by _*c*.

The **SPropAttrArray** structure is defined in IMESSAGE.H.

SPropertyRestriction

The **SPropertyRestriction** structure contains a search restriction to limit a view of the messages in a table based on comparing a property's value to a constant.

```
typedef struct _SPropertyRestriction
{
     ULONG        relop;
     ULONG        ulPropTag;
     LPSPropValue lpProp;
} SPropertyRestriction;
```

Members

relop
Indicates the relational operator to be used in the search. Possible values are:

RELOP_GE
Indicates the comparison is made based on a greater or equal first value.

RELOP_GT
Indicates the comparison is made based on a greater first value.

RELOP_LE
Indicates the comparison is made based on a lesser or equal first value.

RELOP_LT
Indicates the comparison is made based on a lesser first value.

RELOP_NE
Indicates the comparison is made based on unequal values.

RELOP_RE
Indicates the comparison is made based on LIKE (regular expression) values.

RELOP_EQ
Indicates the comparison is made based on equal values.

ulPropTag
Indicates a property tag identifying the property in each message to be compared to the constant.

lpProp
Points to an **SPropValue** structure containing the constant value to be compared against.

Remarks

The **SPropValue** structure pointed to by **lpProp** also contains a **ulPropTag** member. In both tags, MAPI requires only the property type field and ignores the property identifier field. However, the two property types must match, or else the error value MAPI_E_TOO_COMPLEX is returned.

The comparison order is *(property value) (relational operator) (constant value)*.

The result of a property value restriction is undefined when the property does not exist. When a client requires well-defined behavior for such a restriction and is not sure whether the property exists (for example, it is not a required column of a table), it should combine the property restriction with an **SExistsRestriction** in an **SAndRestriction**.

The MV_FLAG is combined using the OR operation to the type portion of the property tags to make the property accommodate an array of the base type. MVI_FLAG is not actually a flag, but instead a combination of the MV_FLAG and MV_INSTANCE.

If supported, property tags with MV_FLAG can be used anywhere single valued property tags can be used. They can be used in **IMAPIProp::SetProps**, **IMAPIProp::GetProps**, **IMAPITable::SetColumns**, **IMAPITable::SortTable**, **IMAPITable::Restrict**.

Property tags with MVI_FLAG are only used on tables and have special semantics.They can be used as input to **IMAPITable::SetColumns** in the **SPropTagArray** structure, **IMAPITable::SortTable** in the **SSortOrderSet** structure and **IMAPITable::Restrict** in the **SRestriction** structure's **ulPropTag** member. They are never found in **SPropValue** or in the output parameters of methods other than **IMAPITable::QueryColumns** and **IMAPITable::QuerySortOrder**.

Columns with MVI_FLAG request the provider to return that column as single value properties, with one row per instance of the MV_FLAG property that is stored in the object. The property tags in **SPropValues** returned by **IMAPITable::QueryRows** are single valued for that column. For example, if you ask for PR_FOO you will get PR_FOO and ~MVI_FLAG in the **ulPropTag** member of the **SPropValue** structure.

SortOrderSet works identically to the **IMAPITable::SetColumns** behavior. Sorts are done based on the single values in the instances, and rows are added based on the expansion of each object's MVI_FLAG columns in both the **SSortOrder** and **SPropTagArray** structures.

Unlike MVI_FLAG in **SSortOrder** and **SPropTagArray**, MVI_FLAG in **SRestriction** does not expand the objects into computed rows. Rather, given that a column has been set by **IMAPITable::SetColumns** or **IMAPITable::SortTable** to be instances of the underlying multivalued property, putting a property tag with the MVI_FLAG in the **SRestriction** structure tells the provider to use that column in restricting the table. The **SPropValue** structure (if any) to restrict against must be a single valued property tag identical to the one that would be returned by **IMAPITable::QueryRows** for the column.

The **SPropertyRestriction** structure is defined in MAPIDEFS.H.

See Also

SPropValue structure, **SRestriction** structure

SPropProblem

The **SPropProblem** structure describes an error relating to a particular property.

```
typedef struct _SPropProblem
{
     ULONG     ulIndex;
     ULONG     ulPropTag;
     SCODE     scode;
} SPropProblem, FAR *LPSPropProblem;
```

Members

ulIndex
Indicates an index in an array of property tags.

ulPropTag
Indicates a property tag for the property.

scode
Indicates an error value indicating the property problem encountered during the update. This value can be any **SCODE** value.

Remarks

The **SPropProblem** structure is defined in MAPIDEFS.H. An **SPropProblem** structure contains an **SCODE** error value that is a result of an operation attempting to modify or delete a MAPI property.

See Also

SCODE data type, **SPropProblemArray** structure

SPropProblemArray

The **SPropProblemArray** structure contains an array of one or more **SPropProblem** structures, each of which holds an SCODE error value that is a result of an operation attempting to modify or delete a MAPI property.

```
typedef struct _SPropProblemArray
{
     ULONG          cProblem;
     SPropProblem   aProblem[MAPI_DIM];
} SPropProblemArray, FAR *LPSPropProblemArray;
```

Members

cProblem
Indicates the number of problem structures in the array indicated by the **aProblem** member.

aProblem
Contains an array of **SPropProblem** structures, each holding a property error.

Remarks

Use the **CbSPropProblemArray** to determine the number of bytes in an existing **SPropProblemArray**.

The syntax is:

int CbSPropProblemArray (**LPSPropProblemArray** *_lparray*)

The *_lparray* parameter specifies a pointer to an **SPropProblemArray** structure. This macro returns the number of bytes of memory in the **SPropProblemArray** structure pointed to by *_lparray*.

The **CbNewSPropProblemArray** macro determines the memory allocation requirements of a **SPropProblemArray** structure containing a specified number of **SPropProblem** structures.

The syntax is:

int CbNewSPropProblemArray (**int**_*cprob*)

The *_cprob* parameter specifies the number of **SPropProblem** structures. This macro returns the number of bytes of memory occupied by an **SPropProblemArray** structure that contains the number of **SPropProblem** structures specified by *_cprob*.

The **SizedSPropProblemArray** creates a structure definition identical to that of **SPropProblemArray** but with a specified number of **SPropProblem** structures. Use the **SizedSPropProblemArray** macro to create property problem arrays with explicit bounds. The syntax is:

The syntax is: **SizedSPropProblemArray** (**int**_*cprob*, *_name*)

The *_cprob* parameter specifies the number of **SPropProblem** structures to be in the property problem array, aProblem. The structure type is defined with the **_SPropProblemArray_** *_name* and type name *_name*.

The **SPropProblemArray** structure is defined in MAPIDEFS.H.

See Also

SCODE data type, **SPropProblem** structure

SPropTagArray

The **SPropTagArray** structure contains an array of property tags.

```
typedef struct _SPropTagArray
{
     ULONG      cValues;
     ULONG      aulPropTag[MAPI_DIM];
} SPropTagArray, FAR *LPSPropTagArray;
```

Members

cValues
Indicates the number of property tags in the array indicated by the **aulPropTag** member.

aulPropTag
Indicates an array of property tags.

Remarks

The **CbSPropTagArray** macro determines the number of bytes occupied by an existing **SPropTagArray**. The syntax is:

int CbSPropTagArray (**LPSPropTagArray** *_lparray*)

The *_lparray* parameter specifies a pointer to an **SPropTagArray** structure. This macro returns the number of bytes of memory occupied by the **SPropTagArray** structure pointed to by *_lparray*.

The **CbNewSPropTagArray** macro determines the memory allocation requirements of an **SPropTagArray** structure containing a specified number of property tags. The syntax is:

int CbNewSPropTagArray (**int** *_ctag*)

The *_ctag* parameter specifies the number of property tags. This macro returns the number of bytes of memory occupied by an **SPropTagArray** structure that contains the number of property tags specified by *_ctag*.

The **SizedSPropTagArray** macro creates a structure definition identical to that of **SPropTagArray** but with a specified number of property tags. Use the **SizedSPropTagArray** macro to create property tag arrays with explicit bounds. The syntax is:

SizedSPropTagArray (**int** *_ctag*, *_name*)

The *_ctag* parameter specifies the number of property tags to be in the property tag array **aulPropTag**. The structure type is defined with the tag _SPropTagArray_ *_name* and type name *_name*.

To use *lpSizedSPropTagArray*, a sized property tag pointer, with a sized property tag in any function call or structure that expects a **LPSPropTagArray** pointer, perform the following cast:

```
lpSPropTagArray = (LPSPropTagArray) lpSizedSPropTagArray
```

The **SPropTagArray** structure is defined in MAPIDEFS.H.

SPropValue

The **SPropValue** structure contains a MAPI property, including its property tag and property value.

```
typedef struct _SPropValue
{
     ULONG     ulPropTag;
     ULONG     dwAlignPad;
     union _PV  Value;
} SPropValue, FAR *LPSPropValue;
```

Members

ulPropTag
Contains a property tag for the property. This tag consists of a code for the property type in the lower 16 bits and a code for the property identifier in the upper 16 bits.

dwAlignPad
Contains the padding bytes to properly align the information indicated by the **Value** member.

Value
Contains the property value from the **_UPV** union.

Remarks

A **_UPV** union, used in the **Value** member of the **SPropValue** structure, defines the possible values for a MAPI property. The syntax for the **_UPV** union is as follows:

```
typedef union _PV
{
     short int             i;
     LONG                  l;
     ULONG                 ul;
     float                 flt;
     double                dbl;
     unsigned short int    b;
     CURRENCY              cur;
     double                at;
     FILETIME              ft;
     LPSTR                 lpszA;
     SBinary               bin;
     LPWSTR                lpszW;
     LPGUID                lpguid;
     LARGE_INTEGER         li;
     SShortArray           MVi;
     SLongArray            MVl;
     SRealArray            MVflt;
     SDoubleArray          MVdbl;
     SCurrencyArray        MVcur;
     SAppTimeArray         MVat;
     SDateTimeArray        MVft;
     SBinaryArray          MVbin;
     SLPSTRArray           MVszA;
     SWStringArray         MVszW;
     SGuidArray            MVguid;
     SLargeIntegerArray    MVli;
     SCODE                 err;
     LONG                  x;
} _UPV;
```

The members of the **_UPV** union contain the following information:

i
: Property value of the property for which the **SPropValue** structure holds information if the property's type is PT_I2.

l
: Property value if the property's type is PT_LONG and the property value is a LONG integer.

ul
: Property value if the property's type is PT_LONG and the property value is an unsigned LONG integer.

flt
: Property value if the property's type is PT_R4.

dbl
: Property value if the property's type is PT_DOUBLE.

b
: Property value if the property's type is PT_BOOLEAN.

cur
: Property value if the property's type is PT_CURRENCY.

at
: Property value if the property's type is PT_APPTIME.

ft
: Property value if the property's type is PT_SYSTIME.

lpszA
: Property value if the property's type is PT_STRING8.

bin
: Property value if the property's type is PT_BINARY.

lpszW
: Property value if the property's type is PT_UNICODE.

lpguid
: Property value if the property's type is PT_CLSID.

li
: Property value if the property's type is PT_I8.

MVi
: Property value if the property's type is PT_MV_I2.

MVl
: Property value if the property's type is PT_MV_LONG.

MVflt
: Property value if the property's type is PT_MV_R4.

MVdbl
: Property value if the property's type is PT_MV_DOUBLE.

MVcur
: Property value if the property's type is PT_MV_CURRENCY.

MVat
: Property value if the property's type is PT_MV_APPTIME.

MVft
: Property value if the property's type is PT_MV_SYSTIME.

MVbin

Property value if the property's type is PT_MV_BINARY.

MVszA
Property value if the property's type is PT_MV_STRING8.

MVszW
Property value if the property's type is PT_MV_UNICODE.

MVguid
Property value if the property's type is PT_MV_CLSID.

MVli
Property value if the property's type is PT_MV_I8.

err
Property value if the property's type is PT_ERROR.

x
Property value if the property's type is PT_NULL or PT_OBJECT.

The following five macros are used to set data types, flags, or to return property, type, or property identifier values.

CHANGE_PROP_TYPE

MVI_PROP

PROP_ID

PROP_TAG

PROP_TYPE

Use the **CHANGE_PROP_TYPE** macro to set the data type of the supplied MAPI property tag without changing the property identifier part of the tag. The syntax is:

ULONG CHANGE_PROP_TYPE (**ULONG** *ulPropTag*,**ULONG** *ulPropType*)

The *ulPropTag* parameter specifies the property tag. The *ulPropType* parameter specifies the value of the property type to set within *ulPropTag*. This macro returns a property tag with the property identifier set to *ulPropTag* and with the property type set to *ulPropType*.

Use the **MVI_PROP** macro to set the MV_FLAG and MV_INSTANCE flags for the supplied property tag. The property identifier and property type are otherwise unchanged. The syntax is:

ULONG MVI_PROP (ULONG *tag***)**

The *tag* parameter is the property tag that will have its MVI_FLAG bits set; this macro returns the property tag with its MVI_FLAG bits set. The MVI_FLAG represents the MV_FLAG and the MV_INSTANCE flags.

The MV_FLAG indicates a multivalued property. The MV_INSTANCE flag is used in table operations to request that a multivalued property be presented as a single-valued property appearing in multiple rows.

For example, when the input property tag contains the type PT_FLOAT, the returned property tag specifies the type PT_MVI_FLOAT; that is, PT_MV_FLOAT with the MV_INSTANCE bit set. All single-valued types have corresponding multivalued types.

The **PROP_ID** macro returns the property identifier value from the supplied property tag. The syntax is:

ULONG PROP_ID (**ULONG** *ulPropTag*)

The *ulPropTag* parameter specifies a property tag for which you want to obtain the property identifier. This macro returns the property identifier in the low-order word (bits 0-15) and zeros in the high-order word (bits 16-31). Bits 0-15 of the return value are equal to bits 16-31 of the supplied *ulPropTag*.

Use the **PROP_TAG** macro to return a property tag constructed from the supplied property type and the supplied property identifier.

The low-order 16 bits of the returned property tag contain the property type, and the high-order 16 bits of the returned tag contain the property identifier. The syntax is:

ULONG PROP_TAG (*ulPropType*, *ulPropID***)**

The *ulPropType* parameter specifies the property type to be used in the property tag. The *ulPropID* parameter specifies the property identifier. This macro returns a property tag with a data type of *ulPropType* and identifier *ulPropID*.

For example, the property tag PR_ENTRYID is formed by using the PROP_TAG macro as follows: PROP_TAG(PT_BINARY, 0x0FFF).

The **PROP_TYPE** macro returns the property type of the supplied property tag. The syntax is:

ULONG PROP_TYPE (**ULONG** *ulPropTag*)

The *ulPropTag* parameter specifies a property tag. This macro returns the low-order 16 bits of the property tag, which contain the value that represents the property type. The high-order 16 bits in the return value are set to zero. For example, PROP_TYPE(PR_ENTRYID) returns the value PT_BINARY.

The MVI_FLAG property tags are never used in the **SPropValue** structure.

The **SPropValue** structure is defined in MAPIDEFS.H.

See Also

About Property Types

SRealArray

The **SRealArray** structure contains a property value of type PT_FLOAT for use in an **SPropValue** structure containing information about a multivalued property.

```
typedef struct _SRealArray
{
     ULONG     cValues;
     float     FAR *lpflt;
} SRealArray;
```

Members

cValues
Indicates the number of values in the array pointed to by the **lpflt** member.

lpflt
Points to an array of float values making up the property.

Remarks

The **SRealArray** structure is defined in MAPIDEFS.H.

See Also

PT_FLOAT property type, **SPropValue** structure

SRestriction

The **SRestriction** structure contains a search restriction, or a set of search restrictions, used to limit a view of the messages in a table.

```
typedef struct _SRestriction
{
     ULONG     rt;
     union
     {
          SComparePropsRestriction    resCompareProps;
          SAndRestriction             resAnd;
          SOrRestriction              resOr;
          SNotRestriction             resNot;
          SContentRestriction         resContent;
          SPropertyRestriction        resProperty;
          SBitMaskRestriction         resBitMask;
          SSizeRestriction            resSize;
          SExistRestriction           resExist;
          SSubRestriction             resSub;
          SCommentRestriction         resComment;
     } res;
} SRestriction;
```

Members

rt

Indicates the restriction type. Possible values are:

RES_COMPAREPROPS
: A property comparison restriction, defined by an **SComparePropsRestriction** structure.

RES_AND
: A logical AND restriction, defined by an **SAndRestriction** structure.

RES_OR
: A logical OR restriction, defined by an **SOrRestriction** structure.

RES_NOT
: A logical NOT restriction, defined by an **SNotRestriction** structure.

RES_CONTENT
: A message content restriction, defined by an **SContentRestriction** structure.

RES_PROPERTY
: A property value restriction, defined by an **SPropertyRestriction** structure.

RES_BITMASK
: A bitmask restriction, defined by an **SBitMaskRestriction** structure.

RES_SIZE
: A size restriction, defined by an **SSizeRestriction** structure.

RES_EXIST
: A property existence restriction, defined by an **SExistRestriction** structure.

RES_SUBRESTRICTION
: A subrestriction restriction, defined by an **SSubRestriction** structure.

RES_COMMENT
: A comment restriction, defined by an **SCommentRestriction** structure.

resCompareProps

Contains an **SComparePropsRestriction** structure. This structure is the first in the union to accommodate static initializations of three-value restrictions.

resAnd
Indicates an **SAndRestriction** structure.

resOr
Indicates an **SOrRestriction** structure.

resContent
Indicates an **SContentRestriction** structure.

resProperty
Indicates an **SPropertyRestriction** structure.

resBitMask
Indicates an **SBitMaskRestriction** structure.

resSize
Indicates an **SSizeRestriction** structure.

resExist
Indicates an **SExistRestriction** structure.

resSub
Indicates an **SSubRestriction** structure.

resComment
Indicates an **SCommentRestriction** structure.

Remarks

A client application uses **SRestriction** structures in calls to the **IMAPITable::Restrict** and (for search-results folders) **IMAPIContainer::SetSearchCriteria** methods. A client can also use **SRestriction** with the **IMAPITable::FindRow** method to find table rows with certain attributes.

These three methods use **SRestriction** structures for locating and selecting an item or items based on the set of criteria incorporated in the **SRestriction** structure. During a search or restriction operation, the provider evaluates each object in a table or folder in terms of the **SRestriction** criteria. Only an object that matches the search restriction is included as a result of the search.

For meaningful results, restrictions on properties that are not required columns in a table should be ANDed with **SExistRestriction** structures on those properties. The provider is not obligated to return consistent or predictable results when a non-existent property is tested in a restriction.

The **SRestriction** structure is defined in MAPIDEFS.H.

See Also

SAndRestriction structure, **SBitMaskRestriction** structure, **SCommentRestriction** structure, **SComparePropsRestriction** structure, **SContentRestriction** structure, **SExistRestriction** structure, **SNotRestriction** structure, **SOrRestriction** structure, **SPropertyRestriction** structure, **SSizeRestriction** structure, **SSubRestriction** structure, **IMAPIContainer::SetSearchCriteria** method, **IMAPITable::FindRow** method, **IMAPITable::Restrict** method

SRow

The **SRow** structure contains a table row containing selected properties for a specific object.

```
typedef struct _SRow
{
     ULONG          ulAdrEntryPad;
     ULONG          cValues;
     LPSPropValue   lpProps;
} SRow, FAR *LPSRow;
```

Members

ulAdrEntryPad
Indicates padding bytes to properly align the information pointed to by the **lpProps** member.

cValues
Indicates the number of values in the array pointed to by **lpProps**.

lpProps
Points to an array of **SPropValue** structures. Each **SPropValue** structure represents a column within the **SRow**.

Remarks

SRow structures typically exist as components of **SRowSet** structures. These structures are used to represent MAPI table rows and MAPI tables, respectively.

Each instance of an **SRow lpProps** member in an **SRowSet** must be allocated (using **MAPIAllocateBuffer**) separately from the **SRowSet**. A row's allocated memory can then be preserved and reused outside of the context of the **SRowSet**.

The **lpProps** members must be deallocated prior to deallocation of the containing **SRowSet** so that pointers to allocated **SPropValue** structures are not lost.

The **SRow** structure is defined in MAPIDEFS.H.

See Also

ADRLIST structure, **SPropValue** structure, **SRowSet** structure

SRowSet

The **SRowSet** structure contains a set of table rows; each row in this set of rows contains selected properties for a specific object.

```
typedef struct _SRowSet
{
     ULONG   cRows;
     SRow    aRow[MAPI_DIM];
} SRowSet, FAR *LPSRowSet;
```

Members

aRow
Indicates an array of **SRow** structures, one for each table row.

Remarks

The MAPI function **HrQueryAllRows** retrieves MAPI table rows into this structure.

The structure member types and allocation rules for **SRow** and **ADRENTRY** structures are identical. **SRowSet** structures can be cast into **ADRLIST** structures to which **IMessage::ModifyRecipients** and **IAddrBook::Address** can then be applied.

See **SRow** for allocation rules for **SRowSet** structures and their associated **SRow** structures.

Use the **CbSRowSet** macro to determine the number of bytes of memory occupied by an existing **SRowSet** structure. The syntax for this macro is:

int CbSRowSet (**LPSRowSet** _*lpSRowSet*)

The _*lpSRowSet* parameter specifies a pointer to an **SRowSet** structure. This macro returns the number of bytes occupied by the **SRowSet** structure pointed to by _*lpSRowSet*.

Use the **cbNewSRowSet** macro to determine the memory allocation requirements of an **SRowSet** structure containing a specified number of rows. The syntax is:

int CbNewSRowSet (**int** _*crow*)

The _*crow* parameter specifies the number of rows, that is the number of elements of type **SRow** in the **aRow** member. This macro returns the number of bytes of memory that an **SRowSet** with the number of rows specified by _*crow* would occupy.

The **SizedSRowSet** macro creates a structure definition identical to that of **SRowSet** but with a specified number of rows. The syntax is:

SizedSRowSet (**int** _*crow*, _*name*)

The _*crow* parameter specifies the number of rows. The structure type is defined with the tag _SRowSet_ _*name* and type name _*name*.

To use a sized property tag array pointer *lpSizedSRowSet* in any function call or structure that expects a **LPSRowSet** pointer, perform the following cast:

```
lpSRowSet = (LPSRowSet) lpSizedSRowSet
```

The **SRowSet** structure is defined in MAPIDEFS.H.

See Also

ADRLIST structure, **HrQueryAllRows** function, **SRow** structure

SShortArray

The **SShortArray** structure contains a property value of type PT_MV_SHORT for use in an **SPropValue** structure containing information about a multivalued property.

```
typedef struct _SShortArray
{
     ULONG              cValues;
     short int      FAR *lpi;
} SShortArray;
```

Members

cValues
Indicates the number of values in the array pointed to by the **lpi** member.

lpi
Points to an array of values making up the property.

Remarks

The **SRowSet** structure is defined in MAPIDEFS.H.

See Also

PT_MV_SHORT property type, **SPropValue** structure

SSizeRestriction

The **SSizeRestriction** structure contains a size search restriction for objects such as tables and message stores.

```
typedef struct _SSizeRestriction
{
     ULONG relop;
     ULONG ulPropTag;
     ULONG cb;
} SSizeRestriction;
```

Members

relop
Relational operator used in the size comparison. Possible values are:

RELOP_GE
Indicates the comparison is made based on a greater or equal first value.

RELOP_GT
Indicates the comparison is made based on a greater first value.

RELOP_LE
Indicates the comparison is made based on a lesser or equal first value.

RELOP_LT
Indicates the comparison is made based on a lesser first value.

RELOP_NE
Indicates the comparison is made based on unequal values.

RELOP_RE
Indicates the comparison is made based on LIKE (regular expression) values.

RELOP_EQ
Indicates the comparison is made based on equal values.

ulPropTag
Contains a property tag.

cb
Indicates the size, in bytes, of the property value.

Remarks

The result of a property value restriction is undefined when the property does not exist. When a client requires well-defined behavior for such a restriction and is not sure whether the property exists (for example, it is not a required column of a table), it should combine the property restriction with an **SExistsRestriction** in an **SAndRestriction**.

The **SSizeRestriction** structure is defined in MAPIDEFS.H.

See Also

SRestriction structure

SSortOrder

The **SSortOrder** structure defines the sort order of data in a table.

```
typedef struct _SSortOrder
{
     ULONG ulPropTag;
     ULONG ulOrder;
} SSortOrder, FAR *LPSSortOrder;
```

Members

ulPropTag
Indicates the property tag of the column on which the table is to be sorted.

ulOrder
Indicates the order in which the data is to be sorted. Possible values are:

TABLE_SORT_ASCEND
Sorts the table in ascending order.

TABLE_SORT_COMBINE
Indicates that the provider should not show this as a separate category, but should instead combine it with the previous category.

TABLE_SORT_DESCEND
Sorts the table in descending order.

Remarks

TABLE_SORT_COMBINE is used on categorized tables to indicate that a particular column should not be its own category. Instead, it should be combined with the previous column. Using this value reduces the number of category rows which are displayed. This value can be used on multiple adjacent columns for multiple combinations. For example, to categorize a table with two name columns by name, yet have a single category, use the TABLE_SORT_COMBINE value.

The MV_FLAG is combined, using the OR operation, with the type portion of the property tags to make the property accommodate an array of the base type. MVI_FLAG is not actually a flag, but instead a combination of the MV_FLAG and MV_INSTANCE.

If supported, property tags with MV_FLAG can be used anywhere single valued property tags can be used. They can be used in **IMAPIProp::SetProps**, **IMAPIProp::GetProps**, **IMAPITable::SetColumns**, **IMAPITable::SortTable**, **IMAPITable::Restrict**.

Property tags with MVI_FLAG are only used on tables and have special semantics.They can be used as input to **IMAPITable::SetColumns** in the **SPropTagArray** structure, **IMAPITable::SortTable** in the **SSortOrderSet** structure and **IMAPITable::Restrict** in the **SRestriction** structure's **ulPropTag** member. They are never found in **SPropValue** or in the output parameters of methods other than **IMAPITable::QueryColumns** and **IMAPITable::QuerySortOrder**.

Columns with MVI_FLAG request the provider to return that column as single value properties, with one row per instance of the MV_FLAG property that is stored in the object. The property tags in **SPropValues** returned by **IMAPITable::QueryRows** are single valued for that column. For example, if you ask for PR_FOO you will get PR_FOO and ~MVI_FLAG in the **ulPropTag** member of the **SPropValue** structure.

SortOrderSet works identically to the **IMAPITable::SetColumns** behavior. Sorts are done based on the single values in the instances, and rows are added based on the expansion of each object's MVI_FLAG columns in both the **SSortOrder** and **SPropTagArray** structures.

Unlike MVI_FLAG in **SSortOrder** and **SPropTagArray**, MVI_FLAG in **SRestriction** does not expand the objects into computed rows. Rather, given that a column has been set by **IMAPITable::SetColumns** or **IMAPITable::SortTable** to be instances of the underlying multivalued property, putting a property tag with MVI_FLAG in the **SRestriction** structure tells the provider to use that column in restricting the table. The **SPropValue** structure (if any) to restrict against must be a single valued property tag identical to the one that would be returned by **IMAPITable::QueryRows** for the column.

The **SSortOrder** structure is defined in MAPIDEFS.H.

See Also

SSortOrderSet structure

SSortOrderSet

The **SSortOrderSet** structure defines a set of sort orders for a table, each order indicating on which column the table is to be sorted.

```
typedef struct _SSortOrderSet
{
     ULONG      cSorts;
     ULONG      cCategories;
     ULONG      cExpanded;
     SSortOrder aSort[MAPI_DIM];
} SSortOrderSet, FAR *LPSSortOrderSet;
```

Members

cSorts
Indicates the number of columns on which to sort the table in the array in the **aSort** member.

cCategories
Indicates the number of categories of data to be sorted. Possible values range from zero, which indicates a noncategorized sort, up to the number indicated by the **cSorts** member.

cExpanded
Indicates the number of categories that start in an expanded condition. Possible values include zero.

aSort
Contains an array of **SSortOrder** structures, each defining a sort order.

Remarks

Use the **CbSSortOrderSet** macro to determine the number of bytes of memory occupied by an existing **SSortOrderSet** structure. The syntax is:

CbSSortOrderSet (*_lpSSortOrderSet*)

The *_lpSSortOrderSet* parameter specifies a pointer to an **SSortOrderSet** structure. This macro returns the number of bytes occupied by the **SSortOrderSet** structure pointed to by *_lpSSortOrderSet*.

The **CbNewSSortOrderSet** macro determines the memory allocation requirements of an **SSortOrderSet** structure containing a specified number of sort orders. The syntax is:

CbNewSSortOrderSet (*_csort*)

The *_csort* parameter specifies the number of sort orders, that is, the number of elements of type **SSortOrder** in the **aSort** member. This macro returns the number of bytes of memory that an **SSortOrderSet** with the number of sort orders specified by *_csort* would occupy.

The **SizedSSortOrderSet** macro creates a structure definition identical to that of **SSortOrderSet** but specifies the columns to sort on. Use the **SizedSSortOrderSet** macro to create sort order sets with explicit bounds. The syntax is:

SizedSSortOrderSet (*_csort*, *_name*)

The *_csort* parameter specifies the number of sort orders. The structure type is defined with the tag _SSortOrderSet_ *_name* and type name *_name*.

To use a sized, sort order set pointer *lpSizedSSortOrderSet* in any function call or structure that expects an **LPSSortOrderSet** pointer, perform the following cast:

```
lpSSortOrderSet = (LPSSortOrderSet) lpSizedSSortOrderSet
```

The **SSortOrderSet** structure is defined in MAPIDEFS.H.

SSubRestriction

The **SSubRestriction** structure contains a search subrestriction for subobjects of table entries.

```
typedef struct _SSubRestriction
{
     ULONG          ulSubObject;
     LPSRestriction lpRes;
} SSubRestriction;
```

Members

ulSubObject
Indicates the subobject identified in the RES_SUBRESTRICTION restriction type supplied for the **rt** member of the **SRestriction** structure. Possible values are PR_MESSAGE_RECIPIENTS or PR_MESSAGE_ATTACHMENTS.

lpRes
Points to an **SRestriction** structure.

Remarks

The most common objects to support subrestrictions are folder contents tables and search-results folders. These objects may support restricting a search using PR_MESSAGE_ATTACHMENTS or PR_MESSAGE_RECIPIENTS as a restriction to find a message that has an attachment or a recipient that meets the other given restrictions. If an implementation does not support subrestrictions, it returns for a subrestricted search the error value MAPI_E_TOO_COMPLEX.

The **SSubRestriction** structure is defined in MAPIDEFS.H.

See Also

SRestriction structure

STATUS_OBJECT_NOTIFICATION

The **STATUS_OBJECT_NOTIFICATION** structure contains information about a notification event indicating that a row of the status table has changed. MAPI uses this structure only as a member of the **NOTIFICATION** structure for the advise sink.

```
typedef struct
{
     ULONG          cbEntryID;
     LPENTRYID      lpEntryID;
     ULONG          cValues;
     LPSPropValue   lpPropVals;
} STATUS_OBJECT_NOTIFICATION;
```

Members

cbEntryID
Indicates the size, in bytes, of the entry identifier of the changed status object.

lpEntryID
Points to the entry identifier of the changed status object.

cValues
Indicates the number of **SPropValue** structures in the array pointed to by the **lpPropVals** member.

lpPropVals
Points to an array of **SPropValue** structures, one for each changed property of the status object.

Remarks

The **STATUS_OBJECT_NOTIFICATION** structure is defined in MAPIDEFS.H.

See Also

NOTIFICATION structure, **SPropValue** structure

STnefProblem

The **STnefProblem** structure contains information about a property or attribute processing problem that occurred during the encoding or decoding of a Transport-Neutral Encapsulation Format (TNEF) stream.

```
typedef struct _STnefProblem
{
     ULONG     ulComponent;
     ULONG     ulAttribute;
     ULONG     ulPropTag;
     SCODE     scode;
} STnefProblem;
```

Members

ulComponent
Indicates the type of processing during which the problem occurred. If the problem occurred during message processing, the **ulComponent** member is set to zero. If the problem occurred during attachment processing, **ulComponent** is set equal to the corresponding attachment's PR_ATTACHMENT_NUM value.

ulAttribute
Indicates the attribute corresponding to the **ulPropTag** member, except when the TNEF processing problem arises while decoding an encapsulation block. In this case, the **ulAttribute** member can have the following possible values:

attMAPIProps
Message level

attAttachment
Attachment level

ulPropTag
Indicates the property tag of the property that caused the TNEF processing problem, except when the problem arises while decoding an encapsulation block, in which case **ulPropTag** is set to zero.

scode
Indicates an error value indicating the problem encountered during processing.

Remarks

If an **STnefProblem** structure is not generated during the processing of an attribute or property, the application can continue under the assumption that the processing of that attribute or property succeeded. The only exception occurs when the problem arose during decoding of an encapsulation block. In this case, the decoding of the component corresponding to the block is halted and decoding is continued in another component.

The **STnefProblem** structure is defined in TNEF.H.

See Also

STnefProblemArray structure

STnefProblemArray

The **STnefProblemArray** structure lists one or more property or attribute processing problems that occurred during the encoding or decoding of a Transport-Neutral Encapsulation Format (TNEF) stream.

```
typedef struct _STnefProblemArray
{
     ULONG....       cProblem;
     STnefProblem    aProblem[MAPI_DIM];
}STnefProblemArray, FAR * LPSTnefProblemArray
```

Members

cProblem
Indicates the number of elements in the array in the **aProblem** member.

aProblem
Contains an array of **STnefProblem** structures. Each structure contains information about a property or attribute processing problem.

Remarks

If a problem occurs during attribute or property processing, an output parameter in the **ITnef::ExtractProps** method and in the **ITnef::Finish** method each receive a pointer to an **STnefProblemArray** structure and **ExtractProps** and **Finish** each return the value MAPI_W_ERRORS_RETURNED. This error value indicates that a problem arose during processing and an **STnefProblemArray** structure was generated.

If an **STnefProblem** structure is not generated during the processing of an attribute or property, the client application can continue under the assumption that the processing of that attribute or property succeeded. The only exception occurs when the problem arose during decoding of an encapsulation block. If the error occurred during this decoding, MAPI_E_UNABLE_TO_COMPLETE can be returned as the SCODE in the structure. In this case, the decoding of the component corresponding to the block is halted and decoding is continued in another component.

The **STnefProblemArray** structure is defined in TNEF.H.

See Also

ITnef::ExtractProps method, **ITnef::Finish** method, **STnefProblem** structure

SWStringArray

The **SWStringArray** structure contains a property value of type PT_MV_UNICODE for use in an **SPropValue** structure containing information about a multivalued property.

```
typedef struct _SWStringArray
{
     ULONG           cValues;
     LPWSTR     FAR *lppszW;
} SWStringArray;
```

Members

cValues
Indicates the number of values in the array pointed to by the **lppszW** member.

lppszW
Points to the array of null-terminated Unicode string values making up the property.

Remarks

The **SWStringArray** structure is defined in MAPIDEFS.H.

See Also

PT_MV_UNICODE property type, **SPropValue** structure

TABLE_NOTIFICATION

The **TABLE_NOTIFICATION** structure contains information about a notification event related to a table, such as a table change, error, addition, or deletion. MAPI uses this structure only as a member of the **NOTIFICATION** structure for the advise sink.

```
typedef struct _TABLE_NOTIFICATION
{
     ULONG        ulTableEvent;
     HRESULT      hResult;
     SPropValue   propIndex;
     SPropValue   propPrior;
     SRow         row;
} TABLE_NOTIFICATION;
```

Members

ulTableEvent
Bitmask of flags used to represent the table event type. The following flags can be set:

TABLE_CHANGED
Indicates something has changed but the table implementation does not have the details. The table's state is as it was before the event. All PR_INSTANCE_KEY properties, bookmarks, current positioning, and user interface selections are still valid. The user interface code that displays the table should re-read the entire table upon receiving this event. Service providers that do not want to implement "rich" table notifications simply send TABLE_CHANGED events rather than more detailed events to indicate a particular type of change.

In response to this flag, the client application can re-read the entire table to get current data.

TABLE_ERROR
Indicates an error has occurred, usually during asynchronous table processing. Errors during calls to **IMAPITable::SortTable**, **SetColumns**, or **Restrict** can generate this type of event. The **IMAPITable::GetLastError** method cannot provide any further information about the error because it was generated at some previous point, not necessarily from the last method call.

TABLE_RELOAD
Indicates a need to re-read the data and start over. Service providers send TABLE_RELOAD when, for example, the underlying data is stored in a database and the database is replaced. When advise sinks receive this event, they should assume that nothing about the table is still valid. All bookmarks, instance keys, status and positioning information are invalid and the data should be reread.

TABLE_RESTRICT_DONE
Indicates a search of the table has completed.

TABLE_ROW_ADDED
Indicates a new row has been added to the table, after a call to **IMAPIProp::SaveChanges**. The **propPrior** instance key is for the row above where the row was added. Advise sinks receiving this event should bear in mind that although the instance key for the row prior to the affected row (**propPrior**) and the property data contained in the **SRow** structure were correct when the notification was generated, it might no longer be correct. Between the time the notification was generated and the time that it was sent, other changes might have occurred. If the added or modified row is now the first row in the table, the property tag in the **propPrior** member is PR_NULL (== 1).

TABLE_ROW_DELETED
Indicates that a row is removed from the table. The **TABLE_NOTIFICATION** structure does not contain a value for the **propPrior** member for this event, only the **propIndex** member.

TABLE_ROW_MODIFIED

Indicates a changed row. The **SRow** structure contains new data for the row. Multiple TABLE_ROW_MODIFIED events should be sent in chronological order with respect to the view that is seen by the user. All TABLE_ROW_MODIFIED events should be sent after changes to the row have been committed and **IMAPIProp::SaveChanges** has been called. If the added or modified row is now the first row in the table, the property tag in the **propPrior** member is PR_NULL (== 1).

TABLE_SETCOL_DONE
Indicates the table's columns have been set.

TABLE_SORT_DONE
Indicates that the table's sort order has changed.

hResult
Contains an HRESULT value for the TABLE_ERROR event listed for the **ulTableEvent** member.

propIndex
Contains an **SPropValue** structure giving the index of the table row that has changed, that is, the current row's index.

propPrior
Contains an **SPropValue** structure giving the index of the row preceding the current one.

row
Contains an **SRow** structure containing the data for the added or modified row. This structure is filled for all table notification events, even if an event of the current type doesn't require it. For table notification events that do not pass row data, **row.cValues** must equal zero and **row.lpProps** must be NULL. This **SRow** structure is read-only, so client applications must copy it to work with it.

Remarks

The properties received in the **TABLE_NOTIFICATION** structure are not necessarily the same as the current column set of the table in question.

TABLE_ERROR can be sent as a result of asynchronous calls to the **Sort**, **IMAPITable::Restrict**, or **IMAPITable::SetColumns** methods. This TABLE_ERROR flag can be written for asynchronous **Sort**, **IMAPITable::Restrict**, or **IMAPITable::SetColumns** calls. It can also be written with underlying processing that attempts to update a table with, for example, new or modified rows.

The TABLE_ROW_ADDED flag that indicates the **propIndex** member identifies the row that the client application has created.

Because client applications handle notifications asynchronously, notification of an addition to a table might arrive after the application is already aware of the change. For example, suppose a notification that a row is added has been generated but not yet sent to the application. The application might read 20 rows, including the added row, before MAPI passes the notification to the application's notification callback function.

Data describing the new position and contents of the row (the **propPrior**, **row.cValues**, and **row.lpProps** members). If the added or modified row is now the first row in the table, the property tag in the **propPrior** member is PR_NULL.

Once a client application receives TABLE_ERROR for a table, the application can no longer rely on the accuracy of the table contents. Notification of changes might be lost. To get additional information about a table for which a TABLE_ERROR event has occurred, the application can call the **GetLastError** method for the table.

The **TABLE_NOTIFICATION** structure is defined in MAPIDEFS.H.

See Also

IMAPITable::Restrict method, **IMAPITable::SetColumns** method, **NOTIFICATION** structure, **SPropValue** structure, **SRow** structure

MAPI Data Types

The following alphabetized entries contain documentation for MAPI Data Types.

BOOKMARK

BOOKMARK is an unsigned long data type that retains in memory a position in a table. The data stored in a bookmark depends on the client application.

```
typedef ULONG BOOKMARK;
```

Remarks

MAPI defines the following bookmarks:

BOOKMARK_BEGINNING
    Seeks to the beginning of the table.
BOOKMARK_CURRENT
    Seeks to the row in the table where the cursor is located.
BOOKMARK_END
    Seeks to the end of the table.

The client application can create additional bookmarks. Created bookmarks are only useful while a table is open. The client must free any created bookmarks when it closes a table.

The BOOKMARK unsigned long data type is defined in MAPIDEFS.H.

See Also

IMAPITable::CreateBookmark method, **IMAPITable::FindRow** method, **IMAPITable::FreeBookmark** method, **IMAPITable::SeekRow** method

BYTE

BYTE is an unsigned character data type that is binary data.

```
typedef unsigned char     BYTE;
```

HRESULT

HRESULT is a data type that is a 32-bit error or warning value.

```
typedef LONG          HRESULT;
```

Remarks

An HRESULT data type is made up of a 1-bit severity flag, an 11-bit handle, a 4-bit facility code indicating status code (SCODE) group, and a 16-bit SCODE information code. A value of zero for the severity flag indicates the success of the operation for which the HRESULT was returned.

An HRESULT type returned as an error value for a function can provide the application that called the function information on the error and how to recover from it. To obtain this information, the application uses the handle of the HRESULT. The HRESULT and SCODE types are not equivalent. OLE includes functions and macros to convert between error values of these two types. To create an HRESULT value from an SCODE value, use **ResultFromScode(**SCODE**)**. To convert an HRESULT form to SCODE value, use **GetScode(**HRESULT**)**. For details about **ResultFromScode** and **GetScode** and for faster ways of making the conversions just mentioned, see the *OLE Programmer's Reference*. For a description of the OLE implementation of HRESULT, see *Inside OLE*, *Second Edition*, by Kraig Brockschmidt.

See Also

SCODE data type

LHANDLE

LHANDLE is a Simple MAPI data type that is a MAPI session handle.

```
typedef unsigned long LHANDLE, FAR *LPLHANDLE;
```

LONG

LONG is a data type that is a 32-bit signed integer.

```
typedef long           LONG;
```

SCODE

SCODE is a data type that is a 32-bit status value. MAPI functions and methods return values of the SCODE type.

```
typedef ULONG     SCODE;
```

Remarks

All MAPI functions and methods return SCODE values; some MAPI functions also return warnings, which are nonzero HRESULT values.

To obtain an SCODE value from an HRESULT value, the client application can use the OLE function **GetScode** as follows:

```
SCODE scode;
HRESULT hresult;

hresult = arbitrary function call;

if (hresult)
{
     scode = GetScode(hresult);
/* Display error based on scode */
}
```

When a MAPI function returns an HRESULT value as a warning, an SCODE return value can be identified as a type of success. For example:

```
hresult = SomeCall(..)
if (hresult !=0)
{
     SCODE scode = GetScode(hresult);
     if (FAILED(scode))
          goto error;
     /*Handle the warning here */
     if (scode == MAPI_W_warning)
     {
     }
}
```

See Also

HRESULT data type

TCHAR

TCHAR is a data type that is a character string on either a Unicode or an ANSI or DBCS platform. For Unicode platforms, this string is defined as having the WCHAR type. For ANSI and DBCS platforms, the string is defined as having the char type.

```
typedef char          TCHAR;
typedef WCHAR  TCHAR;
```

Remarks

The client application can use TCHAR to represent a string of either the WCHAR or char type. Be sure to define the symbolic constant UNICODE and limit the platform where necessary. MAPI will interpret the platform information and internally translate TCHAR to the appropriate string.

ULONG

ULONG is a data type that is a 32-bit unsigned integer.

```
typedef unsigned long     ULONG;
```

WCHAR

WCHAR is a data type that is a Unicode character string.

```
typedef WORD          WCHAR;
```

Common Messaging Calls (CMC)

The Common Messaging Calls (CMC) applications programming interface (API) provides a simple and convenient set of functions for applications that need basic messaging functionality. CMC provides for all the major messaging functionality an application should need, such as accessing message stores for sending and receiving messages, and addressing and name resolution services. The functions in the CMC API are typically high level functions and can each be used in several different ways depending on the arguments to the functions. The CMC API makes it possible for the calling application to know nothing about the underlying messaging system or transport mechanism used to implement the CMC API itself.

Most CMC functions use both input and output parameters. Input parameters provide information the CMC implementation uses to perform the tasks needed by the calling application. The CMC implementation uses output parameters to pass information back to the calling application from a CMC function. Some functions have parameters used for both input and output.

Data structures and symbolic constants for CMC and its extensions are described in this reference. Their C language definitions can be found in the following header files:

Header file name	Header file contents
XCMC.H	CMC data structures and symbolic constants.
XCMCEXT.H	Common CMC extension data structures and symbolic constants.
XCMCMSXT.H	Microsoft CMC extension data structures and symbolic constants.

Functions

The following functions are implemented in compliance with the X.400 API Association's *Common Messaging Call API* specification. These functions make heavy use of extensions, which are described in the Data Extensions section of the CMC reference. The functions are listed in alphabetical order.

cmc_act_on

The **cmc_act_on** function performs the specified operation on a message.

CMC_return_code cmc_act_on (
  CMC_session_id *session*,
  CMC_message_reference * *message_reference*,
  CMC_enum *operation*,
  CMC_flags *act_on_flags*,
  CMC_ui_id *ui_id*,
  CMC_extension FAR * *act_on_extensions*
)

Parameters

session
  Input parameter containing a session handle that represents a MAPI session. The value in the *session* parameter must be a valid session handle, not zero.

message_ reference
  Input parameter pointing to a message reference that identifies the message to be acted upon. A null pointer or a pointer to a message reference of length zero is invalid for any operation requiring a message reference. If the message reference is invalid, the **cmc_act_on** function returns CMC_E_INVALID_MESSAGE_REFERENCE.

operation
  Input parameter containing an enumeration variable that identifies the operation to perform on the message. Possible values for this variable are:

  CMC_ACT_ON_DELETE
    Marks the specified message for deletion from the mailbox. This operation requires a valid *message_reference* parameter.

  CMC_ACT_ON_EXTENDED
    Indicates the operation to be performed is specified in the *act_on_extensions* parameter.

act_on_flags
  Input parameter containing a bitmask of option flags. The following flag can be set:

  CMC_ERROR_UI_ALLOWED
    Displays a dialog box on encountering recoverable errors. If this flag is not set, **cmc_act_on** does not display a dialog box and returns an error value instead.

ui_id
  Input parameter containing a handle that **cmc_act_on** uses to present a dialog box for resolving processing questions.

act_on_extensions
  Input-output parameter pointing to an array of **CMC_extension** structures containing function extensions. On input, this array contains MAPI extensions to the standard **cmc_act_on** function. A value of NULL for the *act_on_extensions* parameter indicates the caller has no extensions for **cmc_act_on** and is expecting no extensions. You can use the CMC_X_COM_SAVE_MESSAGE extension to save a message to the receive folder.

  On output, **cmc_act_on** writes to the array new information about its processing. It writes NULL if it generates no output extensions.

Return Values

CMC_E_FAILURE
  There was a general failure that does not fit the description of any other return value.

CMC_E_INSUFFICIENT_MEMORY
  Insufficient memory was available to complete the requested operation.

CMC_E_INVALID_ENUM
A **CMC_enum** value was invalid.

CMC_E_INVALID_FLAG
A flag set using a flags parameter was invalid.

CMC_E_INVALID_MESSAGE_REFERENCE
The specified message reference is invalid or no longer valid (for example, it has been deleted).

CMC_E_INVALID_PARAMETER
A function parameter was invalid.

CMC_E_INVALID_SESSION_ID
The specified session handle is invalid or no longer valid (for example, after logging off).

CMC_E_INVALID_UI_ID
The specified user-interface identifier is invalid or no longer valid.

CMC_E_MESSAGE_IN_USE
The requested action could not be completed because the message was in use.

CMC_E_UNSUPPORTED_ACTION
The requested action is not supported by the current implementation.

CMC_E_UNSUPPORTED_FLAG
The flag requested is not supported.

CMC_E_UNSUPPORTED_FUNCTION_EXT
The function extension requested is not supported.

See Also

CMC_extension structure, CMC_X_COM_SAVE_MESSAGE

cmc_free

The **cmc_free** function frees memory allocated by the message service through another CMC function.

CMC_return_code cmc_free (
  CMC_buffer *memory*
)

Parameters

memory
  Input parameter pointing to memory previously allocated by CMC. The **cmc_free** function ignores a parameter value of NULL. After this function completes, the pointer to memory is invalid, and the application cannot reference it again.

Return Values

CMC_E_FAILURE
  There was a general failure that does not fit the description of any other return value.

CMC_E_INVALID_MEMORY
  A memory pointer passed is invalid.

Remarks

Results of the **cmc_free** function are unpredictable if the client application calls it with a base pointer to a memory block not allocated by the message service, a base pointer to a memory block already freed, or a nonbase pointer to a complex structure written by another CMC function.

The CMC functions **cmc_list**, **cmc_look_up, cmc_query_configuration**, and **cmc_read** can provide the client application with a base pointer to a complex structure containing several levels of pointers. The client application should free the entire structure or structure array by calling **cmc_free** with the base pointer.

See Also

cmc_list function, **cmc_look_up** function, **cmc_query_configuration** function, **cmc_read** function

cmc_list

The **cmc_list** function lists summary information for messages that meet client application-specified criteria.

CMC_return_code cmc_list (
  CMC_session_id *session*,
  CMC_string *message_type*,
  CMC_flags *list_flags*,
  CMC_message_reference * *seed*,
  CMC_uint32 FAR * *count*,
  CMC_ui_id *ui_id*,
  CMC_message_summary FAR * **FAR** * *result*,
  CMC_extension FAR * *list_extensions*
)

Parameters

session
  Input parameter containing an opaque session handle that represents a MAPI session object indicating a session with a message service. If the session handle is invalid, this function returns CMC_E_INVALID_SESSION_ID.

message_type
  Input parameter pointing to the ASCII name of the type of message for which this function lists information. If the **cmc_list** function does not recognize the specified message type, it returns CMC_E_UNRECOGNIZED_MESSAGE_TYPE. If the function receives a value of NULL for the message type, it lists information for all available message types.

list_flags
  Input parameter containing a bitmask of flags. The following flags can be set:

  CMC_ERROR_UI_ALLOWED
    Displays a dialog box on encountering recoverable errors. If this flag is not set, **cmc_list** does not display a dialog box and returns an error value instead.

  CMC_LIST_COUNT_ONLY
    Lists only a count of messages meeting the specified criteria, not any actual summary information. If this flag is not set, the function lists summary information in the array.

  CMC_LIST_MSG_REFS_ONLY
    Writes only message reference information to the array pointed to by the *result* parameter. If this flag is not set, **cmc_list** writes information to all members of the structures in the array.

  CMC_LIST_UNREAD_ONLY
    Lists unread messages only. If this flag is not set, **cmc_list** can list both read and unread messages.

seed
  Input parameter pointing to a message reference that identifies the message after which **cmc_list** should begin to search. A value of NULL for this parameter indicates that the function should start the search with the first message in the mailbox. A pointer to a message reference of length zero is invalid and causes **cmc_list** to return CMC_E_INVALID_MESSAGE_REFERENCE. If the *seed* parameter is part of a structure that is returned by an earlier CMC call, the structure that was allocated by CMC should be freed before the current session exits.

count
  Input-output parameter containing a message count. On input, this parameter specifies a pointer to the maximum number of messages for which **cmc_list** should provide summary information. A value of zero indicates no maximum.

  On output, the *count* parameter specifies the location to which **cmc_list** writes the number of messages for which it provides summary information. If no messages match the search criteria, or if the mailbox is empty, **cmc_list** writes zero.

ui_id
Input parameter containing the handle of a dialog box for **cmc_list** to present to help resolve processing questions.

result
Output parameter pointing to the location to which **cmc_list** writes the address of the array of **CMC_message_summary** structures that it has written.

list_extensions
Input-output parameter pointing to an array of **CMC_extension** structures specifying function extensions. On input, this array contains MAPI extensions to the standard **cmc_list** function. A value of NULL for the *list_extensions* parameter indicates that the client application has no extensions for **cmc_list** and is expecting no extensions.

On output, **cmc_list** writes to the array new information about its processing of the message summaries. It writes NULL if it generates no output extensions.

Return Values

CMC_E_FAILURE
There was a general failure that does not fit the description of any other return value.

CMC_E_INSUFFICIENT_MEMORY
Insufficient memory was available to complete the requested operation.

CMC_E_INVALID_FLAG
A flag set using a flags parameter was invalid.

CMC_E_INVALID_MESSAGE_REFERENCE
The specified message reference is invalid or no longer valid (for example, it has been deleted).

CMC_E_INVALID_PARAMETER
A function parameter was invalid.

CMC_E_INVALID_SESSION_ID
The specified session handle is invalid or no longer valid (for example, after logging off).

CMC_E_INVALID_UI_ID
The specified user-interface identifier is invalid or no longer valid.

CMC_E_UNRECOGNIZED_MESSAGE_TYPE
The specified message type is not supported by the current implementation.

CMC_E_UNSUPPORTED_FLAG
The flag requested is not supported.

CMC_E_UNSUPPORTED_FUNCTION_EXT
The function extension requested is not supported.

Remarks

The client application can specify a **cmc_list** search to start with a certain message or to start at the first message in the mailbox. It can also specify the maximum number of messages to list. The **cmc_list** function writes the summary information for the specified messages in an array of **CMC_message_summary** structures. Using the message references in these structures, the application can then make calls to the **cmc_read** and **cmc_act_on** functions for additional processing.

Before **cmc_list** writes message summary information, it must allocate memory for the structure array to contain the information. When this memory is no longer needed, the client application should free the entire array with a call to the **cmc_free** function.

See Also

cmc_act_on function, **CMC_extension** structure, **cmc_free** function, **CMC_message_summary**

structure, **cmc_read** function

cmc_logoff

The **cmc_logoff** function logs a client application off a message service.

CMC_return_code cmc_logoff (
  CMC_session_id *session*,
  CMC_ui_id *ui_id*,
  CMC_flags *logoff_flags*,
  CMC_extension FAR * *logoff_extensions*
)

Parameters

session
  Input parameter containing an opaque session handle that represents a MAPI session object indicating a session with a message service. If the session handle is invalid, the **cmc_logoff** function returns CMC_E_INVALID_SESSION_ID. After **cmc_logoff** returns, the session handle is invalid.

ui_id
  Input parameter containing the handle of a dialog box for **cmc_logoff** to present to help resolve processing questions.

logoff_flags
  Input parameter containing a bitmask of flags. The following flags can be set:

  CMC_ERROR_UI_ALLOWED
    Displays a dialog box on encountering recoverable errors. If this flag is not set, **cmc_logoff** does not display a dialog box and returns an error value instead.

  CMC_LOGOFF_UI_ALLOWED
    Indicates **cmc_logoff** can display a dialog box for other purposes than displaying error messages while logging the user off from the session.

logoff_extensions
  Input-output parameter pointing to an array of **CMC_extension** structures containing function extensions. On input, this array contains MAPI extensions to the standard **cmc_logoff** function. A value of NULL for the *logoff_extensions* parameter indicates that the client application has no extensions for **cmc_logoff** and is expecting no extensions.

  On output, **cmc_logoff** writes to the array new information about the logoff operation. It writes NULL if it generates no output extensions.

Return Values

CMC_E_FAILURE
  There was a general failure that does not fit the description of any other return value.

CMC_E_INSUFFICIENT_MEMORY
  Insufficient memory was available to complete the requested operation.

CMC_E_INVALID_FLAG
  A flag set using a flags parameter was invalid.

CMC_E_INVALID_PARAMETER
  A function parameter was invalid.

CMC_E_INVALID_SESSION_ID
  The specified session handle is invalid or no longer valid (for example, after logging off).

CMC_E_INVALID_UI_ID
  The specified user-interface identifier is invalid or no longer valid.

CMC_E_UNSUPPORTED_FLAG
  The flag requested is not supported.

CMC_E_UNSUPPORTED_FUNCTION_EXT

The function extension requested is not supported.

CMC_E_USER_NOT_LOGGED_ON

The user was not logged on and the CMC_LOGON_UI_ALLOWED flag was not set.

See Also

CMC_extension structure

cmc_logon

The **cmc_logon** function logs a client application onto a service provider.

CMC_return_code cmc_logon (
  CMC_string *service*,
  CMC_string *user*,
  CMC_string *password*,
  CMC_object_identifier *character_set*,
  CMC_ui_id *ui_id*,
  CMC_uint16 *caller_CMC_version*,
  CMC_flags *logon_flags*,
  CMC_session_id FAR * *session*,
  CMC_extension FAR * *logon_extensions*
)

Parameters

service
  Input parameter pointing to the location of the service provider for the CMC implementation. Passing NULL for the *service* parameter indicates either that the client application is requesting logon to a service provider that does not require a service name, or that the client is requesting the CMC implementation's logon dialog box.

user
  Input parameter pointing to a MAPI profile name identifying the client application. Passing NULL for the *user* parameter indicates either that the client is requesting logon to a service provider that does not require a user name, or that the client is requesting the CMC implementation's dialog box to prompt for a name.

password
  Input parameter pointing to a MAPI profile password required for access to the CMC implementation. Passing NULL for the *service* parameter indicates either that the client is requesting logon to a service provider that does not require a password, or that the client is requesting the CMC implementation's dialog box to prompt for a password.

character_set
  Input parameter pointing to an object identifier for the character set used by the client application. The client application can call the **cmc_query_configuration** function to retrieve the available values. The CMC implementation requires a non-null value for the *character_set* parameter.

ui_id
  Input parameter containing the handle of a dialog box for the **cmc_logon** function to present to help resolve processing questions or prompt for logon.

caller_CMC_ version
  Input parameter containing the client application's CMC version number, multiplied by 100. For example, version 1 is specified as the integer 100.

logon_flags
  Input parameter containing a bitmask of flags. The following flags can be set:

  CMC_COUNTED_STRING_TYPE
    Indicates the string type the calling application or provider uses for CMC interactions is a CMC_counted_string. If this flag is not set, the function treats all strings as null-terminated strings.

  CMC_ERROR_UI_ALLOWED
    Displays a dialog box on encountering recoverable errors. If this flag is not set, **cmc_logon** does not display a dialog box and returns an error value instead.

  CMC_LOGON_UI_ALLOWED
    Displays a dialog box to prompt for logon if required. If this flag is not set, **cmc_logon** does not

display a dialog box and returns an error value if the user does not supply enough information.

session

Output parameter pointing to the location to which **cmc_logon** writes an opaque session handle. This identifier represents a MAPI session object indicating a session with a message service.

logon_extensions

Input-output parameter pointing to an array of **CMC_extension** structures specifying function extensions. On input, this array contains MAPI extensions to the standard **cmc_logon** function. A value of NULL for the *logon_extensions* parameter indicates that the client application has no extensions for **cmc_logon** and is expecting no extensions.

On output, **cmc_logon** writes to the array new information about the logon operation. It writes NULL if it generates no output extensions.

Return Values

CMC_E_COUNTED_STRING_UNSUPPORTED
This implementation does not support the counted-string type.

CMC_E_FAILURE
There was a general failure that does not fit the description of any other return value.

CMC_E_INSUFFICIENT_MEMORY
Insufficient memory was available to complete the requested operation.

CMC_E_INVALID_FLAG
A flag set using a flags parameter was invalid.

CMC_E_INVALID_PARAMETER
A function parameter was invalid.

CMC_E_INVALID_UI_ID
The specified user-interface identifier is invalid or no longer valid.

CMC_E_LOGON_FAILURE
The service, user name, or password specified was invalid, so logon cannot be completed.

CMC_E_PASSWORD_REQUIRED
A password is required on this message service.

CMC_E_SERVICE_UNAVAILABLE
The service requested was unavailable.

CMC_E_UNSUPPORTED_CHARACTER_SET
The current implementation does not support the character set requested.

CMC_E_UNSUPPORTED_FLAG
The current implementation does not support the flag requested.

CMC_E_UNSUPPORTED_FUNCTION_EXT
The current implementation does not support the function extension requested.

CMC_E_UNSUPPORTED_VERSION
The current implementation cannot support the version specified in the call.

Remarks

The **cmc_logon** function can, at the client application's option, either prompt the user for information through a dialog box or proceed without any user interaction. It writes a session handle that the client application can use in subsequent calls to the CMC implementation.

See Also

CMC_extension structure, **cmc_query_configuration** function

cmc_look_up

The **cmc_look_up** function looks up addressing information in a directory provided by a specified service provider.

CMC_return_code cmc_look_up (
  CMC_session_id *session*,
  CMC_recipient FAR * *recipient_in*,
  CMC_flags *look_up_flags*,
  CMC_ui_id *ui_id*,
  CMC_uint32 FAR * *count*,
  CMC_recipient FAR * **FAR** * *recipient_out*,
  CMC_extension FAR * *look_up_extensions*
)

Parameters

session
  Input parameter containing an opaque session handle that represents a MAPI session object that represents a session with a message service. If the session handle is invalid, the **cmc_look_up** function returns the CMC_E_INVALID_SESSION_ID error value.

recipient_in
  Input parameter pointing to an array of **CMC_recipient** structures containing recipient data. The **cmc_look_up** function interprets the array depending on the flags that the client application has set using the *look_up_flags* parameter. Possible interpretations are as following:

- If the client application has set one of the flags for name resolution, **cmc_look_up** obtains the name to resolve from the name member of the first structure in the array. The function checks the corresponding name-type member to discover what resolution should be performed. The **cmc_look_up** function ignores all recipient structures except the first in the array.
- If the client application has set the CMC_LOOKUP_DETAILS_UI flag, the information in the array must resolve to only one recipient. If it does not, **cmc_look_up** returns CMC_E_AMBIGUOUS_RECIPIENT. The **cmc_look_up** function ignores all recipient structures except the first in the array.
- If the client application has set the CMC_LOOKUP_ADDRESSING_UI flag, **cmc_look_up** displays the recipients specified in the recipient array in the address-list dialog box.

look_up_flags
  Input parameter containing a bitmask of flags. The following flags can be set:

  CMC_COUNTED_STRING_TYPE
    Indicates the string type the calling application or provider uses for CMC interactions is a CMC_counted_string. If this flag is not set, the function treats all strings as null-terminated strings.

  CMC_ERROR_UI_ALLOWED
    Displays a dialog box on encountering recoverable errors. If this flag is not set, **cmc_look_up** does not display a dialog box and returns an error value instead.

  CMC_LOGON_UI_ALLOWED
    Displays a dialog box to prompt for logon if required. If this flag is not set, **cmc_look_up** does not display a dialog box and returns an error value if the user does not supply enough information.

  CMC_LOOKUP_ADDRESSING_UI
    Displays a dialog box to allow creation of a recipient list for addressing a message and general directory browsing. The recipient list passed to the function is the original recipient list for the dialog box. The function returns the list of recipients created by the user. This flag is optional for implementations to support.

  CMC_LOOKUP_DETAILS_UI
    Displays a details dialog box for the recipient pointed to in the *recipient_in* parameter. This dialog

box only acts on the first recipient in the list. If the recipient name indicated resolves to more than one address, **cmc_look_up** does not display the details dialog box and returns CMC_E_AMBIGUOUS_RECIPIENT.

CMC_LOOKUP_RESOLVE_IDENTITY
: Returns a recipient record for the identity of the current user of the message service. If no unique identity can be determined, the implementation carries out ambiguous name resolution to determine the address of the current user.

CMC_LOOKUP_RESOLVE_PREFIX_SEARCH
: Indicates the search method should be by prefix. In a prefix search, all names matching the prefix string, beginning at the first character of the name, are considered matches. If this flag is not set, the search method should be exact-match. CMC implementations must support simple prefix searching. The availability of wildcard or substring searches is optional.

CMC_LOOKUP_RESOLVE_UI
: Attempts to resolve ambiguous names by presenting a name-resolution dialog box to the user. If this flag is not set, resolutions that do not result in a single name return the error value CMC_E_AMBIGUOUS_RECIPIENT for message services that require names to resolve to a single address. Message services that can return multiple addresses can return a list of addresses if the *count* parameter is non-null. The **name_type** field in the *recipient_out* parameter can also be set on input as a hint to aid in resolution of the name. Some CMC implementations might not support this flag. The CMC_LOOKUP_RESOLVE_UI flag is set only when the CMC_LOOKUP_RESOLVE_PREFIX_SEARCH flag is also set.

ui_id
: Input parameter containing the handle of a dialog box for **cmc_look_up** to present to help resolve processing questions.

count
: Input or output parameter containing a maximum name count. On input, this parameter specifies a pointer to the maximum number of names for which **cmc_look_up** can find addressing information. A value of zero indicates no maximum.

  On output, the *count* parameter specifies the location to which **cmc_look_up** writes the number of names that it actually writes to the location indicated by the *recipient_out* parameter. If no names are written, **cmc_look_up** writes zero to the *count* parameter.

recipient_out
: Output parameter pointing to the location to which **cmc_look_up** writes an array of one or more **CMC_recipient** structures containing addressing details for the recipients in the array passed in the *recipient_in* parameter.

look_up_extensions
: Input-output parameter pointing to an array of **CMC_extension** structures specifying function extensions. On input, this array contains MAPI extensions to the standard **cmc_look_up** function. A value of NULL for the *look_up_extensions* parameter indicates that the client application has no extensions for **cmc_look_up** and is expecting no extensions.

  On output, **cmc_look_up** writes to the array new information about the lookup operation. It writes NULL if it generates no output extensions.

Return Values

CMC_E_AMBIGUOUS_RECIPIENT
: The recipient name was ambiguous. Multiple matches were found.

CMC_E_FAILURE
: There was a general failure that does not fit the description of any other return value.

CMC_E_INSUFFICIENT_MEMORY
: Insufficient memory was available to complete the requested operation.

CMC_E_INVALID_FLAG

A flag set using a flags parameter was invalid.

CMC_E_INVALID_PARAMETER
A function parameter was invalid.

CMC_E_INVALID_SESSION_ID
The specified session handle is invalid or no longer valid (for example, after logging off).

CMC_E_INVALID_UI_ID
The specified user-interface identifier is invalid or no longer valid.

CMC_E_LOGON_FAILURE
The service, user name, or password specified was invalid, so logon cannot be completed.

CMC_E_NOT_SUPPORTED
The current implementation does not support the operation requested.

CMC_E_RECIPIENT_NOT_FOUND
One or more of the specified recipients were not found.

CMC_E_UNSUPPORTED_DATA_EXT
The current implementation does not support the data extension requested.

CMC_E_UNSUPPORTED_FLAG
The current implementation does not support the flag requested.

CMC_E_UNSUPPORTED_FUNCTION_EXT
The current implementation does not support the function extension requested.

CMC_E_USER_CANCEL
The operation was canceled by the user.

CMC_E_USER_NOT_LOGGED_ON
The user was not logged on and the CMC_LOGON_UI_ALLOWED flag was not set.

Remarks

A client application calls the **cmc_look_up** function to resolve a display name to a messaging address or to prompt the user to choose among multiple resolved names. A client can also use this function to display a dialog box for creation of recipient lists or to display recipient details.

The **cmc_look_up** function can write multiple addresses. Before it writes addressing information, it must allocate memory for the structure array to contain the information. When this memory is no longer needed, the client application should free the entire array with a call to **cmc_free**.

See Also

CMC_extension structure, **cmc_free** function, **CMC_recipient** structure

cmc_query_configuration

The **cmc_query_configuration** function determines configuration information for the installed CMC implementation.

CMC_return_code cmc_query_configuration (
  CMC_session_id *session***,**
  CMC_enum *item***,**
  CMC_buffer *reference***,**
  CMC_extension FAR * *config_extensions*
)

Parameters

session
Input parameter containing an opaque session handle that represents a MAPI session object indicating a session with a message service. If this parameter is set to zero, there is no session and the **cmc_query_configuration** function returns the default logon information to the buffer indicated by the *reference* parameter. If the *session* parameter is set to a nonzero value, **cmc_query_configuration** returns configuration information as determined by the session. If the value provided for the *session* parameter is invalid, **cmc_query_configuration** returns CMC_E_INVALID_SESSION_ID.

item
Input parameter containing an enumerated variable that identifies the configuration information required by the client application. The **cmc_query_configuration** function will write different values to the buffer or pointer that the *reference* parameter points to depending on the value of the *item* parameter. The caller must allocate this buffer or pointer before calling **cmc_query_configuration**. Possible *item* values are:

CMC_CONFIG_CHARACTER_SET
Indicates the *reference* parameter should be a pointer to a **CMC_object_identifier** structure array. The **cmc_query_configuration** function writes a pointer to the array of **CMC_object_identifier** structures that indicate the character sets supported by the current CMC implementation to the location pointed at by the *reference* parameter. The **cmc_query_configuration** function ends the array with a null **CMC_object_identifier** structure.

The first object identifier in the array is the default character set used if the calling client application or service provider does not specify one explicitly. The calling client or provider uses one of these object identifiers at logon to specify that the implementation use a different character set than the default. This array should be freed with **cmc_free**.

CMC_CONFIG_DEFAULT_SERVICE
Indicates the *reference* parameter should be a pointer to a CMC_string data type. The **cmc_query_configuration** function writes a pointer to the default message service name, if available, to the location indicated by the *reference* parameter. The **cmc_query_configuration** function writes NULL to this location if no default service name is available.

The calling client or provider can use this string, along with the one returned by CMC_CONFIG_DEFAULT_USER, as defaults when prompting the user for the service name, user name, and password. The string is returned in the CMC implementation's default character set.

CMC_CONFIG_DEFAULT_USER
Indicates the *reference* parameter should be a pointer to a CMC_string data type. The **cmc_query_configuration** function writes a pointer to the default user name, if available, to the location indicated by the *reference* parameter. The **cmc_query_configuration** function writes NULL to this location if no default user name is available.

The calling client or provider can use this string, along with the one returned by CMC_CONFIG_DEFAULT_SERVICE, as defaults when prompting the user for the provider name, user name, and password. The string is returned in the CMC implementation's default character set.

CMC_CONFIG_LINE_TERM

Indicates the *reference* parameter should be a pointer to a CMC_enum variable, which is set to CMC_LINE_TERM_CRLF if the line delimiter is a carriage return followed by a line feed, CMC_LINE_TERM_LF if the line delimiter is a line feed, or CMC_LINE_TERM_CR if the line delimiter is a carriage return.

CMC_CONFIG_REQ_PASSWORD

Indicates the *reference* parameter should be a pointer to a CMC_enum variable, which is set to CMC_REQUIRED_NO if the password is not required to log on, CMC_REQUIRED_OPT if the password is optional to log on, or CMC_REQUIRED_YES if the password is required to log on.

CMC_CONFIG_REQ_SERVICE

Indicates the *reference* parameter should be a pointer to a CMC_enum variable, which is set to CMC_REQUIRED_NO if the service name is not required to log on, CMC_REQUIRED_OPT if the service name is optional to log on, or CMC_REQUIRED_YES if the service name is required to log on.

CMC_CONFIG_REQ_USER

Indicates the *reference* parameter should be a pointer to a CMC_enum variable, which is set to CMC_REQUIRED_NO if the user name is not required to log on, CMC_REQUIRED_OPT if the user name is optional to log on, or CMC_REQUIRED_YES if the user name is required to log on.

CMC_CONFIG_SUP_COUNTED_STR

Indicates the *reference* parameter should be a pointer to a CMC_boolean variable, which is set to TRUE if the CMC_COUNTED_STRING_TYPE flag is supported during logon.

CMC_CONFIG_SUP_NOMKMSGREAD

Indicates the *reference* parameter should be a pointer to a CMC_boolean variable, which will be set to TRUE if the **cmc_read** function supports the CMC_DO_NOT_MARK_AS_READ flag.

CMC_CONFIG_UI_AVAIL

Indicates the *reference* parameter should be a pointer to a CMC_boolean variable, which will be set to TRUE if there is a dialog box provided by the CMC implementation.

CMC_CONFIG_VER_IMPLEM

Indicates the *reference* parameter should be a pointer to a CMC_uint16 variable, which is set to the version number for the implementation, multiplied by 100. For example, version 1.01 returns 101.

CMC_CONFIG_VER_SPEC

Indicates the *reference* parameter should be a pointer to a CMC_uint16 variable, which is set to the CMC specification version number for the implementation, multiplied by 100. For example, version 1.00 returns 100.

reference

Output parameter pointing to a buffer or pointer to which **cmc_query_configuration** writes configuration information. The value of *reference* depends on the value of *item*, as previously described.

config_extensions

Input-output parameter pointing to an array of **CMC_extension** structures specifying function extensions. On input, this array contains MAPI extensions to the standard **cmc_query_configuration** function. A value of NULL for the *config_extensions* parameter indicates that the client application has no extensions for **cmc_query_configuration** and is expecting no extensions.

On output, **cmc_query_configuration** writes to the array new information about the query configuration operation. It writes NULL if it generates no output extensions.

Return Values

CMC_E_FAILURE
There was a general failure that does not fit the description of any other return value.

CMC_E_INSUFFICIENT_MEMORY
Insufficient memory was available to complete the requested operation.

CMC_E_INVALID_ENUM
A **CMC_enum** value was invalid.

CMC_E_INVALID_PARAMETER
A function parameter was invalid. Either the *reference* parameter did not point to a writeable location, or the *config_extensions* parameter was badly formed.

CMC_E_NOT_SUPPORTED
The current implementation does not support the operation requested.

CMC_E_UNSUPPORTED_FUNCTION_EXT
The current implementation does not support the function extension requested.

Remarks

The client application must cast the *reference* parameter to the CMC_buffer type before calling **cmc_query_configuration**. The client application must allocate sufficient memory to contain the information passed in the *item* parameter. When this memory is no longer needed, the client should free this memory with whatever memory management routines it is using, or with a call to the **cmc_free** function if required by the previous explanation.

See Also

CMC_extension structure, **cmc_free** function, **cmc_read** function

cmc_read

The **cmc_read** function reads a specified message.

CMC_return_code cmc_read (
  CMC_session_id *session*,
  CMC_message_reference * *message_reference*,
  CMC_flags *read_flags*,
  CMC_message FAR * **FAR** * *message*,
  CMC_ui_id *ui_id*,
  CMC_extension FAR * *read_extensions*
)

Parameters

session
  Input parameter containing an opaque session handle that represents a MAPI session object indicating a session with a message service. If the value provided for the *session* parameter is invalid, the **cmc_read** function returns CMC_E_INVALID_SESSION_ID.

message_reference
  Input parameter pointing to a **CMC_message_reference** structure containing the message reference of the message to be retrieved. A NULL value for this parameter indicates that **cmc_read** should retrieve the first message in the mailbox. If the message reference is invalid, **cmc_read** returns CMC_E_INVALID_MESSAGE_REFERENCE.

read_flags
  Input parameter containing a bitmask of flags. The following flags can be set:

  CMC_DO_NOT_MARK_AS_READ
    Does not mark messages as read when they are returned. This flag also suppresses sending of receipt reports. The calling client application or service provider can query the implementation to see if it supports this flag by calling the **cmc_query_configuration** function.

  CMC_ERROR_UI_ALLOWED
    Displays a dialog box on encountering recoverable errors. If this flag is not set, **cmc_read** does not display a dialog box and returns an error value instead.

  CMC_MSG_AND_ATT_HDRS_ONLY
    Indicates that the *attach_filename* fields in the **CMC_message** structure returned in the *message* parameter will be undefined when **cmc_read** returns and should be ignored. This flag can be used to reduce the amount of data transferred, because the filenames will not be included in the transfer. If this flag is not set, the *attach_filename* fields are returned as usual.

    Note that if the CMC_MSG_TEXT_NOTE_AS_FILE value is set in the flags of the returned message, the first attachment contains the message text. In this case, **cmc_read** returns the *attach_filename* field for that attachment regardless of the setting of the CMC_MSG_AND_ATT_HDRS_ONLY flag.

  CMC_READ_FIRST_UNREAD_MESSAGE
    Returns the first message that is not marked as read. If this flag is not set, **cmc_read** should return the first message in the mailbox, whether it is marked as read or not. This flag can only be set when passing a null message reference to receive the first message in the mailbox.

message
  Output parameter pointing to the location to which **cmc_read** writes the **CMC_message** structure containing the message it has read. The function writes attachment data in files, and the **CMC_message** structure indicates the names of those files in its **attachments** member. If the client application has set the CMC_MSG_AND_ATT_HDRS_ONLY flag, the function does not indicate any attachment files.

ui_id

Input parameter containing the handle of a dialog box for **cmc_read** to present to help resolve processing questions.

read_extensions

Input-output parameter pointing to an array of **CMC_extension** structures specifying function extensions. On input, this array contains MAPI extensions to the standard **cmc_read** function. A value of NULL for the *read_extensions* parameter indicates that the client application has no extensions for **cmc_read** and is expecting no extensions.

On output, **cmc_read** writes to the array new information about the read operation. It writes NULL if it generates no output extensions.

Return Values

CMC_E_ATTACHMENT_OPEN_FAILURE
The specified attachment was found but could not be opened, or the attachment file could not be created.

CMC_E_ATTACHMENT_READ_FAILURE
The specified attachment was found and opened, but there was an error reading it.

CMC_E_ATTACHMENT_WRITE_FAILURE
The attachment file was created successfully, but there was an error writing it.

CMC_E_DISK_FULL
Insufficient disk space was available to complete the requested operation (this can refer to local or shared disk space).

CMC_E_FAILURE
There was a general failure that does not fit the description of any other return value.

CMC_E_INSUFFICIENT_MEMORY
Insufficient memory was available to complete the requested operation.

CMC_E_INVALID_FLAG
A flag set using a flags parameter was invalid.

CMC_E_INVALID_MESSAGE_REFERENCE
The specified message reference is invalid or no longer valid (for example, it has been deleted).

CMC_E_INVALID_PARAMETER
A function parameter was invalid.

CMC_E_INVALID_SESSION_ID
The specified session handle is invalid or no longer valid (for example, after logging off).

CMC_E_INVALID_UI_ID
The specified user-interface identifier is invalid or no longer valid.

CMC_E_TOO_MANY_FILES
The current implementation cannot support the number of files specified.

CMC_E_UNABLE_TO_NOT_MARK_READ
The current implementation cannot support the CMC_DO_NOT_MARK_AS_READ flag.

CMC_E_UNSUPPORTED_FLAG
The current implementation does not support the flag requested.

CMC_E_UNSUPPORTED_FUNCTION_EXT
The current implementation does not support the function extension requested.

Remarks

The **cmc_read** function only reads the first message in the mailbox if the client application passes a null message-reference value.

After processing, **cmc_read** writes the data from the message into the **CMC_message** structure. Unless the client application has set the flag CMC_DO_NOT_MARK_AS_READ on input, the message will be marked as read when **cmc_read** returns. If the client application has set the input flag CMC_MSG_AND_ATT_HDRS_ONLY, **cmc_read** writes only message and attachment headers on output.

The **cmc_read** function can write multiple addresses. Before it writes message information, it must allocate memory for the structure to contain that information. When this memory is no longer needed, the client application should free all structures in the array with a call to the **cmc_free** function.

See Also

CMC_extension structure, **cmc_free** function, **cmc_list** function, **CMC_message** structure, **CMC_message_reference** structure, **cmc_query_configuration** function

cmc_send

The **cmc_send** function sends a message.

CMC_return_code cmc_send (
  CMC_session_id *session***,**
  CMC_message FAR * *message***,**
  CMC_flags *send_flags***,**
  CMC_ui_id *ui_id***,**
  CMC_extension FAR s* *send_extensions*
)

Parameters

session
Input parameter containing an opaque session handle that represents a MAPI session object indicating a session with a message service. If the value provided for the *session* parameter is invalid, the **cmc_send** function returns CMC_E_INVALID_SESSION_ID.

message
Input parameter pointing to a **CMC_message** structure identifying the message to be sent. If the client application has not set the flag CMC_SEND_UI_REQUESTED in the *send_flags* parameter, the message structure must specify at least one primary (TO), carbon-copy (CC), or blind carbon-copy (BCC) recipient. All other structure members are optional. The **cmc_send** function ignores the **time_sent** and **message_reference** members.

send_flags
Input parameter containing a bitmask of flags. The following flags can be set:

CMC_COUNTED_STRING_TYPE
Indicates the string type the calling application or provider uses for CMC interactions is CMC_counted_string. If this flag is not set, the function treats all strings as null-terminated strings.

CMC_ERROR_UI_ALLOWED
Displays a dialog box on encountering recoverable errors. If this flag is not set, **cmc_send** does not display a dialog box and returns an error value instead.

CMC_LOGON_UI_ALLOWED
Displays a dialog box to prompt for logon if required. If this flag is not set, **cmc_send** does not display a dialog box and returns an error value if the caller does not supply enough information.

CMC_SEND_UI_REQUESTED
Displays a dialog box to prompt for recipients, message field information, and other sending options. If this flag is not set, **cmc_send** does not display a dialog box and the caller must specify at least one recipient.

ui_id
Input parameter containing the handle of a dialog box for **cmc_send** to present when resolving processing questions, prompting the user for additional information, or verifying provided information.

send_extensions
Input-output parameter pointing to an array of **CMC_extension** structures specifying function extensions. On input, this array contains MAPI extensions to the standard **cmc_send** function. A value of NULL for the *send_extensions* parameter indicates that the client application has no extensions for **cmc_send** and is expecting no extensions.

On output, **cmc_send** returns to the array new information about the send operation. It returns NULL if it generates no output extensions.

Return Values

CMC_E_AMBIGUOUS_RECIPIENT

The recipient name was ambiguous. Multiple matches were found.

CMC_E_ATTACHMENT_NOT_FOUND
The specified attachment was not found as specified.

CMC_E_ATTACHMENT_OPEN_FAILURE
The specified attachment was found but could not be opened, or the attachment file could not be created.

CMC_E_ATTACHMENT_READ_FAILURE
The specified attachment was found and opened, but there was an error reading it.

CMC_E_ATTACHMENT_WRITE_FAILURE
The attachment file was created successfully, but there was an error writing it.

CMC_E_COUNTED_STRING_UNSUPPORTED
The current implementation does not support the counted-string type.

CMC_E_FAILURE
There was a general failure that does not fit the description of any other return value.

CMC_E_INSUFFICIENT_MEMORY
Insufficient memory was available to complete the requested operation.

CMC_E_INVALID_FLAG
A flag set using a flags parameter was invalid.

CMC_E_INVALID_MESSAGE_PARAMETER
One of the parameters in the message was invalid.

CMC_E_INVALID_PARAMETER
A function parameter was invalid.

CMC_E_INVALID_SESSION_ID
The specified session handle is invalid or no longer valid (for example, after logging off).

CMC_E_INVALID_UI_ID
The specified user-interface identifier is invalid or no longer valid.

CMC_E_LOGON_FAILURE
The service, user name, or password specified were invalid, so logon cannot be completed.

CMC_E_RECIPIENT_NOT_FOUND
One or more of the specified recipients were not found.

CMC_E_TEXT_TOO_LARGE
The size of the text string passed to the current implementation is too large.

CMC_E_TOO_MANY_FILES
The current implementation cannot support the number of files specified.

CMC_E_TOO_MANY_RECIPIENTS
The current implementation cannot support the number of recipients specified.

CMC_E_UNSUPPORTED_DATA_EXT
The current implementation does not support the data extension requested.

CMC_E_UNSUPPORTED_FLAG
The current implementation does not support the flag requested.

CMC_E_UNSUPPORTED_FUNCTION_EXT
The current implementation does not support the function extension requested.

CMC_E_USER_CANCEL
The operation was canceled by the user.

CMC_E_USER_NOT_LOGGED_ON
The user was not logged on and the CMC_LOGON_UI_ALLOWED flag was not set.

Remarks

The **cmc_send** function can, at the client application's option, either use an interface, like a dialog box, to prompt the user for message creation or proceed without any user interaction. A successful return from this function does not necessarily imply recipient validation.

The client application can optionally provide recipient list, subject text, attachments, and note text for the message. If the client application does not provide the required message elements, the **cmc_send** function can prompt the user for them if a dialog box is available. If the client provides one or more recipients, the function can send the message without prompting the user. If the client provides optional parameters and requests a dialog box, the parameters provide the initial values for the dialog box.

The following conditions apply to the **CMC_message** structure members:

message_type
To specify an interpersonal message, use a pointer to the string "CMC:IPM". If the client application provides a pointer value of NULL or a pointer to an empty string, **cmc_send** uses the default string CMC:IPM.

subject
A pointer value of NULL indicates no subject text.

text_note
A pointer value of NULL indicates no message text. If the client application does pass a non-null value to indicate the message text that exceeds the limits of the service provider, the provider can demote the text to an attachment. Alternatively, it can cause **cmc_send** to return CMC_E_TEXT_TOO_LARGE.

recipients
A pointer value of NULL indicates no recipients. If the client application passes a non-null value to indicate recipients in excess of the number of recipients that the service provider allows per message, **cmc_send** returns CMC_E_TOO_MANY_RECIPIENTS.

Note that the **CMC_recipient** structure pointed to by **recipients** can include either the recipient's name, an address, or a name and address pair. If the client application specifies only a name, **cmc_send** resolves the name to an address using name resolution rules defined by the CMC implementation. If the client specifies only an address, **cmc_send** uses this address for delivery and for the recipient display name. Finally, if the client specifies a name and address pair, **cmc_send** does not resolve the name.

The **cmc_send** function does not require a recipient of type originator to send a message.

attachments
A pointer value of NULL indicates no attachments. If the client application passes a non-null value to indicate attachments in excess of the number of attachments that the service provider allows per message, **cmc_send** returns CMC_E_TOO_MANY_FILES.

The **cmc_send** function reads the attachment files before it returns. Thus the caller or user can freely change or delete attachment files after **cmc_send** returns without affecting the message.

message_flags
Bitmask of message flags. The following flag can be set:

CMC_MSG_TEXT_NOTE_AS_FILE
Indicates that the **text_note** member of the *message* parameter is ignored and the message text is contained in the file referred to by the first attachment. If this flag is set to zero, the message text is contained in the **text_note** member.

See Also

CMC_extension structure, **CMC_message** structure, **CMC_recipient** structure

cmc_send_documents

The **cmc_send_documents** function sends a document.

CMC_return_code cmc_send_documents (
  CMC_string *recipient_addresses*,
  CMC_string *subject*,
  CMC_string *text_note*,
  CMC_flags *send_doc_flags*,
  CMC_string *file_paths*,
  CMC_string *file_names*,
  CMC_string *delimiter*,
  CMC_ui_id *ui_id*
)

Parameters

recipient_addresses
Input parameter pointing to the address of the document recipient. When the client application specifies multiple recipients, it should separate the strings using the character specified by the *delimiter* parameter. The **cmc_send_documents** function assumes a recipient to be a primary recipient unless the address is prefixed by CC: (carbon copy) or BCC: (blind carbon copy). The TO: prefix can optionally be used with the primary recipient for consistency with the other recipient types. Passing NULL in the *recipient_addresses* parameter indicates that **cmc_send_documents** should present a dialog box to prompt for recipients.

subject
Input parameter pointing to the subject of the document. Passing NULL in the *subject* parameter indicates no subject text.

text_note
Input parameter pointing to the text note carried with the document. Passing NULL in the *text_note* parameter indicates no text note.

send_doc_flags
Input parameter containing a bitmask of flags used to control how documents are sent. The following flags can be set:

CMC_COUNTED_STRING_TYPE
Indicates the string type the calling application or provider uses for CMC interactions is CMC_counted_string. If this flag is not set, the function treats all strings as null-terminated strings.

CMC_ERROR_UI_ALLOWED
Displays a dialog box on encountering recoverable errors. If this flag is not set, **cmc_send_documents** does not display a dialog box and returns an error value instead.

CMC_FIRST_ATTACH_AS_TEXT_NOTE
Sends the first attachment as the message text. If this flag is not set, the *text_note* field contains the text note.

CMC_LOGON_UI_ALLOWED
Displays a dialog box to prompt for logon if required. If this flag is not set, **cmc_send_documents** does not display a dialog box and returns an error if the user does not supply enough information.

CMC_SEND_UI_REQUESTED
Displays a dialog box to prompt for recipients, message field information, and other sending options. If this flag is not set, **cmc_send_documents** does not display a dialog box but must specify at least one recipient.

file_paths
Input parameter pointing to the actual path for the attachment file. When the client application

specifies multiple paths, it should separate the names using the character indicated by the *delimiter* parameter.

attach_titles

Input parameter pointing to the title of the attachment displayed for the recipient. When the client application specifies multiple titles, it should separate the titles using the character indicated by the *delimiter* parameter.

delimiter

Input parameter pointing to a character used to delimit the names in the *file_paths*, *attach_titles*, and *recipient_addresses* strings. The client application should choose a character that is not used in operating system filenames or recipient names. This parameter cannot be NULL.

ui_id

Input parameter containing the handle of a dialog box for **cmc_send_documents** to present to help resolve processing questions or prompt the user for additional information as required.

Return Values

CMC_E_ATTACHMENT_NOT_FOUND

The specified attachment was not found as specified.

CMC_E_ATTACHMENT_OPEN_FAILURE

The specified attachment was found but could not be opened, or the attachment file could not be created.

CMC_E_ATTACHMENT_READ_FAILURE

The specified attachment was found and opened, but there was an error reading it.

CMC_E_ATTACHMENT_WRITE_FAILURE

The attachment file was created successfully, but there was an error writing it.

CMC_E_COUNTED_STRING_NOT_SUPPORTED

The current implementation does not support counted strings.

CMC_E_FAILURE

There was a general failure that does not fit the description of any other return value.

CMC_E_INSUFFICIENT_MEMORY

Insufficient memory was available to complete the requested operation.

CMC_E_INVALID_FLAG

A flag set using a flags parameter was invalid.

CMC_E_INVALID_PARAMETER

A function parameter was invalid.

CMC_E_INVALID_UI_ID

The specified user-interface identifier is invalid or no longer valid.

CMC_E_LOGON_FAILURE

The service, user name, or password specified was invalid, so logon cannot be completed.

CMC_E_RECIPIENT_NOT_FOUND

One or more of the specified recipients were not found.

CMC_E_TEXT_TOO_LARGE

The size of the text string passed to the current implementation is too large.

CMC_E_TOO_MANY_FILES

The current implementation cannot support the number of files specified.

CMC_E_TOO_MANY_RECIPIENTS

The current implementation cannot support the number of recipients specified.

CMC_E_UNSUPPORTED_FLAG

The current implementation does not support the flag requested.

CMC_E_USER_CANCEL

The operation was canceled by the user.

CMC_E_USER_NOT_LOGGED_ON
The user was not logged on and the CMC_LOGON_UI_ALLOWED flag was not set.

Remarks

The **cmc_send_documents** function is primarily useful for calls from a scripting language application, such as a spreadsheet application using macros, that cannot handle data structures. The **cmc_send_documents** function tries to establish a session without a logon dialog box. If this is not possible, it prompts the user for logon information to establish a session. Before the function returns, it closes the session.

Structures and Data Types

The following data structures and simple data types are used by the CMC implementation to pass information into and out of CMC functions. Wherever possible, you should use these types to maintain compatibility with different CMC implementations.The structures and data types are listed in alphabetical order.

BYTE

BYTE is an unsigned character data type that is binary data.

```
typedef unsigned char     BYTE;
```

CMC_attachment

A **CMC_attachment** structure contains a CMC message attachment.

```
typedef struct
{
     CMC_string     attach_title;
     CMC_object_identifier attach_type;
     CMC_string     attach_filename;
     CMC_flags attach_flags;
     CMC_extension FAR     *attach_extensions;
} CMC_attachment;
```

Members

attach_title
Specifies the optional title for the attachment, for example the original filename of the attachment.

attach_type
Object identifier that specifies the attachment type. Two attachment types have been defined for use by client applications:

CMC_ATT_OID_BINARY
Indicates data in a file is treated as binary data. This attachment type is the default.

CMC_ATT_OID_TEXT
Indicates data in a file is treated as a text string. This attachment type assumes that data exists in the character set for the session on input and maps to the character set for the session on output, if possible.

A NULL value for **attach_type** indicates an attachment of an undefined type.

attach_filename
Specifies the name of the file in which attachment content is located. The location of the file depends on the CMC implementation, which ensures access by the client application.

attach_flags
Bitmask of flags used to describe attachment options. The following flags can be set:

CMC_ATT_APP_OWNS_FILE
Indicates on output that the client application owns the attachment and is responsible for deleting it. This flag is ignored on input. If the flag is not set, it indicates on output that the CMC implementation owns the file and the client application can only read it.

CMC_ATT_LAST_ELEMENT
Identifies the last structure in an array of **CMC_attachment** structures. The structure with this flag set must be at the end of the array. If this flag is set to zero for any structure, that structure is not the last array element.

attach_extensions
Points to the first element in an array of **CMC_extension** structures, each of which is specific to an attachment. Attachment extensions exist to support graphic representation of the attachments in a message when it is displayed. The extensions contain the character positions for the graphic representations. A pointer value of NULL indicates that no extensions are present.

Remarks

A **CMC_message** structure, which contains information about a CMC message, contains a pointer to an array of one or more **CMC_attachment** structures defining attachments for the message, or is NULL if there are no attachments. The array elements should have the same order as the attachments in the message, and the last element in the array should have the CMC_ATT_LAST_ELEMENT flag set in its **attach_flags** member.

See Also

CMC_extension structure, **CMC_message** structure

CMC_boolean

CMC_boolean is a CMC unsigned integer data type that is a Boolean value.

```
typedef      CMC_uint16      CMC_boolean;
```

Remarks

The CMC_boolean data type can contain the symbolic constants CMC_TRUE and CMC_FALSE. The C interface denotes FALSE by using 0 and TRUE by using any other integer. CMC denotes CMC_FALSE by using 0, in line with C interface usage. However, CMC_TRUE is denoted specifically bythe integer 1.

CMC_buffer

CMC_buffer is a CMC data type that points to a memory storage location of an undefined type and size.

```
typedef      void *CMC_buffer;
```

CMC_counted_string

A **CMC_counted_string** structure is an optional structure supporting character sets that allow embedded null characters. The structure contains a counted string and explicitly defines the length of the string.

```
typedef struct {
     CMC_uint32     length;
     char       string[1];
} CMC_counted_string;
```

Members

length

Indicates the length, in bytes, of the string specified by the **string** member.

string

Indicates an array of characters that make up the string. CMC does not require the string to be null-terminated.

Remarks

If a client application uses counted strings instead of null-terminated strings, it must set the CMC_COUNTED_STRING_TYPE flag in the **cmc_logon** function's *logon_flags* parameter when logging on to a MAPI session through **cmc_logon**. The data pointed to by a string of type CMC_string is then assumed to be defined as a **CMC_counted_string** structure.

To determine the character set of a string, the CMC implementation looks at the information for a session identified by a CMC_session_id data type. CMC implementations always attempt to map all strings passed to the client application to the character set for the session. If no session has been created by a call to **cmc_logon**, CMC interprets the string by using the implementation's default character set.

See Also

CMC_string data type

CMC_enum

CMC_enum is a CMC data type that is an enumerated data value.

```
typedef      CMC_sint32          CMC_enum;
```

Remarks

A variable of this type contains a value selected from an enumeration.

CMC_extension

A **CMC_extension** structure contains a CMC data extension for use by the CMC API functions and data structures. A CMC data extension adds parameters to functions or members to data structures.

```
typedef struct {
     CMC_uint32 item_code;
     CMC_uint32 item_data;
     CMC_buffer item_reference;
     CMC_flags extension_flags;
} CMC_extension;
```

Members

item_code
Contains code that uniquely identifies an extension to a function or data structure. The **item_code** member is the mechanism for specifying the extension to be invoked. The client application puts the extension code in this member before calling CMC functions that use extensions. The possible extensions are:

CMC_X_COM_ATTACH_CHARPOS	CMC_X_COM_CAN_SEND_RECIP
CMC_X_COM_CONFIG_DATA	CMC_X_COM_PRIORITY
CMC_X_COM_RECIP_ID	CMC_X_COM_SAVE_MESSAGE
CMC_X_COM_SENT_MESSAGE	CMC_X_COM_SUPPORT_EXT
CMC_X_COM_TIME_RECEIVED	CMC_X_MS_ADDRESS_UI
CMC_X_MS_ATTACH_DATA	CMC_X_MS_FUNCTION_FLAGS
CMC_X_MS_MESSAGE_DATA	CMC_X_MS_SESSION_FLAGS

These extensions are identified by the extension identifiers CMC_XS_COM and CMC_XS_MS. For definitions of these extensions, see Data Extensions.

item_data
Contains item data for the extension. Depending on the value of **item_code**, the **item_data** member might contain the length of the item value, the item value itself, or other information about the item. The specification of the extension describes the interpretation of this member.

item_reference
Contains the item reference for the extension. This value is a pointer to the storage location of the item value. It is NULL if there is no related item storage. The specification of the extension describes the interpretation of this member.

extension_flags
Bitmask of extension flags. The following flags can be set:

CMC_EXT_LAST_ELEMENT
Identifies the last structure in an array of **CMC_extension** structures. The structure with this flag set must be at the end of the array. If this flag is set to zero for any structure, that structure is not the last array element.

CMC_EXT_OUTPUT
Indicates for an output extension that the extension structure contains a pointer to implementation-allocated memory that the client application must release with the **cmc_free**

function. If this flag is set to zero, the implementation has not allocated memory for the extension that the application needs to free. This flag is always set to zero for structure extensions.

CMC_EXT_REQUIRED
: Indicates an error value is returned if this extension cannot be supported. If this flag is set to zero, it enables the CMC implementation to provide any level of support, including no support, for the extension.

Remarks

Extensions are used to add functionality to the CMC API. For example, a client application can implement the **cmc_act_on** function to allow saving a partially completed message in the receive folder (the Inbox) for later updating and sending. To pass the structure defining this partially completed message to CMC and receive back the resulting message reference, the client application can use the CMC_X_COM_SAVE_MESSAGE extension.

An extension can be either an input extension or an output extension; that is, it can be passed either as input from a client application to CMC or as output from CMC to a client application. Whether the information contained in an extension is input or output is implied by the semantics of the particular extension in question and by the presence or absence of CMC_EXT_OUTPUT in the extension's **extension_flags** member after a call to a CMC function. For input extensions, the client application in question allocates memory for the extension structure and any other structures associated with the extension. For output extensions, storage for the extension result, if necessary, is allocated by a CMC function.

For output extensions, a client application must free storage allocated by CMC with calls to the **cmc_free** function. For each **CMC_extension** structure in the array, the caller must call **cmc_free** for the pointer in the **item_reference** member of the structure.

CMC does not require explicit release of a data extension structure, because CMC releases such structures along with the structures that contain them. For example, CMC implicitly releases the message extension array created by the **cmc_read** function when calling **cmc_free** for the enclosing **CMC_message** structure.

See Also

cmc_act_on function, **cmc_free** function, **CMC_message** structure, **cmc_read** function

CMC_flags

CMC_flags is a CMC data type that is a bitmask of flags.

```
typedef          CMC_uint32     CMC_flags;
```

Remarks

A bitmask of this type contains 32 flag bits. The CMC implementation reserves the upper 16 bits for definition by the CMC specification. Any unused bits among the upper 16 must be clear. The implementation reserves the lower 16 flag bits for definition by a CMC data extension.

The meanings of these CMC flags depend on the context in which the client application uses them. CMC reserves all undocumented flags. Unspecified flags should always be set to zero.

CMC_message

A **CMC_message** structure contains information about a CMC message.

```
typedef struct {
     CMC_message_reference     *message_reference;
     CMC_string     message_type;
     CMC_string subject;
     CMC_time      time_sent;
     CMC_string text_note;
     CMC_recipient *recipients;
     CMC_attachment     *attachments;
     CMC_flags message_flags;
     CMC_extension *message_extensions;
} CMC_message;
```

Members

message_reference
Points to the message reference, which is a counted string (that is, a **CMC_counted_string** structure). The message reference is a unique identifier for a message within a mailbox.

message_type
Points to a string that identifies the type of the message. Three different string identifiers are available:

- Object identifiers, which are used for message types identified by object identifiers as defined in *CCITT Recommendation X.208*.
- CMC registered values, which are used for message types defined in the CMC specification.
- Bilaterally defined values, which are used for message types that are unregistered. CMC does not ensure that bilaterally defined values are unique.

For the complete specification of CMC message types, see "Remarks."

subject
Points to a string describing the subject of the message.

time_sent
Contains the **CMC_time** structure containing the date and time when the client application submits the message to the CMC implementation.

text_note
Points to the string containing the message text. If the value of this member is NULL, there is no message text. If the CMC_MSG_TEXT_NOTE_AS_FILE flag is set for the **message_flags** member, the message text is in the first attachment to the message. For information on message text format, see "Remarks."

recipients
Points to the first element in an array of **CMC_recipient** structures defining the message recipients.

attachments
Points to the first element in an array of **CMC_attachment** structures defining the attachments to the message.

message_flags
Bitmask of message flags. The following flags can be set:

CMC_MSG_LAST_ELEMENT
Identifies the last element in an array of **CMC_message** structures. The structure with this flag set must be at the end of the array. If this flag is set to zero for any structure, that structure is not the last array element.

CMC_MSG_READ

Indicates that the message has been read. If this flag is set to zero, the message has not been read.

CMC_MSG_TEXT_NOTE_AS_FILE
: Indicates that the **text_note** member is ignored and the message text is contained in the file referred to by the first attachment. If this flag is set to zero, the message text is contained in the **text_note** member.

CMC_MSG_UNSENT
: Indicates that the client application has not sent the message, for example when the message is a draft. The sender can create such a message with the CMC_X_COM_SAVE_MESSAGE data extension. If this flag is set to zero, the client application has sent the message.

message_extensions
: Points to the first structure in an array of **CMC_extension** structures representing the message extensions.

Remarks

Message types are used to distinguish between various sorts of messages that might require different processing. For example, the message type might be used by the client application to determine how to display the message or by CMC to determine how to send the message.

CMC message types possible in the **message_type** member of a **CMC_message** structure can have the following formats and are case-insensitive.

```
"OID: " + object_identifier
"CMC: " + cmc_registered_value
"BLT: " + string
```

An object_identifier is a string containing a series of whitespace separated integers. See CMC_object_identifier for details.

The following is a list of message types

IPM
: Interpersonal message. An interpersonal message is a memo-like message containing a recipient list, an optional subject, an optional text note, and zero or more attachments. The **CMC_message** structure is optimized to accommodate a message with the registered value IPM.

IP RN
: Read notification for an interpersonal message. A read notification indicates the recipient has opened a message.

IP NRN
: Nonread notification for an interpersonal message. A nonread notification indicates a message has been removed from the recipient's mailbox without being opened. For instance, the service or user has discarded the message or it has been automatically forwarded to another recipient.

DR
: Delivery report. A delivery report indicates the service was able to deliver a message to its recipient.

NDR
: Nondelivery report. A nondelivery report indicates the message service was not able to deliver a message to its recipient.

Bilaterally defined values are arbitrary strings and are used to identify custom message types used by the client application.

As the syntax preceding indicates, the `OID:` type identifier indicates a type identified by an object identifier, the `CMC:` type identifier indicates a type identified by a CMC registered value, and the `BLT:` type identifier indicates a type identified by a bilaterally defined value. Following are examples of valid type identifiers:

```
OID: 1 2 840 113556 3 2 850
CMC: IPM
BLT: my special message type
```

You can format type identifiers as you choose; the CMC implementation also defines a canonical type identifier format that allows a client application to easily compare type identifier strings. The CMC implementation always returns type identifiers in this format, which guarantees the following:

- All tokens are separated with a single space.
- The type identifiers `OID:`, `CMC:`, and `BLT:` are returned in uppercase.

The CMC specification does not define what it will do with type identifier strings that are not in this format.

The formats of messages with the preceding registered values within a **CMC_message** structure depend on the messaging protocols employed by the underlying messaging system. Often, non-IPM messages take the form of a program-generated message, which follows a memo-like format similar to an IPM format but serves instead to convey information about a previously sent message.

Note The cmc_registered_value types correspond to X.400 message types; however, non-X.400 messaging systems can also use them. Thus, these message types are meant to apply generically and not specifically to X.400 services.

Some implementations only support the interpersonal message type (CMC: IPM). Some implementations might treat messages of types other than IPM as IPM messages or might generate an error for such messages.

For the **text_note** member, the format of the message text is a sequence of paragraphs, whether it is passed in memory or in a file. Each paragraph is terminated with the appropriate line terminator for the platform: CR (carriage return) for Macintosh, LF (linefeed) for UNIX, and CR/LF for MS-DOS and Windows. The CMC implementation can word-wrap long lines (paragraphs). There is no guarantee that paragraph formatting will remain constant when a message is saved and read back. For example, the **cmc_read** function can return a long paragraph as a series of shorter paragraphs.

See Also

CMC_attachment structure, **CMC_extension** structure, **CMC_recipient** structure, **cmc_send** function, **CMC_time** structure

CMC_message_reference

A **CMC_message_reference** structure is a **CMC_counted_string** structure containing a message reference, which is the identifier for a message within a message store.

```
typedef          CMC_counted_string     CMC_message_reference;
```

Remarks

The CMC implementation only guarantees the message reference to be valid for the life of the session. The message reference is specific to the message store; CMC does not guarantee that the reference has any correspondence to a message identifier used by the underlying messaging system. Only during the current session can the client application expect the message reference to refer to the same message.

CMC_message_summary

A **CMC_message_summary** structure contains a summary of a message. A message summary includes the type of the message, a reference to it for later retrieval, the size of the message, and other information. Message summaries are useful for previewing messages on low-bandwidth networks without actually retrieving the entire message.

```
typedef struct {
     CMC_message_reference     *message_reference;
     CMC_string     message_type;
     CMC_string     subject;
     CMC_time      time_sent;
     CMC_uint32     byte_length;
     CMC_recipient     *originator;
     CMC_flags     summary_flags;
     CMC_extension     *message_summary_extensions;
} CMC_message_summary;
```

Members

message_reference
Points to the message reference, a **CMC_counted_string** structure containing the mailbox identifier for a message. The message reference is unique within a mailbox.

message_type
Points to a string that identifies the type of the message. See **CMC_message** for details.

subject
Points to a string containing the subject of the message.

time_sent
Indicates a **CMC_time** structure containing the date and time when the client application submits the message to the CMC implementation.

byte_length
Indicates the message size, in bytes. The value should include the size of all the associated features of the message such as attachments and envelope and heading fields. A client application can supply an approximate message size, or if the message size is unknown or unavailable, the value CMC_LENGTH_UNKNOWN.

originator
Points to a **CMC_recipient** structure indicating the message sender.

summary_flags
Bitmask of message summary flags. The following flags can be set:

CMC_SUM_LAST_ELEMENT
Identifies the last structure in an array of **CMC_message_summary** structures. The structure with this flag set must be at the end of the array. If this flag is set to zero for any structure, that structure is not the last array element.

CMC_SUM_READ
Indicates that the message has been read. If this flag is set to zero, the message has not been read.

CMC_SUM_UNSENT
Indicates that the client application has not sent the message, for example when the message is a draft. If this flag is set to zero, the client application has sent the message.

message_summary_extensions
Points to the first structure in an array of **CMC_extension** structures representing the message-summary data extensions, if any.

See Also

CMC_extension structure

CMC_object_identifier

CMC_object_identifier is a data type that is a CMC object identifier string.

```
typedef          CMC_string     CMC_object_identifier;
```

Remarks

An identifier of this type is globally unique. Its syntax must match the format defined in *CCITT Recommendation X.208*. This syntax is:

```
object_identifier          ::= object_id_component*
object_id_component          ::= integer
```

Namely, the object identifier is a sequence of whitespace separated integers.

The following is an example of an object identifier:

```
1 2 840 113556 3 2 850
```

Note The format of the object identifier string is the same as that used in the OID message type. For more information on message types, see **CMC_message**.

See Also

CMC_message structure

CMC_recipient

A **CMC_recipient** structure contains information about a messaging user, either a message recipient or the message sender.

```
typedef struct {
     CMC_string     name;
     CMC_enum      name_type;
     CMC_string     address;
     CMC_enum role;
     CMC_flags recip_flags;
     CMC_extension *recip_extensions;
} CMC_recipient;
```

Members

name
: Points to a string that identifies the recipient or sender display name. When the CMC implementation resolves the name to an address, it determines whether it should interpret the name as the name of an individual first and then as the name of a group if the individual name is not found, or vice versa, according to the value of the **name_type** member.

name_type
: Enumeration that indicates whether the structure contains information for a message recipient or a message sender. Possible values are:

  CMC_TYPE_GROUP
  : Indicates the recipient or sender name belongs to a distribution list.

  CMC_TYPE_INDIVIDUAL
  : Indicates the recipient or sender name belongs to an individual messaging user.

  CMC_TYPE_UNKNOWN
  : Indicates an unknown recipient or originator name.

  The **name_type** member is meaningful only if the **name** member is present. The CMC implementation sets **name_type** on output. On input, the **name_type** information can be used by the addressing mechanism in the message service to optimize resolution of the name.

address
: Points to a recipient or sender address string in a format recognized by the underlying messaging system. CMC does not define the format of the string. This member therefore accommodates any string notations supported by the CMC implementation, as configured at installation.

role
: Enumeration that indicates the role of the message recipient or sender. Possible values are:

  CMC_ROLE_AUTHORIZING_USER
  : Indicates the user authorizing the message, in cases of messages sent by proxy or in the name of another user.

  CMC_ROLE_BCC
  : Indicates a blind carbon copy (BCC) recipient.

  CMC_ROLE_CC
  : Indicates a carbon copy (CC) recipient.

  CMC_ROLE_ORIGINATOR
  : Indicates the sender of the message.

  CMC_ROLE_TO
  : Indicates a primary recipient.

recip_flags
: Bitmask of recipient flags. The following flags can be set:

CMC_RECIP_IGNORE
Indicates that CMC should ignore the specified recipient. This flag is useful for reusing an incoming message's recipient list for a reply. If this flag is set to zero, it indicates that the recipient should not be ignored.

CMC_RECIP_LAST_ELEMENT
Identifies the last structure in an array of **CMC_recipient** structures. The structure with this flag set must be at the end of the array. If this flag is set to zero for any structure, that structure is not the last array element.

CMC_RECIP_LIST_TRUNCATED
Indicates that CMC has not written all recipient or originator structures requested. The client application uses this flag only for the **cmc_look_up** function when the complete list of recipients matching the search name cannot be written. The function only sets this flag in the last structure in the array of **CMC_recipient** structures. If the flag is set to zero, **cmc_look_up** has written a complete recipient array.

recip_extensions
Points to the first structure in an array of **CMC_extension** structures that contain the recipient or sender data extensions, if any.

Remarks

If the underlying messaging system does not support carbon copy recipients, CMC can convert such a recipient to a primary recipient. Services that cannot support blind carbon-copy recipients should reject messages containing them. If a user designates the same recipient in more than one role, the client application should place multiple recipient entries in the recipient list, each differing from the others in role.

On output, the CMC implementation writes an array of **CMC_recipient** structures in a specific order. The message sender's structure should be the first element in the array, followed by the primary, carbon copy, and blind carbon-copy recipient structures grouped together in that order. If there is an authorizing user structure, it should be the final element in the array. The CMC implementation does not require ordering of the **CMC_recipient** structures on input.

See Also

CMC_extension structure, **cmc_look_up** function

CMC_return_code

CMC_return_code is a data type that is a 32-bit value returned by a CMC function.

```
typedef      CMC_uint32          CMC_return_code;
```

Remarks

A nonzero return value for a CMC function indicates an error and is associated with one of the defined CMC return values. A return value of zero for a function indicates success. The CMC implementation reserves values in the low-order 16 bits of the return value for standard CMC-defined error values. The high-order 16 bits of the return value are reserved for error values that are defined specifically for the implementation.

CMC client applications can resolve errors within the scope of the CMC implementation. For example, a client application can resolve errors by prompting the user with a dialog box defined through the CMC user interface. If the error remains unresolved after the dialog box has closed, CMC sets the CMC_ERROR_UI_DISPLAYED flag in the return value to indicate that a dialog box regarding the error has already been displayed.

CMC_session_id

CMC_session_id is a data type that is a 32-bit CMC session handle.

```
typedef      uint32          CMC_session_id;
```

Remarks

The context identified by the session handle contains information about the current session, such as the character set in use and handles for any open sessions with underlying message services. The **cmc_logon** function creates the CMC session handle, and the **cmc_logoff** function invalidates it.

See Also

cmc_logoff function, **cmc_logon** function

CMC_string

CMC_string is a CMC data type that is a pointer to a character string.

```
typedef       char      *CMC_string;
```

Remarks

By default, the CMC implementation interprets the string that this data type points to as a null-terminated array of characters. The chosen character set determines the width of a character and the corresponding null-terminating character.

If a client application uses counted strings instead of null-terminated strings, it must set the CMC_COUNTED_STRING_TYPE flag in the *logon_flags* parameter when logging onto a MAPI session through **cmc_logon**. The data pointed to by a string of type CMC_string is then defined as a **CMC_counted_string** structure.

To determine the character set of characters in the string, the CMC implementation looks at the character set in the session attributes, which are chosen by the client application when the call to **cmc_logon** is made. CMC always attempts to map all strings passed to the client application to the character set for the session. If the client application has not called **cmc_logon** and is using an implicit session, the CMC implementation interprets input strings by using its default character set.

See Also

CMC_counted_string structure, **cmc_logon** function

CMC_time

A **CMC_time** structure contains a time value in CMC-compatible form for use in a message.

```
typedef struct{
     CMC_sint8     second;
     CMC_sint8     minute;
     CMC_sint8     hour;
     CMC_sint8     day;
     CMC_sint8     month;
     CMC_sint8     year;
     CMC_sint8     isdst;
     CMC_sint8      unused1;
     CMC_sint16     tmzone;
     CMC_sint16     unused2;
} CMC_time;
```

Members

second
Indicates seconds; possible values range from 0 through 59.

minute
Indicates minutes; possible values range from 0 through 59.

hour
Indicates hours since midnight; possible values range from 0 through 23.

day
Indicates day of the month; possible values range from 1 through 31.

month
Indicates months since January; possible values range from 0 through 11.

year
Indicates years since 1900.

isdst
Value for daylight saving time. A nonzero value means daylight saving time is in force.

unused1
Reserved. Do not use.

tmzone
Indicates time zone, measured in minutes relative to Greenwich mean time. The value CMC_NO_TIMEZONE indicates that time zone information is not available.

unused2
Reserved. Do not use.

Remarks

The CMC time implementation is based on the assumption that all time values reflect the appropriate local time. For example, the **time_sent** members in the **CMC_message** and **CMC_message_summary** structures reflect the local time of the sender's location.

See Also

CMC_message structure, **CMC_message_summary** structure

CMC_ui_id

CMC_ui_id is a data type that is a CMC user interface handle.

```
typedef CMC_uint32          CMC_ui_id;
```

Remarks

The CMC implementation uses a data value of this type for passing user interface information to CMC functions. For example, in a Windows-based operating environment, the parent window handle for the client application is a data value of this type.

A value of NULL for CMC_ui_id is always valid. The CMC implementation defines a default behavior. For example, some implementations might treat NULL as a request to use a user interface defined within the implementation, while other implementations might not have any internally defined interface at all.

CMC_X_COM_configuration

A **CMC_X_COM_configuration** structure contains configuration data written by the **cmc_query_configuration** function for the CMC_X_COM_CONFIG_DATA data extension.

```
typedef struct {
     CMC_uint16          ver_spec;
     CMC_uint16          ver_implem;
     CMC_object_identifier     *character_set;
     CMC_enum               line_term;
     CMC_string          default_service;
     CMC_string          default_user;
     CMC_enum               req_password;
     CMC_enum               req_service;
     CMC_enum               req_user;
     CMC_boolean          ui_avail;
     CMC_boolean          sup_nomkmsgread;
     CMC_boolean          sup_counted_str;
} CMC_X_COM_configuration;
```

Members

ver_spec
Contains a CMC specification version number.

ver_implem
Contains a CMC version number multiplied by 100. For example, version 1.00 is represented as 100.

character_set
Points to a **CMC_object_identifier** structure.

line_term
Enumeration that indicates the type of line delimiter for the message text in a CMC_message. Possible values are:

CMC_LINE_TERM_CRLF
Indicates the line delimiter is a carriage return followed by a line feed.

CMC_LINE_TERM_LF
Indicates the line delimiter is a line feed.

CMC_LINE_TERM_CR
Indicates the line delimiter is a carriage return.

default_service
Points to a string identifying the default message service.

default_user
Points to a string identifying the default user name of the user accessing the CMC implementation. This default can be used when prompting the user for a user name.

req_password
Enumeration that indicates if a password is required to access the service. Possible values are:

CMC_REQUIRED_NO
Indicates no password is required.

CMC_REQUIRED_YES
Indicates a password is required.

CMC_REQUIRED_OPT
Indicates a password is optional.

req_service

Enumeration that indicates if the message service name is required for logon. Possible values are:

CMC_REQUIRED_NO
: Indicates no service name is required.

CMC_REQUIRED_YES
: Indicates a service name is required.

CMC_REQUIRED_OPT
: Indicates a service name is optional.

req_user

Enumeration that indicates if the messaging user name is required for logon. Possible values are:

CMC_REQUIRED_NO
: Indicates no user name is required.

CMC_REQUIRED_YES
: Indicates a user name is required.

CMC_REQUIRED_OPT
: Indicates a user name is optional.

ui_avail

Boolean value that is TRUE if the CMC implementation in use provides a user interface and FALSE otherwise.

sup_nomkmsgread

Boolean value that indicates whether the **cmc_read** function supports not marking messages as read. The value is TRUE if the **cmc_read** function supports the CMC_DO_NOT_MARK_AS_READ flag and FALSE otherwise.

sup_counted_str

Boolean value that indicates whether the **cmc_logon** function supports the use of the CMC_counted_string type. The value is TRUE if the **cmc_logon** function supports the CMC_COUNTED_STRING_TYPE flag and FALSE otherwise.

Remarks

The **cmc_query_configuration** function writes a value into the buffer pointed to by its *reference* parameter. That value is a copy of a particular member of the CMC implementation's **CMC_X_COM_configuration** structure, depending on the value the client application passes for the *item* parameter. For example, to see whether the CMC implementation defines its own user interface for logon and error resolution, the client application should use CMC_CONFIG_UI_AVAIL as the value of *item*. When **cmc_query_configuration** returns, the CMC_buffer pointed to by *reference* will contain TRUE or FALSE to indicate whether an interface is available. The client application must ensure that the *reference* parameter points to a buffer of sufficient size to hold the type of data that the *item* parameter specifies.

See Also

cmc_free function, **cmc_logon** function, **CMC_object_identifier** data type, **cmc_query_configuration** function, **cmc_read** function, CMC_X_COM_CONFIG_DATA extension

CMC_X_COM_support

A **CMC_X_COM_support** structure contains information about MAPI support for a particular CMC data extension or extension set, and is used in an array pointed to by the **item_reference** member of a **CMC_extension** structure.

```
typedef struct {
     CMC_uint32 item_code;
     CMC_flags flags;
} CMC_X_COM_support;
```

Members

item_code

Contains code for the CMC data extension whose support the client application is querying about. The client application sets this member during calls to the **cmc_logon** and **cmc_query_configuration** functions. The possible extensions are:

- CMC_XS_COM
- CMC_X_COM_ATTACH_CHARPOS
- CMC_X_COM_CAN_SEND_RECIP
- CMC_X_COM_CONFIG_DATA
- CMC_X_COM_PRIORITY
- CMC_X_COM_RECIP_ID
- CMC_X_COM_SAVE_MESSAGE
- CMC_X_COM_SENT_MESSAGE
- CMC_X_COM_SUPPORT_EXT
- CMC_X_COM_TIME_RECEIVED
- CMC_XS_MS
- CMC_X_MS_ADDRESS_UI
- CMC_X_MS_ATTACH_DATA
- CMC_X_MS_FUNCTION_FLAGS
- CMC_X_MS_MESSAGE_DATA
- CMC_X_MS_SESSION_FLAGS

flags

Bitmask of extension code flags. The following flags can be set:

CMC_X_COM_SUP_EXCLUDE
: On input this flag removes the item represented by the value of the **item_code** member from consideration when deciding whether the implementation supports an extension set. If this flag is set on input for the **cmc_logon** function, the implementation will not attach the item to extension structures for this session even if other entries request it. If this flag is set on input, then none of the remaining flags in this list will be set on output.

CMC_X_COM_SUPPORTED
: Indicates the CMC implementation supports the item represented by the **item_code** member. For whole extension sets (for example, CMC_XS_COM), this flag indicates the CMC implementation supports all the function and structure extensions in the set.

CMC_X_COM_NOT_SUPPORTED
: Indicates the CMC implementation does not support the item represented by the **item_code** member. If this flag applies to a whole extension set containing both function and structure extensions, it indicates the CMC implementation does not support some or all function and structure extensions in the set. If this flag applies to a structure extension or an extension set containing structure extensions, it indicates the CMC implementation will not attach the structure extensions to structures for this session.

CMC_X_COM_DATA_EXT_SUPPORTED

Indicates the CMC implementation supports all requested structure extensions for an extension set. The client application must request function extensions separately. When the **cmc_logon** function returns this flag, the implementation will attach structure extensions to structures for this session.

CMC_X_COM_FUNC_EXT_SUPPORTED

Indicates the CMC implementation supports all requested function extensions for an extension set. The client application must request structure extensions separately. Unlike the CMC_X_COM_SUPPORTED flag, if the **cmc_logon** function returns this flag, structure extensions will not be attached to structures for this session and must be requested separately.

See Also

CMC_extension structure,**cmc_logon** function, CMC_X_COM_SUPPORT_EXT extension

CMC_X_MS_ATTACH

CMC_X_MS_ATTACH data structures are used by the CMC_X_MS_ATTACH_DATA and CMC_X_MS_ATTACH_MESSAGE extensions to support message attachments.

```
typedef struct {
     CMC_message_reference FAR * message;
     CMC_uint32 id;
     CMC_buffer object;
} CMC_X_MS_ATTACH;
```

Data Extensions

The functionality of the CMC data structures and functions can be augmented through the use of CMC data extensions. Data extensions are used to add additional fields to data structures and additional parameters to a function.

A standard generic data structure, **CMC_extension**, specifies the item code, item data, item reference, and a bitmask of flags. The item code is the name of the extension and is used to identify it. The item data, depending on the item code, contains either the length of the item value, the item value itself, or other information about the item. The item reference points to where the extension value is stored or is NULL if there is no related item storage. The flags are set to zero or to values that describe options for the extension. To use an extension in a CMC function, a client application sets the item code member of a **CMC_extension** structure to a valid name and passes the structure as part of the parameter list.

Extensions that are additional parameters to a function can be either input or output parameters. If the extension is passed as an input parameter, the calling client application or service provider allocates memory for the **CMC_extension** structure and any other structures that are associated with the extension. If the extension is passed as an output parameter, CMC allocates the storage for the extension result and the caller frees it by calling the **cmc_free** function.

The following extensions make up the CMC common extension set, the extensions that are common to most CMC implementations but are not in the CMC base specification. The extensions are listed in alphabetical order. Each reference entry describes the purpose of the extension.

CMC_XS_COM

CMC_XS_COM is a CMC extension identifier used for all extensions in the common extension set, that is the extensions that are common to most CMC implementations but are not in the CMC base specification. For a complete list of common extensions, see CMC_X_COM_support.

See Also

cmc_logon function, **cmc_query_configuration** function, CMC_X_COM_SUPPORT_EXT extension

CMC_X_COM_ATTACH_CHARPOS

CMC_X_COM_ATTACH_CHARPOS is a CMC extension that supports display of a graphical representation (usually an icon) of an attachment in the message text. The extension holds the character position for the representation.

Input Usage

item_data
Zero-based character offset of the attachment within the message text. Note that this offset is a character offset, not a byte offset (an important distinction when multibyte character sets are in use).

item_reference
NULL.

extension_flags
All flags used with the **CMC_extension** structure are valid. No further flags are defined.

Output Usage

item_data
Zero-based character offset of the attachment.

item_reference
Unchanged.

extension_flags
Unchanged.

Adds to

The **CMC_attachment** structure.

Remarks

At logon, CMC passes the CMC_X_COM_ATTACH_CHARPOS **item_data** member and any flags in a **CMC_X_COM_support** structure. Doing so indicates that CMC supports the CMC_X_COM_ATTACH_CHARPOS extension and that the client application can attach a **CMC_extension** containing an item code and a graphic representation to the **CMC_attachment** structure during the session.

See Also

CMC_attachment structure, **CMC_extension** structure, **CMC_X_COM_support** structure

CMC_X_COM_CAN_SEND_RECIP

CMC_X_COM_CAN_SEND_RECIP is a CMC extension that checks whether the message service is ready to send to the specified recipient.

Input Usage

item_data
Zero.

item_reference
NULL. On input to the **cmc_look_up** function, the *recipient_in* parameter contains the recipient about which the message service is being queried. If there is more than one recipient passed in *recipient_in*, **cmc_look_up** only looks at the first recipient.

extension_flags
Indicates all flags used with the **CMC_extension** structure are valid. No further flags are defined.

Output Usage

item_data
Bitmask of extension flags. The following flags can be set:

CMC_X_COM_DEFER
Indicates the service provider will accept the message but defer it until a transport provider is ready.

CMC_X_COM_NOT_READY
Indicates no transport provider is available for the specified recipient type.

CMC_X_COM_READY
Indicates the message can be sent immediately.

item_reference
Unchanged.

extension_flags
Unchanged.

Adds to

The **cmc_look_up** function.

See Also

CMC_extension structure, **cmc_look_up** function

CMC_X_COM_CONFIG_DATA

CMC_X_COM_CONFIG_DATA is a CMC extension that obtains all available configuration information.

Input Usage

item_data
Zero.

item_reference
NULL.

extension_flags
Indicates all flags used with the **CMC_extension** structure are valid. No further flags are defined.

Output Usage

item_data
Unchanged.

item_reference
Points to a **CMC_X_COM_configuration** structure containing all the information available from the **cmc_query_configuration** function.

extension_flags
Bitmask of extension flags. If the call successfully returns a structure, the function sets the CMC_EXT_OUTPUT flag.

Adds to

The **cmc_query_configuration** function.

Remarks

The buffer pointed to by the **item_reference** member should be freed with one call to **cmc_free**.

See Also

CMC_extension structure, **cmc_query_configuration** function, **CMC_X_COM_configuration** structure

CMC_X_COM_PRIORITY

CMC_X_COM_PRIORITY is a CMC extension indicating message priority.

Input Usage

item_data
Bitmask of extension flags. The following flags can be set:

CMC_X_COM_LOW
Indicates a low priority for the message.

CMC_X_COM_NORMAL
Indicates a normal priority for the message.

CMC_X_COM_URGENT
Indicates a high priority for the message.

item_reference
NULL.

extension_flags
Indicates all flags used with the **CMC_extension** structure are valid. No further flags are defined.

Output Usage

item_data
Bitmask of extension flags that describe the urgency of the message. The following flags can be set:

CMC_X_COM_LOW
Indicates a low priority for the message.

CMC_X_COM_NORMAL
Indicates a normal priority for the message.

CMC_X_COM_URGENT
Indicates a high priority for the message.

item_reference
Unchanged.

extension_flags
Unchanged.

Adds to

The **CMC_message** and **CMC_message_summary** structures.

Remarks

At logon, the caller passes CMC_X_COM_PRIORITY in the **CMC_X_COM_SUPPORT_EXT** extension to indicate that this extension should be attached to the **CMC_message** and **CMC_message_summary** structures during the session.

See Also

CMC_extension structure, **CMC_message** structure, **CMC_message_summary** structure, **CMC_X_COM_SUPPORT_EXT** extension.

CMC_X_COM_RECIP_ID

CMC_X_COM_RECIP_ID is a CMC extension that adds a unique recipient identifier to a **CMC_recipient** structure. A recipient identifier is an opaque object that the underlying messaging system uses to uniquely represent a recipient, and can be thought of as the result of resolving an address.

Input Usage

item_data
Indicates the length in bytes of the recipient identifier.

item_reference
Points to the recipient identifier.

extension_flags
Indicates all flags used with the **CMC_extension** structure are valid. No further flags are defined.

Output Usage

item_data
Indicates the length in bytes of the recipient identifier.

item_reference
Points to the recipient identifier.

extension_flags
Unchanged.

Adds to

The **CMC_recipient** structure.

Remarks

The CMC implementation handles the CMC_X_COM_RECIP_ID extension during recipient name resolution. The client application can use this to avoid further name resolution during sending. Some message services allow the client application to reuse the recipient identifier returned by a previous call to **cmc_look_up**. The client application can attach it to a recipient structure that the service provider would otherwise try to resolve.

At logon, the client application passes the **item_code** member of CMC_X_COM_RECIP_ID in the CMC_X_COM_SUPPORT_EXT array to request that the CMC implementation should attach recipent identifiers to **CMC_recipient** structures during the session.

See Also

CMC_extension structure, **CMC_recipient** structure, **CMC_X_COM_support** structure

CMC_X_COM_SAVE_MESSAGE

CMC_X_COM_SAVE_MESSAGE is a CMC extension that saves a message (that is, a **CMC_message** structure) to the receive folder (the Inbox).

Input Usage

item_data
Zero.

item_reference
Points to the **CMC_message** structure to save to the receive folder. To indicate that an unsent message has not been sent, the CMC implementation sets the CMC_MSG_UNSENT flag in the **message_flags** member of the **CMC_message** structure. To indicate that the operation to be performed (save to receive folder) is contained in a CMC_X_COM_SAVE_MESSAGE extension, the **cmc_act_on** function's *operation* parameter must be set to the CMC_ACT_ON_EXTENDED flag.

extension_flags
Indicates all flags used with the **CMC_extension** structure are valid. No further flags are defined. To indicate that the CMC implementation should carry out a save action rather than a deletion, the flag CMC_EXT_REQUIRED must be set in the **extension_flags** member.

Output Usage

item_data
Unchanged.

item_reference
Points to the message reference of the message that was saved to the receive folder (the Inbox). The client application must free the message reference using the **cmc_free** function.

extension_flags
If **cmc_act_on** has successfully saved the message and returned the message reference, the CMC_EXT_OUTPUT flag is set.

Adds to

The **cmc_act_on** function.

See Also

cmc_act_on function, **CMC_extension** structure, **CMC_message** structure, **CMC_message_reference** structure

CMC_X_COM_SENT_MESSAGE

CMC_X_COM_SENT_MESSAGE is a CMC extension that creates a **CMC_message** structure containing information for the message just sent.

Input Usage

item_data
Zero.

item_reference
NULL.

extension_flags
Indicates all flags used with the **CMC_extension** structure are valid. No further flags are defined.

Output Usage

item_data
Unchanged.

item_reference
Points to a **CMC_message** structure containing information for the message just sent. The client application must free this pointer by calling the **cmc_free** function.

extension_flags
Indicates **cmc_send** has successfully put a pointer to a **CMC_message** structure in the *item_reference* parameter if the CMC_EXT_OUTPUT flag is set.

Adds to

The **cmc_send** function.

Remarks

CMC_X_COM_SAVE_MESSAGE is used to obtain information about a message that has just been sent when some or all of the members of the **CMC_message** structure are set by the user interface defined by the CMC implementation rather than by the client application before the **cmc_send** call.

See Also

CMC_extension structure, **cmc_free** function, **CMC_message** structure, **cmc_send** function

CMC_X_COM_SUPPORT_EXT

CMC_X_COM_SUPPORT_EXT is a CMC extension that client applications use to query the CMC implementation about the extensions it supports. If the implementation supports any extensions, it must support CMC_X_COM_SUPPORT_EXT.

Input Usage

item_data
Indicates the number of items in an array pointed to by the **item_reference** member.

item_reference
Points to the first element in an array of **CMC_X_COM_support** structures listing extensions the client application requests the CMC implementation to support. See **CMC_X_COM_support** for details.

extension_flags
Indicates all flags used with CMC are valid. No further flags are defined.

Output Usage

item_data
Unchanged.

item_reference
Points to an array of **CMC_X_COM_support** structures where the CMC implementation sets the flags in the **flags** member to indicate support for the requested extensions. The implementation does not set these flags if CMC_X_COM_SUP_EXCLUDE was set on input. See **CMC_X_COM_support** for details.

extension_flags
Unchanged.

Adds to

The **cmc_query_configuration** and **cmc_logon** functions.

Remarks

Client applications can use this extension before establishing a session to get preliminary information about support before logging on. When a client application uses the extension with **cmc_logon**, it indicates which data extensions the client wants added to the data structures for the session.

Note The CMC implementation supports different extensions, based on the service with which the client application creates a session. Thus client applications should use CMC_X_COM_SUPPORT_EXT at logon to verify extension support.

See Also

CMC_extension structure, **cmc_logon** function, **cmc_query_configuration** function, **CMC_X_COM_support** structure

CMC_X_COM_TIME_RECEIVED

CMC_X_COM_TIME_RECEIVED is a CMC extension that provides a **CMC_time** structure containing the delivery time of a message.

Input Usage

CMC ignores this extension on input.

Output Usage

item_data
Zero.

item_reference
Points to the **CMC_time** structure containing the time the message was received.

extension_flags
NULL.

Adds to

The **CMC_message** and **CMC_message_summary** structures.

Remarks

At logon, the client application passes the CMC_X_COM_TIME_RECEIVED extension in the CMC_X_COM_SUPPORT_EXT array to indicate that the CMC implementation should attach this extension to **CMC_message** and **CMC_message_summary** structures during the session.

See Also

CMC_extension structure, **CMC_message** structure, **CMC_message_summary** structure, **CMC_time** structure, **CMC_X_COM_support** structure

CMC_XS_MS

CMC_XS_MS is a CMC extension identifier used for all extensions in the Microsoft extension set. For a full list of Microsoft extensions, see CMC_X_COM_support.

See Also

cmc_logon function, **cmc_query_configuration** function, CMC_X_COM_SUPPORT_EXT extension

CMC_X_MS_ADDRESS_UI

CMC_X_MS_ADDRESS_UI is a CMC extension that adds options to the address-book dialog box.

Input Usage

item_data
Indicates the number of edit boxes in the dialog box.

item_reference
Points to an array of two **CMC_string** data types. The first string is the caption for the address-book dialog box. To not provide a label, the client application sets the label string to NULL. The second string is a label for the recipient well if there is only one recipient well. If there are multiple recipient wells (for example, TO: and CC:) this string is ignored.

extension_flags
Indicates all flags used with the **CMC_extension** structure are valid. No further flags are defined.

Output Usage

item_data
Unchanged.

item_reference
Unchanged.

extension_flags
Unchanged.

Adds to

The **cmc_look_up** function.

See Also

CMC_extension structure, **cmc_look_up** function

CMC_X_MS_ATTACH_DATA

CMC_X_MS_ATTACH_DATA is a CMC extension that contains bitmasks of flags used to provide data on message attachments.

Input Usage

item_data
Bitmask of attachment flags. The following flags can be set:

CMC_X_MS_ATTACH_OLE
Indicates the CMC implementation supports OLE attachments.

CMC_X_MS_ATTACH_OLE_STATIC
Indicates the CMC implementation supports static OLE attachments.

CMC_X_MS_ATTACH_MESSAGE
Indicates the CMC implementation supports attached messages.

item_reference
Object identifier for the attachment encoding. This member can point to a **CMC_X_MS_ATTACH** structure, to allow a message reference to be added as an embedded message in the current message.

extension_flags
Indicates all flags used with the **CMC_extension** structure are valid. No further flags are defined.

Output Usage

item_data
Bitmask of attachment flags. CMC can set the following flags:

CMC_X_MS_ATTACH_OLE
Indicates the CMC implementation supports OLE attachments.

CMC_X_MS_ATTACH_OLE_STATIC
Indicates the CMC implementation supports static OLE attachments.

CMC_X_MS_ATTACH_MESSAGE
Indicates the CMC implementation supports attached messages.

item_reference
Object identifier for the attachment encoding.

extension_flags
Unchanged.

Adds to

The **CMC_attachment** structure.

See Also

CMC_attachment structure, **CMC_extension** structure

CMC_X_MS_FUNCTION_FLAGS

CMC_X_MS_FUNCTION_FLAGS is a CMC extension that contains a bitmask of flags used for CMC functions that serve purposes other than session handling.

Input Usage

item_data
Bitmask of extension flags. The following flags can be set:

CMC_X_MS_AB_NO_MODIFY
Disallows changes to the address book through the address-book details dialog box if a user interface is being used with the **cmc_look_up** function.

CMC_X_MS_LIST_GUARANTEE_FIFO
Causes the CMC implementation to return messages in date order when the **cmc_list** function is called.

CMC_X_MS_READ_BODY_AS_FILE
Causes the CMC implementation to put an attachment containing the message text in the message body rather than putting the message text in the message body directly when returning a message with the **cmc_read** function.

CMC_X_MS_READ_ENV_ONLY
Causes the CMC implementation to return only the message header information when returning a message with the **cmc_read** function.

item_reference
NULL.

extension_flags
Indicates all flags used with the **CMC_extension** structure are valid. No further flags are defined.

Output Usage

extension_flags
Unchanged.

item_data
Unchanged.

item_reference
Unchanged.

Adds to

The **cmc_read**, **cmc_look_up**, and **cmc_list** functions.

See Also

CMC_extension structure, **cmc_list** function, **cmc_look_up** function, **cmc_read** function

CMC_X_MS_MESSAGE_DATA

CMC_X_MS_MESSAGE_DATA is a CMC extension that contains a bitmask of flags used to provide extra data about messages.

Input Usage

item_data
: Bitmask of extension flags. The following flag can be set:

  CMC_X_MS_MSG_RECEIPT_REQ
  : Indicates the message sender requests notification of message receipt.

item_reference
: Points to a string containing the conversation thread.

extension_flags
: Indicates all flags used with the **CMC_extension** structure are valid. No further flags are defined.

Output Usage

item_data
: Bitmask of extension flags. The following flag can be set:

  CMC_X_MS_MSG_RECEIPT_REQ
  : Indicates the message sender requests notification of message receipt.

item_reference
: Unchanged.

extension_flags
: Unchanged.

Adds to

The **CMC_message** structure.

See Also

CMC_extension structure, **CMC_message** structure

CMC_X_MS_SESSION_FLAGS

CMC_X_MS_SESSION_FLAGS is a CMC extension that contains a bitmask of flags used to provide information when logging on and off a session.

Input Usage

item_data
Bitmask of logon and logoff flags. The following flags can be set for use with the **cmc_logon** function:

CMC_X_MS_NEW_SESSION
Indicates that the message sender requests a new session instead of requesting use of a shared session.

CMC_X_MS_FORCE_DOWNLOAD
Indicates that the CMC implementation should attempt to download all new messages before the **cmc_logon** function returns. If this flag is not set, downloading might take place in the background after **cmc_logon** returns.

The following flags can be used with the **cmc_logoff** function:

CMC_X_LOGOFF_SHARED
Indicates the CMC implementation should close all shared sessions.

item_reference
NULL.

extension_flags
Indicates all flags used with the **CMC_extension** structure are valid. No further flags are defined.

Output Usage

item_data
Unchanged.

item_reference
Unchanged.

extension_flags
Unchanged.

Adds to

The **cmc_logon**, **cmc_look_up**, and **cmc_send** functions.

See Also

CMC_extension structure, **cmc_list** function, **cmc_logon** function, **cmc_look_up** function, **cmc_send** function

Simple MAPI

Simple MAPI is a set of functions and related data structures you can use to add messaging functionality to C, C++, or Visual Basic Windows applications. The Simple MAPI functions are available in C and C++ and Visual Basic versions.

The following table provides an overview of the Simple MAPI functions.

Simple MAPI function	**Description**
MAPIAddress	Addresses a message.
MAPIDeleteMail	Deletes a message.
MAPIDetails	Displays a recipient-details dialog box.
MAPIFindNext	Returns the identifier of the first or next message of a specified type.
MAPIFreeBuffer	Frees memory allocated by the messaging system.
MAPILogoff	Ends a session with the messaging system.
MAPILogon	Establishes a messaging session.
MAPIReadMail	Reads a message.
MAPIResolveName	Displays a dialog box to resolve an ambiguous recipient name.
MAPISaveMail	Saves a message.
MAPISendDocuments	Sends a standard message using a dialog box.
MAPISendMail	Sends a message, allowing greater flexibility than **MAPISendDocuments** in message generation.

To use the Simple MAPI functions, compile your source code with MAPI.H. MAPI.H contains definitions for all of the functions, return value constants, and data types. To call a Simple MAPI function, load MAPI.DLL and use the Win32 **GetProcAddress** function to acquire an entry point. The function calling conventions should be FAR PASCAL.

All strings passed to all MAPI calls and returned by all MAPI calls are null-terminated and must be specified in the current character set or code page of the caller's operating system process.

Functions for C and C++

The following alphabetical entries contain documentation for the Simple MAPI functions for C and C++.

MAPIAddress

The **MAPIAddress** function creates or modifies a set of address list entries.

ULONG FAR PASCAL MAPIAddress(
  LHANDLE *lhSession*,
  ULONG *ulUIParam*,
  LPTSTR *lpszCaption*,
  ULONG *nEditFields*,
  LPTSTR *lpszLabels*,
  ULONG *nRecips*,
  lpMapiRecipDesc *lpRecips*,
  FLAGS *flFlags*,
  ULONG *ulReserved*,
  LPULONG *lpnNewRecips*,
  lpMapiRecipDesc FAR * *lppNewRecips*
)

Parameters

lhSession
Input parameter specifying either a session handle that represents a Simple MAPI session or zero. If the value of the *lhSession* parameter is zero, MAPI logs on the user and creates a session that exists only for the duration of the call. This temporary session can be an existing shared session or a new one. If necessary, a logon dialog box is displayed.

ulUIParam
Input parameter specifying either a parent window handle or zero, indicating that if a dialog box is displayed, it is application modal. If the *ulUIParam* parameter contains a parent window handle, it is of type HWND (cast to a ULONG). If no dialog box is displayed during the call, *ulUIParam* is ignored.

lpszCaption
Input parameter specifying either a pointer to the caption for the address list dialog box, NULL, or an empty string. When the *lpszCaption* parameter is NULL or points to an empty string, **MAPIAddress** uses the default caption "Address Book."

nEditFields
Input parameter specifying the number of edit controls that should be present in the address list. The values 0 through 4 are valid. If the value of the *nEditFields* parameter is 4, each recipient class supported by the underlying messaging system has an edit control. If the value of *nEditFields* is zero, only address list browsing is possible. Values of 1, 2, or 3 control the number of edit controls present.

However, if the number of recipient classes in the array pointed to by the *lpRecips* parameter is greater than the value of *nEditFields*, the number of classes in *lpRecips* is used to indicate the number of edit controls instead of the value of *nEditFields.* If the value of *nEditFields* is 1 and more than one kind of entry exists in *lpRecips*, then the *lpszLabels* parameter is ignored.

Entries selected for the different controls are differentiated by the **ulRecipClass** member in the returned recipient structure.

lpszLabels
Input parameter pointing to a string to be used as an edit control label in the address-list dialog box. When the *nEditFields* parameter is set to any value other than 1, the *lpszLabels* parameter is ignored and should be NULL or point to an empty string. Also, if the caller requires the default control label "To," *lpszLabels* should be NULL or point to an empty string.

nRecips
Input parameter specifying the number of entries in the array indicated by the *lpRecips* parameter. If the value of the *nRecips* parameter is zero, *lpRecips* is ignored.

lpRecips
Input parameter pointing to an array of **MapiRecipDesc** structures defining the initial recipient entries to be used to populate the address-list dialog box. The entries do not need to be grouped by recipient class; they are differentiated by the values of the **ulRecipClass** members of the **MapiRecipDesc** structures in the array. If the number of different recipient classes is greater than the value indicated by the *nEditFields* parameter, the *nEditFields* and *lpszLabels* parameters are ignored.

flFlags
Input parameter containing a bitmask of option flags. The following flags can be set:

MAPI_LOGON_UI
Indicates that a dialog box should be displayed to prompt the user to log on if required. When the MAPI_LOGON_UI flag is not set, the client application does not display a logon dialog box and returns an error value if the user is not logged on.

MAPI_NEW_SESSION
Indicates an attempt should be made to create a new session rather than acquire the environment's shared session. If the MAPI_NEW_SESSION flag is not set, **MAPIAddress** uses an existing shared session.

ulReserved
Reserved; must be zero.

lpnNewRecips
Output parameter pointing to the number of entries in the *lppNewRecips* recipient output array. If the value of the *lpnNewRecips* parameter is zero, the *lppNewRecips* parameter is ignored.

lppNewRecips
Output parameter pointing to an array of **MapiRecipDesc** structures containing the final list of recipients. This array is allocated by **MAPIAddress**, cannot be NULL, and must be freed using **MAPIFreeBuffer**, even if there are no new recipients. Recipients are grouped by recipient class in the following order: MAPI_TO, MAPI_CC, MAPI_BCC.

Return Values

MAPI_E_FAILURE
One or more unspecified errors occurred while addressing the message. No list of recipient entries was returned.

MAPI_E_INSUFFICIENT_MEMORY
There was insufficient memory to proceed. No list of recipient entries was returned.

MAPI_E_INVALID_EDITFIELDS
The value of the *nEditFields* parameter was outside the range of 0 through 4. No list of recipient entries was returned.

MAPI_E_INVALID_RECIPS
One or more of the recipients in the address list was not valid. No list of recipient entries was returned.

MAPI_E_INVALID_SESSION
An invalid session handle was used for the *lhSession* parameter. No list of recipient entries was returned.

MAPI_E_LOGIN_FAILURE
There was no default logon, and the user failed to log on successfully when the logon dialog box was displayed. No list of recipient entries was returned.

MAPI_E_NOT_SUPPORTED
The operation was not supported by the underlying messaging system.

MAPI_E_USER_ABORT
The user canceled one of the dialog boxes. No list of recipient entries was returned.

SUCCESS_SUCCESS

The call succeeded and a list of recipient entries was returned.

Remarks

The **MAPIAddress** function displays a standard address-list dialog box to show an initial set of zero or more recipients. The user can choose new entries to add to the set or make changes to existing entries. This dialog box cannot be suppressed, but the caller can set dialog box characteristics. The changed set of recipients is returned to the caller.

Before **MAPIAddress** writes new or changed recipient information, it must allocate memory for the structure array that will contain the information. Memory is also allocated as part of preloading the address book, regardless of whether new or changed recipient data is written. Client applications must call the **MAPIFreeBuffer** function to free this memory after **MAPIAddress** returns. If any error occurs, no memory was allocated and clients do not need to call **MAPIFreeBuffer**.

See Also

MAPIFreeBuffer function, **MAPILogon** function, **MapiRecipDesc** structure

MAPIDeleteMail

The **MAPIDeleteMail** function deletes a message.

ULONG FAR PASCAL MAPIDeleteMail(
  LHANDLE *lhSession*,
  ULONG *ulUIParam*,
  LPTSTR *lpszMessageID*,
  FLAGS *flFlags*,
  ULONG *ulReserved*
)

Parameters

lhSession
  Input parameter specifying a session handle that represents a valid Simple MAPI session. The value of the *lhSession* parameter must represent a valid session; it cannot be zero.

ulUIParam
  Input parameter specifying either a parent window handle or zero, indicating that if a dialog box is displayed, it is application modal. If the *ulUIParam* parameter contains a parent window handle, it is of type HWND (cast to a ULONG). If no dialog box is displayed during the call, *ulUIParam* is ignored.

lpszMessageID
  Input parameter specifying the identifier for the message to be deleted. This identifier is messaging system-specific and will be invalid when **MAPIDeleteMail** successfully returns.

flFlags
  Reserved; must be zero.

ulReserved
  Reserved; must be zero.

Return Values

MAPI_E_FAILURE
  One or more unspecified errors occurred while deleting the message. No message was deleted.

MAPI_E_INSUFFICIENT_MEMORY
  There was insufficient memory to proceed. No message was deleted.

MAPI_E_INVALID_MESSAGE
  An invalid message identifier was passed in the *lpszMessageID* parameter. No message was deleted.

MAPI_E_INVALID_SESSION
  An invalid session handle was passed in the *lhSession* parameter. No message was deleted.

SUCCESS_SUCCESS
  The call succeeded and the message was deleted.

Remarks

To find the message to be deleted, call the **MAPIFindNext** function before calling the **MAPIDeleteMail** function. Because message identifiers are opaque, messaging system-specific, and can be invalidated at any time, **MAPIDeleteMail** considers a message identifier to be valid only for the current session. **MAPIDeleteMail** handles invalid message identifiers by returning the MAPI_E_INVALID_MESSAGE value.

See Also

MAPIFindNext function, **MAPILogon** function, **MAPISaveMail** function

MAPIDetails

The **MAPIDetails** function displays a dialog box containing the details of a selected address list entry.

ULONG FAR PASCAL MAPIDetails(
  LHANDLE *lhSession*,
  ULONG *ulUIParam*,
  lpMapiRecipDesc *lpRecip*,
  FLAGS *flFlags*,
  ULONG *ulReserved*
)

Parameters

lhSession
  Input parameter specifying either a session handle that represents a Simple MAPI session or zero. If the value of the *lhSession* parameter is zero, MAPI logs on the user and creates a session that exists only for the duration of the call. This temporary session can be an existing shared session or a new one. If additional information is required from the user to successfully complete the logon, a dialog box is displayed.

ulUIParam
  Input parameter specifying either a parent window handle or zero, indicating that if a dialog box is displayed, it is application modal. If the *ulUIParam* parameter contains a parent window handle, it is of type HWND (cast to a ULONG). If no dialog box is displayed during the call, *ulUIParam* is ignored.

lpRecip
  Input parameter pointing to the recipient for which details are to be displayed. **MAPIDetails** ignores all members of this **MapiRecipDesc** structure except the **ulEIDSize** and **lpEntryID** members. If the value of **ulEIDSize** is non-zero, **MAPIDetails** resolves the recipient entry. If the value of **ulEIDSize** is zero, **MAPIDetails** returns the MAPI_E_AMBIGUOUS_RECIP value.

flFlags
  Input parameter containing a bitmask of option flags. The following flags can be set:

  MAPI_AB_NOMODIFY
    Indicates the caller is requesting that the dialog box be read-only, prohibiting changes. **MAPIDetails** might or might not honor the request.

  MAPI_LOGON_UI
    Indicates that a dialog box should be displayed to prompt the user to log on if required. When the MAPI_LOGON_UI flag is not set, the client application does not display a logon dialog box and returns an error value if the user is not logged on.

  MAPI_NEW_SESSION
    Indicates an attempt should be made to create a new session rather than acquire the environment's shared session. If the MAPI_NEW_SESSION flag is not set, **MAPIDetails** uses an existing shared session.

ulReserved
  Reserved; must be zero.

ReturnValues

MAPI_E_AMBIGUOUS_RECIPIENT
  The dialog box could not be displayed because the **ulEIDSize** member of the structure pointed to by the *lpRecips* parameter was zero.

MAPI_E_FAILURE
  One or more unspecified errors occurred. No dialog box was displayed.

MAPI_E_INSUFFICIENT_MEMORY

There was insufficient memory to proceed. No dialog box was displayed.

MAPI_E_INVALID_RECIPS
: The recipient specified in the *lpRecip* parameter was unknown or the recipient had an invalid **ulEIDSize** value. No dialog box was displayed.

MAPI_E_LOGIN_FAILURE
: There was no default logon, and the user failed to log on successfully when the logon dialog box was displayed. No dialog box was displayed.

MAPI_E_NOT_SUPPORTED
: The operation was not supported by the underlying messaging system.

MAPI_E_USER_ABORT
: The user canceled either the logon dialog box or the details dialog box.

SUCCESS_SUCCESS
: The call succeeded and the details dialog box was displayed.

Remarks

The **MAPIDetails** function presents a dialog box that shows the details of a particular address list entry. The display name and address are the minimum attributes that are displayed in the dialog box; more information can be shown, depending on the address book provider. The details dialog box cannot be suppressed, but the caller can request that it be read-only or modifiable.

Details can only be shown for resolved address list entries. An entry is resolved if the value of the **ulEIDSize** member of the **MapiRecipDesc** structure is nonzero. Entries are resolved when they are returned by the **MAPIAddress** or **MAPIResolveName** functions and as the result being recipients of read mail.

See Also

MAPIAddress function, **MAPILogon** function, **MapiRecipDesc** structure, **MAPIResolveName** function

MAPIFindNext

The **MAPIFindNext** function retrieves the next (or first) message identifier of a specified type of incoming message.

ULONG FAR PASCAL MAPIFindNext(
  LHANDLE *lhSession*,
  ULONG *ulUIParam*,
  LPTSTR *lpszMessageType*,
  LPTSTR *lpszSeedMessageID*,
  FLAGS *flFlags*,
  ULONG *ulReserved*,
  LPTSTR *lpszMessageID*
)

Parameters

lhSession
: Input parameter specifying a session handle that represents a Simple MAPI session. The value of the *lhSession* parameter must represent a valid session; it cannot be zero.

ulUIParam
: Input parameter specifying either a parent window handle or zero, indicating that if a dialog box is displayed, it is application modal. If the *ulUIParam* parameter contains a parent window handle, it is of type HWND (cast to a ULONG). If no dialog box is displayed during the call, *ulUIParam* is ignored.

lpszMessageType
: Input parameter pointing to a string identifying the message class to search. To find an interpersonal message (IPM), specify NULL in the *lpszMessageType* parameter or have it point to an empty string. Messaging systems whose only supported message class is IPM can ignore this parameter.

lpszSeedMessageID
: Input parameter pointing to a string containing the message identifier seed for the request. If the *lpszSeedMessageID* parameter is NULL or points to an empty string, **MAPIFindNext** retrieves the first message that matches the type specified in the *lpszMessageType* parameter.

flFlags
: Input parameter containing a bitmask of option flags. The following flags can be set:

  MAPI_GUARANTEE_FIFO
  : Indicates the message identifiers returned should be in the order of time received. **MAPIFindNext** calls can take longer if this flag is set. Some implementations cannot honor this request and return the MAPI_E_NO_SUPPORT value.

  MAPI_LONG_MSGID
  : Indicates that the returned message identifier can be as long as 512 characters. If this flag is set, the *lpszMessageID* parameter must be large enough to accomodate 512 characters.

    Older versions of MAPI supported smaller message identifiers (64 bytes) and did not include this flag. **MAPIFindNext** will succeed without this flag set as long as *lpszMessageID* is large enough to hold the message identifier. If *lpszMessageID* cannot hold the message identifier, **MAPIFindNext** will fail.

  MAPI_UNREAD_ONLY
  : Indicates that only unread messages of the specified type should be enumerated. If this flag is not set, **MAPIFindNext** can return any message of the specified type.

ulReserved
: Reserved; must be zero.

lpszMessageID
: Output parameter specifying a pointer to the returned message identifier. The caller is responsible

for allocating the memory. To ensure compatibility, allocate 512 characters and set MAPI_LONG_MSGID in the *flFlags* parameter. A smaller buffer is sufficient only if the returned message identifier is always 64 characters or less.

Return Values

MAPI_E_FAILURE
One or more unspecified errors occurred while matching the message type. The call failed before message type matching could take place.

MAPI_E_INSUFFICIENT_MEMORY
There was insufficient memory to proceed. No message was found.

MAPI_E_INVALID_MESSAGE
An invalid message identifier was passed in the *lpszSeedMessageID* parameter. No message was found.

MAPI_E_INVALID_SESSION
An invalid session handle was passed in the *lhSession* parameter. No message was found.

MAPI_E_NO_MESSAGES
A matching message could not be found.

SUCCESS_SUCCESS
The call succeeded and the message identifier was returned.

Remarks

The **MAPIFindNext** function allows a client application to enumerate messages of a given type. This function can be called repeatedly to list all messages in the folder. Message identifiers returned from **MAPIFindNext** can be used in other Simple MAPI calls to retrieve message contents and delete messages. This function is for processing incoming messages, not for managing received messages.

MAPIFindNext looks for messages in the folder in which new messages of the specified type are delivered. **MAPIFindNext** calls can be made only in the context of a valid Simple MAPI session established with the **MAPILogon** function.

When the *lpszSeedMessageID* parameter is NULL or points to an empty string, **MAPIFindNext** returns the message identifier for the first message of the type specified by the *lpszMessageType* parameter. When *lpszSeedMessageID* contains a valid identifier, the function returns the next matching message of the type specified by *lpszMessageType*. Repeated calls to **MAPIFindNext** ultimately result in a return of the MAPI_E_NO_MESSAGES value, which means the enumeration is complete.

Message type matching is done against message class strings. All message types whose names match (up to the length specified in *lpszMessageType*) are returned.

Because message identifiers are messaging system-specific and can be invalidated at any time, message identifiers are valid only for the current session. If the message identifier passed in *lpszSeedMessageID* is invalid, **MAPIFindNext** returns the MAPI_E_INVALID_MESSAGE value.

See Also

MAPILogon function

MAPIFreeBuffer (Simple MAPI)

The **MAPIFreeBuffer** function frees memory allocated by the messaging system.

ULONG FAR PASCAL MAPIFreeBuffer(
  LPVOID *pv*
)

Parameters

pv
  Input parameter specifying a pointer to memory allocated by the messaging system. This pointer is returned by the **MAPIReadMail**, **MAPIAddress**, and **MAPIResolveName** functions.

Return Values

MAPI_E_FAILURE
  One or more unspecified errors occurred. The memory could not be freed.

SUCCESS_SUCCESS
  The call succeeded and the memory was freed.

See Also

MAPILogoff function

MAPILogoff

The **MAPILogoff** function ends a session with the messaging system.

ULONG FAR PASCAL MAPILogoff (
LHANDLE *lhSession***,**
ULONG *ulUIParam***,**
FLAGS *flFlags***,**
ULONG *ulReserved*
)

Parameters

lhSession
Input parameter specifying a handle for the Simple MAPI session to be terminated. Session handles are returned by the **MAPILogon** function and invalidated by **MAPILogoff**. The value of the *lhSession* parameter must represent a valid session; it cannot be zero.

ulUIParam
Input parameter specifying either a parent window handle or zero, indicating that if a dialog box is displayed, it is application modal. If the *ulUIParam* parameter contains a parent window handle, it is of type HWND (cast to a ULONG). If no dialog box is displayed during the call, *ulUIParam* is ignored.

flFlags
Reserved; must be zero.

ulReserved
Reserved; must be zero.

Return Values

MAPI_E_FAILURE
The *flFlags* parameter is invalid or one or more unspecified errors occurred.

MAPI_E_INSUFFICIENT_MEMORY
There was insufficient memory to proceed. The session was not terminated.

MAPI_E_INVALID_SESSION
An invalid session handle was used for the *lhSession* parameter. The session was not terminated.

SUCCESS_SUCCESS
The call succeeded and the session was terminated.

See Also

MAPILogon function

MAPILogon

The **MAPILogon** function begins a Simple MAPI session, loading the default message store and address book providers.

ULONG FAR PASCAL MAPILogon(
  ULONG *ulUIParam***,**
  LPTSTR *lpszProfileName***,**
  LPTSTR *lpszPassword***,**
  FLAGS *flFlags***,**
  ULONG *ulReserved***,**
  LPLHANDLE *lplhSession*
)

Parameters

ulUIParam
: Input parameter specifying either a parent window handle or zero, indicating that if a dialog box is displayed, it is application modal. If the *ulUIParam* parameter contains a parent window handle, it is of type HWND (cast to a ULONG). If no dialog box is displayed during the call, *ulUIParam* is ignored.

lpszProfileName
: Input parameter pointing to a null-terminated profile name string, limited to 256 characters or less. This is the profile to use when logging on. If the *lpszProfileName* parameter is NULL or points to an empty string, and the *flFlags* parameter is set to MAPI_LOGON_UI, **MAPILogon** displays a logon dialog box with an empty name field.

lpszPassword
: Input parameter pointing to a null-terminated credential string, limited to 256 characters or less. If the messaging system does not require password credentials, or if it requires that the user enter them, the *lpszPassword* parameter should be NULL or point to an empty string. When the user must enter credentials, the *flFlags* parameter must be set to MAPI_LOGON_UI to allow a logon dialog box to be displayed.

flFlags
: Input parameter containing a bitmask of option flags. The following flags can be set:

  MAPI_FORCE_DOWNLOAD
  : Indicates an attempt should be made to download all of the user's messages before returning. If the MAPI_FORCE_DOWNLOAD flag is not set, messages can be downloaded in the background after the function call returns.

  MAPI_NEW_SESSION
  : Indicates an attempt should be made to create a new session rather than acquire the environment's shared session. If the MAPI_NEW_SESSION flag is not set, **MAPILogon** uses an existing shared session.

  MAPI_LOGON_UI
  : Indicates that a logon dialog box should be displayed to prompt the user for logon information. If the user needs to provide a password and profile name to enable a successful logon, MAPI_LOGON_UI must be set.

  MAPI_PASSWORD_UI
  : Indicates that **MAPILogon** should only prompt for a password and not allow the user to change the profile name. Either MAPI_PASSWORD_UI or MAPI_LOGON_UI should not be set, since the intent is to select between two different dialog boxes for logon.

ulReserved
: Reserved; must be zero.

lplhSession

Output parameter specifying a Simple MAPI session handle.

Return Values

MAPI_E_FAILURE
One or more unspecified errors occurred during logon. No session handle was returned.

MAPI_E_INSUFFICIENT_MEMORY
There was insufficient memory to proceed. No session handle was returned.

MAPI_E_LOGIN_FAILURE
There was no default logon, and the user failed to log on successfully when the logon dialog box was displayed. No session handle was returned.

MAPI_E_TOO_MANY_SESSIONS
The user had too many sessions open simultaneously. No session handle was returned.

MAPI_E_USER_ABORT
The user canceled the logon dialog box. No session handle was returned.

SUCCESS_SUCCESS
The call succeeded and a Simple MAPI session was established.

Remarks

The **MAPILogon** function begins a session with the messaging system, returning a handle that can be used in subsequent MAPI calls to explicitly provide user credentials to the messaging system. To request the display of a logon dialog box if the credentials presented fail to validate the session, set the *flFlags* parameter to MAPI_LOGON_UI.

The client application tests for an existing session by calling **MAPILogon** with a NULL value for the *lpszProfileName* parameter, a NULL value for the *lpszPassword* parameter and by not setting the MAPI_LOGON_UI flag in *flFlags*. If there is an existing session, the call succeeds and returns a valid LHANDLE for the session. Otherwise, the call fails.

See Also

MAPILogoff function

MAPIReadMail

The **MAPIReadMail** function retrieves a message for reading.

ULONG FAR PASCAL MAPIReadMail(
  LHANDLE *lhSession*,
  ULONG *ulUIParam*,
  LPTSTR *lpszMessageID*,
  FLAGS *flFlags*,
  ULONG *ulReserved*,
  lpMapiMessage FAR * *lppMessage*
)

Parameters

lhSession
  Input parameter specifying a handle to a Simple MAPI session. The value of the *lhSession* parameter must represent a valid session; it cannot be zero.

ulUIParam
  Input parameter specifying either a parent window handle or zero, indicating that if a dialog box is displayed, it is application modal. If the *ulUIParam* parameter contains a parent window handle, it is of type HWND (cast to a ULONG). If no dialog box is displayed during the call, *ulUIParam* is ignored.

lpszMessageID
  Input parameter pointing to a message identifier string for the message to be read. The string is allocated by the caller.

flFlags
  Input parameter containing a bitmask of option flags. The following flags can be set:

  MAPI_BODY_AS_FILE
    Indicates **MAPIReadMail** should write the message text to a temporary file and add it as the first attachment in the attachment list.

  MAPI_ENVELOPE_ONLY
    Indicates **MAPIReadMail** should read the message header only. File attachments are not copied to temporary files, and neither temporary file names nor message text is written. Setting this flag enhances performance.

  MAPI_PEEK
    Indicates **MAPIReadMail** does not mark the message as read. Marking a message as read affects its appearance in the user interface and generates a read receipt. If the messaging system does not support this flag, **MAPIReadMail** always marks the message as read. If **MAPIReadMail** encounters an error, it leaves the message unread.

  MAPI_SUPPRESS_ATTACH
    Indicates **MAPIReadMail** should not copy file attachments but should write message text into the **MapiMessage** structure. **MAPIReadMail** ignores this flag if the calling application has set the MAPI_ENVELOPE_ONLY flag. Setting the MAPI_SUPPRESS_ATTACH flag enhances performance.

ulReserved
  Reserved; must be zero.

lppMessage
  Output parameter pointing to the location where the message is written. Messages are written to a **MapiMessage** structure which can be freed with a single call to the **MAPIFreeBuffer** function.

  When MAPI_ENVELOPE_ONLY and MAPI_SUPPRESS_ATTACH are not set, attachments are written to temporary files pointed to by the **lpFiles** member of the **MapiMessage** structure. It is the caller's responsibility to delete these files when they are no longer needed.

Return Values

MAPI_E_ATTACHMENT_WRITE_FAILURE
: An attachment could not be written to a temporary file. Check directory permissions.

MAPI_E_DISK_FULL
: An attachment could not be written to a temporary file because there was not enough space on the disk.

MAPI_E_FAILURE
: One or more unspecified errors occurred while reading the message.

MAPI_E_INSUFFICIENT_MEMORY
: There was insufficient memory to read the message.

MAPI_E_INVALID_MESSAGE
: An invalid message identifier was passed in the *lpszMessageID* parameter.

MAPI_E_INVALID_SESSION
: An invalid session handle was passed in the *lhSession* parameter. No message was retrieved.

MAPI_E_TOO_MANY_FILES
: There were too many file attachments in the message. The message could not be read.

MAPI_E_TOO_MANY_RECIPIENTS
: There were too many recipients of the message. The message could not be read.

SUCCESS_SUCCESS
: The call succeeded and the message was read.

Remarks

The **MAPIReadMail** function returns one message, breaking the message content into the same parameters and structures used in the **MAPISendMail** function. **MAPIReadMail** fills a block of memory with the **MapiMessage** structure containing message elements, such as the subject, message class, delivery time, and the sender. File attachments are saved to temporary files, and the names are returned to the caller in the message structure. Recipients, attachments, and contents are copied from the message before **MAPIReadMail** returns to the caller, so later changes to the files do not affect the contents of the message.

A flag is provided to specify that only envelope information is to be returned from the call. Another flag (in the **MapiMessage** structure) specifies whether the message is marked as sent or unsent.

The caller is responsible for freeing the **MapiMessage** structure by calling the **MAPIFreeBuffer** function and deleting any files associated with attachments included with the message.

Before calling **MAPIReadMail**, use the **MAPIFindNext** function to verify that the message to be read is the one you want to be read. Because message identifiers are system-specific and opaque and can be invalidated at any time, **MAPIReadMail** considers a message identifier to be valid only for the current Simple MAPI session.

See Also

MAPIFreeBuffer function, **MAPILogon** function, **MapiMessage** structure

MAPIResolveName

The **MAPIResolveName** function transforms a message recipient's name as entered by a user to an unambiguous address list entry.

ULONG FAR PASCAL MAPIResolveName(
  LHANDLE *lhSession***,**
  ULONG *ulUIParam***,**
  LPTSTR *lpszName***,**
  FLAGS *flFlags***,**
  ULONG *ulReserved***,**
  lpMapiRecipDesc FAR * *lppRecip*
)

Parameters

lhSession
  Input parameter specifying either a handle that represents a Simple MAPI session or zero. If the value of the *lhSession* parameter is zero, MAPI logs on the user and creates a session that exists only for the duration of the call. This temporary session can be an existing shared session or a new one. If necessary, the logon dialog box is displayed.

ulUIParam
  Input parameter specifying either a parent window handle or zero, indicating that if a dialog box is displayed, it is application modal. If the *ulUIParam* parameter contains a parent window handle, it is of type HWND (cast to a ULONG). If no dialog box is displayed during the call, *ulUIParam* is ignored.

lpszName
  Input parameter pointing to the name to be resolved.

flFlags
  Input parameter containing a bitmask of option flags. The following flags can be set:

  MAPI_AB_NOMODIFY
    Indicates the caller is requesting that the dialog box be read-only, prohibiting changes. **MAPIResolveName** ignores this flag if MAPI_DIALOG is not set.

  MAPI_DIALOG
    Indicates that a dialog box should be displayed for name resolution. If this flag is not set and the name cannot be resolved, **MAPIResolveName** returns the MAPI_E_AMBIGUOUS_RECIPIENT value.

  MAPI_LOGON_UI
    Indicates that a dialog box should be displayed to prompt the user to log on if required. When the MAPI_LOGON_UI flag is not set, the client application does not display a logon dialog box and returns an error value if the user is not logged on.

  MAPI_NEW_SESSION
    Indicates an attempt should be made to create a new session rather than acquire the environment's shared session. If the MAPI_NEW_SESSION flag is not set, **MAPIResolveName** uses an existing shared session.

ulReserved
  Reserved; must be zero.

lppRecip
  Output parameter pointing to a recipient structure if the resolution results in a single match. The recipient structure contains the resolved name and related information. Memory for this structure must be freed using the **MAPIFreeBuffer** function.

Return Values

MAPI_E_AMBIGUOUS_RECIPIENT
: The recipient requested has not been or could not be resolved to a unique address list entry.

MAPI_E_UNKNOWN_RECIPIENT
: The recipient could not be resolved to any address. The recipient might not exist or might be unknown.

MAPI_E_FAILURE
: One or more unspecified errors occurred. The name was not resolved.

MAPI_E_INSUFFICIENT_MEMORY
: There was insufficient memory to proceed. The name was not resolved.

MAPI_E_LOGIN_FAILURE
: There was no default logon, and the user failed to log on successfully when the logon dialog box was displayed. The name was not resolved.

MAPI_E_NOT_SUPPORTED
: The operation was not supported by the underlying messaging system.

MAPI_E_USER_ABORT
: The user canceled one of the dialog boxes. The name was not resolved.

SUCCESS_SUCCESS
: The call succeeded and the name was resolved.

Remarks

The **MAPIResolveName** function resolves a message recipient's name (as entered by a user) to an unambiguous address list entry, optionally prompting the user to choose between possible entries, if necessary. A recipient descriptor structure containing fully resolved information about the entry is allocated and returned. The caller should free this **MAPIRecipDesc** structure at some point by calling the **MAPIFreeBuffer** function. If **MAPIResolveName** returns an error value, it is not necessary to deallocate memory with **MAPIFreeBuffer**.

See Also

MAPIFreeBuffer function, **MAPILogon** function, **MapiRecipDesc** structure

MAPISaveMail

The **MAPISaveMail** function saves a message into the message store.

ULONG FAR PASCAL MAPISaveMail(
  LHANDLE *lhSession*,
  ULONG *ulUIParam*,
  lpMapiMessage *lpMessage*,
  FLAGS *flFlags*,
  ULONG *ulReserved*,
  LPTSTR *lpszMessageID*
)

Parameters

lhSession
Input parameter specifying either a handle for a Simple MAPI session or zero. The value of the *lhSession* parameter must not be zero if the *lpszMessageID* parameter contains a valid message identifier. However, if *lpszMessageID* does not contain a valid message identifier, and the value of *lhSession* is zero, MAPI logs on the user and creates a session that exists only for the duration of the call. This temporary session can be an existing shared session or a new one. If necessary, the logon dialog box is displayed.

ulUIParam
Input parameter specifying either a parent window handle or zero, indicating that if a dialog box is displayed, it is application modal. If the *ulUIParam* parameter contains a parent window handle, it is of type HWND (cast to a ULONG). If no dialog box is displayed during the call, *ulUIParam* is ignored.

lpMessage
Input parameter pointing to a **MapiMessage** structure containing the contents of the message to be saved. The **lpOriginator** member is ignored. Applications can either ignore the **flFlags** member, or if the message has never been saved, can set the MAPI_SENT and MAPI_UNREAD flags.

flFlags
Input parameter containing a bitmask of option flags. The following flags can be set:

MAPI_LOGON_UI
Indicates that a dialog box should be displayed to prompt the user to logon if required. When the MAPI_LOGON_UI flag is not set, the client application does not display a logon dialog box and returns an error value if the user is not logged on. **MAPISaveMail** ignores this flag if the *lpszMessageID* parameter is empty.

MAPI_LONG_MSGID
Indicates that the returned message identifier is expected to be 512 characters. If this flag is set, the *lpszMessageID* parameter must be large enough to accomodate 512 characters.

MAPI_NEW_SESSION
Indicates an attempt should be made to create a new session rather than acquire the environment's shared session. If the MAPI_NEW_SESSION flag is not set, **MAPISaveMail** uses an existing shared session.

MAPI_LONG_MSGID
When the input message is NULL, this flag should be passed into **MAPISaveMail**, and the accompanying message identifier buffer should be 512 bytes long.

ulReserved
Reserved; must be zero.

lpszMessageID
Input-output parameter pointing to either the message identifier to be replaced by the save operation or an empty string, indicating that a new message is to be created. The string must be allocated by

the caller and must be able to hold at least 512 characters if the *flFlags* parameter is set to MAPI_LONG_MSGID. If the *flFlags* parameter is not set to MAPI_LONG_MSGID, the message identifier string can hold 64 characters.

Return Values

MAPI_E_ATTACHMENT_NOT_FOUND
An attachment could not be located at the specified path. Either the drive letter was invalid, the path was not found on that drive, or the file was not found in that path.

MAPI_E_BAD_RECIPTYPE
The recipient type in the *lpMessage* was invalid.

MAPI_E_FAILURE
One or more unspecified errors occurred while saving the message. No message was saved.

MAPI_E_INSUFFICIENT_MEMORY
There was insufficient memory to save the message. No message was saved.

MAPI_E_INVALID_MESSAGE
An invalid message identifier was passed in the *lpszMessageID* parameter; no message was saved.

MAPI_E_INVALID_RECIPS
One or more recipients of the message were invalid or could not be identified.

MAPI_E_INVALID_SESSION
An invalid session handle was passed in the *lhSession* parameter. No message was saved.

MAPI_E_LOGIN_FAILURE
There was no default logon, and the user failed to log on successfully when the logon dialog box was displayed. No message was saved.

MAPI_E_NOT_SUPPORTED
The operation was not supported by the underlying messaging system.

MAPI_E_USER_ABORT
The user canceled one of the dialog boxes. No message was saved.

SUCCESS_SUCCESS
The call succeeded and the message was saved.

Remarks

The **MAPISaveMail** function saves a message, optionally replacing an existing message. Before calling **MAPISaveMail**, use the **MAPIFindNext** function to verify that the message to be saved is the one you want saved. The elements of the message identified by the *lpszMessageID* parameter are replaced by the elements in the *lpMessage* parameter. If *lpszMessageID* is empty, a new message is created. All replaced messages are saved in their appropriate folders. New messages are saved in the folder appropriate for incoming messages of that class.

Not all messaging systems support storing messages. If the underlying messaging system does not support message storage, **MAPISaveMail** returns the MAPI_E_NOT_SUPPORTED value.

Because message identifiers are system-specific and opaque and can be invalidated at any time, **MAPISaveMail** considers a message identifier to be valid only for the current Simple MAPI session. **MAPISaveMail** handles invalid message identifiers by returning the MAPI_E_INVALID_MESSAGE value.

See Also

MAPILogon function, **MapiMessage** structure

MAPISendDocuments

The **MAPISendDocuments** function sends a standard message with one or more attached files and a cover note. The cover note is a dialog box that allows the user to enter a list of recipients and an optional message. **MAPISendDocuments** differs from the **MAPISendMail** function in that it allows less flexibility in message generation.

ULONG FAR PASCAL MAPISendDocuments(
  ULONG *ulUIParam***,**
  LPTSTR *lpszDelimChar***,**
  LPTSTR *lpszFullPaths***,**
  LPTSTR *lpszFileNames***,**
  ULONG *ulReserved*
)

Parameters

ulUIParam
: Input parameter specifying either a parent window handle or zero, indicating that if a dialog box is displayed, it is application modal. If the *ulUIParam* parameter contains a parent window handle, it is of type HWND (cast to a ULONG). If no dialog box is displayed during the call, *ulUIParam* is ignored.

lpszDelimChar
: Input parameter pointing to a character that the caller uses to delimit the names pointed to by the *lpszFullPaths* and *lpszFileNames* parameters. The caller should select a character for the delimiter that is not used in operating system filenames.

lpszFullPaths
: Input parameter pointing to a string containing a list of full paths (including drive letters) to attachment files. This list is formed by concatenating correctly formed file paths separated by the character specified in the *lpszDelimChar* parameter and followed by a null terminator. An example of a valid list is:

```
C:\TMP\TEMP1.DOC;C:\TMP\TEMP2.DOC
```

The files specified in this parameter are added to the message as file attachments. If this parameter is NULL or contains an empty string, the Send Note dialog box is displayed with no attached files.

lpszFileNames
: Input parameter pointing to a null-terminated list of the original filenames as they should appear in the message. When multiple names are specified, the list is formed by concatenating the filenames separated by the character specified in the *lpszDelimChar* parameter and followed by a null terminator. An example is:

```
TEMP3.DOC;TEMP4.DOC
```

If there is no value for the *lpszFileNames* parameter or if it is empty, **MAPISendDocuments** sets the filenames set to the filename values indicated by the *lpszFullPaths* parameter.

ulReserved
: Reserved; must be zero.

Return Values

MAPI_E_ATTACHMENT_OPEN_FAILURE
: One or more files in the *lpszFilePaths* parameter could not be located. No message was sent.

MAPI_E_ATTACHMENT_WRITE_FAILURE
: An attachment could not be written to a temporary file. Check directory permissions.

MAPI_E_FAILURE

One or more unspecified errors occurred while sending the message. It is not known if the message was sent.

MAPI_E_INSUFFICIENT_MEMORY
: There was insufficient memory to proceed.

MAPI_E_LOGIN_FAILURE
: There was no default logon, and the user failed to log on successfully when the logon dialog box was displayed. No message was sent.

MAPI_E_USER_ABORT
: The user canceled one of the dialog boxes. No message was sent.

SUCCESS_SUCCESS
: The call succeeded and the message was sent.

Remarks

The **MAPISendDocuments** function sends a standard message, always displaying a cover note dialog box so the user can provide recipients and other sending options. This function tries to establish a session using the messaging system's shared session. If no shared session exists, it prompts the user for logon information to establish a session. Before **MAPISendDocuments** returns, it ends the session.

Message attachments can include the active document or all the currently open documents in the client application that called **MAPISendDocuments**. This function is used primarily for calls from a macro or scripting language, often found in applications such as spreadsheet or word-processing programs.

MAPISendDocuments creates as many file attachments as there are paths specified by the *lpszFullPaths* parameter in spite of the fact that there can be different numbers of paths and filenames. The caller is responsible for deleting temporary files created when using **MAPISendDocuments**.

See Also

MAPISendMail function

MAPISendMail

The **MAPISendMail** function sends a message. This function differs from the **MAPISendDocuments** function in that it allows greater flexibility in message generation.

ULONG FAR PASCAL MAPISendMail(
  LHANDLE *lhSession*,
  ULONG *ulUIParam*,
  lpMapiMessage *lpMessage*,
  FLAGS *flFlags*,
  ULONG *ulReserved*
)

Parameters

lhSession
 Input parameter specifying either a handle to a Simple MAPI session or zero. If the value of the *lhSession* parameter is zero, MAPI logs on the user and creates a session that exists only for the duration of the call. This temporary session can be an existing shared session or a new one. If necessary, the logon dialog box is displayed.

ulUIParam
 Input parameter specifying either a parent window handle or zero, indicating that if a dialog box is displayed, it is application modal. If the *ulUIParam* parameter contains a parent window handle, it is of type HWND (cast to a ULONG). If no dialog box is displayed during the call, *ulUIParam* is ignored.

lpMessage
 Input parameter pointing to a **MapiMessage** structure containing the message to be sent. If the MAPI_DIALOG flag is not set, the **nRecipCount** and **lpRecips** members must be valid for successful message delivery. Client applications can set the **flFlags** member to MAPI_RECEIPT_REQUESTED to request a read report. All other members are ignored and unused pointers should be NULL.

flFlags
 Input parameter containing a bitmask of option flags. The following flags can be set:

 MAPI_DIALOG
  Indicates that a dialog box should be displayed to prompt the user for recipients and other sending options. When MAPI_DIALOG is not set, at least one recipient must be specified.

 MAPI_LOGON_UI
  Indicates that a dialog box should be displayed to prompt the user to log on if required. When the MAPI_LOGON_UI flag is not set, the client application does not display a logon dialog box and returns an error value if the user is not logged on. **MAPISendMail** ignores this flag if the *lpszMessageID* parameter is empty.

 MAPI_NEW_SESSION
  Indicates an attempt should be made to create a new session rather than acquire the environment's shared session. If the MAPI_NEW_SESSION flag is not set, **MAPISendMail** uses an existing shared session.

ulReserved
 Reserved; must be zero.

Return Values

MAPI_E_AMBIGUOUS_RECIPIENT
 A recipient matched more than one of the recipient descriptor structures and MAPI_DIALOG was not set. No message was sent.

MAPI_E_ATTACHMENT_NOT_FOUND

The specified attachment was not found. No message was sent.

MAPI_E_ATTACHMENT_OPEN_FAILURE

The specified attachment could not be opened. No message was sent.

MAPI_E_BAD_RECIPTYPE

The type of a recipient was not MAPI_TO, MAPI_CC, or MAPI_BCC. No message was sent.

MAPI_E_FAILURE

One or more unspecified errors occurred. No message was sent.

MAPI_E_INSUFFICIENT_MEMORY

There was insufficient memory to proceed. No message was sent.

MAPI_E_INVALID_RECIPS

One or more recipients were invalid or did not resolve to any address.

MAPI_E_LOGIN_FAILURE

There was no default logon, and the user failed to log on successfully when the logon dialog box was displayed. No message was sent.

MAPI_E_TEXT_TOO_LARGE

The text in the message was too large. No message was sent.

MAPI_E_TOO_MANY_FILES

There were too many file attachments. No message was sent.

MAPI_E_TOO_MANY_RECIPIENTS

There were too many recipients. No message was sent.

MAPI_E_UNKNOWN_RECIPIENT

A recipient did not appear in the address list. No message was sent.

MAPI_E_USER_ABORT

The user canceled one of the dialog boxes. No message was sent.

SUCCESS_SUCCESS

The call succeeded and the message was sent.

Remarks

The **MAPISendMail** function sends a standard message, with or without any user interaction. The profile must be configured so that **MAPISendMail** can open the default service providers without requiring user interaction. However, if the *flFlags* parameter is set to MAPI_NEW_SESSION, disallowing the use of a shared session, and the profile requires a password, MAPI_LOGON_UI must be set or the function will fail. Client applications can avoid this situation by using an explicit profile without a password or by using the default profile without a password.

Client applications can provide a full or partial list of recipient names, subject text, file attachments, or message text. If any information is missing, **MAPISendMail** can prompt the user for it. If no information is missing, either the message can be sent as is or the user can be prompted to verify the information, changing values if necessary.

A successful return from **MAPISendMail** does not necessarily imply recipient validation. The message might not have been sent to all recipients. Depending on the transport provider, recipient validation can be a lengthy process.

A NULL value for the **lpszSubject** member of the **MapiMessage** structure pointed to by the *lpMessage* parameter indicates that there is no text for the subject of the message. A NULL value for the **lpszNoteText** member indicates that there is no message text. Some client applications can truncate subject lines that are too long or contain carriage returns, line feeds, or form feeds.

Each paragraph should be terminated with a CR (0x0d), an LF (0x0a), or a CRLF pair (0x0d0a). **MAPISendMail** wraps lines as appropriate. If the text exceeds system limits, the function returns the MAPI_E_TEXT_TOO_LARGE value.

The **lpszMessageType** member of the **MapiMessage** structure pointed to by *lpMessage* is used only by non-IPM applications. Applications that handle IPM messages can set it to NULL or have it point to an empty string.

The number of attachments per message can be limited in some messaging systems. If the limit is exceeded, the MAPI_E_TOO_MANY_FILES value is returned. If no files are specified, a pointer value of NULL should be assigned to the **lpFiles** member of the structure pointed to by *lpMessage*. File attachments are copied to the message before **MAPISendMail** returns; therefore, later changes to the files do not affect the contents of the message. The files must be closed when they are copied. Do not attempt to display attachments outside the range of the message text.

Some messaging systems can limit the number of recipients per message. A pointer value of NULL for the **lpRecips** member in the **MapiMessage** structure pointed to by *lpMessage* indicates no recipients. If the client application passes a non-NULL value indicating a number of recipients exceeding the system limit, **MAPISendMail** returns the MAPI_E_TOO_MANY_RECIPIENTS value. If the value of the **nRecipCount** member in the **MapiMessage** structure pointed to by *lpMessage* is 0, the MAPI_DIALOG flag must be present in the call to **MAPISendMail**.

Note that the **lpRecips** member in the **MapiMessage** structure can include either an entry identifier, the recipient's name, an address, or a name and address pair. The following table shows how **MAPISendMail** handles the variety of information that can be specified:

Information	**Action**
entry identifier	No name resolution; the name and address are ignored.
name	Name resolved using the Simple MAPI resolution rules.
address	No name resolution; address is used for both message delivery and for displaying the recipient name.
name and address	No name resolution; name used only for displaying the recipient name.

Client applications that send messages to custom recipients can avoid name resolution. Such clients should set the **lpszAddress** member of the **MapiRecipDesc** structure pointed to by the **lpRecips** member of the **MapiMessage** structure pointed to by the *lpMessage* parameter to the custom address.

MAPISendMail does not require an originator-type recipient to send a message.

See Also

MAPILogon function, **MapiMessage** structure, **MapiRecipDesc** structure

Structures for C and C++

The following alphabetized entries contain documentation for the Simple MAPI structures for C and C+
+.

MapiFileDesc (Simple MAPI)

A **MapiFileDesc** structure contains information about a file containing a message attachment stored as a temporary file. That file can contain a static OLE object, an embedded OLE object, an embedded message, and other types of files.

```
typedef struct {
     ULONG ulReserved;
     ULONG flFlags;
     ULONG nPosition;
     LPTSTR lpszPathName;
     LPTSTR lpszFileName;
     LPVOID lpFileType;
} MapiFileDesc, FAR *lpMapiFileDesc;
```

Members

ulReserved
Reserved; must be zero.

flFlags
Contains a bitmask of attachment flags. The following flags can be set:

MAPI_OLE
Indicates the attachment is an OLE object. If MAPI_OLE_STATIC is also set, the attachment is a static OLE object. If MAPI_OLE_STATIC is not set, the attachment is an embedded OLE object.

MAPI_OLE_STATIC
Indicates the attachment is a static OLE object.

If neither flag is set, the attachment is treated as a data file.

nPosition
Contains an integer used to indicate where in the message text to render the attachment. Attachments replace the character found at a certain position in the message text. That is, attachments replace the character in the **MapiMessage** structure field **NoteText[nPosition]**. A value of - 1 (0xFFFFFFFF) means the attachment position is not indicated; the client application will have to provide a way for the user to access the attachment.

lpszPathName
Points to the fully qualified path of the attached file. This path should include the disk drive letter and directory name.

lpszFileName
Points to the attachment filename seen by the recipient, which may differ from the filename in the **lpszPathName** member if temporary files are being used. If the **lpszFileName** member is empty or NULL, the filename from **lpszPathName** is used.

lpFileType
Points to the attachment file type, which can be represented with a **MapiFileTagExt** structure. A value of NULL indicates an unknown file type or a file type determined by the operating system.

Remarks

Simple MAPI works with three kinds of embedded attachments:

- Data file attachments
- Editable OLE object file attachments
- Static OLE object file attachments

Data file attachments are simply data files. OLE object file attachments are OLE objects that are displayed in the message text. If the OLE attachment is editable, the recipient can double-click it and its source application will be started to handle the edit session. If the OLE attachment is static, the object cannot be edited. The flag set in the **flFlags** member of the **MapiFileDesc** structure determines the kind of a particular attachment. Embedded messages can be identified by a .MSG extension in the **lpszFileName** member.

OLE object files are file representations of OLE object streams. The client application can recreate an OLE object from the file by calling the OLE function **OleLoadFromStream** with an OLESTREAM object that reads the file contents. If an OLE file attachment is included in an outbound message, the OLE object stream should be written directly to the file used as the attachment.

When using the **MapiFileDesc** member **nPosition**, the client application should not place two attachments in the same location. Client applications might not display file attachments at positions beyond the end of the message text.

See Also

MapiFileTagExt structure

MapiFileTagExt (Simple MAPI)

A **MapiFileTagExt** structure specifies a message attachment's type at its creation and its current form of encoding so that it can be restored to its original type at its destination.

```
typedef struct {
     ULONG ulReserved;
     ULONG cbTag;
     LPBYTE lpTag;
     ULONG cbEncoding;
     LPBYTE lpEncoding
} MapiFileTagExt, FAR *lpMapiFileTagExt;
```

Members

ulReserved
Reserved; must be zero.

cbTag
Indicates the size, in bytes, of the value defined by the **lpTag** member.

lpTag
Points to an X.400 object identifier indicating the type of the attachment in its original form, for example "Microsoft Excel worksheet".

cbEncoding
Indicates the size, in bytes, of the value defined by the **lpEncoding** member.

lpEncoding
Points to an X.400 object identifier indicating the form in which the attachment is currently encoded, for example MacBinary, UUENCODE, or binary.

Remarks

A **MapiFileTagExt** structure defines the type of an attached file for purposes such as encoding and decoding the file, choosing the correct application to launch when opening it, or any use that requires full information regarding the file type. Client applications can use information in the **lpTag** and **lpEncoding** members of this structure to determine what to do with an attachment.

See Also

PR_ATTACH_TAG property **PR_ATTACH_ENCODING** property **MapiFileDesc** structure

MapiMessage (Simple MAPI)

A **MapiMessage** structure contains information about a message.

```
typedef struct {
     ULONG ulReserved;
     LPTSTR lpszSubject;
     LPTSTR lpszNoteText;
     LPTSTR lpszMessageType;
     LPTSTR lpszDateReceived;
     LPTSTR lpszConversationID;
     FLAGS flFlags;
     lpMapiRecipDesc lpOriginator;
     ULONG nRecipCount;
     lpMapiRecipDesc lpRecips;
     ULONG nFileCount;
     lpMapiFileDesc lpFiles;
} MapiMessage, FAR *lpMapiMessage;
```

Members

ulReserved
Reserved; must be zero.

lpszSubject
Points to the text string describing the message subject, typically limited to 256 characters or less. If this member is empty or NULL, the user has not entered subject text.

lpszNoteText
Points to a string containing the message text. If this member is empty or NULL, there is no message text.

lpszMessageType
Points to a string indicating a non-IPM type of message. Client applications can select message types for their non-IPM messages. Clients that only support IPM messages can ignore the **lpszMessageType** member when reading messages and set it to empty or NULL when sending messages.

lpszDateReceived
Points to a string indicating the date when the message was received. The format is YYYY/MM/DD HH:MM, using a 24-hour clock.

lpszConversationID
Points to a string identifying the conversation thread to which the message belongs. Some messaging systems can ignore and not return this member.

flFlags
Contains a bitmask of message status flags. The following flags can be set:

MAPI_RECEIPT_REQUESTED
Indicates a receipt notification is requested. Client applications set this flag when sending a message.

MAPI_SENT
Indicates the message has been sent.

MAPI_UNREAD
Indicates the message has not been read.

lpOriginator
Points to a **MapiRecipDesc** structure containing information about the sender of the message.

nRecipCount

Indicates the number of message recipient structures in the array pointed to by the **lpRecips** member. A value of zero indicates no recipients are included.

lpRecips

Points to an array of **MapiRecipDesc** structures, each containing information about a message recipient.

nFileCount

Indicates the number of structures describing file attachments in the array pointed to by the **lpFiles** member. A value of zero indicates no file attachments are included.

lpFiles

Points to an array of **MapiFileDesc** structures, each containing information about a file attachment.

See Also

MapiFileDesc structure, **MapiRecipDesc** structure

MapiRecipDesc (Simple MAPI)

A **MapiRecipDesc** structure contains information about a message sender or recipient.

```
typedef struct {
     ULONG ulReserved
     ULONG ulRecipClass;
     LPTSTR lpszName;
     LPTSTR lpszAddress;
     ULONG ulEIDSize;
     LPVOID lpEntryID;
} MapiRecipDesc, FAR *lpMapiRecipDesc;
```

Members

ulReserved
Reserved; must be zero.

ulRecipClass
Contains a numeric value that indicates the type of recipient. Possible values are:

Value	Constant	Meaning
0	MAPI_ORIG	Indicates the original sender of the message.
1	MAPI_TO	Indicates a primary message recipient.
2	MAPI_CC	Indicates a recipient of a message copy.
3	MAPI_BCC	Indicates a recipient of a blind copy.

lpszName
Points to the display name of the message recipient or sender.

lpszAddress
Optional pointer to the recipient or sender's address; this address is provider-specific message delivery data. Generally, the messaging system provides such addresses for inbound messages. For outbound messages, the **lpszAddress** member can point to an address entered by the user for a recipient not in an address book (that is, a custom recipient).

The format of an address pointed to by the **lpszAddress** member is [*address type*][*e-mail address*]. Examples of valid addresses are FAX:206-555-1212 and SMTP:M@X.COM.

ulEIDSize
Indicates the size, in bytes, of the entry identifier pointed to by the **lpEntryID** member.

lpEntryID
Points to an opaque entry identifier used by a messaging system service provider to identify the message recipient. Entry identifiers have meaning only for the service provider; client applications will not be able to decipher them. The messaging system uses this member to return valid entry identifiers for all recipients or senders listed in the address book.

Functions for Visual Basic

Visual Basic uses a different set of calling and programming conventions than C and C++ use. Different structure and parameter definitions support the Visual Basic representation of strings and of structures, which in Visual Basic are called *types*. The following list describes how programming Simple MAPI Visual Basic applications differs from programming Simple MAPI C and C++ applications:

- Because the concept of a pointer is foreign to Visual Basic, developers use extra function parameters instead of the complex pointer structures used in C and C++.
- Because the Visual Basic MAPI functions are declared, it is not necessary to explicitly cast passed arguments using **ByVal**.
- An empty string in a string variable is equivalent to a NULL value.
- Arrays must be dynamically declared so that they are redimensioned when the Simple MAPI function is executed.
- Visual Basic manages memory, eliminating the need for calling the **MAPIFreeBuffer** function.
- All structures used in the Visual Basic version of Simple MAPI are Visual Basic types rather than C-language structures.
- All strings used in the Visual Basic version of Simple MAPI are Visual Basic strings rather than C-language strings.

The Simple MAPI functions for Visual Basic work with Visual Basic 3, Visual Basic 4, and Visual Basic for Applications. Note that slight differences in the 16-bit and 32-bit Visual Basic runtime DLLs mean that some Simple MAPI functions have different declarations depending on which runtime is being used. The 32-bit declarations use explicit Visual Basic array notation. The alternate declarations are documented with the functions that have them.

MAPIAddress (VB)

The Visual Basic **MAPIAddress** function enables users to create or modify a set of recipients. **MAPIAddress** generates an address-book dialog box that shows the contents of the recipient set and allows the user to select new entries or change existing entries.

MAPIAddress(

Session as **Long**,
UIParam as **Long**,
Caption as **String**,
EditFields as **Long**,
Label as **String**,
RecipCount as **Long**,
*Recipients***()** as **MapiRecip**,
Flags as **Long**,
Reserved as **Long**) as **Long**

Parameters

Session
Input parameter specifying either a session handle that represents a Simple MAPI session or zero. If the value of the *Session* parameter is zero, MAPI logs on the user and creates a session that exists only for the duration of the call. This temporary session can be an existing shared session or a new one. If necessary, a logon dialog box is displayed.

UIParam
Input parameter specifying either a parent window handle or zero, indicating that if a dialog box is displayed, it is application modal. If no dialog box is displayed during the call, the *UIParam* parameter is ignored.

Caption
Input parameter specifying the caption of the address-list dialog box. If this parameter is an empty string, the default value "Address Book" is used.

EditFields
Input parameter specifying the number of edit controls that should be present in the address list. The values 0 through 4are valid. If the value of the *nEditFields* parameter is 4, each recipient class supported by the underlying messaging system has an edit control. If the value of *EditFields* is zero, only address list browsing is possible. Values of 1, 2, or 3 control the number of edit controls present.

However, if the number of recipient classes in the *Recipients* parameter is greater than the value of *EditFields*, the number of classes in *Recipients* is used to indicate the number of edit controls instead of the value of *EditFields.* If the value of *EditFields* is 1 and more than one kind of entry exists in *Recipients*, then the *Labels* parameter is ignored.

Entries selected for the different controls are differentiated by the **ulRecipClass** member in the returned recipient structure.

Label
Input parameter specifying an edit control label in the address-list dialog box. The *Label* parameter is ignored and should be an empty string except when the value of the *EditFields* parameter is 1. If you want a default control label "To:", *Label* should be an empty string.

RecipCount
Input parameter specifying the number of entries in the *Recipients* parameter. If the value of the *RecipCount* parameter is zero, *Recipients* is ignored.

Recipients
Input parameter specifying the initial array of recipient entries to be used to populate edit controls in the address-list dialog box. Recipient entries need not be grouped by recipient class. If the value of

the greatest recipient class present is greater than the value of the *EditFields* parameter, the *EditFields* and *Label* parameters are ignored. This array is redimensioned as necessary to accommodate the entries made by the user in the address-list dialog box.

Flags

Input parameter containing a bitmask of flags. The following flags can be set:

MAPI_LOGON_UI

Indicates that a dialog box should be displayed to prompt the user to log on if required. When the MAPI_LOGON_UI flag is not set, the client application does not display a logon dialog box and returns an error value if the user is not logged on.

MAPI_NEW_SESSION

Indicates an attempt should be made to create a new session rather than acquire the environment's shared session. If the MAPI_NEW_SESSION flag is not set, **MAPIAddress** uses an existing shared session.

Reserved

Reserved; must be zero.

Return Values

MAPI_E_FAILURE

One or more unspecified errors occurred while building recipient lists or browsing the address book. No list of recipients was returned.

MAPI_E_INSUFFICIENT_MEMORY

There was insufficient memory to proceed. No list of recipients was returned.

MAPI_E_INVALID_EDITFIELDS

The value of the *nEditFields* parameter was outside the range of 0 through 4. No list of recipients was returned.

MAPI_E_INVALID_RECIPS

One or more of the recipients in the address list was not valid or the *Recipients* parameter was not a valid array. No list of recipients was returned.

MAPI_E_INVALID_SESSION

An invalid session handle was used for the *lhSession* parameter. No list of recipients was returned.

MAPI_E_LOGIN_FAILURE

There was no default logon, and the user failed to log on successfully when the logon dialog box was displayed. No list of recipients was returned.

MAPI_E_NOT_SUPPORTED

The operation was not supported by the underlying messaging system. A list of recipients might have been returned.

MAPI_E_USER_ABORT

The user canceled one of the dialog boxes. No list of recipients was returned.

SUCCESS_SUCCESS

The call succeeded and a list of address entries was returned.

Remarks

The **MAPIAddress** function makes it possible for users to create or modify a set of address-list entries using a standard address-list dialog box. The dialog box cannot be suppressed, but function parameters allow the caller to set characteristics of the dialog box.

The call is made with an initial, and possibly empty, set of recipients. The address-list dialog box shows the contents of the recipient set; users can choose new entries to add to the set. The final set of recipients is returned to the caller in the *RecipCount* and *Recipients* parameters, destroying their initial values.

The declaration of this function for the 32-bit Visual Basic runtime is:

MAPIAddress(

Session As Long,
    UIParam As Long,
    Caption As String,
    nEditFields As Long,
    Label As String,
    nRecipients As Long,
    Recips() As MapiRecip,
    Flags As Long,
    Reserved As Long) **As Long**

MAPIDeleteMail (VB)

The Visual Basic **MAPIDeleteMail** function deletes a message.

MAPIDeleteMail(

Session as **Long**,
UIParam as **Long**,
MessageID as **String**,
Flags as **Long**,
Reserved as **Long**) as **Long**

Parameters

Session
Input parameter specifying a session handle that represents a valid Simple MAPI session. The value of the *Session* parameter cannot be zero.

UIParam
Input parameter specifying either a parent window handle or zero, indicating that if a dialog box is displayed, it is application modal. If no dialog box is displayed during the call, the *UIParam* parameter is ignored.

MessageID
Input parameter specifying the identifier for the message to be deleted. This string identifier is messaging system-specific and will be invalid when the **MAPIDeleteMail** function successfully returns. Both the **MAPIFindNext** and **MAPISaveMail** functions return message identifiers.

Flags
Reserved; must be zero.

Reserved
Reserved; must be zero.

Return Values

MAPI_E_FAILURE
One or more unspecified errors occurred while deleting the message. No message was deleted.

MAPI_E_INSUFFICIENT_MEMORY
There was insufficient memory to proceed. No message was deleted.

MAPI_E_INVALID_MESSAGE
An invalid message identifier was passed in for the *MessageID* parameter. No message was deleted.

MAPI_E_INVALID_SESSION
An invalid session handle was passed in for the *Session* parameter. No message was deleted.

SUCCESS_SUCCESS
The call succeeded and the message was deleted.

Remarks

To find the message to be deleted, call the **MAPIFindNext** function before calling **MAPIDeleteMail**.

The declaration of this function for the 32-bit Visual Basic runtime is:

MAPIDeleteMail(

ByVal *Session*&,
ByVal *UIParam*&,
ByVal *MsgID*$,
ByVal *Flags*&,

ByVal *Reserved*&) **As Long**

MAPIDetails (VB)

The Visual Basic **MAPIDetails** function displays a dialog box containing the details of a selected address-list entry.

MAPIDetails(

Session as **Long**,
UIParam as **Long**,
Recipient as **MapiRecip**,
Flags as **Long**,
Reserved as **Long**) as **Long**

Parameters

Session
Input parameter specifying either a session handle that represents a Simple MAPI session or zero. If the value of the *Session* parameter is zero, MAPI logs on the user and creates a session that exists only for the duration of the call. This temporary session can be an existing shared session or a new one. If necessary, a logon dialog box is displayed.

UIParam
Input parameter specifying either a parent window handle or zero, indicating that if a dialog box is displayed, it is application modal. If no dialog box is displayed during the call, the *UIParam* parameter is ignored.

Recipient
Input parameter specifying a recipient descriptor containing the entry whose details are to be displayed. **MAPIDetails** ignores all members of the **MapiRecip** type except **EIDSize** and **EntryID**. If the value of the **EIDSize** member is non-zero, **MAPIDetails** resolves the recipient entry. If the value of **EIDSize** is zero, the MAPI_E_AMBIGUOUS_RECIPIENT value is returned.

Flags
Input parameter containing a bitmask of flags. The following flags can be set:

MAPI_AB_NOMODIFY
Indicates the caller is requesting that the dialog box be read-only, prohibiting changes. **MAPIDetails** might or might not honor the request.

MAPI_LOGON_UI
Indicates that a dialog box should be displayed to prompt the user to log on if required. When the MAPI_LOGON_UI flag is not set, the client application does not display a logon dialog box and returns an error value if the user is not logged on.

MAPI_NEW_SESSION
Indicates an attempt should be made to create a new session rather than acquire the environment's shared session. If the MAPI_NEW_SESSION flag is not set, **MAPIDetails** uses an existing shared session.

Reserved
Reserved; must be zero.

Return Values

MAPI_E_AMBIGUOUS_RECIPIENT
The recipient requested has not been or could not be resolved to a unique address list entry.

MAPI_E_FAILURE
One or more unspecified errors occurred. No dialog box was displayed.

MAPI_E_INSUFFICIENT_MEMORY
There was insufficient memory to proceed. No dialog box was displayed.

MAPI_E_INVALID_RECIPS

The recipient specified in the *Recipient* parameter was unknown. No dialog box was displayed.

MAPI_E_LOGIN_FAILURE
: There was no default logon, and the user failed to log on successfully when the logon dialog box was displayed. No dialog box was displayed.

MAPI_E_NOT_SUPPORTED
: The operation was not supported by the underlying messaging system.

MAPI_E_USER_ABORT
: The user canceled either the logon dialog box or the details dialog box.

SUCCESS_SUCCESS
: The call succeeded and the details dialog box was displayed.

Remarks

The **MAPIDetails** function presents a dialog box that shows the details of a particular address list entry. The display name and address are the minimum attributes that are displayed in the dialog box; more information can be shown depending on the directory to which the entry belongs. The details dialog box cannot be suppressed, but the caller can request that it be read-only or modifiable.

Details can only be shown for resolved address list entries. An entry is resolved if the **EIDSize** member of the **MapiRecip** type is nonzero. Entries are resolved when they are returned by the **MAPIAddress** or **MAPIResolveName** functions and as the result of being recipients of read mail.

The declaration of this function for the 32-bit Visual Basic runtime is:

MAPIDetails(

ByVal *Session*&,
    ByVal *UIParam*&,
    Recipient As MapiRecip,
    ByVal *Flags*&,
    ByVal *Reserved*&) **As Long**

MAPIFindNext (VB)

The Visual Basic **MAPIFindNext** function retrieves the next (or first) message identifier of a specified type of incoming message.

MAPIFindNext(

Session as **Long**,
UIParam as **Long**,
MessageType as **String**,
SeedMessageID as **String,**
Flags as **Long**,
Reserved as **Long,**
MessageID as **String**) as **Long**

Parameters

Session
Input parameter specifying a session handle that represents a Simple MAPI session. The value of the *Session* parameter must represent a valid session; it cannot be zero.

UIParam
Input parameter specifying either a parent window handle or zero, indicating that if a dialog box is displayed, it is application modal. If no dialog box is displayed during the call, the *UIParam* parameter is ignored.

MessageType
Input parameter specifying the type of message to search. To find an interpersonal message (IPM), use an empty string, "".

SeedMessageID
Input parameter specifying the message identifier seed for the request. If the *SeedMessageID* parameter is an empty string, **MAPIFindNext** retrieves the first message that matches the type specified in the *MessageType* parameter.

Flags
Input parameter containing a bitmask of option flags. The following flags can be set:

MAPI_GUARANTEE_FIFO
Indicates the message identifiers returned should be in the order of time received. **MAPIFindNext** calls can take longer if this flag is set. Some implementations cannot honor this request and return the MAPI_E_NO_SUPPORT value.

MAPI_NEW_SESSION
Indicates an attempt should be made to create a new session rather than acquire the environment's shared session. If the MAPI_NEW_SESSION flag is not set, **MAPIFindNext** uses an existing shared session.

MAPI_UNREAD_ONLY
Indicates that only unread messages of the specified type should be enumerated. When this flag is not set, **MAPIFindNext** can return any message of the specified type.

Reserved
Reserved; must be zero.

MessageID
Output parameter specifying the returned message identifier. The *MessageID* parameter is a variable-length string allocated by the caller. To ensure compatibility, allocate 512 characters. A smaller buffer is sufficient only if the returned message identifier is always 64 characters or less.

Return Values

MAPI_E_FAILURE

One or more unspecified errors occurred while matching the message type. The call failed before message type matching could take place.

MAPI_E_INSUFFICIENT_MEMORY
: There was insufficient memory to proceed. No message was found.

MAPI_E_INVALID_MESSAGE
: An invalid message identifier was passed in the *SeedMessageID* parameter. No message was found.

MAPI_E_INVALID_SESSION
: An invalid session handle was passed in the *lhSession* parameter. No message was found.

MAPI_E_NO_MESSAGES
: A matching message could not be found.

SUCCESS_SUCCESS
: The call succeeded and the message identifier was returned.

Remarks

The **MAPIFindNext** function allows a client application to enumerate messages of a given type. This function can be called repeatedly to list all messages in the folder. Message identifiers returned from **MAPIFindNext** can be used in other Simple MAPI calls to retrieve message contents and delete messages. This function is for processing incoming messages, not for managing received messages.

When the value of the *SeedMessageID* parameter is NULL or empty, **MAPIFindNext** returns the message identifier for the first message of the type specified by the *MessageType* parameter. When *SeedMessageID* contains a valid identifier, **MAPIFindNext** returns the next matching message of the type specified by *MessageType*. Repeated calls to **MAPIFindNext** ultimately result in a return of the MAPI_E_NO_MESSAGES value, which means the enumeration is complete.

Because message identifiers are messaging system-specific and can be invalidated at any time, message identifiers are valid only for the current session. If the message identifier passed in with *SeedMessageID* is invalid, **MAPIFindNext** returns the MAPI_E_INVALID_MESSAGE value.

Message type matching is done against message class strings. All message types whose names match, up to the length specified in *MessageType*, are returned.

The declaration of this function for the 32-bit Visual Basic runtime is:

MAPIFindNext(

ByVal *Session*&,
ByVal *UIParam*&,
MsgType$,
SeedMsgID$,
ByVal *Flag*&,
ByVal *Reserved*&,
MsgID$) **As Long**

MAPILogoff (VB)

The Visual Basic **MAPILogoff** function ends a session with the messaging system.

MAPILogoff(

Session as **Long**,
UIParam as **Long**,
Flags as **Long**,
Reserved as **Long**) as **Long**

Parameters

Session
Input parameter specifying a handle for a Simple MAPI session to be terminated. Session handles are returned by **MAPILogon** and invalidated by **MAPILogoff**. The value of the *Session* parameter must represent a valid session; it cannot be zero.

UIParam
Input parameter specifying either a parent window handle or zero, indicating that if a dialog box is displayed, it is application modal. If no dialog box is displayed during the call, the *UIParam* parameter is ignored.

Flags
Reserved; must be zero.

Reserved
Reserved; must be zero.

Return Values

MAPI_E_FAILURE
One or more unspecified errors occurred.

MAPI_E_INSUFFICIENT_MEMORY
There was insufficient memory to proceed. The session was not terminated.

MAPI_E_INVALID_SESSION
An invalid session handle was used for the *Session* parameter. The session was not terminated.

SUCCESS_SUCCESS
The call succeeded and the session was terminated.

The declaration of this function for the 32-bit Visual Basic runtime is:

MAPILogoff(

ByVal *Session*&,
ByVal *UIParam*&,
ByVal *Flags*&,
ByVal *Reserved*&) As Long

MAPILogon (VB)

The Visual Basic **MAPILogon** function begins a Simple MAPI session, loading the default message store and address book providers.

MAPILogon(

UIParam ByVal as **Long**,
User as **String**,
Password as **String,**
Flags as **Long**,
Reserved as **Long,**
Session as **Long**) as **Long**

Parameters

UIParam
Input parameter specifying either a parent window handle or zero, indicating that if a dialog box is displayed, it is application modal. If no dialog box is displayed during the call, the *UIParam* parameter is ignored.

User
Input parameter specifying a client account-name string, limited to 256 characters or less. This is the name to use when logging on. If the *User* parameter is empty, and the *Flags* parameter is set to MAPI_LOGON_UI, **MAPILogon** displays a logon dialog box with an empty name field.

Password
Input parameter specifying a credential string, limited to 256 characters or less. If the messaging system does not require password credentials, or if it requires that the user enter them, the *Password* parameter should be empty. When the user must enter credentials, the *Flags* parameter must be set to MAPI_LOGON_UI to allow a logon dialog box to be displayed.

Flags
Input parameter containing a bitmask of option flags. The following flags can be set:

MAPI_FORCE_DOWNLOAD
Indicates an attempt should be made to download all of the user's messages before returning. If the MAPI_FORCE_DOWNLOAD flag is not set, messages can be downloaded in the background after the function call returns.

MAPI_LOGON_UI
Indicates that a logon dialog box should be displayed to prompt the user for logon information. If the user needs to provide information to enable a successful logon, MAPI_LOGON_UI must be set.

MAPI_NEW_SESSION
Indicates an attempt should be made to create a new session rather than acquire the environment's shared session. If the MAPI_NEW_SESSION flag is not set, **MAPILogon** uses an existing shared session.

Reserved
Reserved; must be zero.

Session
Output parameter specifying a Simple MAPI session handle.

Return Values

MAPI_E_FAILURE
One or more unspecified errors occurred during logon. No session handle was returned.

MAPI_E_INSUFFICIENT_MEMORY
There was insufficient memory to proceed. No session handle was returned.

MAPI_E_LOGIN_FAILURE
There was no default logon, and the user failed to log on successfully when the logon dialog box was displayed. No session handle was returned.

MAPI_E_TOO_MANY_SESSIONS
The user had too many sessions open simultaneously. No session handle was returned.

MAPI_E_USER_ABORT
The user canceled the process. No session handle was returned.

SUCCESS_SUCCESS
The call succeeded and a session was established.

Remarks

The **MAPILogon** function begins a session with the messaging system, returning a handle that can be used in subsequent MAPI calls to explicitly provide user credentials to the messaging system. To request the display of a logon dialog box if the credentials presented fail to validate the session, set the *Flags* parameter to MAPI_LOGON_UI.

The declaration of this function for the 32-bit Visual Basic runtime is:

MAPILogon(

ByVal *UIParam*&,
ByVal *User*$,
ByVal *Password*$,
ByVal *Flags*&,
ByVal *Reserved*&,
Session&) **As Long**

MAPIReadMail (VB)

The Visual Basic **MAPIReadMail** function retrieves a message for reading.

MAPIReadMail(

Session as **Long,**
UIParam as **Long**,
MessageID as **String**,
Flags as **Long**,
Reserved as **Long,**
Message as **MapiMessage,**
Originator as **MapiRecip,**
Recips() as **MapiRecip,**
Files() as **MapiFile**) as **Long**

Parameters

Session
Input parameter specifying a handle to a Simple MAPI session. The value of the *Session* parameter must represent a valid session; it cannot be zero.

UIParam
Input parameter specifying either a parent window handle or zero, indicating that if a dialog box is displayed, it is application modal. If no dialog box is displayed during the call, the *UIParam* parameter is ignored.

MessageID
Input parameter specifying the message identifier of the message to be read. The *MessageID* parameter is a variable-length string that is obtained from the **MAPIFindNext** and **MAPISaveMail** functions.

Flags
Input parameter containing a bitmask of flags. The following flags can be set:

MAPI_BODY_AS_FILE
Indicates **MAPIReadMail** should write the message text to a temporary file and add it as the first attachment in the attachment list.

MAPI_ENVELOPE_ONLY
Indicates **MAPIReadMail** should read the message header only. File attachments are not copied to temporary files, and neither temporary file names nor message text are written. Setting this flag makes **MAPIReadMail** processing faster.

MAPI_PEEK
Indicates **MAPIReadMail** does not mark the message as read. Marking a message as read affects its appearance in the user interface and generates a read receipt. If the messaging system does not support this flag, **MAPIReadMail** always marks the message as read. If **MAPIReadMail** encounters an error, it leaves the message unread.

MAPI_SUPPRESS_ATTACH
Indicates **MAPIReadMail** should not copy file attachments but should write message text into the **MapiMessage** type. **MAPIReadMail** ignores this flag if the calling application has set the MAPI_ENVELOPE_ONLY flag. Setting this flag makes **MAPIReadMail** processing faster.

Reserved
Reserved; must be zero.

Message
Output parameter specifying a type set by **MAPIReadMail** to a message containing the message contents.

Originator

Output parameter specifying the originator of the message.

Recips
Output parameter specifying an array of recipients. This array is redimensioned as necessary to accommodate the number of recipients chosen by the user.

Files
Output parameter specifying an array of attachment files written when the message is read. When **MAPIReadMail** is called, all message attachments are written to temporary files. It is the caller's responsibility to delete these files when they are no longer needed. When MAPI_ENVELOPE_ONLY or MAPI_SUPPRESS_ATTACH is set, no temporary files are written and no temporary names are filled into the file attachment descriptors. This array is redimensioned as necessary to accommodate the number of files attached by the user.

Return Values

MAPI_E_ATTACHMENT_WRITE_FAILURE
An attachment could not be written to a temporary file. Check directory permissions.

MAPI_E_DISK_FULL
The disk was full.

MAPI_E_FAILURE
One or more unspecified errors occurred while reading the message.

MAPI_E_INSUFFICIENT_MEMORY
There was insufficient memory to read the message.

MAPI_E_INVALID_MESSAGE
An invalid message identifier was passed in the *MessageID* parameter.

MAPI_E_INVALID_SESSION
An invalid session handle was passed in the *Session* parameter. No message was retrieved.

MAPI_E_TOO_MANY_FILES
There were too many file attachments in the message. The message could not be read.

MAPI_E_TOO_MANY_RECIPIENTS
There were too many recipients of the message. The message could not be read.

SUCCESS_SUCCESS
The call succeeded and the message was read.

Remarks

The **MAPIReadMail** function returns one message, breaking the message content into the same parameters and types used in the **MAPISendMail** function. **MAPIReadMail** fills a block of memory with the **MapiMessage** type containing message elements. File attachments are saved to temporary files, and the names are returned to the caller in the message type. Recipients, attachments, and contents are copied from the message before **MAPIReadMail** returns to the caller, so later changes to the files do not affect the contents of the message.

A flag is provided to specify that only envelope informationis to be returned from the call. Another flag in the **MapiMessage** type specifies whether the message is marked as sent or unsent.

All strings are null-terminated and must be specified in the current character set or code page of the client application's operating system process. In Microsoft Windows, the character set is ANSI.

The sender, recipients, and file attachments are written into the appropriate parameters of the Visual Basic call. The *Recips* and *Files* parameters should be dynamically allocated arrays of their respective types.

The declaration of this function for the 32-bit Visual Basic runtime is:

MAPIReadMail(

Session As Long,
    UIParam As Long,
    MessageID As String,
    Flags As Long,
    Reserved As Long,
    message As MAPIMessage,
    Orig As MapiRecip,
    RecipsOut() As MapiRecip,
    FilesOut() As MapiFile) **As Long**

MAPIResolveName (VB)

The Visual Basic **MAPIResolveName** function transforms a message recipient's name as entered by a user to an unambiguous address list entry.

MAPIResolveName(

Session as **Long,**
UIParam as **Long**,
UserName as **String**,
Flags as **Long**,
Reserved as **Long,**
Recipient as **MapiRecip**) as **Long**

Parameters

Session
Input parameter specifying either a session handle that represents a Simple MAPI session or zero. If the value of the *Session* parameter is zero, MAPI logs on the user and creates a session that exists only for the duration of the call. This temporary session can be an existing shared session or a new one. If necessary, a logon dialog box is displayed.

UIParam
Input parameter specifying either a parent window handle or zero, indicating that if a dialog box is displayed, it is application modal. If no dialog box is displayed during the call, the *UIParam* parameter is ignored.

UserName
Input parameter specifying the name to be resolved.

Flags
Input parameter containing a bitmask of option flags. The following flags can be set:

MAPI_AB_NOMODIFY
Indicates the caller is requesting that the dialog box be read-only, prohibiting changes. **MAPIResolveName** ignores this flag if MAPI_DIALOG is not set.

MAPI_DIALOG
Indicates that a dialog box should be displayed for name resolution. If this flag is not set and the name cannot be resolved, **MAPIResolveName** returns the MAPI_E_AMBIGUOUS_RECIPIENT value.

MAPI_LOGON_UI
Indicates that a dialog box should be displayed to prompt the user to log on if required. When the MAPI_LOGON_UI flag is not set, the client application does not display a logon dialog box and returns an error value if the user is not logged on.

MAPI_NEW_SESSION
Indicates an attempt should be made to create a new session rather than acquire the environment's shared session. If the MAPI_NEW_SESSION flag is not set, **MAPIResolveName** uses an existing shared session.

Reserved
Reserved; must be zero.

Recipient
Output parameter specifying a recipient-type set returned by **MAPIResolveName** if the resolution results in a single match. The type contains the recipient information of the resolved name. The descriptor can then be used in calls to the **MAPISendMail**, **MAPISaveMail**, and **MAPIAddress** functions.

Return Values

MAPI_E_AMBIGUOUS_RECIPIENT
    The recipient requested has not been or could not be resolved to a unique address list entry.

MAPI_E_FAILURE
    One or more unspecified errors occurred. The name was not resolved.

MAPI_E_INSUFFICIENT_MEMORY
    There was insufficient memory to proceed. The name was not resolved.

MAPI_E_LOGIN_FAILURE
    There was no default logon, and the user failed to log on successfully when the logon dialog box was displayed. The name was not resolved.

MAPI_E_NOT_SUPPORTED
    The operation was not supported by the underlying messaging system.

MAPI_E_USER_ABORT
    The user canceled the resolution. The name was not resolved.

SUCCESS_SUCCESS
    The call succeeded and the name was resolved.

Remarks

The **MAPIResolveName** function resolves a message recipient's name (as entered by a user) to an unambiguous address list entry, optionally prompting the user to choose between possible entries, if necessary. A recipient descriptor containing fully resolved information about the entry is allocated and returned.

The declaration of this function for the 32-bit Visual Basic runtime is:

MAPIResolveName(

ByVal *Session*&,
    ByVal *UIParam*&,
    ByVal *UserName*$,
    ByVal *Flags*&,
    ByVal *Reserved*&,
    Recipient As MapiRecip) **As Long**

MAPISaveMail (VB)

The Visual Basic **MAPISaveMail** function saves a message.

MAPISaveMail(

Session as **Long,**
UIParam as **Long**,
Message as **MapiMessage**,
Recips as **MapiRecip**,
Files as **MapiFile**,
Flags as **Long,**
Reserved ByVal as **Long,**
MessageID as **String**) as **Long**

Parameters

Session
Input parameter specifying either a session handle that represents a Simple MAPI session or zero. If the value for the *Session* parameter is zero, MAPI logs on the user and creates a session that exists only for the duration of the call. This temporary session can be an existing shared session or a new one. If necessary, a logon dialog box is displayed.

UIParam
Input parameter specifying either a parent window handle or zero, indicating that if a dialog box is displayed, it is application modal. If no dialog box is displayed during the call, the *UIParam* parameter is ignored.

Message
Input parameter specifying the contents of the message to be saved. Client applications can either ignore the **Flags** member of the **MapiMessage** type, or if the message has never been saved, can set the MAPI_SENT and MAPI_UNREAD flags.

Recips
Input parameter specifying the first element in an array of recipients. When the value of the **RecipCount** member in the **MapiMessage** type is zero, this parameter is ignored. The recipient string can include either the recipient's name or the recipient's name-address pair. If only a name is specified, the name is resolved to an address using implementation-defined address-book search rules. If an address is also specified, a search for the name is not performed. The address is in an implementation-defined format and is assumed to have been obtained from the implementation some other way. When the address is specified, the name is used for display to the user and the address is used for delivery. When the **EntryID** member for a particular recipient is used, no search is performed and the display-name and address are ignored. (A name and address are associated with the **EntryID** within the messaging system.)

Files
Input parameter specifying the first element in an array of attachment files written when the message is read. The number of attachments per message can be limited in some systems. If the limit is exceeded, the MAPI_E_TOO_MANY_FILES value is returned. When the value of the **FileCount** member in the **MapiMessage** type is zero, this parameter is ignored. Attachment files are read and attached to the message before the call returns. Do not attempt to display attachments outside the range of the message text.

Flags
Input parameter containing a bitmask of option flags. The following flags can be set:

MAPI_LOGON_UI
Indicates that a dialog box should be displayed to prompt the user to log on if required. When the MAPI_LOGON_UI flag is not set, the client application does not display a logon dialog box and returns an error value if the user is not logged on. **MAPISaveMail** ignores this flag if the

MessageID parameter is empty.

MAPI_LONG_MSGID
Indicates that the returned message identifier is expected to be 512 characters. If this flag is set, the *MessageID* parameter must be large enough to accomodate 512 characters.

MAPI_NEW_SESSION
Indicates an attempt should be made to create a new session rather than acquire the environment's shared session. If the MAPI_NEW_SESSION flag is not set, **MAPISaveMail** uses an existing shared session.

Reserved
Reserved; must be zero.

MessageID
Input parameter specifying a variable-length, caller-allocated string identifier for the message, returned either by the **MAPIFindNext** function or a previous call to **MAPISaveMail**, or a null string. If the *MessageID* parameter contains a valid message identifier, the message is overwritten. If *MessageID* contains a null string, a new message is created.

Return Values

MAPI_E_FAILURE
One or more unspecified errors occurred while saving the message. No message was saved.

MAPI_E_BAD_RECIPTYPE
The type of a recipient was not MAPI_TO, MAPI_CC, or MAPI_BCC. No message was sent.

MAPI_E_INSUFFICIENT_MEMORY
There was insufficient memory to save the message. No message was saved.

MAPI_E_INVALID_MESSAGE
An invalid message identifier was passed in the *MessageID* parameter. No message was saved.

MAPI_E_INVALID_SESSION
An invalid session handle was passed in the *Session* parameter. No message was saved.

MAPI_E_LOGIN_FAILURE
There was no default logon, and the user failed to log on successfully when the logon dialog box was displayed. No message was saved.

MAPI_E_NOT_SUPPORTED
The operation was not supported by the underlying messaging system.

MAPI_E_USER_ABORT
The user canceled the process. No message was saved.

SUCCESS_SUCCESS
The call succeeded and the message was saved.

Remarks

To replace an existing message, the caller first calls the **MAPIFindNext** function to locate the message to be replaced and then calls the **MAPISaveMail** function with the *MessageID* parameter set with a valid message identifier. The elements of the message identified by *MessageID* are replaced by the elements in the **MapiMessage** type pointed to by the *Message* parameter.

To create a new message, the caller passes an empty string for *MessageID*. New messages are saved in the folder appropriate for incoming messages of that class. The new message identifier is returned in *MessageID* on completion.

The *MessageID* parameter must be a variable-length string. The elements of the message identified by *MessageID* are replaced by the elements in the *Message* parameter. If *MessageID* is empty, a new message is created.

MAPISaveMail takes the recipients and file attachments from the *Recips* and *Files* parameters, which should each be the first element of dynamically allocated arrays of their respective types. These arrays are not redimensioned.

The declaration of this function for the 32-bit Visual Basic runtime is:

MAPISaveMail(

ByVal *Session*&,
    ByVal *UIParam*&,
    message As MAPIMessage,
    Recipient() As MapiRecip,
    File() As MapiFile,
    ByVal *Flags*&,
    ByVal *Reserved*&,
    MsgID$) **As Long**

MAPISendDocuments (VB)

The Visual Basic **MAPISendDocuments** function sends a standard message with one or more attached files and a cover note. The cover note is a dialog box that allows the user to enter a list of recipients and an optional message.

MAPISendDocuments(

UIParam as **Long**,
DelimChar as **String**,
FullPaths as **String**,
FileNames as **String**,
Reserved as **Long**) as **Long**

Parameters

UIParam
Input parameter specifying either a parent window handle or zero, indicating that if a dialog box is displayed, it is application modal. If no dialog box is displayed during the call, the *UIParam* parameter is ignored.

DelimChar
Input parameter specifying a string containing the character used to delimit the names in the *FullPaths* and *FileNames* parameters. This character should not be used in filenames on your operating system.

FullPaths
Input parameter specifying a string containing the list of full paths, including drive letters, for the attached files. The list is formed by concatenating correctly formed file paths separated by the character specified in the *DelimChar* parameter. An example of a valid list is:

```
C:\TMP\TEMP1.DOC;C:\TMP\TEMP2.DOC
```

The files specified in *FullPaths* are added to the message as file attachments. If *FullPaths* contains an empty string, the Send Note dialog box is displayed with no attached files.

FileNames
Input parameter specifying a string containing the list of the original filenames as they should be displayed in the message. When multiple names are specified, the list is formed by concatenating the filenames separated by the character specified in the *DelimChar* parameter. An example is:

```
MEMO.DOC;EXPENSES.DOC
```

If there is no value for the *FileNames* parameter or if it is empty, **MAPISendDocuments** sets the filenames set to the filename values indicated by the *FullPaths* parameter.

Reserved
Reserved; must be zero.

Return Values

MAPI_E_ATTACHMENT_NOT_FOUND
An attachment could not be located in the specified path. Either the drive letter was invalid, the path was not found on that drive, or the file was not found in that path.

MAPI_E_ATTACHMENT_OPEN_FAILURE
One or more files in the *FullPaths* parameter could not be located. No message was sent.

MAPI_E_ATTACHMENT_WRITE_FAILURE
An attachment could not be written to a temporary file. Check directory permissions.

MAPI_E_FAILURE
One or more unspecified errors occurred while sending the message. It is not known if the message

was sent.

MAPI_E_INSUFFICIENT_MEMORY
There was insufficient memory to proceed.

MAPI_E_LOGIN_FAILURE
There was no default logon, and the user failed to log on successfully when the logon dialog box was displayed. No message was sent.

MAPI_E_USER_ABORT
The user canceled the process. No message was sent.

SUCCESS_SUCCESS
The call succeeded and the message was sent.

Remarks

Calling the **MAPISendDocuments** function displays a Send Note dialog box, which prompts the user to send a message with data file attachments. Attachments can include the active document or all the currently open documents in the Windows-based application that called **MAPISendDocuments**. This function is used primarily for calls from a macro or scripting language, often found in applications such as spreadsheet or word-processing programs.

There is no default identification when **MAPISendDocuments** is called; a standard logon dialog box appears. After the user provides a mailbox name and password, the Send Note dialog box appears.

The user's default messaging options are used as the default dialog box values. The caller is responsible for deleting temporary files created when using **MAPISendDocuments**.

The declaration of this function for the 32-bit Visual Basic runtime is:

MAPISendDocuments(

ByVal *UIParam*&,
ByVal *DelimStr*$,
ByVal *FilePaths*$,
ByVal *FileNames*$,
ByVal *Reserved*&) **As Long**

MAPISendMail (VB)

The Visual Basic **MAPISendMail** function sends a standard message.

MAPISendMail(

Session as **Long,**.
UIParam as **Long**,
Message as **MapiMessage**,
Recips as **MapiRecip**,
Files as **MapiFile**,
Flags as **Long,**
Reserved as **Long**) as **Long**

Parameters

Session

Input parameter specifying either a session handle that represents a Simple MAPI session or zero. If the value of the *Session* parameter is zero, MAPI logs on the user and creates a session that exists only for the duration of the call. This temporary session can be an existing shared session or a new one. If necessary, a logon dialog box is displayed.

UIParam

The parent window handle for the dialog box. A value of zero indicates that any dialog box displayed is application modal.

Message

Input parameter specifying the message to be sent. An empty string indicates no text. Each paragraph should be terminated with either a carriage return (0x0d), a line feed (0x0a), or a carriage return-line feed pair (0x0d0a). The implementation wraps lines as appropriate. Implementations can place limits on the size of the text. The MAPI_E_TEXT_TOO_LARGE value is returned if this limit is exceeded. Client applications can set MAPI_RECEIPT_REQUESTED in the **Flags** member in the **MapiMessage** type pointed to by the *Message* parameter to prompt for a read report.

Recips

Input parameter specifying the first element of an array of recipients. When the the value of the **RecipCount** member in the **MapiRecip** type pointed to by the *Message* parameter is zero, the *Recips* parameter is ignored. The *Recips* parameter can include either an entry identifier, the recipient's name, an address, or a name and address pair. Depending on the type and amount of information passed, **MAPISendMail** will perform varied levels of name resolution. If an entry identifier in the **EntryID** member for a particular recipient is specified, **MAPISendMail** performs no lookup and ignores the name and address. If only a name is specified, **MAPISendMail** resolves the name to a valid address using name resolution rules defined by Simple MAPI. If only an address is specified, **MAPISendMail** uses this address for both message delivery and for displaying the recipient name; no name resolution occurs. If both a name and address are specified, again **MAPISendMail** does not resolve the name. The specified name is used as the display name and not for resolution.

Files

Input parameter specifying the first element of an array of attachment files written when the message is read. The number of attachments per message might be limited in some systems. If the limit is exceeded, the MAPI_E_TOO_MANY_FILES value is returned. When the value of the **FileCount** member in the **MapiMessage** type pointed to by the *Message* parameter is zero, the *Files* parameter is ignored. Attachment files are read and attached to the message before the call returns. Do not attempt to display attachments outside the range of the message text.

Flags

Input parameter containing a bitmask of option flags. The following flags can be set:

MAPI_DIALOG

Indicates that a dialog box should be displayed to prompt the user for recipients and other sending options. Set the MAPI_LOGON_UI flag if **MAPISendMail** should display a dialog box to prompt the user to log on. When this flag is not set, **MAPISendMail** does not display a dialog box and returns a message if the user is not logged on.

MAPI_LOGON_UI
Indicates that a dialog box should be displayed to prompt the user to log on if required. When the MAPI_LOGON_UI flag is not set, the client application does not display a logon dialog box and returns an error value if the user is not logged on. **MAPISaveMail** ignores this flag if the *MessageID* parameter is empty.

MAPI_NEW_SESSION
Indicates an attempt should be made to create a new session rather than acquire the environment's shared session. If the MAPI_NEW_SESSION flag is not set, **MAPISendMail** uses an existing shared session.

Reserved
Reserved; must be zero.

Return Values

MAPI_E_AMBIGUOUS_RECIPIENT
A recipient matched more than one of the recipient descriptor structures and MAPI_DIALOG was not set. No message was sent.

MAPI_E_ATTACHMENT_NOT_FOUND
The specified attachment was not found. No message was sent.

MAPI_E_ATTACHMENT_OPEN_FAILURE
The specified attachment could not be opened. No message was sent.

MAPI_E_FAILURE
One or more unspecified errors occurred. No message was sent.

MAPI_E_INSUFFICIENT_MEMORY
There was insufficient memory to proceed. No message was sent.

MAPI_E_LOGIN_FAILURE
There was no default logon, and the user failed to log on successfully when the logon dialog box was displayed. No message was sent.

MAPI_E_TEXT_TOO_LARGE
The text in the message was too large. No message was sent.

MAPI_E_TOO_MANY_FILES
There were too many file attachments. No message was sent.

MAPI_E_TOO_MANY_RECIPIENTS
There were too many recipients. No message was sent.

MAPI_E_UNKNOWN_RECIPIENT
A recipient did not appear in the address list. No message was sent.

MAPI_E_USER_ABORT
The user canceled the process. No message was sent.

SUCCESS_SUCCESS
The call succeeded and the message was sent.

Remarks

The **MAPISendMail** function sends a standard message, with or without any user interaction. If recipient names, file attachments, or message text is provided, **MAPISendMail** can send the files or note without prompting users. If the optional parameters are specified and a dialog box is requested by use of the MAPI_DIALOG flag, the parameters provide the initial values for the dialog box.

File attachments are copied to the message before **MAPISendMail** returns; therefore, later changes to the files do not affect the contents of the message. The files must be closed when they are copied.

MAPISendMail takes the recipients and file attachments from the *Recips* and *Files* parameters, which should each be the first element of dynamically allocated arrays of their respective types. These arrays are not redimensioned.

All strings must be specified in the current character set or code page of the client application's operating system process.

The declaration of this function for the 32-bit Visual Basic runtime is:

MAPISendMail(

ByVal *Session*&,
    ByVal *UIParam*&,
    message As MAPIMessage,
    Recipient() As MapiRecip,
    File() As MapiFile,
    ByVal *Flags*&,
    ByVal *Reserved*&) **As Long**

Data Types for Visual Basic

Visual Basic uses a different set of calling and programming conventions than C and C++ use. Different structure and parameter definitions support the Visual Basic representation of strings and of structures, which in Visual Basic are called *types*. The following list describes how programming Simple MAPI Visual Basic applications differs from programming Simple MAPI C and C++ applications:

- In C and C++, structures can contain pointers to other structures. Because the concept of a pointer is foreign to Visual Basic, extra function parameters are used instead of these complex structures.
- Because the Visual Basic MAPI functions are declared, it is not necessary to explicitly cast passed arguments using **ByVal**.
- An empty string in a string variable is equivalent to a NULL value.
- Arrays must be dynamically declared so that they are redimensioned when the Simple MAPI function is executed.
- Visual Basic manages memory, eliminating the need to call the **MAPIFreeBuffer** function.
- All structures used in the Visual Basic version of Simple MAPI are Visual Basic types rather than C-language structures.
- All strings used in the Visual Basic version of Simple MAPI are Visual Basic strings rather than C-language strings.

The following alphabetized entries contain documentation for the Visual Basic Data Types

MapiFile (VB)

The Visual Basic **MapiFile** type contains file attachment information.

```
Type MapiFile
     Reserved as Long
     Flags as Long
     Position as Long
     PathName as String
     FileName as String
     FileType as String
End Type
```

Members

Reserved
Reserved; must be zero.

Flags
A bitmask of flags. The following flags can be set:

MAPI_OLE
Indicates the attachment is an OLE object file attachment. If MAPI_OLE_STATIC is also set, the object is static. If neither flag is set, the attachment is simply a data file.

MAPI_OLE_STATIC
Indicates the attachment is a static OLE object file attachment.

Position
Contains an integer used to determine where the attachment should be placed in the message text. Attachments replace the character found at a certain position in the message text; in other words, attachments replace the **MapiMessage** member **NoteText[Position]**. Applications cannot place two attachments in the same location within a message, and attachments cannot be placed beyond the end of the message text. **MAPIReadMail** does not return an attachment with the value of **Position** equal to -1 unless the MAPI_BODY_AS_FILE flag is set.

PathName
Contains the full path of the attached file. The file should be closed before this call is made.

FileName
Contains the filename seen by the recipient. This name can differ from the filename in the **PathName** member if temporary files are being used. If the **FileName** member is empty, the filename from **PathName** is used. If the attachment is an OLE object, **FileName** contains the class name of the object, such as "Microsoft Excel Worksheet."

FileType
A reserved descriptor that indicates to the recipient the type of the attached file. An empty string indicates an unknown or operating system-determined file type. With this release, you must use an empty string, "", for this parameter.

Remarks

Simple MAPI for Visual Basic supports the following kinds of attachments:

- Data files
- Embedded OLE objects
- Static OLE objects

The **Flags** member determines the kind of attachment. OLE object files are file representations of OLE object streams. You can re-create an OLE object from the file by calling the OLE function **OleLoadFromStream** with an OLESTREAM object that reads the file contents. If an OLE file attachment is included in an outbound message, the OLE object stream should be written directly to the file used as the attachment.

MapiMessage (VB)

The Visual Basic **MapiMessage** type contains message information.

```
Type MapiMessage
     Reserved as Long
     Subject as String
     NoteText as String
     MessageType as String
     DateReceived as String
     ConversiondID as String
     Flags as Long
     Originator as Long
     RecipCount as Long
     FileCount as Long
End Type
```

Members

Reserved
Reserved; must be zero.

Subject
Contains the subject text, limited to 256 characters or less. Messages saved with the **MAPISaveMail** function are not limited to 256 characters. An empty string indicates no subject text.

NoteText
Contains a string containing text in the message. An empty string indicates no text. For inbound messages, each paragraph is terminated with a carriage return-line feed pair (0x0d0a). For outbound messages, paragraphs can be delimited with a carriage return, a line feed, or a carriage return-line feed pair (0x0d, 0x0a, or 0x0d0a).

MessageType
Contains a message type string used by applications other than interpersonal electronic mail. An empty string indicates an interpersonal message (IPM) type.

DateReceived
Contains a string indicating the date a message is received. The format is YYYY/MM/DD HH:MM; hours are measured on a 24-hour clock.

ConversationID
Contains a string indicating the conversation thread identifier to which this message belongs.

Flags
Contains a bitmask of flags. The following flags can be set:

MAPI_RECEIPT_REQUESTED
Indicates a receipt notification is requested.

MAPI_SENT
Indicates the message has been sent.

MAPI_UNREAD
Indicates the message has not been read.

Originator
Contains a **MapiFile** type describing the sender of the message.

RecipCount
Contains a count of the recipient descriptor types. A value of 0 indicates that no recipients are included.

FileCount
Contains a count of the file attachment descriptor types. A value of 0 indicates that no file

attachments are included.

MapiRecip (VB)

The Visual Basic **MAPIRecip** type contains recipient information.

```
Type MapiRecip
     Reserved as Long
     RecipClass as Long
     Name as String
     Address as String
     EIDSize as Long
     EntryID as String
End Type
```

Members

Reserved
Reserved; must be zero.

RecipClass
Classifies the recipient of the message. (Messages can be sorted by recipient class.) This member can also contain information about the originator of an inbound message.

Name
Contains the name of the recipient that is displayed by the messaging system.

Address
Contains provider-specific message delivery data. This can be used by the messaging system to identify custom recipients who are not in an address list.

EIDSize
Indicates the size, in bytes, of the data in the **EntryID** member.

EntryID
Contains a string used by the messaging system to uniquely identify the recipient. Unlike the contents of the **Address** member, this data is opaque and is not printable. The messaging system returns valid **EntryID** members for recipients or senders included in the address list.

Property Identifiers and Types

All MAPI properties are represented by property tags. A property tag is a 32-bit unsigned integer value that contains the property's identifier in the high order 16 bits and the property's type in the low order 16 bits. Property tags for all of the properties defined by MAPI are included in the MAPITAGS.H header file.

Property identifiers are used to indicate what a property is used for and who is responsible for it. Property identifiers are divided by MAPI into ranges; where an identifier falls in the range indicates its use and ownership.

Property types are used to indicate the format of the property's data. MAPI defines all of the valid types. Clients and service providers creating new properties must use one of these types. All of the property types are included in the MAPIDEFS.H header file.

List of Property Identifier Ranges

The following table summarizes the different ranges for property identifiers, describing the owner for the properties in each range.

Identifier range	Description
0000	Reserved by MAPI for the special value PR_NULL.
0001 - 0BFF	Message envelope properties defined by MAPI.
0C00 - 0DFF	Recipient properties defined by MAPI.
0E00 - 0FFF	Non-transmittable message properties defined by MAPI.
1000 - 2FFF	Message content properties defined by MAPI.
3000 - 3FFF	Properties for objects other than messages and recipients defined by MAPI.
4000 - 57FF	Message envelope properties defined by transport providers.
5800 - 5FFF	Recipient properties defined by transport and address book providers.
6000 - 65FF	Non-transmittable message properties defined by clients.
6600 - 67FF	Non-transmittable properties defined by a service provider. These properties can be visible or invisible to users.
6800 - 7BFF	Message content properties for custom message classes defined by creators of those classes.
7C00 - 7FFF	Non-transmittable properties for custom message classes defined by creators of those classes.
8000 - FFFE	Properties defined by clients and occasionally service providers that are identified by name through the **IMAPIProp::GetNamesFromIDs** and **IMAPIProp::GetIDsFromNames** methods.
FFFF	Reserved by MAPI for the special error value PROP_ID_INVALID.

The range between 3000 and 3FFF is reserved for properties that are not related to either messages or recipients. MAPI divides this range into sub-ranges by types of object; the following table shows this further breakdown.

Identifier range	Type of property
3000 - 33FF	Common properties that appear on multiple objects, such as PR_DISPLAY_NAME and PR_ENTRYID.
3400 - 35FF	Message store properties
3600 - 36FF	Folder and address book container properties
3700 - 38FF	Attachment properties
3900 - 39FF	Address book properties
3A00 - 3BFF	Messaging user properties

3C00 - 3CFF	Distribution list properties
3D00 - 3DFF	Profile properties
3E00 - 3FFF	Status object properties

List of Property Types

MAPI supports both single-valued and multivalued properties. With a single-valued property, there is one value of the base type for the property. With a multivalued property, there are multiple values of the base type.

The single-valued and multivalued property types that are supported by MAPI are described as follows. For each single-valued type that has a corresponding multivalued type, the multivalued type appears in parentheses after the single-valued type.

PT_APPTIME (PT_MV_APPTIME)
: Double value that is interpreted as date and time. This property type is the same as the OLE type VT_DATE and is compatible with the Visual Basic time representation.

PT_BINARY (PT_MV_BINARY)
: **SBinary** structure value, a counted byte array.

PT_BOOLEAN (PT_MV_12)
: 16-bit Boolean value where zero equals FALSE and non-zero equals TRUE. This property type is the same as the OLE type VT_BOOL.

PT_CLSID (PT_MV_CLSID)
: **CLSID** structure value. This property type is the same as the OLE type VT_CLSID.

PT_CURRENCY (PT_MV_CURRENCY)
: 64-bit integer intepreted as decimal. This property type is compatible with the Visual Basic CURRENCY type and is the same as the OLE type VT_CY.

PT_DOUBLE (PT_MV_DOUBLE)
: Double value; 64-bit floating point value. This property type is the same as PT_R8 and the OLE type VT_R8.

PT_ERROR
: SCODE value; 32-bit unsigned integer. This property type is the same as the OLE type VT_ERROR.

PT_FLOAT (PT_MV_FLOAT)
: 32-bit floating point value. This property type is the same as PT_R4 and the OLE type VT_R4.

PT_I2 (PT_MV_I2)
: Signed 16-bit integer. This property type is the same as PT_SHORT and the OLE type VT_I2.

PT_I4 (PT_MV_I4)
: Signed or unsigned 32-bit integer. This property type is the same as PT_LONG and the OLE type VT_I4.

PT_I8 (PT_MV_I8)
: Signed or unsigned 64-bit integer that uses the **LARGE_INTEGER** structure. This property type is the same as the OLE type VT_I8.

PT_LONG (PT_MV_LONG)
: Signed or unsigned 32-bit integer. This property type is the same as PT_I4 and the OLE type VT_I4.

PT_LONGLONG (PT_MV_LONGLONG)
: Signed or unsigned 64-bit integer. This property type is the same as PT_I8 and the OLE type VT_I8.

PT_NULL
: Indicates no property value. This property type is reserved for use with interface methods and is the same as the OLE type VT_NULL.

PT_OBJECT
: Pointer to an object that implements the **IUnknown** interface. This property type is similar to several OLE types such as VT_UNKNOWN.

PT_R4 (PT_MV_R4)
: 4-byte floating point value. This property type is the same as the OLE type VT_R4.

PT_R8 (PT_MV_R8)

8-byte floating point value. This property type is the same as the OLE type VT_DOUBLE.

PT_SHORT (PT_MV_SHORT)
: Signed 16-bit integer. This property type is the same as PT_SHORT and the OLE type VT_I2.

PT_STRING8 (PT_MV_STRING8)
: Null-terminated 8-bit character string. This property type is the same as the OLE type VT_LPSTR.

PT_SYSTIME (PT_MV_SYSTIME)
: 64-bit integer data and time value in the form of a **FILETIME** structure. This property type is the same as the OLE type VT_FILETIME.

PT_TSTRING (PT_MV_TSTRING)
: Properties with this type have the property type reset to PT_UNICODE when compiling with the UNICODE symbol and to PT_STRING8 when not compiling with the UNICODE symbol. This property type is the same as the OLE type VT_LPSTR for resulting PT_STRING8 properties and VT_LPWSTR for PT_UNICODE properties

PT_UNSPECIFIED
: Indicates that the property type is unknown. This property type is reserved for use with interface methods.

MAPI Versions of 32-Bit Windows Functions

This appendix documents functions from the Win32 application programming interface (API) useful in the 16-bit environment used by MAPI client and server developers. Most of the functions documented in this appendix have counterparts published in the Win32 SDK, and some have limitations relative to their Win32 SDK counterparts. Such limitations are noted for each function. Where possible, if the calling implementation requests unsupported functionality, the function returns a value that indicates failure.

Some of the 32-bit Windows functions documented here have been implemented specifically for MAPI. The MAPIWIN.H header file includes definitions to aid in developing single-source service providers that run on both Win32 and Win16 API platforms.

MAPI works only with filenames, and other strings passed to it, in the ANSI character set. Applications that use filenames in the OEM character set must convert them to ANSI before calling MAPI.

Accessing Win32 Information in the MAPIWIN.H Header File

MAPIWIN.H has three sections. The first section defines how to call an available function by different methods in Win16. Functions included in the first section manage per-instance global variables for dynamic-link libraries (DLLs). They work on the assumption that all of a DLL's per-instance global variables exist in a single block of memory.

The second section specifically defines for the Win16 environment functionality that is generally available in the Win32 environment. This section consists largely of Win32 file input-output functions that are not supported under Win16 but are implemented in MAPI.DLL using MS-DOS calls. Some functions of this type have limitations relative to their Win32 counterparts; the limitations are spelled out in this Appendix. The third section defines conventions that simplify certain common operations.

The following functions have no meaning on Win16, but Microsoft's MAPI implementation defines macros to make it easier to write common code:

CloseMutexHandle
CreateMutex
DeleteCriticalSection
EnterCriticalSection
InitializeCriticalSection
LeaveCriticalSection
ReleaseMutex
WaitforSingleObject

Syntax and Limitations for Win32 Functions Useful in MAPI Development

The remainder of this appendix lists Win32 functions useful to MAPI developers that have limits, adaptations, or differences when used in the MAPI development environment. Where applicable, these limitations are described. Where there are no limitations specifically defined, only the syntax of a function is included. For detailed descriptions of the Win32 functions, see the *Win32 Programmer's Reference.*

Three limitations apply to most of the Win32 functions that MAPI implements. Error codes returned from these functions or from **GetLastError** come from MS-DOS and may not always match the Win32 counterpart. Second, MAPI works only with filenames, and other strings passed to it, in the ANSI character set. Applications that use filenames in the OEM character set must convert them to ANSI before calling MAPI. And third, security attributes are ignored.

CloseHandle

This function entry is part of a MAPI-specific list of Win32 functions. For a complete description, see Win32 Functions.

```
BOOL CloseHandle(
  HANDLE hObject
 )
```

Limitations

This function will close only file handles.

CompareFileTime

This function entry is part of a MAPI-specific list of Win32 functions. For a complete description, see Win32 Functions.

LONG CompareFileTime(
CONST FILETIME * *lpft1***,**
CONST FILETIME * *lpft2*
)

CompareStringA

int CompareStringA(
    LCID *Locale***,**
    DWORD *dwCmpFlags***,**
    LPCSTR *lpString1***,**
    int *cchCount1***,**
    LPCSTR *lpString2***,**
    int *cchCount2*
)

CompareStringW

int CompareStringW(
    LCID *lcid*,
    DWORD *fdwStyle*,
    LPCWSTR *lpString1*,
    int *cch1*,
    LPCWSTR *lpString2*,
    int *cch2*
)

Limitations

Only accurate for character values less than 128. Characters whose values are greater may not be compared accurately because the 16-bit implementation does not have the necessary Unicode mapping tables.

CopyFile

This function entry is part of a MAPI-specific list of Win32 functions. For a complete description, see Win32 Functions.

BOOL CopyFile(
  LPCTSTR *lpszExistingFile***,**
  LPCTSTR *lpszNewFile***,**
  BOOL *fFailIfExists*
)

CopyMemory

This function entry is part of a MAPI-specific list of Win32 functions. For a complete description, see Win32 Functions.

VOID CopyMemory (
    PVOID *Destination***,**
    CONST VOID ** Source***,**
    DWORD *Length*
)

CreateDirectory

This function entry is part of a MAPI-specific list of Win32 functions. For a complete description, see Win32 Functions.

BOOL CreateDirectory(
  LPCTSTR *lpszPath***,**
  LPSECURITY_ATTRIBUTES *lpsa*
)

Limitations:

LPSECURITY_ATTRIBUTES is not supported; fails if non-null.

CreateFile

This function entry is part of a MAPI-specific list of Win32 functions. For a complete description, see Win32 Functions.

HANDLE CreateFile(
  LPCTSTR *lpszName***,**
  DWORD *fdwAccess***,**
  DWORD *fdwShareMode***,**
  LPSECURITY_ATTRIBUTES *lpsa***,**
  DWORD *fdwCreate***,**
  DWORD *fdwAttrsAndFlags***,**
  HANDLE *hTemplateFile*
)

Limitations

Several differences from ordinary Win32 usage occur when using the MAPI version of the **CreateFile** function:

- **dwFlagsAndAttributes** and **dwDesiredAccess** are ignored.
- **lpSecurityAttributes** is not supported and the function fails if it is not NULL.
- *hTemplateFile* is not supported and the function fails if it is nonzero.

DeleteFile

This function entry is part of a MAPI-specific list of Win32 functions. For a complete description, see Win32 Functions.

BOOL DeleteFile(
  LPCTSTR *lpszFileName*
)

DosDateTimeToFileTime

This function entry is part of a MAPI-specific list of Win32 functions. For a complete description, see Win32 Functions.

BOOL DosDateTimeToFileTime(
  WORD *wDOSDate***,**
  WORD *wDOSTime***,**
  LPFILETIME *lpft*
)

FBadReadPtr

BOOL FBadReadPtr(
  CONST VOID * *lpvPtr***,**
  UINT *cbBytes*
)

Limitations

The **FBadReadPtr** function behaves as does the **IsBadReadPtr** function but returns FALSE if the *cbBytes* parameter is zero regardless of the value of the *lpvPtr* parameter. This matches the behavior of Win32 **IsBadReadPtr**, rather than Win16 **IsBadReadPtr**.

FileTimeToLocalFileTime

This function entry is part of a MAPI-specific list of Win32 functions. For a complete description, see Win32 Functions.

BOOL FileTimeToLocalFileTime(
  CONST FILETIME * *lpft***,**
  LPFILETIME *lpftLocal*
)

Limitations

Depends on time zone information in WIN.INI. The UI is included in the mail and fax control panel applet, and is set by the **SetTimeZoneInformation** call listed in this Appendix.

FileTimeToDosDateTime

This function entry is part of a MAPI-specific list of Win32 functions. For a complete description, see Win32 Functions.

BOOL FileTimeToDosDateTime(
  CONST FILETIME * *lpft***,**
  LPWORD *lpwDOSDate***,**
  LPWORD *lpwDOSTime*
)

FileTimeToSystemTime

This function entry is part of a MAPI-specific list of Win32 functions. For a complete description, see Win32 Functions.

BOOL FileTimeToSystemTime(
CONST FILETIME * *lpft***,**
LPSYSTEMTIME *lpst*
)

FillMemory

This function entry is part of a MAPI-specific list of Win32 functions. For a complete description, see Win32 Functions.

VOID FillMemory (
  PVOID *Destination***,**
  DWORD *Length***,**
  BYTE *Fill*
)

FindClose

This function entry is part of a MAPI-specific list of Win32 functions. For a complete description, see Win32 Functions.

```
BOOL FindClose(
  HANDLE hFindFile
 )
```

FindFirstFile

This function entry is part of a MAPI-specific list of Win32 functions. For a complete description, see Win32 Functions.

```
HANDLE FindFirstFile(
   LPCTSTR lpFileName,
   LPWIN32_FIND_DATA lpFindFileData
  )
```

Limitations

The **dwReserved0**, **dwReserved1** and **cAlternateFileName** members are not supported in the **WIN32_FIND_DATA** structure.

FindNextFile

This function entry is part of a MAPI-specific list of Win32 functions. For a complete description, see Win32 Functions.

BOOL FindNextFile(
  HANDLE *hFindFile***,**
  LPWIN32_FIND_DATA *lpFindFileData*
)

Limitations

The **dwReserved0**, **dwReserved1** and **cAlternateFileName** members are not supported in the **WIN32_FIND_DATA** structure.

GetACP

This function entry is part of a MAPI-specific list of Win32 functions. For a complete description, see Win32 Functions.

UINT GetACP(VOID)

GetCurrentProcessID

This function entry is part of a MAPI-specific list of Win32 functions. For a complete description, see Win32 Functions.

DWORD GetCurrentProcessId(VOID)

Limitations

Returns HTASK. Value is subject to reuse by the operating system.

GetFileAttributes

This function entry is part of a MAPI-specific list of Win32 functions. For a complete description, see Win32 Functions.

DWORD GetFileAttributes(
   LPCTSTR *lpFileName*
)

Limitations

This function won't work on Novell NetWare without FILESCAN rights.

GetFileSize

This function entry is part of a MAPI-specific list of Win32 functions. For a complete description, see Win32 Functions.

DWORD GetFileSize(
    HANDLE *hFile***,**
    LPDWORD *lpdwFileSizeHigh*
)

GetFileTime

BOOL GetFileTime(
    HANDLE *hFile***,**
    LPFILETIME *lpftCreation***,**
    LPFILETIME *lpftLastAccess***,**
    LPFILETIME *lpftLastWrite*
)

Limitations

The time that the file in question was last modified is supported, but not the file creation or access time. The function fails if either of the latter is requested.

GetFullPathName

This function entry is part of a MAPI-specific list of Win32 functions. For a complete description, see Win32 Functions.

DWORD GetFullPathName(
  LPCTSTR *lpszFile***,**
  DWORD *cchPath***,**
  LPTSTR *lpszPath***,**
  LPTSTR * *lpszFilePart*
)

Limitations

This function does not handle ".." path components in *lpFileName*.

GetLastError

This function entry is part of a MAPI-specific list of Win32 functions. For a complete description, see Win32 Functions.

DWORD GetLastError(VOID)

Limitations

This function won't be as reliable as it is on Windows NT. If a function in this module fails because an unsupported feature was requested, no error is returned.

GetLocalTime

This function entry is part of a MAPI-specific list of Win32 functions. For a complete description, see Win32 Functions.

VOID GetLocalTime(
LPSYSTEMTIME *lpst*
)

Limitations

This function returns the current time as local time.

GetSystemTime

This function entry is part of a MAPI-specific list of Win32 functions. For a complete description, see Win32 Functions.

VOID GetSystemTime(
   LPSYSTEMTIME *lpst*
)

Limitations

This function depends on the time zone and returns the current time as Greenwich mean time (GMT). Requires that an earlier call to **SetTimeZoneInformation** has been made to get the local time zone.

GetTempFileName

This function entry is part of a MAPI-specific list of Win32 functions. For a complete description, see Win32 Functions.

UINT GetTempFileName(
    LPCTSTR *lpszPath***,**
    LPCTSTR *lpszPrefix***,**
    UINT *uUnique***,**
    LPTSTR *lpszTempFile*
)

GetTempFileName32

UINT WINAPI GetTempFileName32 (
    LPCSTR *lpPathName***,**
    LPCSTR *lpPrefixString***,**
    UINT *uUnique***,**
    LPSTR *lpTempFileName*
)

GetTempPath

This function entry is part of a MAPI-specific list of Win32 functions. For a complete description, see Win32 Functions.

DWORD GetTempPath(
 DWORD *cchBuffer***,**
 LPTSTR *lpszTempPath*
)

GetTimeZoneInformation

This function entry is part of a MAPI-specific list of Win32 functions. For a complete description, see Win32 Functions.

```
DWORD GetTimeZoneInformation(
  LPTIME_ZONE_INFORMATION lptzi
 )
```

GetUserDefaultLCID

This function entry is part of a MAPI-specific list of Win32 functions. For a complete description, see Win32 Functions.

LCID GetUserDefaultLCID(VOID)

InterlockedDecrement

This function entry is part of a MAPI-specific list of Win32 functions. For a complete description, see Win32 Functions.

LONG InterlockedDecrement(
    LPLONG *lplVal*
)

Limitations

This function relies on cooperative multitasking.

InterlockedIncrement

This function entry is part of a MAPI-specific list of Win32 functions. For a complete description, see Win32 Functions.

LONG InterlockedIncrement(
  LPLONG *lplVal*
)

Limitations

This function relies on cooperative multitasking.

IsBadBoundedStringPtr

BOOL WINAPI IsBadBoundedStringPtr(
    const void FAR* *lpsz***,**
    UINT *cchMax*
)

IsBadReadPtr

This function entry is part of a MAPI-specific list of Win32 functions. For a complete description, see Win32 Functions.

BOOL IsBadReadPtr(
  CONST VOID * *lpvPtr***,**
  UINT *cbBytes*
)

Limitations

This function has been redefined to work as on the Win32 version of **FBadReadPtr**.

IsBadStringPtrW

BOOL IsBadStringPtrW(
    LPCWSTR *lpszStr*,
    UINT *cchMax*
)

LocalFileTimeToFileTime

This function entry is part of a MAPI-specific list of Win32 functions. For a complete description, see Win32 Functions.

BOOL LocalFileTimeToFileTime(
    CONST FILETIME * *lpftLocal***,**
    LPFILETIME *lpft*
)

Limitations

Requires that an earlier call to **SetTimeZoneInformation** has been made to get the local time zone.

lstrlenW

int lstrlenW(
   LPCWSTR *lpszString*
)

lstrcmpW

int lstrcmpW(
LPCWSTR *lpszString1*,
LPCWSTR *lpszString2*
)

lstrcpyW

LPWSTR lstrcpyW(
    LPWSTR *lpszString1*,
    LPCWSTR *lpszString2*
)

MoveFile

This function entry is part of a MAPI-specific list of Win32 functions. For a complete description, see Win32 Functions.

BOOL MoveFile(
  LPCTSTR *lpszExisting***,**
  LPCTSTR *lpszNew*
)

Limitations

The MAPI version of the **MoveFile** function won't move a directory. This function always works across drives, and it always performs a copy operation then deletes the original file, as opposed to truly performing a move operation.

MoveMemory

This function entry is part of a MAPI-specific list of Win32 functions. For a complete description, see Win32 Functions.

```
VOID MoveMemory (
    PVOID Destination,
    CONST VOID * Source,
    DWORD Length
  )
```

MulDiv32

int MulDiv(
  int *nMultiplicand***,**
  int *nMultiplier***,**
  int *nDivisor*
)

Limitations

The **MulDiv32** function supports the **MULDIV** macro contained in the MAPIWIN.H header file. It takes 32-bit arguments, unlike the native Win16 **MulDiv**. The MAPI **MulDiv32** does not check for overflow on the *nMultiplier* parameter.

MultiByteToWideChar

This function entry is part of a MAPI-specific list of Win32 functions. For a complete description, see Win32 Functions.

int MultiByteToWideChar(
  UINT *CodePage***,**
  DWORD *dwFlags***,**
  LPCSTR *lpMultiByteStr***,**
  int *cchMultiByte***,**
  LPWSTR *lpWideCharStr***,**
  int *cchWideChar*
)

Limitations

This function is only accurate for character values less than 128. Characters whose values are greater may not be compared accurately because the 16-bit implementation does not have the necessary Unicode mapping tables; it does not support Unicode single-byte conversion; works reliably only for ASCII characters.

ReadFile

This function entry is part of a MAPI-specific list of Win32 functions. For a complete description, see Win32 Functions.

BOOL ReadFile(
  HANDLE *hFile*,
  LPVOID *lpBuffer*,
  DWORD *NumberOfBytesToRead*,
  LPDWORD *lpNumberOfBytesRead*,
  LPOVERLAPPED *lpOverlapped*
)

Limitations

The count is limited to 64K. The *lpOverlapped* parameter is not supported and the function fails if *lpOverlapped* is not NULL.

RemoveDirectory

This function entry is part of a MAPI-specific list of Win32 functions. For a complete description, see Win32 Functions.

```
BOOL RemoveDirectory(
  LPCTSTR lpszDir
 )
```

SetEndOfFile

This function entry is part of a MAPI-specific list of Win32 functions. For a complete description, see Win32 Functions.

BOOL SetEndOfFile(
    HANDLE *hFile*
)

SetFilePointer

This function entry is part of a MAPI-specific list of Win32 functions. For a complete description, see Win32 Functions.

DWORD SetFilePointer(
  HANDLE *hFile***,**
  LONG *lDistanceToMove***,**
  PLONG *lpDistanceToMoveHigh***,**
  DWORD *dwMoveMethod*
)

Limitations

Distance is limited to 2 gigabytes (signed 32-bits). The **SetFilePointer** function fails if the *lpDistanceToMoveHigh* parameter is present and nonzero, unless the value it holds is the sign extension of a negative distance.

SetTimeZoneInformation

This function entry is part of a MAPI-specific list of Win32 functions. For a complete description, see Win32 Functions.

```
BOOL SetTimeZoneInformation(
    CONST TIME_ZONE_INFORMATION * lptzi
  )
```

Limitations

Depends on time zone information in WIN.INI. The UI is included in the mail and fax control panel applet.

Sleep

This function entry is part of a MAPI-specific list of Win32 functions. For a complete description, see Win32 Functions.

```
VOID Sleep(
   DWORD cMilliseconds
  )
```

Limitations

This function does not handle the ALT+TAB and ALT+ESC key combinations for task switching.

SystemTimeToFileTime

This function entry is part of a MAPI-specific list of Win32 functions. For a complete description, see Win32 Functions.

BOOL SystemTimeToFileTime(
  CONST SYSTEMTIME * *lpst***,**
  LPFILETIME *lpft*
)

WideCharToMultiByte

This function entry is part of a MAPI-specific list of Win32 functions. For a complete description, see Win32 Functions.

int WideCharToMultiByte(
  UINT *CodePage***,**
  DWORD *dwFlags***,**
  LPCWSTR *lpWideCharStr***,**
  int *cchWideChar***,**
  LPSTR *lpMultiByteStr***,**
  int *cchMultiByte***,**
  LPCSTR *lpDefaultChar***,**
  LPBOOL *lpUsedDefaultChar*
)

Limitations

This function is only accurate for character values less than 128. Characters whose values are greater may not be compared accurately because the 16-bit implementation does not have the necessary Unicode mapping tables; it does not support Unicode single-byte conversion; works reliably only for ASCII characters.

WriteFile

This function entry is part of a MAPI-specific list of Win32 functions. For a complete description, see Win32 Functions.

BOOL WriteFile(
  HANDLE *hFile***,**
  PCVOID *lpBuffer***,**
  DWORD *nNumberOfBytesToWrite***,**
  PDWORD *lpNumberOfBytesWritten***,**
  POVERLAPPED *lpOverlapped*
)

Limitations

The *lpOverlapped* parameter is not supported, and the function fails if it is not NULL.

ZeroMemory

This function entry is part of a MAPI-specific list of Win32 functions. For a complete description, see Win32 Functions.

VOID ZeroMemory (
    PVOID *Destination***,**
    DWORD *Length*
)

Address Types

E-mail addresses in MAPI are defined by two properties. The address type, PR_ADDRTYPE, is a string property that names the messaging system the address is destined for. It is used by MAPI to assign responsibility for an e-mail address to the right transport provider. The e-mail address itself, PR_EMAIL_ADDRESS, is another string whose format varies depending upon PR_ADDRTYPE. It is not interpreted by MAPI, but by the transport provider and by the messaging system itself.

This appendix lists address types known to MAPI at this time and the corresponding e-mail address formats. Several of these address types, including SMTP, X400, and FAX are used by a large number of messaging systems. It is important that all MAPI components agree on the type name and address format for such common types, so they can interoperate smoothly. MAPI address book providers, MAPI transport providers, and gateways to messaging systems with MAPI-based clients should all take note of this specification.

To define a new address type, to make comments or suggestions about the information in this appendix, or to request more information, send e-mail to mapi@microsoft.com. The following table, and many of the address type specifications that follow, are work in progress.

Messaging system	**Address type (PR_ADDRTYPE)**	**Owner or reference document**
3Com® 3+Mail®	3COM	3Com (3+Open®)
AT&T Easylink Services	ATT	AT&T
cc:Mail™ (proposed)	CCMAIL	IBM
CompuServe (proposed)	COMPUSERVE	Compuserve
Microsoft Exchange Server	EX	Microsoft
Facsimile	FAX MSFAX	
MCI MAIL	MCI	MCI Communications Corp.
Novell® Message Handling System	MHS	Novell
Microsoft Mail Server for PC Networks	MS	Microsoft
Microsoft Mail Server for Appletalk Networks	MSA	Starnine Technologies®
The Microsoft Network	MSN	Microsoft
Professional Office System	PROFS	
Internet	SMTP	Internet Request for Comments (RFC) 822
SNA Distribution	SNADS	

System		
Telex (proposed)	TELEX	
X.400 Message Handling System	X400	CCITT X.400 series specifications
X.500 Directory Services	X500	CCITT X.500 series specifications

3+Mail (3COM)

This information is based on the Microsoft Mail Server for PC Networks gateway to 3+Mail. The custom recipient address is in the following format:

[3COM:*UserName***:***Domain***:***Organization***]**

- Or -

[3COM:*UserName***]**

- Or -

[3COM:*UserAlias***:***Domain***:***Organization***]**

AT&T EasyLink (ATT)

This information is based on the Microsoft Mail for PC Networks gateway to AT&T Easylink. The custom recipient addresses are in the following formats:

AT&T Mail Address

[ATT:attmail!*UserName***]**

attmail
The "gate name" for the AT&T Mail Network.

!
A delimiter.

UserName
A unique identifier for an individual user.

AT&T MailFAX

[ATT:fax!+*FaxTelephoneNumber* **(/***RecipientName***)** *Option***]**

fax!
Indicates to the AT&T Mail Network that this message is being delivered to a fax machine.

+
Must be specified for AT&T Mail Network to insert the correct country access code.

FaxTelephoneNumber
The local fax telephone number, country code, area code and local exchange.

RecipientName
The recipient of the fax.

Options
Any AT&T Mail options.

An example of this address is:

```
[ATT:fax!+19085551234(/John Doe) delivery].
```

AT&T Telex

[ATT:telex!*TelexNumber* **(/***Attention***)]**

telex!
Indicates to the AT&T Mail Network that this is a message for a telex machine.

TelexNumber
The number of the telex machine to which the message is being delivered.

Attention
The name of the recipient of the message.

An example of this address is:

```
[ATT:telex!900123(Jones MFG CO/John Doe)].
```

AT&T MailPRINT

[ATT:printer!*PrinterTelephoneNumber* **(/***RecipientName***)** *Options***]**

printer!
Indicates to the AT&T Mail Network that this message is being delivered to a printer.

PrinterTelephoneNumber

The local printer telephone number, country code, area code, and local exchange.

RecipientName
: The recipient of the message.

Options
: Any AT&T Mail option.

An example of this address type is:

```
[ATT:printer!19085551234 (/John Doe)].
```

AT&T Paper

[ATT:*Address***]**

Address
: The postal address is entered on one line with semicolons (;) used to separate the address parts. If the address contains a semicolon, surround it with quotation marks (";").

An example of this address type is:

```
[ATT:John Doe;123 Main Street; N.Y. 01234].
```

Facsimile (FAX, MSFAX)

Support is required for custom recipient addresses in the following format:

[FAX:*Dialable* **|** *Name @ Canonical* **|** *SpecialDialing*]

Dialable
: A dialable fax phone number that consists of any of the following characters: 0-9, ( ), -, space. Examples are 64545 and 9, 1-206-8828080.

Name
: Optional field used for inbound routing by Microsoft At Work Fax. MAPI providers that do not support Microsoft At Work Fax should simply ignore this field. The name field is interpreted by the receiving machine and can contain any information which helps route the fax to its intended recipient. For example, this field might be a recipient's e-mail name. This field can include the at sign character (@).

Canonical
: Must be a canonical fax phone number. If a canonical fax phone number is included in the address, and the provider can process canonical numbers, the provider should always use the canonical number instead of the dialable phone number.

|
: Separator character, ASCII Hex 0x7C. The separator between the subscriber number and special dialing instructions (such as a postfix or TIA/EIA/IS-141 subaddress) is optional. If the phone number is in canonical form and a separator is not present, then the special dialing instructions are assumed to begin with the first character in the local number which is not one of the following: the numerals 0-9, hyphen (-), space.

SpecialDialing
: Must be a postfix, TIA/EIA/IS-141 subaddress or ISDN subaddress. MAPI 1.0 providers can optionally support these special dialing extensions.

Examples of this address type are:

```
[FAX:936-7329]
[FAX:1-206-9367329]
[FAX:936-7329 | +1 (206) 936-7329]
[FAX:936-7329 | tedst@ +1 (206) 936-7329]
[FAX:936-7329 | tedst@+1 (206) 936-7329]
[FAX:936-7329 | +1 (206) 9367329|#546]
[FAX:936-7329 | +1 (206) 9364344|,,,,,567]
```

Note Some service providers create 'MSFAX' formatted addresses but with the 'FAX' address type. Providers can either handle these addresses, or not accept responsibility for them as they see fit; either method is acceptable.

Support is optional (but recommended) for custom recipient addresses in the following format:

[MSFAX:*Name@ PhoneNumber* **|** *SpecialDialing***]**

Name
: Optional field used for inbound routing by Microsoft At Work Fax. MAPI providers that do not support Microsoft At Work Fax should simply ignore this field. The name field is interpreted by the receiving machine and can contain any information which helps route the fax to its intended recipient. For example, this field might be a recipient's e-mail name. This field can include the at sign character (@).

PhoneNumber
: Must be a canonical phone number or dialable phone number.

|
: Separator character, ASCII Hex 7c. This is the separator between the subscriber number and special dialing instructions (such as a postfix or TIA/EIA/IS-141 subaddress). This separator is optional. If the phone number is in canonical form and a separator is not present, then the special dialing instructions are assumed to begin with the first character in the local number which is not one of the following: the numerals 0-9, hyphen (-), period (.), and space.

DialablePhoneNumber
: A dialable address is a phone number that consists of any of the following characters: the numerals 0-9, letters A-D, asterisk (*), pound (#), comma (,), exclamation mark (!), uppercase and lowercase w (W, w), uppercase and lowercase p (P, p), uppercase and lowercase t (T, t), at sign (@), dollar sign ($), question mark (?), hyphen (-), period (.), and space. For example 64545. Dialable addresses are intended to be used to represent internal phone extensions.

SpecialDialing
: Must be a postfix, TIA/EIA/IS-141 subaddress or ISDN subaddress. MAPI 1.0 providers can optionally support these special dialing extensions.

Examples of this address type are:

```
[MSFAX: 936-7329]
[MSFAX: +1 (206) 936-7329]
[MSFAX: tedst@+1 (206) 936-7329]
[MSFAX: +1 (206) 9367329|#546]
[MSFAX: +1 (206) 9364344|,,,,,567]
```

The following table includes descriptions of the components of this address:

Component	**Description**
canonical phone number	A phone number in the form: **+***CountryCode* space (*AreaCode*) space *LocalNumber*. Examples are +1 (206) 936-4479 and +43 443444. *CountryCode* and *AreaCode* can only contain the numerals 0 through 9. If the area code is present it must be preceded by exactly one ASCII left parenthesis character (0x28), and be followed by exactly one ASCII right parenthesis character (0x29) and one ASCII space character (0x20). The local number must contain one or more of the numerals 0 through 9 but must not include any of the following characters: AaBbCcDdPpTtWw*#!,@$?;()|^ **CRLF**
special dialing	Must be a postfix, TIA/EIA/IS-141 subaddress or ISDN subaddress. MAPI 1.0 providers can optionally support these special dialing extensions.
postfix	A string of dialing control

	characters and digits used to complete calls which require DTMF routing, manual dialing, or pauses in the dialing sequence. The string must begin with one of the following characters: $, ? W w @ and can contain any of the following characters: 0-9 A-D * # , ! W w P p T t @ $?. An example is +1 (206) 9367329|,,,,,567. This waits for several seconds after the main number is dialed before entering the DTMF tones 567.
TIA/EIA/IS-141 subaddress	Used for inbound routing according to the U.S. standard TIA/EIA/IS-141. MAPI 1.0 providers that cannot transmit this subaddress information should remove this field before dialing. This field must begin with the number sign (#). In typical use, this value uniquely identifies a recipient. An example is +1 (206) 9367329|#543. This example enables inbound routing to employee 543
ISDN Subaddress	For faxing over ISDN. MAPI 1.0 providers that cannot transmit ISDN subaddresses should remove this field before dialing. An ISDN subaddress can begin with any character except the following: # $, ? W w @
$	Wait for "billing signal", such as a credit card prompt tone.
,	Indicates that dialing is to be paused. The duration of a pause is device specific.
?	Indicates that the user is to be prompted before continuing with dialing.
W w	Dialing should proceed only after a dialing tone has been detected.
@	Indicates that dialing is to "wait for quiet answer" before dialing the remainder of the number. This means to wait for at least one ringback tone followed by several seconds of silence.

P p	Indicates that pulse dialing is to be used for the digits following it.
T t	Indicates that tone (DTMF) dialing is to be used for the digits following it.

Note Microsoft supports two FAX address formats. FAX is intended to be used for compatibility with downlevel systems – including the Microsoft Mail for PC Networks FAX gateway. Although it is optional, all transport writers are encouraged to support and use the MSFAX standard. The 'FAX' standard will likely be eliminated in favor of the 'MSFAX' standard in a future release.

For more information on dialable and canonical addresses, see the Win32 Telephony (TAPI) documentation.

MCI MAIL (MCI)

This information is based on the Microsoft Mail for PC Networks gateway to MCI MAIL. It may be incomplete or out of date in certain respects. The custom recipient address is in the following format:

[MCI:*UserName***\r EMS:***EMSName***\r MBX:***MCIMailbox***]**

MCI
MCI MAIL address indicator.

UserName
Name of mail recipient. Not used for mail delivery.

EMS
Indicates the name or number in the next field is the recipient's REMS (remote e-mail system) account.

EMSName
The actual EMS name or number. This can also be an MCI service provided by MCI; consult MCI Help for more information.

MBX
Indicates this is additional information required by the remote system.

MCIMailbox
Actual MBX (address) for the recipient on the remote system. If the remote system is Microsoft Mail, *MCIMailbox* is the full Microsoft Mail address. You can only have one complete MBX address.

There is always a space in front of the **EMS:** token and another space in front of each **MBX:** token. These spaces are required. The carriage-return symbol at the end of the line is in addition to whatever code your text editor inserts.

The actual message text requires a header separator after the TEXT: token. That is, the first line after TEXT: should be a line of 78 dash (-) characters to separate the header from the message text. Include these 78 characters in the TEXT count.

An example of this address type is:

```
[MCI:John Doe\r EMS:MCI Mail\r MBX:123-4567\r]
```

The following are other MCI addresses:

MCI MAIL

Example:

[MCI:John Doe at MCI\r EMS:MCI Mail\r MBX:123-4567]

MCI Paper

Example:

[MCI:John Doe\r EMS:MCI Mail\r MBX:Company:ABC, Inc.\r
MBX:Country:Canada\r MBX:Line1:100 - 123 Main Street\r
MBX:Line2:\r MBX:City:Vancouver\r MBX:State:B.C.\r
MBX:Code:V6B 1A1]

EMS

This address type is for Microsoft Mail recipients.

Example:

[MCI:John Doe\r EMS:ems_name\r
MBX:network/postoffice/JohnDoe\r

MBX:\r MBX:\r MBX:\r MBX:\r MBX:]

REMS

This address type is for non-Microsoft Mail recipients.

Example:

[MCI:John Doe\r EMS:Internet\r MBX:JohnDoe@aaa.bbb.bc.ca\r
MBX:\r MBX:\r MBX:\r MBX:\r MBX:]

TELEX

Example:

[MCI:John Doe\r EMS:MCI Mail\r MBX:Country:-\r
MBX:Telex:6501234567\r MBX:Answerback:6507654321]

Fax

Example:

[MCI:John Doe\r EMS:MCI Mail\r MBX:FAXNo:604-123-4567\r
MBX:Retry:4.0\r MBX:Company:ABC Company Inc.\r
MBX:SFax:604-111-1111\r MBX:SPhone:604-222-2222]

Novell Message Handling System (MHS)

This information is based on the Microsoft Mail for PC Networks gateway to MHS and the SMF 71 specification. It may be incomplete or out of date in certain respects. The custom recipient address is in the following format:

[MHS:*User@Host***]** for a MHS user on MHS host.

Host
The name of the MHS host.

- Or -

[MHS:*Mailbox@Gateway***]** for a MSMail user on a MHS gateway

Mailbox
8 ASCII characters

Gateway
The name of the MHS gateway, not MHS host.

- Or -

[MHS: User@Gateway {*Network*/*Postoffice*/*Mailbox*}**]** for a MSMail user on MHS gateway/downstream postoffice.

User
A place holder.

Network/Postoffice/Mailbox
MSMail for PC Networks address.

- Or -

[MHS: User@Gateway *LocalAddress***]** for non-MSMail user on MHS gateway (general case).

LocalAddress
The user's native address. For example, MSMail, X.400.

Microsoft Mail for PC Networks (MS)

The custom recipient address is in the following format:

[MS:*Network*/*PostOffice*/*UserID***]**

Network
A 10 char (max) name of a MICROSOFT Mail Network.

PostOffice
A 10 character (max) Postoffice name.

UserID
A 10 chararacter (max) name of a PC Mail user.

/
A delimiter.

This address type is also used at times by Microsoft Exchange Server.

Microsoft Mail for Appletalk Networks (MSA)

The custom recipient address is in the following format:

[MSA:*UserID@MS Mail Server***]**

UserID
Name of Microsoft Mail for AppleTalk user.

MS Mail Server
Name of Microsoft Mail for AppleTalk server.

The Microsoft Network (MSN)

The custom recipient address is in the following format:

[MSN:*UserID***]**

UserID
: The name of the MSN member. For example, [MSN:patsmith].

MSN allows the sender to change the recipient's display name by using the following addressing style: Pat Smith[MSN:PatSm]. This will resolve on the client to a friendly name of 'Pat Smith' and an address alias of PatSm. The recipient will see the friendly name when they read the message.

Professional Office System (PROFS)

This information is based on the Microsoft Mail for PC Networks gateway to PROFS. The GATEWAY field is part of the gateway architecture, not the PROFS address. The custom recipient address is in the following format:

[PROFS:*Gateway*/*Node*/*UserID***]**

Gateway
The gateway name. Limited to 10 characters.

Node
VM node name. Limited to 8 characters.

UserID
VM user ID. Limited to 8 characters.

Internet (SMTP)

The format of Internet e-mail addresses is defined in RFC 822. MAPI components should handle any address that complies with that standard. However, there is a particular form of RFC 822 address that best encodes MAPI addresses:

display name **<***e-mail address***>**

The angle brackets are included as literals. Blanks are common in display names; they need not be quoted. A typical address might look like this one, which belongs to one of the coauthors of RFC 1521:

Nathaniel Borenstein <nsb@bellcore.com>

If the display name contains characters that have special meaning in SMTP addresses, such as < or @, the entire display name should be quoted using double quotes. On outbound mail, if the total length of the e-mail address plus display name exceeds 255 characters, the display name should be dropped.

SNA Distribution System (SNADS)

This information is based on the Microsoft Mail for PC Networks gateway to SNADS. The GATEWAY field is part of the gateway architecture, not the SNADS address. The custom recipient address is in the following format:

[SNADS:*Gateway*/*DistributionGroupName*/*DistributionElementName***]**

Gateway
Gateway name. Limited to 10 characters.

DistributionGroupName
Distribution Group Name. Limited to 8 characters.

DistributionElementName
Distribution Element Name. Limited to 8 characters.

Telex (TELEX) - Proposed

This address type definition is only a proposal. Comments and suggestions are welcome. The custom recipient address is in the following format:

[TELEX:*Recipient@TelexNumber***]**

At least a telex number is required. For example [TELEX:foobar @ 1-2066354657].

X.400 Message Handling System (X400)

This information is based on the Microsoft Mail for PC Networks gateway to X.400 mail systems. Certain fields, particularly "DDA," and the length limitations may not apply in all X.400-based messaging systems. The custom recipient address is in the following format:

[X400:g=*GivenName***;s=***Surname***;o=***Organization***;ou=***OrganizationalUnit***;**
p=*PRMD***;a=***ADMD***;c=***Country***;]**

Label	Description	Maximum Length
G=	Given name	16
I=	Initials	5
S=	Surname (required if G, I or Q are used)	40
Q=	Generation Qualifier	3
CN=	Common name	64
X.121=	X.121 address	15
N-ID=	User agent numeric identifier	32
T-TY=	Terminal type	3
T-ID=	Terminal identifier	24
O=	Organization	64
OU1=	Organizational unit #1	32
OU2=	Organizational unit #2	32
OU3=	Organizational unit #3	32
OU4=	Organizational unit #4	32
P=	Private Management Domain (PRMD)	16
A=	Administrative Management Domain (ADMD - required)	16
C=	Country (required)	2 or 3
DDA=	Domain Defined Attribute (format => dda:<type>=<value>;)	8, 128

X.500 Directory Service (X500)

This address type is based on CCITT Recommendations X.500 and the associated APIA - X/OPEN API specifications and the proposed annex to F.401, Annex F. The custom recipient address is in the following format:

[X500:/C=*CountryCode***/O=***Organization***/OU=***OrganizationUnit***/CN=***CommonName***]**

The following field length limits apply:

Field	**Length limit**
CountryCode	2 characters
Organization	Up to 64 characters
OrganizationUnit	Up to 32 characters
CommonName	

The following guidelines apply:

- Labels can be uppercase or lowercase.
- Delimiters are slash marks (/) and must be specified before the first value as well as between values.
- Delimiters can be followed by a space.

An example of this address is:

```
[X500:/ c=US/ o=Microsoft/ ou=WGA/ cn=TedSt]
```

Transport-Neutral Encapsulation Format (TNEF)

TNEF is a serialization of MAPI properties. Here is a summary of the format: The file begins with a 32-bit signature followed by a 16-bit unsigned integer that is used as a key to cross-reference attachments to their location within the tagged message text. The remainder of the file is a sequence of TNEF attributes. Each attribute consists of a class byte, an attribute identifier, the attribute size, the attribute data, and a 16-bit unsigned checksum of the attachment data. Message attributes appear first in the TNEF stream, and attachment attributes follow. Attributes belonging to a particular attachment are grouped together, beginning with the attAttachRenddata attribute.

TNEF Encoding Example

In the following TNEF encoding, all integers are specified in hexadecimal format. Nonterminal elements are in italics; constants and anything that always appears exactly as shown are in bold. In addition, sequential elements run across, and alternative elements run down.

Stream:
TNEF_SIGNATURE *Key Object*

Key:
a nonzero 16-bit unsigned integer

Object:
Message_Seq
Message_Seq Attach_Seq
Attach_Seq

Message_Seq:
attTnefVersion
attTnefVersion Msg_Attribute_Seq
attTnefVersion attMessageClass
attTnefVersion attMessageClass Msg_Attribute_Seq
attMessageClass
attMessageClass Msg_Attribute_Seq
Msg_Attribute_Seq

attTnefVersion:
LVL_MESSAGE attTnefVersion sizeof(ULONG) 0x00010000 *checksum*

attMessageClass:
LVL_MESSAGE attMessagClass *msg_class_length msg_class checksum*

Msg_Attribute_Seq:
Msg_Attribute
Msg_Attribute Msg_Attribute_Seq

Msg_Attribute:
LVL_MESSAGE *attribute-ID attribute-length attribute-data checksum*

Attach_Seq:
attRenddata
attRenddata Att_Attribute_Seq

attRenddata:
LVL_ATTACHMENT attRenddata sizeof(RENDDATA) *renddata checksum*

Att_Attribute_Seq:
Att_Attribute
Att_Attribute Att_Attribute_Seq

Att_Attribute:
LVL_ATTACHMENT *attribute-ID attribute-length attribute-data checksum*

The key is a nonzero, 16-bit unsigned integer that signifies the initial value of the attachment reference keys. The attachment reference keys are assigned sequentially beginning with the initial value. For example, if the key was 0x01AF, the first attachment reference key would be 0x01AF, the second would be 0x01B0, and so on. The TNEF implementation uses the attachment reference keys to link specific attachments with their position within the tagged message text. The initial key value is passed to the **OpenTnefStream** function when the stream is encoded. The key value should be random so that two different messages do not use the same key.

The TNEF implementation uses the attribute identifier to map attributes to their corresponding MAPI properties. The attribute identifier is a 32-bit unsigned integer made up of two word values. The high-order byte is an indication of the data type, such as string or binary, and the low-order byte is a relatively unique identifier.

All attribute lengths are unsigned long integers. If the data stored is a string or text attribute, the terminating null character is included in the length.

The checksum is a 16-bit unsigned value that is simply the summation of the individual bytes in the attribute data. The header information is not included in the checksum.

All numbers in the TNEF format are stored in little endian (that is, in Intel format).

Mapping of TNEF Message Attributes to MAPI Properties

The following table lists all the message attributes possible in a TNEF stream and their mappings to MAPI properties. In some cases, multiple MAPI properties are encoded as a single attribute. In these cases, multiple MAPI properties appear listed for a single TNEF attribute. For further explanation of specific mappings, see "Comments About the Attributes" later in this chapter.

TNEF attribute	MAPI property or properties
attAidOwner	PR_OWNER_APPT_ID
attBody	PR_BODY
attConversationID	PR_CONVERSATION_KEY
attDateEnd	PR_END_DATE
attDateModified	PR_LAST_MODIFICATION_TIME
attDateRecd	PR_MESSAGE_DELIVERY_TIME
attDateSent	PR_CLIENT_SUBMIT_TIME
attDateStart	PR_START_DATE
attFrom	PR_SENDER_ENTRYID and PR_SENDER_NAME
attMAPIProps	For information about this mapping, see "Comments About the Attributes" later in this chapter
attMessageClass	PR_MESSAGE_CLASS
attMessageID	PR_SEARCH_KEY
attMessageStatus	PR_MESSAGE_FLAGS
attOriginalMessageClass	PR_ORIG_MESSAGE_CLASS
attOwner	PR_RCVD_REPRESENTING_ENTRYID and PR_RCVD_REPRESENTING_NAME or PR_SENT_REPRESENTING_ENTRYID and PR_SENT_REPRESENTING_NAME
attParentID	PR_PARENT_KEY
attPriority	PR_PRIORITY
attRecipTable	PR_MESSAGE_RECIPIENTS
attRequestRes	PR_RESPONSE_REQUESTED
attSentFor	PR_SENT_REPRESENTING_ENTRYID
attSubject	PR_SUBJECT
attTnefVersion	For information about this mapping, see "Comments About the Attributes" later in this chapter

Mapping of TNEF Attachment Attributes to MAPI Properties

Attachment attributes are mapped in the same way message attributes are. The following table lists all the attachment attributes possible in a TNEF stream and their mappings to MAPI properties. For further explanation of specific mappings, see "Comments About the Attributes" later in this chapter.

TNEF attribute	MAPI property or properties
attAttachCreateDate	PR_CREATION_TIME
attAttachData	PR_ATTACH_DATA_BIN or PR_ATTACH_DATA_OBJ
attAttachment	For information about this mapping, see "Comments About the Attributes" later in this chapter
attAttachMetaFile	PR_ATTACH_RENDERING
attAttachModifyDate	PR_LAST_MODIFICATION_TIME
attAttachRenddata	PR_ATTACH_METHOD, PR_RENDERING_POSITION
attAttachTitle	PR_ATTACH_FILENAME
attAttachTransportFilename	PR_ATTACH_TRANSPORT_NAME

Comments About the Attributes

This section provides additional information about the TNEF attribute to MAPI property mapping for certain attributes. For more information about the MAPI properties that the attributes are mapped to, see the reference entries for the individual properties.

attMAPIProps

The attMAPIProps attribute is special in that it can be used to encapsulate any MAPI property that does not have a counterpart in the set of existing TNEF-defined attributes. The attribute data is a counted set of MAPI properties laid end-to-end. The format of attMAPIProps, which allows for any configuration of MAPI properties, is as follows:

Property_Seq:
  property-count Property_Values, ...

Property_Values:
  proptag Property
  proptag Proptag_Name Property

Property:
  Value
  value-count Value, ...

Value:
  value-data
  value-size value-data padding
  value-size value-IID value-data padding

Proptag_Name:
  name-guid name-kind name-id
  name-guid name-kind name-string-length name-string padding

The encapsulation of each property varies in the following ways based on the property identifier and the property type:

If the property falls in the named property range, then the property tag is immediately followed by the MAPI property name, consisting of a globally unique identifier (GUID), a kind, and either an identifier or a Unicode string.

If the property is either multivalued or is of variable length, such as the PT_BINARY, PT_STRING8, PT_UNICODE, or PT_OBJECT properties, then the number of values, which are encoded as a 32-bit unsigned long, falls next in the encapsulation followed by the individual values. Each variable-length value is preceded by its size in bytes encoded as a 32-bit unsigned long. Additionally, each individual value is padded out to 4-byte boundaries; the padding is not included in the value size.

If the property is of type PT_OBJECT, the value size is followed by the interface identifier (IID) of the object. The current implementation of TNEF only supports IID_IMessage, IID_IStorage, and IID_IStream. The size of the IID is included in the value size.

If the object is an embedded message,that is, if the object has a property type of PT_OBJECT and an IID of IID_IMessage, the value data is encoded as a TNEF stream. The actual encoding of an embedded message in the MAPI implementation of TNEF is done by opening a second TNEF object for the original stream and processing the stream inline.

Attributes with the attDate Prefix

All date properties are stored as **DTR** structures. A **DTR** structure is very similar to the **SYSTEMTIME** structure defined in the 32-bit Windows header files. The **DTR** is encoded in TNEF as a sizeof(**DTR**) bytes starting at &dtr.wYear. The dates and times for attachment attributes are encoded as **DTR** structures. Any MAPI property that does not map to a down-level attribute is encoded as a MAPI encapsulation in attAttachment.

attOriginalMessageClass

A message class is stored as a string. The encoded string usually holds the MAPI-specified name of the message class. The exception is that, to keep compatibility with Microsoft Mail for Windows for Workgroups 3.1, the following MAPI message classes are mapped to down-level message classes:

MAPI message class	**Windows for Workgroups Mail 3.*x***
IPM	IPM.Microsoft Mail.Note
IPM.Note	IPM.Microsoft Mail.Note
IPM.Schedule.Meeting.Canceled	IPM.Microsoft Schedule.MtgCncl
IPM.Schedule.Meeting.Request	IPM.Microsoft Schedule.MtgReq
IPM.Schedule.Meeting.Resp.Neg	IPM.Microsoft Schedule.MtgRespN
IPM.Schedule.Meeting.Resp.Pos	IPM.Microsoft Schedule.MtgRespP
IPM.Schedule.Meeting.Resp.Tent	IPM.Microsoft Schedule.MtgRespA
Report.IPM.Note.NDR	IPM.Microsoft Mail.Non-Delivery
Report.IPM.Note.RN	IPM.Microsoft Mail.Read Receipt

attConversationID and attParentID

The Windows for Workgroups 3.1 Mail conversation key is a textual string. The MAPI equivalent is a binary value. TNEF converts the binary data to text and adds a terminating null character.

attFrom

The attFrom attribute is encoded as two **TRP** structures laid end-to-end. The format for attFrom is as follows:

attFrom:
    trpidOneOff
    (sizeof(TRP) *2) + length display-name + terminator + pad to 2 byte boundary
    length address-type**:**email-address + terminator
    display-name terminated and padded
    address-type**:**email-address terminated
    zero-fill sizeof(TRP)

attOwner

The attOwner attribute is encoded as counted strings laid end-to-end. The format for attOwner is as follows:

attOwner:
16bit-length-display-name (terminator included)
display-name
16bit-length-address-type**:**email-address (terminator included)
address-type**:**email-address

The mapping of the attOwner attribute is dependent on the message class of the message being encoded. If the message is either a Microsoft Schedule+ meeting request or cancellation, the attribute maps to one of the PR_SENT_REPRESENTING_*X* properties. If the message is a Microsoft Schedule+ meeting response of any type, the attribute maps to one of the PR_RCVD_REPRESENTING_*X* properties.

attSentFor

The attSentFor attribute is encoded as counted strings laid end-to-end. The format for attSentFor is as follows:

attSentFor:
  16bit-length-display-name (terminator included)
  display-name
  16bit-length-address-type**:**email-address (terminator included)
  address-type**:**email-address

attRecipTable

When a recipient table is being encoded, each recipients is encoded as a row of MAPI properties. The format is as follows:

Row_Seq:
  row-count Property_Seq, ...

attPriority

MAPI message priorities are also mapped to TNEF for down-level compatibility. MAPI identifies - 1, 0, and 1 as low, normal, and high priority respectively. The down-level priorities are 3, 2, and 1.

attMessageStatus

MAPI message flags must also be mapped to down-level values. All the flags are grouped together and encoded in a single byte. The mappings are as follows:

MAPI message flags	**Down-level message flags**
MSGFLAG_READ	fmsRead
MSGFLAG_UNMODIFED	not fmsModified
MSGFLAG_SUBMIT	fmsSubmitted
MSGFLAG_HASATTACH	fmsHasAttach
MSGFLAG_UNSENT	fmsLocal

attAttachRenddata

The **RENDDATA** structure describes how and where the attachment is rendered in the message text.

The rendering data is encoded as sizeof(**RENDDATA**) bytes beginning at &rd.atyp. If the value of the **RENDDATA** structure's **dwFlags** member is set to **MAC_BINARY**, then the attachment data is stored in MacBinary format; otherwise, the attachment data is encoded as usual.

OLE Attachments

OLE attachments, when encoded for compatibility, are encoded as OLE 1.0 stream objects. This coding standard means that if the original object is really an OLE 2.0 IStorage object, then the object must be converted to an OLE 1.0 stream. This conversion is performed using **OleConvertIStorageToOLESTREAM** function, which is supplied by the OLE DLLs; examples of this conversion can be found in *OLE Programmer's Reference, Volume One.*

Mapping of X.400 P2 Attributes to MAPI Properties

The X/Open CAE Specification API to Electronic Mail (X.400), published by the X/Open Company Limited and X.400 API Association (1991), describes a recommended implementation of the X.400 (1984) and X.400 (1988) Blue Book specifications.

This appendix describes mappings between the recommended implementation's P2 attributes and MAPI properties.

The reference information is presented in two different ways in this appendix:

- X.400 attributes are organized by object.
- All X.400 attributes are combined into one comprehensive list and presented in alphabetical order.

X.400 Attributes By Object

The following entries contain tables for each object listing the mappings from X.400 P2 attributes to their corresponding MAPI properties.

OMP_O_IM_C_BD_PRT

The class OMP_O_IM_C_BD_PRT does not have any attributes that are unique to the class.

OMP_O_IM_C_BILAT_DEF_BD_PRT

The attributes of the class OMP_O_IM_C_BILAT_DEF_BD_PRT map to MAPI properties as follows:

MH ID/Type	MAPI Property
IM_BILATERAL_DATA	PR_ATTACH_DATA_BIN

OMP_O_IM_C_EXTERN_DEF_BD_PRT

The attributes of the class OMP_O_IM_C_EXTERN_DEF_BD_PRT map to MAPI properties as follows:

MH ID/Type	MAPI Property
IM_EXTERNAL_DATA	PR_ATTACH_DATA_BIN, PR_ATTACH_FILENAME, PR_ATTACH_TAG
IM_ EXTERNAL_PARAMETERS	PR_ATTACHMENT_X400_PARAMETERS

OMP_O_IM_C_G3_FAX_BD_PRT

The attributes of the class OMP_O_IM_C_G3_FAX_BD_PRT are not mapped to MAPI properties.

MH ID/Type	**MAPI Property**
IM_G3_FAX_NBPS	Not mapped to a MAPI property.
IM_IMAGES	Not mapped to a MAPI property.

OMP_O_IM_C_G4_CLASS_1_BD_PRT

The attributes of the class OMP_O_IM_C_G4_CLASS_1_BD_PRT are not mapped to MAPI properties.

MH ID/Type	MAPI Property
IM_G4_CLASS_1_DOCUMENT	Not mapped to a MAPI property.

OMP_O_IM_C_GENERAL_TEXT_BD_PRT

The attributes of the class OMP_O_IM_C_GENERAL_TEXT_BD_PRT are not mapped to MAPI properties.

MH ID/Type	MAPI Property
IM_CHAR_SET_REG	Not mapped to a MAPI property.
IM_TEXT	Not mapped to a MAPI property.

OMP_O_IM_C_IA5_TEXT_BD_PRT

The attributes of the class OMP_O_IM_C_IA5_TEXT_BD_PRT are mapped to the following MAPI properties:

MH ID/Type	MAPI Property
IM_REPERTOIRE	Not mapped to a MAPI property.
IM_TEXT	PR_BODY

OMP_O_IM_C_INTERPERSONAL_MSG

The attributes of the class OMP_O_IM_C_INTERPERSONAL_MSG map to MAPI properties as follows:

MH ID/Type	MAPI Property
IM_AUTHORIZING_USERS	PR_AUTHORIZING_USERS
IM_AUTO_FORWARDED	PR_AUTO_FORWARDED
IM_BLIND_COPY_RECIPIENTS	PR_DISPLAY_BCC
IM_BODY	PR_BODY
IM_COPY_RECIPIENTS	PR_DISPLAY_CC
IM_EXPIRY_TIME	PR_EXPIRY_TIME
IM_IMPORTANCE	PR_IMPORTANCE
IM_INCOMPLETE_COPY	PR_INCOMPLETE_COPY
IM_LANGUAGES	PR_LANGUAGES
IM_OBSOLETED_IPMS	PR_OBSOLETED_IPMS
IM_ORIGINATOR	PR_CALLBACK_TELEPHONE_NUMBER, PR_SENDER_ADDRTYPE, PR_SENDER_EMAIL_ADDRESS, PR_SENDER_ENTRYID, PR_SENDER_NAME, PR_SENDER_SEARCH_KEY (and all PR_SENT_REPRESENTING_* properties)
IM_PRIMARY_RECIPIENTS	PR_DISPLAY_TO
IM_RELATED_IPMS	PR_RELATED_IPMS
IM_REPLIED_TO_IPM	PR_PARENT_KEY
IM_REPLY_RECIPIENTS	PR_REPLY_RECIPIENT_ENTRIES, PR_REPLY_RECIPIENT_NAMES
IM_REPLY_TIME	PR_REPLY_TIME
IM_SENSITIVITY	PR_SENSITIVITY
IM_SUBJECT	PR_NORMALIZED_SUBJECT, PR_SUBJECT, PR_SUBJECT_PREFIX
IM_THIS_IPM	PR_SEARCH_KEY

The following constant values are mapped from IM_IMPORTANCE to PR_IMPORTANCE:

Importance	MAPI value
IM_HIGH	2
IM_LOW	0
IM_ROUTINE	1

The following constant values are mapped from IM_SENSITIVITY to PR_SENSITIVITY:

IM_SENSITIVITY value	PR_SENSITIVITY value
IM_COMPANY_CONFIDENTIAL	SENSITIVITY_COMPANY_CONFIDENTIAL
IM_NOT_SENSITIVE	SENSITIVITY_NONE
IM_PERSONAL	SENSITIVITY_PERSONAL
IM_PRIVATE	SENSITIVITY_PRIVATE

OMP_O_IM_C_INTERPERSONAL_NOTIF

The attributes of the class OMP_O_IM_C_INTERPERSONAL_NOTIF map to MAPI properties as follows:

MH ID/Type	MAPI Property
IM_CONVERSION_EITS	PR_CONVERSION_EITS
IM_IPM_INTENDED_RECIPIENT	PR_ORIGINALLY_INTENDED_RECIPIENT_NAME
IM_IPN_ORIGINATOR	PR_CALLBACK_TELEPHONE_NUMBER, PR_SENDER_ADDRTYPE, PR_SENDER_EMAIL_ADDRESS, PR_SENDER_ENTRYID, PR_SENDER_NAME, PR_SENDER_SEARCH_KEY (and all PR_SENT_REPRESENTING_* properties)
IM_SUBJECT_IPM	PR_ORIGINAL_SEARCH_KEY

OMP_O_IM_C_IPM_IDENTIFIER

The attributes of the class OMP_O_IM_C_IPM_IDENTIFIER map to MAPI properties as follows:

MH ID/Type	MAPI Property
IM_USER, IM_USER_RELATIVE_IDENTIFIER	Construct a GUID from IM_USER_RELATIVE_IDENTIFIER

OMP_O_IM_C_ISO_6937_TEXT_BD_PRT

The attributes of the class OMP_O_IM_C_ISO_6937_TEXT_BD_PRT are mapped to the following MAPI properties:

MH ID/Type	**MAPI Property**
IM_REPERTOIRE	Not mapped to a MAPI property.
IM_TEXT	PR_ATTACH_DATA_BIN

OMP_O_IM_C_MESSAGE_BD_PRT

The attributes of the class OMP_O_IM_C_MESSAGE_BD_PRT are mapped as follows:

MH ID/Type	MAPI Property
IM_ENVELOPE	Not mapped to a MAPI property.
IM_IPM	Mapped to an IMessage object embedded in the message.

OMP_O_IM_C_MIXED_MODE_BD_PRT

The attributes of the class OMP_O_IM_C_MIXED_MODE_BD_PRT are not mapped to any MAPI properties.

MH ID/Type	MAPI Property
IM_MIXED_MODE_DOCUMENT	Not mapped to a MAPI property.

OMP_O_IM_C_NATIONAL_DEF_BD_PRT

The attributes of the class OMP_O_IM_C_NATIONAL_DEF_BD_PRT are not mapped to MAPI properties.

MH ID/Type	MAPI Property
IM_NATIONAL_DATA	Not mapped to a MAPI property.

OMP_O_IM_C_NON_RECEIPT_NOTIF

The attributes of the class OMP_O_IM_C_NON_RECEIPT_NOTIF map to MAPI properties (when PR_NON_RECEIPT_NOTIFICATION_REQUESTED = 1 in Recipient table or when the message has PR_READ_RECEIPT_REQUESTED = 1), as follows:

MH ID/Type	MAPI Property
IM_AUTO_FORWARD_COMMENT	PR_AUTO_FORWARD_COMMENT
IM_DISCARD_REASON	PR_DISCARD_REASON
IM_NON_RECEIPT_REASON	PR_NON_RECEIPT_REASON
IM_RETURNED_IPM	(Object; no corresponding property)

The following constant values are mapped from IM_DISCARD_REASON to PR_DISCARD_REASON:

IM_DISCARD_REASON value	PR_DISCARD_REASON value
IM_NO_DISCARD	-1
IM_IPM_EXPIRED	0
IM_IPM_OBSOLETED	1
IM_USER_TERMINATED	2

The following constant values are mapped from IM_NON_RECEIPT_REASON to PR_NON_RECEIPT_REASON:

IM_NON_RECEIPT_REASON value	PR_NON_RECEIPT_REASON value
IM_IPM_AUTO_FORWARDED	1
IM_IPM_DISCARDED	0

OMP_O_IM_C_ODA_BD_PRT

The attributes of the class OMP_O_IM_C_ODA_BD_PRT are not mapped to MAPI properties.

MH ID/Type	MAPI Property
IM_APPLICATION_PROFILE	Not mapped to a MAPI property.
IM_ARCHITECTURE_CLASS	Not mapped to a MAPI property.
IM_ODA_DOCUMENT	Not mapped to a MAPI property.

OMP_O_IM_C_OR_DESCRIPTOR

The attributes of the class OMP_O_IM_C_OR_DESCRIPTOR map to MAPI properties as follows:

MH ID/Type	**MAPI Property**
IM_FORMAL_NAME	Extract name using PR_ENTRYID; for X.400 address types, build an OR Address from string address.
IM_FREE_FORM_NAME	PR_SENDER_NAME for originator, PR_DISPLAY_NAME for recipient, or extract the display name using PR_SENDER_ENTRYID
IM_TELEPHONE_NUMBER	PR_CALLBACK_TELEPHONE_NUMBER for originator, PR_PRIMARY_TELEPHONE_NUMBER for recipient

OMP_O_IM_C_RECEIPT_NOTIF

The attributes of the class OMP_O_IM_C_RECEIPT_NOTIF map to MAPI properties, as follows:

MH ID/Type	**MAPI Property**
IM_ACKNOWLEDGEMENT_MODE	PR_ACKNOWLEDGEMENT_MODE
IM_RECEIPT_TIME	PR_RECEIPT_TIME
IM_SUPPLEMENTARY_RECEIPT_INFO	PR_REPORT_TEXT

The following constant values are mapped from IM_ACKNOWLEDGEMENT_MODE to PR_ACKNOWLEDGEMENT_MODE:

IM_ACKNOWLEDGEMENT_ MODE value	PR_ACKNOWLEDGEMENT_ MODE value
IM_AUTOMATIC	1
IM_MANUAL	0

OMP_O_IM_C_RECIPIENT_SPECIFIER

The attributes of the class OMP_O_IM_C_OR_DESCRIPTOR map to MAPI properties as follows:

MH ID/Type	**MAPI Property**
IM_IPM_RETURN_REQUESTED	PR_IPM_RETURN_REQUESTED
IM_NOTIFICATION_REQUEST	PR_NON_RECEIPT_NOTIFICATION_REQUESTED, PR_READ_RECEIPT_REQUESTED
IM_RECIPIENT	PR_ADDRTYPE, PR_DISPLAY_NAME, PR_EMAIL_ADDRESS, PR_ENTRYID, PR_OFFICE_LOCATION, PR_PRIMARY_TELEPHONE_NUMBER, PR_SEARCH_KEY
IM_REPLY_REQUESTED	PR_REPLY_REQUESTED

The following constant values are mapped from IM_NOTIFICATION_REQUEST to the corresponding MAPI properties:

IM_NOTIFICATION_REQUEST Value	MAPI Values
IM_ALWAYS	PR_NON_RECEIPT_NOTIFICATION_REQUESTED = TRUE, PR_READ_RECEIPT_REQUESTED = TRUE
IM_NEVER	PR_NON_RECEIPT_NOTIFICATION_REQUESTED = FALSE, PR_READ_RECEIPT_REQUESTED = FALSE
IM_NON_RECEIPT	PR_NON_RECEIPT_NOTIFICATION_REQUESTED = TRUE, PR_READ_RECEIPT_REQUESTED = FALSE

OMP_O_IM_C_TELETEX_BD_PRT

The attributes of the class OMP_O_IM_C_TELETEX_BD_PRT are mapped to the following MAPI properties:

MH ID/Type	MAPI Property
IM_TELETEX_COMPATIBLE	Not mapped to a MAPI property.
IM_TELETEX_DOCUMENT	PR_BODY
IM_TELETEX_NBPS	Not mapped to a MAPI property.

OMP_O_IM_C_UNIDENTIFIED_BD_PRT

The attributes of the class OMP_O_IM_C_UNIDENTIFIED_BD_PRT are not mapped to MAPI properties.

MH ID/Type	MAPI Property
IM_UNIDENTIFIED_DATA	Not mapped to a MAPI property.
IM_UNIDENTIFIED_TAG	Not mapped to a MAPI property.

OMP_O_IM_C_USA_NAT_DEF_BD_PRT

The attributes of the class OMP_O_IM_C_USA_NAT_DEF_BD_PRT are mapped to the following MAPI properties:

MH ID/Type	MAPI Property
IM_BODY_PART_NUMBER	Not mapped to a MAPI property.
IM_USA_DATA	Not mapped to a MAPI property.

OMP_O_IM_C_VIDEOTEX_BD_PRT

The attributes of the class OMP_O_IM_C_VIDEOTEX_BD_PRT are not mapped to MAPI properties.

MH ID/Type	**MAPI Property**
IM_VIDEOTEX_DATA	Not mapped to a MAPI property.
IM_VIDEOTEX_SYNTAX	Not mapped to a MAPI property.

Comprehensive X.400 Attributes Reference

The following table contains mappings between X.400 P2 attributes and MAPI properties, listed in alphabetical order. The name of the object appears in parentheses.

MH ID/Type	MAPI Property
IM_ACKNOWLEDGEMENT_MODE (OMP_O_IM_C_RECEIPT_NOTIF)	PR_ACKNOWLEDGEMENT_MODE
IM_APPLICATION_PROFILE (OMP_O_IM_C_ODA_BD_PRT)	Not mapped to a MAPI property.
IM_ARCHITECTURE_CLASS (OMP_O_IM_C_ODA_BD_PRT)	Not mapped to a MAPI property.
IM_AUTHORIZING_USERS (OMP_O_IM_C_INTERPERSONAL_MSG)	PR_AUTHORIZING_USERS
IM_AUTO_FORWARD_COMMENT (OMP_O_IM_C_NON_RECEIPT_NOTIF)	PR_AUTO_FORWARD_COMMENT
IM_AUTO_FORWARDED (OMP_O_IM_C_INTERPERSONAL_MSG)	PR_AUTO_FORWARDED
IM_BILATERAL_DATA (OMP_O_IM_C_BILAT_DEF_BD_PRT)	PR_ATTACH_DATA_BIN
IM_BLIND_COPY_RECIPIENTS (OMP_O_IM_C_INTERPERSONAL_MSG)	PR_DISPLAY_BCC
IM_BODY (OMP_O_IM_C_INTERPERSONAL_MSG)	PR_BODY
IM_BODY_PART_NUMBER (OMP_O_IM_C_USA_NAT_DEF_BD_PRT)	Not mapped to a MAPI property.
IM_CHAR_SET_REG (OMP_O_IM_C_GENERAL_TEXT_BD_PRT)	Not mapped to a MAPI property.
IM_CONVERSION_EITS (OMP_O_IM_C_INTERPERSONAL_NOTIF)	PR_CONVERSION_EITS
IM_COPY_RECIPIENTS (OMP_O_IM_C_INTERPERSONAL_MSG)	PR_DISPLAY_CC
IM_DISCARD_REASON (OMP_O_IM_C_NON_RECEIPT_NOTIF)	PR_DISCARD_REASON
IM_ENVELOPE (OMP_O_IM_C_MESSAGE_BD_PRT)	Not mapped to a MAPI property.
IM_EXPIRY_TIME (OMP_O_IM_C_INTERPERSONAL_MSG)	PR_EXPIRY_TIME
IM_EXTERNAL_DATA (OMP_O_IM_C_EXTERN_DEF_BD_PRT)	PR_ATTACH_DATA_BIN, PR_ATTACH_FILENAME, PR_ATTACH_TAG
IM_EXTERNAL_PARAMETERS (OMP_O_IM_C_EXTERN_DEF_BD_PRT)	PR_ATTACHMENT_X400_PARAMETERS
IM_FORMAL_NAME (OMP_O_IM_C_OR_DESCRIPTOR)	Extract name using PR_ENTRYID; for X.400 address types, build an OR Address from string address.
IM_FREE_FORM_NAME (OMP_O_IM_C_OR_DESCRIPTOR)	PR_SENDER_NAME for originator, PR_DISPLAY_NAME for recipient, or extract the display name using

	PR_SENDER_ENTRYID
IM_G3_FAX_NBPS (OMP_O_IM_C_G3_FAX_BD_PRT)	Not mapped to a MAPI property.
IM_G4_CLASS_1_DOCUMENT (OMP_O_IM_C_G4_CLASS_1_BD_PRT)	Not mapped to a MAPI property.
IM_IMAGES (OMP_O_IM_C_G3_FAX_BD_PRT)	Not mapped to a MAPI property.
IM_IMPORTANCE (OMP_O_IM_C_INTERPERSONAL_MSG)	PR_IMPORTANCE
IM_INCOMPLETE_COPY (OMP_O_IM_C_INTERPERSONAL_MSG)	PR_INCOMPLETE_COPY
IM_IPM (OMP_O_IM_C_MESSAGE_BD_PRT)	Mapped to an IMessageobject embedded in the message.
IM_IPM_INTENDED_RECIPIENT (OMP_O_IM_C_INTERPERSONAL_NOTIF)	PR_ORIGINALLY_INTENDED_RECIPIENT_NAME
IM_IPM_RETURN_REQUESTED (OMP_O_IM_C_OR_DESCRIPTOR)	PR_IPM_RETURN_REQUESTED
IM_IPN_ORIGINATOR (OMP_O_IM_C_INTERPERSONAL_NOTIF)	PR_CALLBACK_TELEPHONE_NUMBER, PR_SENDER_ADDRTYPE, PR_SENDER_EMAIL_ADDRESS, PR_SENDER_ENTRYID, PR_SENDER_NAME, PR_SENDER_SEARCH_KEY (and all PR_SENT_REPRESENTING_* properties)
IM_LANGUAGES (OMP_O_IM_C_INTERPERSONAL_MSG)	PR_LANGUAGES
IM_MIXED_MODE_DOCUMENT (OMP_O_IM_C_MIXED_MODE_BD_PRT)	Not mapped to a MAPI property.
IM_NATIONAL_DATA (OMP_O_IM_C_NATIONAL_DEF_BD_PRT)	Not mapped to a MAPI property.
IM_NON_RECEIPT_REASON (OMP_O_IM_C_NON_RECEIPT_NOTIF)	PR_NON_RECEIPT_REASON
IM_NOTIFICATION_REQUEST (OMP_O_IM_C_OR_DESCRIPTOR)	PR_NON_RECEIPT_NOTIFICATION_REQUESTED, PR_READ_RECEIPT_REQUESTED
IM_OBSOLETED_IPMS (OMP_O_IM_C_INTERPERSONAL_MSG)	PR_OBSOLETED_IPMS
IM_ODA_DOCUMENT (OMP_O_IM_C_ODA_BD_PRT)	Not mapped to a MAPI property.
IM_ORIGINATOR (OMP_O_IM_C_INTERPERSONAL_MSG)	PR_CALLBACK_TELEPHONE_NUMBER, PR_SENDER_ADDRTYPE, PR_SENDER_EMAIL_ADDRESS, PR_SENDER_ENTRYID, PR_SENDER_NAME, PR_SENDER_SEARCH_KEY (and all PR_SENT_REPRESENTING_* properties)
IM_PRIMARY_RECIPIENTS (OMP_O_IM_C_INTERPERSONAL_MSG)	PR_DISPLAY_TO
IM_RECEIPT_TIME	PR_RECEIPT_TIME

(OMP_O_IM_C_RECEIPT_NOTIF)	
IM_RECIPIENT (OMP_O_IM_C_OR_DESCRIPTOR)	PR_ADDRTYPE, PR_DISPLAY_NAME, PR_EMAIL_ADDRESS, PR_ENTRYID, PR_OFFICE_LOCATION, PR_PRIMARY_TELEPHONE_NUMBER, PR_SEARCH_KEY
IM_RELATED_IPMS (OMP_O_IM_C_INTERPERSONAL_MSG)	PR_RELATED_IPMS
IM_REPERTOIRE (OMP_O_IM_C_IA5_TEXT_BD_PRT)	Not mapped to a MAPI property.
IM_REPERTOIRE (OMP_O_IM_C_ISO_6937_TEXT_BD_PRT)	Not mapped to a MAPI property.
IM_REPLIED_TO_IPM (OMP_O_IM_C_INTERPERSONAL_MSG)	PR_PARENT_KEY
IM_REPLY_RECIPIENTS (OMP_O_IM_C_INTERPERSONAL_MSG)	PR_REPLY_RECIPIENT_ENTRIES, PR_REPLY_RECIPIENT_NAMES
IM_REPLY_REQUESTED (OMP_O_IM_C_OR_DESCRIPTOR)	PR_REPLY_REQUESTED
IM_REPLY_TIME (OMP_O_IM_C_INTERPERSONAL_MSG)	PR_REPLY_TIME
IM_RETURNED_IPM (OMP_O_IM_C_NON_RECEIPT_NOTIF)	(Object; no corresponding property)
IM_SENSITIVITY (OMP_O_IM_C_INTERPERSONAL_MSG)	PR_SENSITIVITY
IM_SUBJECT (OMP_O_IM_C_INTERPERSONAL_MSG)	PR_NORMALIZED_SUBJECT, PR_SUBJECT, PR_SUBJECT_PREFIX
IM_SUBJECT_IPM (OMP_O_IM_C_INTERPERSONAL_NOTIF)	PR_ORIGINAL_SEARCH_KEY
IM_SUPPLEMENTARY_RECEIPT_INFO (OMP_O_IM_C_RECEIPT_NOTIF)	PR_REPORT_TEXT
IM_TELEPHONE_NUMBER (OMP_O_IM_C_OR_DESCRIPTOR)	PR_CALLBACK_TELEPHONE_NUMBER for Originator, PR_PRIMARY_TELEPHONE_NUMBER for Recipient
IM_TELETEX_COMPATIBLE (OMP_O_IM_C_TELETEX_BD_PRT)	Not mapped to a MAPI property.
IM_TELETEX_DOCUMENT (OMP_O_IM_C_TELETEX_BD_PRT)	PR_BODY
IM_TELETEX_NBPS (OMP_O_IM_C_TELETEX_BD_PRT)	Not mapped to a MAPI property.
IM_TEXT (OMP_O_IM_C_GENERAL_TEXT_BD_PRT)	Not mapped to a MAPI property.
IM_TEXT (OMP_O_IM_C_IA5_TEXT_BD_PRT)	PR_BODY
IM_TEXT (OMP_O_IM_C_ISO_6937_TEXT_BD_PRT)	PR_ATTACH_DATA_BIN
IM_THIS_IPM (OMP_O_IM_C_INTERPERSONAL_MSG)	PR_SEARCH_KEY
IM_UNIDENTIFIED_DATA	Not mapped to a MAPI property.

(OMP_O_IM_C_UNIDENTIFIED_BD_PRT)	
IM_UNIDENTIFIED_TAG (OMP_O_IM_C_UNIDENTIFIED_BD_PRT)	Not mapped to a MAPI property.
IM_USA_DATA (OMP_O_IM_C_USA_NAT_DEF_BD_PRT)	Not mapped to a MAPI property.
IM_USER, IM_USER_RELATIVE_IDENTIFIER (OMP_O_IM_C_IPM_IDENTIFIER)	Construct a GUID from IM_USER_RELATIVE_IDENTIFIER
IM_VIDEOTEX_DATA (OMP_O_IM_C_VIDEOTEX_BD_PRT)	Not mapped to a MAPI property.
IM_VIDEOTEX_SYNTAX (OMP_O_IM_C_VIDEOTEX_BD_PRT)	Not mapped to a MAPI property.

Mapping of Internet Mail Attributes to MAPI Properties

This appendix describes how a MAPI transport provider or MAPI-aware gateway which connects to the Internet should translate between MAPI message properties and Simple Message Transport Protocol (SMTP) message attributes. SMTP is the messaging protocol used on much of the Internet. SMTP defines a set of message headers (the message envelope) and a message content format. SMTP is fully documented in a set of two docments, RFC 821 and RFC 822, which can be found at a number of FTP and WWW sites on the Internet.

The goal of mapping SMTP message attributes to MAPI properties (and vice versa) is to ensure that the full content of MAPI messages, over and above that which can be encoded using native SMTP message attributes, can be reliably exchanged among different MAPI components that must communicate over the Internet. This document is based on work already done on such components at Microsoft. How to translate between MAPI message properties and X.400 message attributes is described in the appendix, Mapping of X.400 P2 Attributes to MAPI Properties.

This document assumes familiarity with MAPI transports, TNEF, and SMTP mail. It strives to be concise rather than abundantly clear.

As a convention, "outbound" refers to mail traveling from a MAPI-compliant UA or MTA to the Internet, and "inbound" refers to mail traveling from the Internet to a MAPI component.

Addressing

The format of SMTP e-mail addresses is defined in RFC 822. MAPI components should handle any address that complies with that standard. However, there is a particular form of RFC 822 address that best encodes MAPI addresses:

display-name <*email-address*>

The angle brackets are included as literals. Blanks are common in display names; they need not be quoted. A typical address might look like this one, which belongs to one of the coauthors of RFC 1521:

Nathaniel Borenstein <nsb@bellcore.com>

If the display name contains characters that have special meaning in SMTP addresses, such as < or @, the entire display name should be quoted using double quotes. On outbound mail, if the total length of the e-mail address plus display name exceeds 255 characters, the display name should be dropped.

The parts of an SMTP address map into MAPI properties as follows:

SMTP address component	**MAPI property**
display-name for all recipients	PR_DISPLAY_NAME
display-name for From field	PR_SENDER_NAME
display-name for Sender field	PR_SENT_REPRESENTING_NAME
email-address	PR_EMAIL_ADDRESS
implicit, always "SMTP"	PR_ADDRTYPE

If there is no display name for an address on inbound mail, the entire e-mail address should be used instead. The address type is always SMTP.

Recipient properties are taken from the MAPI message's recipient table; sender properties are taken from the message itself.

Message Envelope

RFC 822 headers are mapped to MAPI properties as follows. PR_SENDER_* is an abbreviation for the following 5 properties:

PR_SENDER_NAME
PR_SENDER_ADDRTYPE
PR_SENDER_EMAIL_ADDRESS
PR_SENDER_SEARCH_KEY
PR_SENDER_ENTRYID

Similar abbreviations are used for PR_SENT_REPRESENTING_* and other groups of message properties.

SMTP header	**MAPI property**
From:	Outbound: PR_SENDER_*; inbound: PR_SENDER_* and PR_SENT_REPRESENTING_*
Date:	Outbound: current time; inbound: PR_MESSAGE_DELIVERY_TIME
To:	PR_DISPLAY_NAME and PR_EMAIL_ADDRESS for recipients where PR_RECIPIENT_TYPE is MAPI_TO
Cc:	PR_DISPLAY_NAME and PR_EMAIL_ADDRESS for recipients where PR_RECIPIENT_TYPE is MAPI_CC
Bcc:	PR_DISPLAY_NAME and PR_EMAIL_ADDRESS for recipients where PR_RECIPIENT_TYPE is MAPI_BCC
Received:	No corresponding MAPI property; put local host name and your component name here
Return-receipt-to:	PR_REPORT_NAME and PR_REPORT_ENTRYID
Reply-to:	PR_REPLY_RECIPIENT_ENTRIES and PR_REPLY_RECIPIENT_NAMES
Subject:	PR_SUBJECT No particular length limitation.
MIME-version:	Always "1.0"
X-MS-Attachment:	For compatibility with MS Mail SMTP gateway. *filename size mm-dd-yyy hh:mm* Details below.
entire SMTP message	PR_TRANSPORT_MESSAGE_HEAD

envelope	ERS
header name TBD	PR_SEND_RICH_INFO *for sender only.* The TBD header should be used to determine whether the sender is capable of interpreting TNEF content in a reply.
MessageID:	PR_TNEF_CORRELATION_KEY
Content-type	Either text/plain or multipart/mixed. See "Message Content" section.

The X-MS-Attachment header is formatted as four tokens, separated by blanks:

name size date time

The first token is the filename, which may contain embedded blanks, so this header should be parsed from the right on inbound messages. The size is in bytes; the date is formatted as *mm-dd-yyyy,* and the time as *hh:mm.*

Note MessageID is not mapped to PR_SEARCH_KEY because the SMTP domain has specific requirements on the format of the message identifier which make it impossible to encode an arbitrary MAPI message identifier. Instead, MessageID is mapped to PR_TNEF_CORRELATION_KEY. This property is a transport-defined property that is set by the transport sending an outbound message and used by a transport receiving an inbound message. For more information, see Developing a TNEF-Enabled Transport Provider.

Message Content

There are two possible encodings for the message content: one using MIME, the other using uuencode. MIME is the preferred encoding. In addition, MAPI defines a per-recipient property, PR_SEND_ RICH_INFO, that governs whether or not TNEF information should be included in an outgoing message. So there are a total of four ways of encoding message content:

- MIME with TNEF
- MIME without TNEF
- uuencode with TNEF
- uuencode without TNEF

How to choose MIME or uuencode for outbound messages is not specified.

The following properties are excluded from TNEF: PR_SENDER_*, PR_ATTACH_DATA_*, PR_BODY. All other transmittable message properties are included in the TNEF stream.

Message Text

For outbound messages in MIME mode, the content-type depends on whether there are attachments and what the message text looks like. If there are attachments, the Content-type is *multipart/mixed;* the message text and each attachment become a separate part of the message content, each with its own content-type. If there are no attachments, the content-type of the message is *text/plain* and there is only one part.

The message text is not line-wrapped unless some line exceeds 140 characters in length. If one does, the entire text is wrapped to 76 columns and the *quoted-printable* encoding is used to preserve line breaks. The content-type depends on what characters are found in the message text, as follows:

- If only 7-bit characters are found and no line exceeds 140 characters in length, the message is ASCII text.

  Content-type: text/plain; charset=us-ascii
  (Content-Transfer-Encoding=7bit is assumed.)
- If long lines or 8-bit characters are found, the message is text and the character set is determined by the locale. It should be chosen from the character sets defined by ISO standard 8859.

  Content-type: text/plain; charset=iso-8859-1 (or another valid charset)
  Content-Transfer-Encoding: quoted-printable

For inbound MIME messages, if the first message content part has *Content-type: text/** (that is, any text type) and its character set is recognized, it is mapped to PR_BODY. A first message content part not meeting this criterion becomes an attachment. Any subsequent parts also become attachments.

In uuencode mode, message text in outbound messages is line-wrapped to 78 columns, as for MS Mail 3.x. The content-type is "text/plain." To preserve the original message's paragraph breaks under these circumstances, observe the following conventions in the wrapped text. There are three possible reasons for ending a line of text, each with its own character sequence:

- Line-break. The original text contained a newline entered by the user (paragraph mark). In the transport, this maps to a newline with *no* preceding blanks. If the user enters a newline preceded by blanks, the blanks should be stripped out.
- Line-nobreak. The original text contained a word too long to fit on a single line of the message. In the transport, this maps to a newline preceded by *two* blanks.
- Line-wrap. The original text contained no newline, the text is too long to fit on a single line of the message, but it can be broken between two words. In the transport, this maps to a newline preceded by a *single* blank.

Attached Files and Messages

MIME with TNEF. All attachment properties and content are in the TNEF stream. The TNEF itself is a single, binary attached file named WINMAIL.DAT, encoded as described for *MIME without TNEF.*

MIME without TNEF. Attached files are sent as MIME message content parts. The file name is placed in the *name* parameter to the *Content-type* header for the attachment. The character set for the attachment is placed in the *charset* parameter to the *Content-type;* it and the content-transfer-encoding are determined by scanning the entire attachment content. URL attachments are treated specially:

- If the attachment is a URL (an attached file with extension .URL), and the access mode defined in it is anonymous FTP, it is encoded as an external message, and the content of the file (the URL) is copied into the header of the external message.

  Content-type: message/external-body; access-type=anon-ftp
  (Content-Transfer-Encoding: 7bit is assumed.)
- If only 7-bit characters are found and no line exceeds 140 characters in length, the attachment is ASCII text.

  Content-type: text/plain; charset=us-ascii
  Content-Transfer-Encoding: 7bit
- If long lines or up to 25% 8-bit characters are found, the attachment content is text and the character set is determined by the locale. It should be chosen from the character sets defined by ISO standard 8859.

  Content-type: text/plain; charset=ISO-8859-1 (or what have you)
  Content-Transfer-Encoding: quoted-printable
- If 25% or more of the characters have the high bit set, the attachment is binary. It is encoded using the Base64 algorithm.

  Content-type: application/octet-stream (by default; based on file extension)
  Content-Transfer-Encoding: base64

On outbound messages, the content-type should be derived from the filename's three-letter extension. This mapping exists in the system registry; under there is a string value named "Content Type" that gives the MIME content type if one is defined. This example is for a TIFF image file:

```
HKEY_LOCAL_MACHINE\
    Software\
        Microsoft\
            Classes\
                .tif
                    Content Type = "image/tiff"
```

If there is no mapping for the file extension, the default *application/octet*-stream should be used. Windows 95 now supports this system registry mapping. Windows NT will support it in a forthcoming release. Windows 3.x does not support a system registry; implementations are free to choose how they store this mapping.

On inbound messages, the content-type for an attachment should always be copied to the MAPI property PR_ATTACH_MIME_TAG. Even if a filename is defined for an attached file, the extension mapped by the content-type should be used in the PR_ATTACH_FILENAME and PR_ATTACH_EXTENSION properties.

The *name* parameter is officially "deprecated" by RFC 821. As standards evolve, Microsoft will consider specifying an alternate mapping for attached filenames.

Outbound attached messages are sent as
Content-type: message/rfc822
Messages within attached messages are encoded recursively, in their proper place. Inbound message content parts with *Content-Type: multipart/digest* are also mapped to embedded messages.

Uuencode with TNEF. All attachment properties and content are in the TNEF stream. The TNEF itself is a single, binary attached file named WINMAIL.DAT, encoded as described for *Uuencode without TNEF.*

Uuencode without TNEF. All attached files are treated as binary and uuencoded, following the message text. The file name is present in the uuencode header:

```
    begin 0755 WINMAIL.DAT
    ... data ...
    end
```

Attached messages are textized into the message text. The hierarchy of attached messages is always flattened; that is, messages within attached messages are pulled out to the top level.

Embedded OLE objects are discarded.

General. Attachment rendering positions are transmitted literally, using the property PR_ATTACH_RENDERING_POSITION in the TNEF. If TNEF is not used, they are lost. Incoming attachments with no rendering position (including when there is no TNEF) have their rendering position set to 0xFFFFFFFF, i.e. no position within the message text.

Suggested Configuration Parameters

The following are suggestions only, not requirements. The intent is to provide a convenient list of parameters which the implementation can choose how to support.

Whether to encode using MIME or uuencode for outbound messages: boolean.

Character set to use for outbound messages: string (copied directly to charset parameter) or enumeration (translated internally to charset string).

(possibly, if not supplied by the OS) List of mappings between file extensions and content-types.

References

RFC 821 (for information on the SMTP protocol used to communicate with SMTP-based mail agents)

RFC 822 (for addressing and standard message headers)

RFC 1521 (for MIME)

Regular Expressions

MAPI supports a limited form of regular expression notation. A regular expression specifies a set of character strings. A member of this set of strings is said to match the regular expression. The regular expressions allowed by MAPI are constructed as follows:

The following one-character regular expressions match a single character:

1.1. An ordinary character (not one of those discussed in 1.2 below) is a one-character regular expression that matches itself.

1.2. You form regular expressions using special characters, ".", "*", "[", and the like. If you want to use a special character literally − without its special meaning − in a regular expression, you must quote it by preceding it with "\". A backslash (\) followed by any special character is a one-character regular expression that matches the special character itself. MAPI interprets special characters in strings as regular expressions *only* when the relop is RELOP_RE, so do not be tempted to quote the special characters everywhere.

The special characters are:

a. .,*, [, and \ (dot, star,left square bracket, and backslash, respectively), which are always special, except when they appear within square brackets ([]; see 1.4).

b. ^ (caret), which is special at the beginning of an entire regular expression (see 3.1 and 3.2), or when it immediately follows the left of a pair of square brackets ([]) (see 1.4).

c. $ (dollar sign), which is special at the end of an entire regular expression (see 3.2).

1.3. A period (.) is a one-character regular expression that matches any character except NEWLINE.

1.4. A nonempty string of characters enclosed in square brackets ([]) is a one-character regular expression that matches any single character in that string. If, however, the first character of the string is a caret (^), the one-character regular expression matches any character except NEWLINE and the remaining characters in the string. The star (*) has this special meaning only if it occurs first in the string. The dash (-) can be used to indicate a range of consecutive ASCII characters; for example, [0-9] is equivalent to [0123456789]. The dash (-) loses this special meaning if it occurs first (after an initial caret (^), if any) or last in the string. The right square bracket (]) does not terminate such a string when it is the first character within it (after an initial caret (^), if any); for example, []a-f], matches either a right square bracket (]) or one of the letters "a" through "f". Dot, star, left bracket, and the backslash lose their special meaning within such a string of characters.

Use the following rules to construct regular expressions from one-character regular expressions:

2.1 A one-character regular expression matches itself.

2.2 A one-character regular expression followed by a star (*) is a regular expression that matches zero or more occurrences of the one-character regular expression. If there is any choice, the longest leftmost string that permits a match is chosen.

2.3 A one-character regular expression followed by \{m\}, \{m,\}, or \{m,n\} is a regular expression that matches a range of occurrences of the one-character regular expression. The values of m and n must be nonnegative integers less than 255; \{m\} matches exactly m occurrences; \{m,\} matches at least m occurrences; \{m,n\} matches any number of occurrences between m and n, inclusive. Whenever a choice exists, the regular expression matches as many occurrences as possible.

2.4 A concatenation of regular expressions is a regular expression that matches the concatenation of the strings matched by each component of the regular expression.

2.5 A regular expression enclosed between the character sequences \(and \) is a regular expression that matches whatever the original regular expression matches. See 2.6 for a discussion of why this is useful.

2.6 The \n expression matches the same string of characters that was matched by an expression enclosed between \(and \) earlier in the same regular expression. Here n is a digit; the subexpression specified is that expression beginning with the nth occurrence of \(counting from the left. For example, the expression matches a line consisting of two repeated appearances of the same string.

Finally, you can constrain an entire regular expression to match only an initial segment or final segment of a line (or both):

3.1 A caret (^) at the beginning of an entire regular expression constrains that regular expression to match an initial segment of a line.

3.2 A dollar sign ($) at the end of an entire regular expression constrains that regular expression to match a final segment of a line. The construction ^entire regular expression$ constrains the entire regular expression to match the entire line.

Functionality Groups

This appendix provides reference information grouped by functionality. It identifies and describes each group and lists the reference entries that belong in it.

Function Groups

The following are groups of related functions.

Arithmetic Functions

The following functions are used by clients and service providers:

FtAddFt
FtMulDw
FtMulDwDw
FtNegFt
FtSubFt
HexFromBin
ScBinFromHexBounded

Base Functions

The following functions are used only by clients:

MAPIInitialize
MAPIUninitialize
MAPILogonEx

Forms Functions

The following functions are used only by clients:

MAPIOpenFormMgr
MAPIOpenLocalFormContainer

Identifier Functions

The following functions are used by clients and service providers:

HrSzFromEntryID
HrEntryIDFromSz
WrapStoreEntryID
HrComposeEID
HrDecomposeEID
HrComposeMsgID
HrDecomposeMsgID

Idle Functions

The following functions are used by clients and service providers:

DeregisterIdleRoutine
ChangeIdleRoutine
FtgRegisterIdleRoutine
EnableIdleRoutine

Initialization Functions

The following entry point functions are implemented by service providers:

HPProviderInit
MSProviderInit
XPProviderInit
ABProviderInit

IStorage Functions

The following functions are used by clients and service providers:

GetAttribIMsgOnIStg
HrIStorageFromStream
MapStorageSCode
OpenStreamOnFile
OpenIMsgOnIStg
SetAttribIMsgOnIStg

IUnknown Functions

The following functions are used by clients and service providers:

UlAddRef
UlRelease

Memory Management Functions

The following functions are used by clients and service providers:

MAPIFreeBuffer
MAPIAllocateBuffer
MAPIAllocateMore
MAPIGetDefaultMalloc
FreePadrlist

Message Session Functions

The following functions are used by clients and service providers:

OpenIMsgSession
CloseIMsgSession

Notification Functions

The following functions are used by clients and service providers:

HrAllocAdviseSink
HrThisThreadAdviseSink
ScCopyNotifications
ScCountNotifications
ScRelocNotifications

Parameter Validation Functions

The following functions are used by clients, service providers, and MAPI:

CheckParameters
CheckParms
UIValidateParameters
UIValidateParms
ValidateParameters
ValidateParms

Path Name Functions

The following functions are used by clients and service providers:

ScLocalPathFromUNC
ScUNCFromLocalPath

Preprocessor Functions

The following functions are implemented by service providers:

PreprocessMessage
RemovePreprocessInfo

Properties Functions

The following functions are used by clients and service providers:

CreateIProp
FPropCompareProp
FPropContainsProp
FPropExists
GetInstance
HrSetOneProp
HrGetOneProp
LPropCompareProp
PpropFindProp
PropCopyMore
ScCopyProps
ScCountProps
ScDupPropset
ScRelocProps
UlPropSize

RTF Synchronization Functions

The following functions are used by clients and service providers:

RTFSync
WrapCompressedRTFStream

String Operator Functions

The following functions are used by clients and service providers:

FBinFromHex
FEqualNames
SzFindCh
SzFindLastCh
SzFindSz
UFromSz
UlFromSzHex

Structure Validation Functions

The following functions are used by clients and service providers:

FBadEntryList
FBadProp
FBadColumnSet
FBadPropTag
FBadRestriction
FBadRglpNameID
FBadRglpszW
FBadRow
FBadRowSet
FBadSortOrderSet

Table Functions

The following functions are used by clients and service providers:

CreateTable
FreeProws
HrAddColumnsEx
HrQueryAllRows

Tnef Functions

The following functions are used only by service providers:

OpenTnefStream
OpenTnefStreamEx

Property Groups

Many of the MAPI properties fall into groups based on which ones are used in conjunction with each other. The following lists enumerate the groups and provide additional references where appropriate.

Actual Recipient Properties

The following are the address properties for the messaging user that actually receives the message:

PR_RECEIVED_BY_ADDRTYPE
PR_RECEIVED_BY_EMAIL_ADDRESS
PR_RECEIVED_BY_ENTRYID
PR_RECEIVED_BY_NAME
PR_RECEIVED_BY_SEARCH_KEY

For more information on the address properties, see About Base Address Properties.

Alternate Recipient Properties

The following are the alternate autoforwarding recipient properties:

PR_ALTERNATE_RECIPIENT
PR_ALTERNATE_RECIPIENT_ALLOWED
PR_ORIGINATOR_REQUESTED_ALTERNATE_RECIPIENT

For more properties related to autoforwarding, see Autoforwarding Properties and Originally Intended Recipient Properties.

Attachment Properties

The following are the attachment properties for a message:

PR_ATTACH_DATA_BIN
PR_ATTACH_DATA_OBJ
PR_ATTACH_ENCODING
PR_ATTACH_EXTENSION
PR_ATTACH_FILENAME
PR_ATTACH_LONG_FILENAME
PR_ATTACH_LONG_PATHNAME
PR_ATTACH_METHOD
PR_ATTACH_MIME_TAG
PR_ATTACH_NUM
PR_ATTACH_PATHNAME
PR_ATTACH_RENDERING
PR_ATTACH_SIZE
PR_ATTACH_TAG
PR_ATTACH_TRANSPORT_NAME
PR_HASATTACH
PR_PHYSICAL_RENDITION_ATTRIBUTES
PR_RENDERING_POSITION
PR_TNEF_CORRELATION_KEY

For more information on attachments, see About Message Attachments.

Autoforwarding Properties

The following are the autoforwarding properties for a message:

PR_AUTO_FORWARD_COMMENT
PR_AUTO_FORWARDED
PR_REDIRECTION_HISTORY

For more properties related to autoforwarding, see Alternate Recipient Properties and Originally Intended Recipient Properties.

Base Address Properties

The following are the base address properties for all messaging users:

PR_ADDRTYPE
PR_DISPLAY_NAME
PR_EMAIL_ADDRESS
PR_ENTRYID
PR_SEARCH_KEY

For more information on the address properties, see About Base Address Properties.

Certificate Properties

The following are the ASN.1 certificate properties:

PR_ORIGINATOR_CERTIFICATE
PR_RECIPIENT_CERTIFICATE
PR_USER_CERTIFICATE

Content Properties

The following are the content properties in the envelope of a message:

PR_CONTENT_CONFIDENTIALITY_ALGORITHM_ID
PR_CONTENT_CORRELATOR
PR_CONTENT_IDENTIFIER
PR_CONTENT_INTEGRITY_CHECK
PR_CONTENT_LENGTH
PR_CONTENT_RETURN_REQUESTED

For more information on message content and envelopes, see Types of Message Properties.

Conversation Properties

The following are the conversation thread properties for a message:

PR_CONVERSATION_INDEX
PR_CONVERSATION_TOPIC
PR_OBSOLETED_IPMS
PR_RELATED_IPMS

For more properties related to conversation threads, see Original Author Properties and Original Represented Sender Properties.

For more information on conversations, see About Conversation Tracking.

Conversion Properties

The following are the message text conversion properties:

PR_CONVERSION_EITS
PR_CONVERSION_PROHIBITED
PR_CONVERSION_WITH_LOSS_PROHIBITED
PR_CONVERTED_EITS
PR_EXPLICIT_CONVERSION
PR_IMPLICIT_CONVERSION_PROHIBITED
PR_ORIGINAL_EITS

Criticality Properties

The following are the criticality properties for a message:

PR_IMPORTANCE
PR_ORIGINAL_SENSITIVITY
PR_PRIORITY
PR_SECURITY
PR_SENSITIVITY

For more information on criticality properties, see About Message Delivery Options.

Dialog Box Control Properties

The following are the properties for a dialog box control element:

PR_CONTROL_FLAGS
PR_CONTROL_ID
PR_CONTROL_STRUCTURE
PR_CONTROL_TYPE
PR_DELTAX
PR_DELTAY
PR_XPOS
PR_YPOS

For more information on dialog boxes, see About Display Tables.

Folder Properties

The following are the properties relating to folders:

PR_ASSOC_CONTENT_COUNT
PR_COMMON_VIEWS_ENTRYID
PR_CONTENT_COUNT
PR_CONTENT_UNREAD
PR_DEFAULT_VIEW_ENTRYID
PR_FINDER_ENTRYID
PR_FOLDER_TYPE
PR_PARENT_ENTRYID
PR_STATUS
PR_SUBFOLDERS
PR_VALID_FOLDER_MASK
PR_VIEWS_ENTRYID

For more information on folders, see Folders.

Form Properties

The following are the properties for a form:

PR_FORM_CATEGORY
PR_FORM_CATEGORY_SUB
PR_FORM_CLSID
PR_FORM_CONTACT_NAME
PR_FORM_DESIGNER_GUID
PR_FORM_DESIGNER_NAME
PR_FORM_HIDDEN
PR_FORM_HOST_MAP
PR_FORM_MESSAGE_BEHAVIOR
PR_FORM_VERSION
PR_ICON
PR_MINI_ICON

For more information on forms, see MAPI Form Architecture.

General Object Properties

The following are the properties for any general object:

PR_ACCESS
PR_ACCESS_LEVEL
PR_COMMENT
PR_MAPPING_SIGNATURE
PR_OBJECT_TYPE
PR_RECORD_KEY

Identity Properties

The following are the address properties constituting a service provider's identity as defined within a messaging system:

PR_IDENTITY_DISPLAY
PR_IDENTITY_ENTRYID
PR_IDENTITY_SEARCH_KEY

For more information on the address properties, see About Base Address Properties.

For more information on identity, see About Providing Session Identity.

IPM Properties

The following are the interpersonal message (IPM) properties:

PR_IPM_OUTBOX_ENTRYID
PR_IPM_SENTMAIL_ENTRYID
PR_IPM_SUBTREE_ENTRYID
PR_IPM_WASTEBASKET_ENTRYID
PR_SENTMAIL_ENTRYID

For more information on interpersonal message properties, see About the IPM Subtree.

Message Properties

The following are the general properties for a message:

PR_ACKNOWLEDGEMENT_MODE
PR_AUTHORIZING_USERS
PR_CORRELATE
PR_CORRELATE_MTSID
PR_DELETE_AFTER_SUBMIT
PR_DISCLOSURE_OF_RECIPIENTS
PR_DL_EXPANSION_HISTORY
PR_DL_EXPANSION_PROHIBITED
PR_INCOMPLETE_COPY
PR_LANGUAGES
PR_MESSAGE_CLASS
PR_MESSAGE_DELIVERY_ID
PR_MESSAGE_FLAGS
PR_MESSAGE_SECURITY_LABEL
PR_MESSAGE_SIZE
PR_MESSAGE_SUBMISSION_ID
PR_MESSAGE_TOKEN
PR_MSG_STATUS
PR_ORIG_MESSAGE_CLASS
PR_RECIPIENT_REASSIGNMENT_PROHIBITED
PR_REPLY_RECIPIENT_ENTRIES
PR_REPLY_RECIPIENT_NAMES
PR_REPLY_REQUESTED
PR_REQUESTED_DELIVERY_METHOD
PR_TRANSPORT_MESSAGE_HEADERS
PR_X400_CONTENT_TYPE

For more information on message properties, see Types of Message Properties.

Message Service Properties

The following are the properties for a message service:

PR_SERVICE_DELETE_FILES
PR_SERVICE_DLL_NAME
PR_SERVICE_ENTRY_NAME
PR_SERVICE_EXTRA_UIDS
PR_SERVICE_NAME
PR_SERVICE_SUPPORT_FILES
PR_SERVICE_UID

For more information on message services, see About Message Services.

Original Author Properties

The following are the address properties for the original author of a message:

PR_ORIGINAL_AUTHOR_ADDRTYPE
PR_ORIGINAL_AUTHOR_EMAIL_ADDRESS
PR_ORIGINAL_AUTHOR_ENTRYID
PR_ORIGINAL_AUTHOR_NAME
PR_ORIGINAL_AUTHOR_SEARCH_KEY

For more information on the address properties, see About Base Address Properties.

Original Messaging User Properties

The following are the address properties for a messaging user or distribution list entry copied from one address book to another:

PR_ORIGINAL_DISPLAY_NAME
PR_ORIGINAL_ENTRYID
PR_ORIGINAL_SEARCH_KEY

For more information on the address properties, see About Base Address Properties.

Original Represented Sender Properties

The following are the address properties for a messaging user being represented by the original sender:

PR_ORIGINAL_SENT_REPRESENTING_ADDRTYPE
PR_ORIGINAL_SENT_REPRESENTING_EMAIL_ADDRESS
PR_ORIGINAL_SENT_REPRESENTING_ENTRYID
PR_ORIGINAL_SENT_REPRESENTING_NAME
PR_ORIGINAL_SENT_REPRESENTING_SEARCH_KEY

For more information on the address properties, see About Base Address Properties.

Original Sender Properties

The following are the address properties for the original sender of a message:

PR_ORIGINAL_SENDER_ADDRTYPE
PR_ORIGINAL_SENDER_EMAIL_ADDRESS
PR_ORIGINAL_SENDER_ENTRYID
PR_ORIGINAL_SENDER_NAME
PR_ORIGINAL_SENDER_SEARCH_KEY

For more information on the address properties, see About Base Address Properties.

Originally Intended Recipient Properties

The following are the properties for the messaging user originally intended to be the recipient:

PR_ORIGINALLY_INTENDED_RECIP_ADDRTYPE
PR_ORIGINALLY_INTENDED_RECIP_EMAIL_ADDRESS
PR_ORIGINALLY_INTENDED_RECIP_ENTRYID
PR_ORIGINALLY_INTENDED_RECIPIENT_NAME

For more information on the address properties, see About Base Address Properties. Note that PR_ORIGINALLY_INTENDED_RECIPIENT_NAME is not one of the address properties.

For more properties related to autoforwarding, see Alternate Recipient Properties and Autoforwarding Properties.

Origination Properties

The following are the origination properties for a message:

PR_ORIGIN_CHECK
PR_ORIGINATING_MTA_CERTIFICATE
PR_ORIGINATOR_AND_DL_EXPANSION_HISTORY
PR_ORIGINATOR_DELIVERY_REPORT_REQUESTED
PR_ORIGINATOR_NON_DELIVERY_REPORT_REQUESTED
PR_ORIGINATOR_RETURN_ADDRESS

Physical Delivery Properties

The following are the physical delivery properties for a message:

PR_PHYSICAL_DELIVERY_BUREAU_FAX_DELIVERY
PR_PHYSICAL_DELIVERY_MODE
PR_PHYSICAL_DELIVERY_REPORT_REQUEST
PR_PHYSICAL_FORWARDING_ADDRESS
PR_PHYSICAL_FORWARDING_ADDRESS_REQUESTED
PR_PHYSICAL_FORWARDING_PROHIBITED
PR_REGISTERED_MAIL_TYPE

Proof of Progress Properties

The following are the properties for proof of message progress:

PR_PROOF_OF_DELIVERY
PR_PROOF_OF_DELIVERY_REQUESTED
PR_PROOF_OF_SUBMISSION
PR_PROOF_OF_SUBMISSION_REQUESTED

Provider Properties

The following are the properties for service providers:

PR_AB_PROVIDER_ID
PR_MDB_PROVIDER
PR_OWN_STORE_ENTRYID
PR_PROVIDER_DISPLAY
PR_PROVIDER_DLL_NAME
PR_PROVIDER_ORDINAL
PR_PROVIDER_UID
PR_RESOURCE_FLAGS
PR_RESOURCE_PATH
PR_RESOURCE_TYPE
PR_STATUS_CODE
PR_STATUS_STRING
PR_STORE_ENTRYID
PR_STORE_RECORD_KEY
PR_STORE_STATE
PR_STORE_SUPPORT_MASK

For more information on service providers, see Service Provider Basics.

Recipient List Properties

The following are the recipient list properties for a message:

PR_DISPLAY_BCC
PR_DISPLAY_CC
PR_DISPLAY_TO
PR_MESSAGE_CC_ME
PR_MESSAGE_RECIP_ME
PR_MESSAGE_TO_ME
PR_ORIGINAL_DISPLAY_BCC
PR_ORIGINAL_DISPLAY_CC
PR_ORIGINAL_DISPLAY_TO

Remote Transfer Properties

The following are the properties for a remote transfer:

PR_MESSAGE_DOWNLOAD_TIME
PR_REMOTE_PROGRESS
PR_REMOTE_PROGRESS_TEXT
PR_REMOTE_VALIDATE_OK

For more information on remote transfers, see Remote Transport Architecture.

Report Properties

The following are the properties for a report message:

PR_DELIVERY_POINT
PR_DISCARD_REASON
PR_DISCRETE_VALUES
PR_IPM_RETURN_REQUESTED
PR_NDR_DIAG_CODE
PR_NDR_REASON_CODE
PR_NON_RECEIPT_NOTIFICATION_REQUESTED
PR_NON_RECEIPT_REASON
PR_READ_RECEIPT_REQUESTED
PR_REPORT_TAG
PR_REPORT_TEXT
PR_REPORTING_DL_NAME
PR_REPORTING_MTA_CERTIFICATE
PR_RETURNED_IPM
PR_SUPPLEMENTARY_INFO
PR_TYPE_OF_MTS_USER

For more information on reports, see About Report Messages.

For more properties related to reports, see Report Recipient Properties.

Report Recipient Properties

The following are the address properties for the recipient of a report:

PR_READ_RECEIPT_ENTRYID
PR_READ_RECEIPT_SEARCH_KEY
PR_REPORT_ENTRYID
PR_REPORT_NAME
PR_REPORT_SEARCH_KEY

For more information on the address properties, see About Base Address Properties.

For more properties related to reports, see Report Properties.

Represented Recipient Properties

The following are the address properties for a messaging user being represented by the receiving user:

PR_RCVD_REPRESENTING_ADDRTYPE
PR_RCVD_REPRESENTING_EMAIL_ADDRESS
PR_RCVD_REPRESENTING_ENTRYID
PR_RCVD_REPRESENTING_NAME
PR_RCVD_REPRESENTING_SEARCH_KEY

For more information on the address properties, see About Base Address Properties.

Represented Sender Properties

The following are the address properties for a messaging user being represented by the sender:

PR_SENT_REPRESENTING_ADDRTYPE
PR_SENT_REPRESENTING_EMAIL_ADDRESS
PR_SENT_REPRESENTING_ENTRYID
PR_SENT_REPRESENTING_NAME
PR_SENT_REPRESENTING_SEARCH_KEY

For more information on the address properties, see About Base Address Properties.

Reserved Properties

The following properties are reserved for use by MAPI 1.0:

PR_AB_DEFAULT_DIR
PR_AB_DEFAULT_PAB
PR_AB_PROVIDERS
PR_AB_SEARCH_PATH
PR_AB_SEARCH_PATH_UPDATE
PR_PREPROCESS
PR_RECIPIENT_STATUS
PR_RTF_SYNC_BODY_COUNT
PR_RTF_SYNC_BODY_CRC
PR_RTF_SYNC_BODY_TAG
PR_RTF_SYNC_PREFIX_COUNT
PR_RTF_SYNC_TRAILING_COUNT
PR_SERVICES
PR_STORE_PROVIDERS
PR_TRANSPORT_KEY
PR_TRANSPORT_PROVIDERS

Rich Text Properties

The following are the Rich Text Format (RTF) properties:

PR_RTF_COMPRESSED
PR_RTF_IN_SYNC
PR_SEND_RICH_INFO

For more information on formatted text, see About Supporting Formatted Text.

Schedule Properties

The following are the properties for a schedule message:

PR_DELEGATION
PR_END_DATE
PR_OWNER_APPT_ID
PR_RESPONSE_REQUESTED
PR_START_DATE

For more information on schedule messages, see About Scheduling Properties.

Sender Properties

The following are the address properties for the current message sender:

PR_SENDER_ADDRTYPE
PR_SENDER_EMAIL_ADDRESS
PR_SENDER_ENTRYID
PR_SENDER_NAME
PR_SENDER_SEARCH_KEY

For more information on the address properties, see About Base Address Properties.

Specialized Usage Properties

The following are the properties with specialized usage:

PR_ANR
PR_NULL
PR_SEARCH

Subject Properties

The following are the message subject properties:

PR_NORMALIZED_SUBJECT
PR_ORIGINAL_SUBJECT
PR_SUBJECT
PR_SUBJECT_PREFIX

For more information on subject properties, see About Message Subject Properties.

Supplemental Name Properties

The following are the supplemental name properties for a messaging user:

PR_7BIT_DISPLAY_NAME
PR_MHS_COMMON_NAME
PR_TRANSMITTABLE_DISPLAY_NAME

Table Access Properties

The following are the table access properties:

PR_CONTAINER_CONTENTS
PR_CONTAINER_HIERARCHY
PR_FOLDER_ASSOCIATED_CONTENTS
PR_MESSAGE_ATTACHMENTS
PR_MESSAGE_RECIPIENTS
PR_RECEIVE_FOLDER_SETTINGS

For more information on table access, see Types of Tables.

Table Row Properties

The following are the properties for a row entry in a table:

PR_DEPTH
PR_DISPLAY_TYPE
PR_INSTANCE_KEY
PR_PARENT_DISPLAY
PR_ROW_TYPE
PR_ROWID

For more information on table rows, see Retrieving Rows.

Template Properties

The following are the template dialog box properties for an address book container:

PR_CREATE_TEMPLATES
PR_DEF_CREATE_DL
PR_DEF_CREATE_MAILUSER
PR_SELECTABLE

For more information on template identifiers, see About Types of Address Book Identifiers.

Time Properties

The following are the properties for tracking the progress of a message through time:

PR_CLIENT_SUBMIT_TIME
PR_CREATION_TIME
PR_DEFERRED_DELIVERY_TIME
PR_DELIVER_TIME
PR_EXPIRY_TIME
PR_LAST_MODIFICATION_TIME
PR_LATEST_DELIVERY_TIME
PR_MESSAGE_DELIVERY_TIME
PR_ORIGINAL_DELIVERY_TIME
PR_ORIGINAL_SUBMIT_TIME
PR_PROVIDER_SUBMIT_TIME
PR_RECEIPT_TIME
PR_REPLY_TIME
PR_REPORT_TIME

For more information on time properties on messages, see About Message Delivery Options.

Unused Properties

The following properties are not used or supported in MAPI 1.0:

PR_ATTACHMENT_X400_PARAMETERS
PR_CAPABILITIES_TABLE
PR_CONTAINER_CLASS
PR_CONTAINER_MODIFY_VERSION
PR_CONTENTS_SORT_ORDER
PR_CONVERSATION_KEY
PR_CREATION_VERSION
PR_CURRENT_VERSION
PR_DISC_VAL
PR_FILTERING_HOOKS
PR_HEADER_FOLDER_ENTRYID
PR_IPM_ID
PR_IPM_OUTBOX_SEARCH_KEY
PR_IPM_SENTMAIL_SEARCH_KEY
PR_IPM_SUBTREE_SEARCH_KEY
PR_IPM_WASTEBASKET_SEARCH_KEY
PR_MODIFY_VERSION
PR_PARENT_KEY
PR_PRIMARY_CAPABILITY
PR_TRANSPORT_STATUS
PR_X400_DEFERRED_DELIVERY_CANCEL

User Personal Information Properties

The following are the user personal information properties for a messaging user:

PR_ACCOUNT
PR_ASSISTANT
PR_ASSISTANT_TELEPHONE_NUMBER
PR_BUSINESS_FAX_NUMBER
PR_BUSINESS_TELEPHONE_NUMBER
PR_BUSINESS2_TELEPHONE_NUMBER
PR_CALLBACK_TELEPHONE_NUMBER
PR_CAR_TELEPHONE_NUMBER
PR_COMPANY_NAME
PR_COUNTRY
PR_DEPARTMENT_NAME
PR_GENERATION
PR_GIVEN_NAME
PR_GOVERNMENT_ID_NUMBER
PR_HOME_FAX_NUMBER
PR_HOME_TELEPHONE_NUMBER
PR_HOME2_TELEPHONE_NUMBER
PR_INITIALS
PR_ISDN_NUMBER
PR_KEYWORD
PR_LOCALITY
PR_LOCATION
PR_MOBILE_TELEPHONE_NUMBER
PR_OFFICE_LOCATION
PR_ORGANIZATIONAL_ID_NUMBER
PR_OTHER_TELEPHONE_NUMBER
PR_PAGER_TELEPHONE_NUMBER
PR_POST_OFFICE_BOX
PR_POSTAL_ADDRESS
PR_POSTAL_CODE
PR_PRIMARY_FAX_NUMBER
PR_PRIMARY_TELEPHONE_NUMBER
PR_RADIO_TELEPHONE_NUMBER
PR_STATE_OR_PROVINCE
PR_STREET_ADDRESS
PR_SURNAME
PR_TELEX_NUMBER
PR_TITLE

For more information on the user personal information properties, see About Messaging User Objects.

Structure Groups

The following are groups of related structures.

Address Structures

The following structures are used by clients and service providers:

ADRENTRY
ADRLIST
ADRPARM

Display Table Structures

The following structures are used by clients and service providers:

DTBLBUTTON
DTBLCHECKBOX
DTBLCOMBOBOX
DTBLDDLBX
DTBLEDIT
DTBLGROUPBOX
DTBLLABEL
DTBLLBX
DTBLMVDDLBOX
DTBLMVLISTBOX
DTBLPAGE
DTBLRADIOBUTTON
DTCTL
DTPAGE

Form Structures

The following structures are used by clients and service providers:

FORMPRINTSETUP
SMAPIFormInfoArray
SMAPIFormProp
SMAPIFormPropArray
SMAPIFormPropEnumVal
SMAPIVerb
SMAPIVerbArray
SMessageClassArray

ID Structures

The following structures are used by clients and service providers:

ENTRYID
ENTRYLIST
FLATENTRY
FLATENTRYLIST
FLATMTSIDLIST
GUID
IID
MAPINAMEID
MAPIUID
MTSID

Notification Structures

The following structures are used by clients and service providers:

ERROR_NOTIFICATION
EXTENDED_NOTIFICATION
NEWMAIL_NOTIFICATION
NOTIFICATION
NOTIFKEY
OBJECT_NOTIFICATION
STATUS_OBJECT_NOTIFICATION
TABLE_NOTIFICATION

Property Structures

The following structures are used by clients and service providers:

SPropAttrArray
SPropProblem
SPropProblemArray
SPropTagArray
SPropValue

Restriction Structures

The following structures are used by clients and service providers:

SAndRestriction
SBitMaskRestriction
SCommentRestriction
SComparePropsRestriction
SContentRestriction
SExistRestriction
SNotRestriction
SOrRestriction
SPropertyRestriction
SRestriction
SSizeRestriction
SSubRestriction

Simple MAPI Structures

The following structures are used by clients and service providers:

MapiFileDesc
MapiMessage
MapiRecipDesc

Table Structures

The following structures are used by clients and service providers:

SRow
SRowSet
SSortOrder
SSortOrderSet

TNEF Structures

The following structures are used by clients and service providers:

STnefProblem
STnefProblemArray

A

address book
A MAPI object that manages a collection of one or more types of recipient available to a client application. A recipient can be a messaging user or a distribution list. The recipients are stored in address book containers that are organized hierarchically. At a given workstation an address book includes recipients from every address book container furnished by every address book provider in the current profile. Address books implement the **IAddrBook : IUnknown** interface.

address book container
A MAPI object that contains recipient information furnished by an address book provider. Address book containers implement the **IABContainer:IMAPIContainer** interface.

address book provider
A MAPI service provider object that manages zero or more address book containers, enabling users to address messages and create recipients. An address book provider can furnish containers or templates or both. A fax address book provider, for example, typically furnishes messaging user and distribution list templates but no containers. Address book providers implement the **IABProvider:IUnknown** interface. *See also* address book container, container, message, template.

address properties
A fundamental set of five properties for every recipient that describes the recipient's e-mail address and address type, display name, entry identifier, and search key. For more information, see About Base Address Properties.

advise sink
A MAPI object that registers for and receives notifications of specific events in other objects. The notification of an event is asynchronous to the event itself, that is, it is communicated at an indeterminate time following the event, and the object generating the event does not wait for any response to the notification. An advise sink provides a callback function for events that occur in a message store, address book, or session. Advise sinks implement the **IMAPIAdviseSink:IUnknown** interface.

ambiguous name resolution
(ANR) *See* name resolution.

application form library
A form library that supplies forms to a particular workstation. Forms installed in an application form library are available to every MAPI client application on that workstation, regardless of what message store or profile is currently active. *See also* folder form library, personal form library.

ASN
(Abstract Syntax Notation) An ISO standard for encoding of human-readable symbols, such as header tags, into condensed binary form. ASN.1 corresponds to CCITT standards X.208 (for the notation) and X.209 (for the encoding rules). In turn it forms part of the specifications for the X.400 and X.500 series of standards, as well as for various other specifications.

associated contents table
A table of information associated with a folder and normally stored in the "hidden" or associated part of that folder. It typically contains forms that are installed into the folder and view descriptors that are associated with the folder. *See also* associated information, contents table, form, standard

contents table, view descriptor.

associated information
Additional information kept in a folder with a specific purpose for a client application, such as a view or a form definition. Associated information is typically not visible to the user. *See also* associated contents table.

attachment
A MAPI object that contains additional data, such as a file or OLE object, associated with a message. Attachments implement the **IAttach:IMAPIProp** interface.

attachment table
A MAPI table object that provides access to information about a message's attachments. Each row represents a message attachment.

B

bookmark
A marker that identifies a position within a table.

C

canonical form
A conventional and normalized form of any entity, such as an address, phone number, or identifier, that could exist or be presented in several different forms. An entity's canonical form is regarded as the most straightforward of its possible forms and the one that best facilitates interactions with other entities of the same type, particularly comparisons. In MAPI, canonical form is most often applied to e-mail addresses.

CCITT
The International Telegraph and Telephone Consultative Committee, an international standards committee and division of the United Nations that defines standards, such as the Electronic Data Interchange (EDI) data standard. Now called the International Telecommunications Union (ITU). CCITT stands for Comite Consultatif International Telegraphique et Telephonique, the committee's original French name. *See also* X.400, X.435, X.500.

client
See client application.

client application
A program that enables its user to interact with an underlying messaging system by calling functions or interface methods implemented by the MAPI subsystem or a service provider. These functions and methods are known collectively as the client interface. *See also* messaging application, service provider interface.

client extension
A program component that adds to the feature set of a client application, for example a handler for custom commands. Client extensions implement the **IExchExt:IUnknown** interface or one of its sibling interfaces. For more information, see Interfaces for Extending the Microsoft Exchange Client.

client interface
The set of interfaces and functions used by a client application. The client interface has four components: MAPI, Simple MAPI, Common Messaging Calls (CMC), and the OLE Messaging Library. The last three components are layers between the client application and the MAPI component, and they make calls to MAPI. The client interface can use one or more of these four components in any combination. For more information, see Selecting a Client Interface.

CMC
See Common Messaging Calls.

common dialog box
A Microsoft Windows dialog box that is provided with the MAPI SDK. A client application uses common dialog boxes to promote an appearance both consistent within the application and consistent with other Windows-based applications. *See also* dialog box.

Common Messaging Calls
(CMC) A set of functions that is based upon the standard developed by the X.400 Application Programming Interface Association (XAPIA) and that is a component of the MAPI client interface. CMC is a cross-platform function set that enables a client application to be independent of the actual messaging system, operating system, and hardware used. It provides a layer between the client application and the MAPI client interface component. *See also* Simple MAPI, OLE

Messaging Library.

Common Messaging Calls data extension
A data structure that is used to add features to Common Messaging Calls (CMC) functions and structures. Data extensions can add members to existing data structures or parameters to existing CMC functions.

compound entry identifier
An entry identifier that is created by combining the entry identifier of a message store and the entry identifier of a message within the store. Compound entry identifiers are used by a client application to open messages from a nondefault message store provider. MAPI, Simple MAPI, and Common Messaging Calls applications can all use compound entry identifiers.

configuration file
A formatted file used to define a form and to install it into a form library.using the **IMAPIFormContainer::InstallForm** method. Configuration files must have the .CFG filename extension.

connection number
A number that uniquely identifies an individual registration for a notification. A client application uses this number to cancel its registration.

container
A MAPI object that holds one or more other MAPI objects. Examples of containers are address book containers, distribution lists, and folders. Containers implement the **IMAPIContainer:IMAPIProp** interface. *See also* address book container, distribution list, folder, form container.

contents table
A MAPI table object that provides access to a summary view of entries in a folder or in an address book container. Each row represents a folder, a message, or a recipient, depending on which provider is implementing the table. *See also* associated contents table, standard contents table.

conversation thread
A series of messages that pertain to the same topic. The topic is held in the PR_CONVERSATION_TOPIC property of each message in the thread. This property and the PR_CONVERSATION_INDEX property facilitate sorting of messages by threads.

custom recipient
See one-off address.

custom recipient entry identifier
See one-off entry identifier.

custom recipient table
See one-off table.

custom recipient template entry identifier
See one-off template entry identifier.

D

data extension
See Common Messaging Calls data extension.

DBCS
(Double-Byte Character Set) A mapping of a collection of characters to a set of values each of which can be either one or two bytes. Commonly used for encoding languages requiring more than 256 characters, such as Japanese Kanji. *See also* Unicode.

default profile
Configuration information about a session's set of message services, with which a client application logs on if no other such configuration information is specified. *See also* message service, profile, session.

delegate access
The ability of one messaging user to send or receive messages on behalf of another user and to access that other user's message store. The messaging users in a delegation are commonly human users, but either the delegator or the delegate could be a software application.

Deleted Items folder
A folder within the interpersonal message (IPM) subtree that is designated to hold messages marked for deletion. Also referred to as the wastebasket. *See also* IPM subtree.

delivery report
(DR) A report message sent to a message sender that indicates the message has been delivered to a particular recipient. A delivery report can be generated by the MAPI spooler, a message transfer agent, or a transport provider. The recipient may or may not have read the message.

dialog box
A window displayed in the user interface (UI) to solicit input from the user. Its contents are controlled by a display table.

disk instance
A temporary file that contains executable files for a form if those executable files are not yet resident on the user's local disk. A disk instance is created any time a form is loaded from a form library.

display name
A character string that represents a MAPI object in the user interface (UI). Service providers assign display names to their objects by setting each object's PR_DISPLAY_NAME property.

display table
A MAPI table object that describes the layout of a dialog box. Each row in the table represents a control in the dialog box. Each column in the table holds data about one property of the control, such as its location, size, or type. A display table is used to display details of a messaging user, a distribution list, or a container. It can also be used to display and edit configuration parameters for a service provider.

distribution list
A MAPI object that identifies a grouping of message recipients. A distribution list can contain

individual messaging users and other distribution lists. Distribution lists implement the **IDistList:IMAPIContainer** interface. *See also* messaging user, recipient.

domain
See messaging domain.

E

Electronic Data Interchange
(EDI) A standard for integrating data with various native formats into a message, which has been defined by the International Telegraph and Telephone Consultative Committee (CCITT) standards body, now called the International Telecommunications Union (ITU), and is implemented in the X.435 message-handling standard.

entry identifier
A binary value that distinguishes a MAPI object from other objects of the same type and permits clients and providers to open and access the object. Part of the entry identifier represents the service provider that defines the object, and part represents the individual object. Entry identifiers allow the MAPI subsystem to determine which service provider should handle a particular object. An object's entry identifier is stored in its PR_ENTRYID property. *See also* compound entry identifier, one-off entry identifier, one-off template entry identifier, long-term entry identifier, short-term entry identifier, record key, search key.

entry point function
A function in a message service dynamic-link library (DLL) called by MAPI at logon time for configuration purposes. For more information, see About Message Service Entry Point Functions.

event
A change in an object's state that can generate a notification, such as a critical error occurring or the object being modified.

event mask
A bitmask that is used to indicate one or more types of event notifications. Advise sink objects register for specific notifications using the event mask. *See also* advise sink, notification.

extension
See client extension, Common Messaging Calls data extension.

F

folder
A message store container object that holds messages and other folders. MAPI folders implement the **IMAPIFolder:IMAPIContainer** interface. *See also* header folder, IPM subtree, receive folder, root folder, search-results folder.

folder form library
A form library that supplies forms to a particular folder. Forms installed in a folder form library are available to every user of that folder. *See also* application form library, personal form library.

foreign system
An X.400 messaging term indicating a messaging system outside the messaging domain of the local X.400 network.

form
A MAPI object that enables a user of a client application to interact with a message of a particular message class. User interaction is controlled by a form server application and is based on MAPI properties corresponding to controls on the form. Form objects implement the **IMAPIForm:IUnknown** and **IMAPIFormInfo:IMAPIProp** interfaces.

form activation
The process of starting a form server to enable a user to work with a particular form within a client application.

form container
A MAPI object that stores form definitions. A form container can be used to locate a form definition and to activate the appropriate form server. Form containers implement the **IMAPIFormContainer:IUnknown** interface. *See also* form resolution.

form library
A form container as seen through the user interface (UI). MAPI distinguishes several types of form library, including the application form library, folder form library, and personal form library.

form library provider
A MAPI service provider object that manages one or more form libraries. Like most service providers it is a dynamic-link library (DLL). Form library providers implement the **IMAPIFormMgr:IUnknown** interface. *See also* form library.

form resolution
The process of mapping the message class of a particular message to the class identifier of the form server for that message, and of locating the form for that message in the appropriate form container. Form resolution determines which piece of code should be activated to manage interaction with a particular message.

form server
An application that manages a user's interaction with a form, for example responding to menu commands. Form servers implement the **IMAPIForm:IUnknown** and **IPersistMessage:IUnknown** interfaces, and can optionally implement the **IMAPIViewAdviseSink:IUnknown** interface. *See also* form viewer.

form viewer
A client application that is capable of launching a form. *See also* form server.

fuzzy level
A value that describes the degree of exactness or looseness desired when searching through a container for a target string. Lower levels of fuzziness indicate more exact matching. Higher levels return matches for more varied forms of the target string and usually require more execution time.

G

GAL
See global address list.

gateway
Software that links two or more dissimilar messaging systems, that is, systems that use different transport protocols. A gateway provides address translation and protocol and storage conversion. *See also* messaging system, messaging domain.

global address list
(GAL) A MAPI address book container that holds recipient entries for an entire organization and is available to all e-mail users in that organization.

H

hands-off state
A condition in which a form's storage object is unavailable for either read or write access. A form is placed in the hands-off state during a save operation to permanent storage. *See also* uninitialized state, normal state, no-scribble state. For more information, see About Form States.

header folder
A special folder exposed by a remote transport provider. A client application can open the header folder to view only the message headers at the remote message store and decide for each message whether it is worth downloading for reading or can simply be deleted at the remote site.

hierarchy table
A MAPI table object that provides access to the tree organization of a MAPI container. Each row represents the hierarchical position of a container held within the parent container. For example, a hierarchy table is used by message store providers to display the folder hierarchy.

hook
See messaging hook provider.

hook provider
See messaging hook provider.

I

IID
See interface identifier.

Inbox
A folder within the interpersonal message (IPM) subtree that is designated as the default destination for incoming messages. *See also* IPM subtree, receive folder.

information service
See message service.

interface
A collection of related methods exposed by a given class of objects. There is normally a one-to-one correspondence between an object class and an interface, for example address book objects and the **IAddrBook** interface. All interfaces inherit either directly or indirectly from **IUnknown**. *See also* MAPI interface.

interface identifier
(IID) A constant that represents a particular interface and is used to request a pointer to the interface in order to call its methods. For example, the IID for the **IAddrBook** interface is IID_IAddrBook. Interface identifiers are defined using the **IID** structure, which is a specialized **GUID** structure.

International Telecommunications Union
(ITU) *See* CCITT.

interpersonal message
(IPM) A message that is sent or received by a human user rather than an application or process. Interpersonal messages have message class IPM.Note. *See also* IPC message, non-IPM message.

interprocess communication
(IPC) The exchange of data between two or more processes or applications. This exchange takes place solely through software, without any human intervention.

IPC message
A message that is sent and received by interprocess communication rather than by human users. See also interpersonal message, non-IPM message.

IPM
See interpersonal message.

IPM subtree
(interpersonal message subtree) An area of a folder hierarchy reserved by MAPI for interpersonal message (IPM) applications. Only information stored within the IPM subtree is visible to IPM users. Folders are commonly created to handle incoming, outgoing, sent, and deleted interpersonal messages. These, together with any other folders created for IPM usage, constitute a subtree of the message store which can be accessed through the message store's PR_IPM_SUBTREE_ENTRYID property. The default names for the common folders are the Inbox, Outbox, Sent Items folder, and Deleted Items folder.

K

key

A binary value associated with a MAPI object that can be compared with another binary value for the purpose of determining the relationship between the objects. Commonly used keys include the record key and the search key.

L

local message store
A message store provider that keeps its data on the user's local disk.

logon
The process by which a messaging user establishes a session with a messaging system through the MAPI subsystem. Logon typically involves the verification of the user's name and password and the user's selection of a valid profile. A user can log on to more than one messaging system at the same time.

logoff
The process by which a messaging user ends a session with the MAPI subsystem and the messaging system or systems it was logged on to.

long-term entry identifier
An entry identifier associated permanently with a MAPI object and unique in a global scope. It is stored with the object and in other appropriate places so that it can be used for multiple operations. *See also* short-term entry identifier.

M

mail user
See messaging user.

MAPI
(Messaging Application Programming Interface) A messaging architecture and a client interface component.

As a messaging architecture, MAPI enables multiple applications to interact with multiple messaging systems across a variety of hardware platforms. *See also* MAPI subsystem, messaging system.

As a client interface component, MAPI is the complete set of functions and object-oriented interfaces that forms the foundation for the MAPI subsystem's client application and service provider interfaces. In comparison with Simple MAPI, Common Messaging Calls (CMC), and the OLE Messaging Library, MAPI provides the highest performance and greatest degree of control to messaging-based applications and service providers.

MAPI interface
A set of related methods that describe the behavior of a MAPI object. All MAPI interfaces derive from the OLE base interface, **IUnknown**.

MAPI object
An object that supports the OLE Component Object Model (COM), implementing one or more interfaces derived from the **IUnknown** interface. It is implemented or used by a MAPI compliant service provider or client application, or by MAPI itself.

MAPI spooler
A MAPI process that handles the sending and receiving of messages between client applications and most messaging systems. A tightly coupled message store and transport provider does not use the spooler. *See also* client application, messaging system.

MAPI subsystem
The set of dynamic-link libraries (DLLs) provided by MAPI that enable interaction between client applications and service providers. The MAPI subsystem DLLs implement the MAPI spooler, the client interface, the service provider interface, and the support object.

message
A MAPI object containing information that can be sent to one or more recipients by means of a messaging system, or posted to a public folder. The principal parts of a message are the message content and the message envelope. Messages implement the **IMessage:IMAPIProp** interface. *See also* message text, post, recipient.

message body
See message text.

message class
A character string property that is assigned to every MAPI message, identifying the type of the message. The message class determines behavior such as selecting the appropriate receive folder and launching a form for the message. Message class strings are stored in the PR_MESSAGE_CLASS property.

message content
The portion of a message representing the information the sender wishes to communicate to each recipient. The message content includes the message text and any attachment objects. It excludes the message envelope. Messages of the same message class are expected to have similar content structure.

message envelope
The portion of a message used for routing and delivery to each recipient. The message envelope includes the address properties for each recipient, header information such as the sender and subject, and specifications for responding and exception handling. It excludes the message content. *See also* message text.

message preprocessor
A dynamic-link library (DLL) that runs on a client workstation and operates on outbound messages before they are given to any transport provider. For example, a message preprocessor might make a local copy of every outgoing message. A message preprocessor can be called selectively based on each recipient's address type or entry identifier. *See also* messaging hook provider.

message service
A group of one or more related service providers installed and configured by a single body of code. For example, CompuServe might furnish an address book provider, message store provider, and transport provider all designed to interface with the CompuServe message service.

message service table
A MAPI table object that provides access to information about every message service in the current profile, such as the service's name, service providers, and associated files.

message site
A MAPI object that handles the manipulation of form objects. Message site objects implement the **IMAPIMessageSite:IUnknown** interface.

message store
A MAPI object that contains messages and folders organized hierarchically and that implements the **IMsgStore:IMAPIProp** interface. *See also* folder, message.

message store provider
A MAPI service provider object that manages a message store, handling message distribution, organization, and storage. Message store providers implement the **IMSProvider:IUnknown** interface.

message store table
A MAPI table object that provides access to information about every message store in the current profile.

message text
The principal content of an interpersonal message, that is, a message of message class IPM.Note. Message text can optionally be used in other message classes. It is the main portion of the message content, typically displayed to each recipient as an immediate result of opening the message. It excludes any attachment objects. Also referred to as the message body.

message transfer agent
(MTA) The X.400 term for the part of a message transfer system (MTS) that interfaces with clients

of that MTS. A MAPI transport provider commonly interfaces with a message transfer agent.

message transfer system
(MTS) The X.400 term for a messaging system.

messaging application
A program that uses the MAPI client interface to pass messaging requests, such as requests to send and receive messages, to and from a messaging system. A messaging application is a type of client application.

messaging domain
A collection of interconnected messaging users that share a common addressing scheme and transport protocol. Communication with a messaging user in the same messaging domain does not involve reformatting or address translation. Communication with a messaging user in another messaging domain typically requires the use of a gateway. Also referred to as a domain or a site.

messaging hook provider
A MAPI service provider object that runs on the same machine as the MAPI spooler and performs special processing on inbound and outbound messages. Messaging hook providers implement the **ISpoolerHook:IUnknown** interface. A messaging hook provider can be called for all inbound messages, all outbound messages, or both. Also referred to as a spooler hook or a hook provider. *See also* message preprocessor.

messaging service
See message service.

messaging system
A product that enables electronic communication over a network, such as fax, CompuServe, or the Internet. Typical clients of a messaging system include an individual computer with a modem and a local area network with a gateway.

messaging transport provider
See transport provider.

messaging user
A MAPI object that describes an individual recipient of a message. Messaging users implement the **IMailUser:IMAPIProp** interface. *See also* distribution list.

MTA
See message transfer agent.

MTS
See message transfer system.

multivalued property
A property that can contain many values of the same type. Its property type has MV_FLAG set, and its property value contains multiple values of the specified property type. *See also* single-valued property. For more information, see About Property Types.

N

name resolution
The process of associating a string with a valid address for a particular messaging system. An unresolved name lacks an entry identifier. The names of all recipients for a message must be resolved before the message can be sent. *See also* recipient, resolved recipient, unresolved recipient.

name space
The set of all possible named property names within a property set. The **GUID** that is part of each property's name is unique to each property set and guarantees that no property names from different name spaces can be the same.

named property
A user-defined property whose principal designation is by a unique name rather than by a property identifier. Because of name-identifier mapping, a named property is valid in any messaging domain and any session. Every named property belongs to a property set, each member of which uses the same **GUID** for the first part of its name. For more information, see About Named Properties.

name-identifier mapping
A bidirectional, persistent one-to-one mapping between the name and the property identifier of a named property. Name-identifier mapping is provided by certain implementations of the **IMAPIProp::GetIDsFromNames** and **IMAPIProp::GetNamesFromIDs** methods. For more information, see About Support for Named Properties.

NDR
See nondelivery report.

no-scribble state
A condition in which a form's storage object is available only for read access; write access is prohibited. A form is placed in the no-scribble state during a save operation to enable the operation to finish uninterrupted. *See also* uninitialized state, normal state, hands-off state. For more information, see About Form States.

non-IPM message
(non-interpersonal message) A message that is meant to be sent or received by an application rather than by a human user, such as a notification from a workgroup scheduling application or an IPC message. *See also* interpersonal message.

nondelivery report
(NDR) A report message sent to a message sender that indicates the message could not be delivered to a particular recipient. A nondelivery report can be generated by the MAPI spooler, a message transfer agent, or a transport provider. Many situations can cause nondelivery reports to be generated, such as an inaccurate recipient address, unavailable transport providers, or an inoperative network.

nonread notification
(NRN) *See* nonread report.

nonread report
(NRN) A report message from a message store provider to a message sender that indicates the

message was not read by a particular recipient, that is, the client application did not display the message contents to the recipient before the recipient deleted the message or before a specified expiration time.

nontransmittable property

A property that is not sent along with the message it is associated with. Nontransmittable properties are typically those that apply only in the sending environment, such as PR_DELETE_AFTER_SUBMIT, or only in the receiving environment, such as PR_HASATTACH. *See also* transmittable property.

normal state

A condition in which a form's storage object is available for both read and write access. A form is typically in the normal state unless a save or close operation is in process. *See also* uninitialized state, no-scribble state, hands-off state. For more information, see About Form States.

notification

A communication advising a MAPI object of the occurrence of an event in another MAPI object. The notification is asynchronous to the event, that is, it is communicated at an indeterminate time following the event, and the object generating the event does not wait for any response to the notification. The object receiving the notification is referred to as the advise sink. Advise sinks register for notification of specific events in an object, such as critical errors, by calling that object's **Advise** method. See also registration.

O

OID
(object identifier) A value specifying the nature of an X.400 object. MAPI uses OIDs for various purposes, such as indicating the types of attached files. "Object" is a much more generalized concept in X.400 than in object-oriented programming. It can mean an algorithm, application, body part type, character set, external parameter, or message type, among other things. Also, it usually refers to a class of things, such as CRC-32, rather than one particular implementation of a 32-bit CRC (cyclic redundancy code).

OLE Messaging Library
An OLE Automation programming interface that is a component of the MAPI client interface and is used primarily by Visual Basic and Visual C and C++ client application developers. The OLE Messaging Library furnishes programmable objects, like Microsoft Excel objects and Microsoft Access objects, that make available properties and methods that can then be managed by Visual Basic (VB) and Visual Basic for Applications (VBA) programs. It provides a layer between the client application and the MAPI client interface component. *See also* Simple MAPI, Common Messaging Calls.

OLE Messaging collection
An object that contains zero or more OLE Messaging objects of the same type. The OLE Messaging Library supports two types of collections: large collections and small collections. With large collections, you can use the object's methods to get the first, next, last, and previous items within the collection. With small collections, you can access all items in the collection using an implied index.

OLE Messaging object
An object contained in the OLE Messaging Library, such as a session object, folder object, or message object, that exposes properties and methods. See the OLE Messaging Library documentation in the MAPI SDK.

one-off
British jargon for something that is used only once and not retained permanently.

one-off address
A messaging address that represents a recipient that is not in the current address book. A one-off address is created by a client application or a service provider, either by presenting a complete address string for name resolution or by using a template supplied by an address book provider. An address string without an explicit address type is assumed to be an Internet address. Also referred to as a custom recipient. *See also* one-off template entry identifier.

one-off entry identifier
An entry identifier created from a one-off address and a display name. Also referred to as a custom recipient entry identifier. For more information, see About Types of Address Book Identifiers.

one-off table
A MAPI table object that provides access to information about supported one-off templates. Also referred to as a custom recipient table.

one-off template
A template for creating a one-off address.

one-off template entry identifier
An entry identifier held in an address book provider's one-off table and used to access a one-off template for creating a one-off address. Also referred to as a custom recipient template entry identifier. For more information, see About Types of Address Book Identifiers.

Outbox
A folder within the interpersonal message (IPM) subtree that is designated to hold outgoing messages until they are sent. *See also* IPM subtree.

P

PAB
See personal address book.

personal address book
(PAB) A modifiable MAPI address book container that holds recipient entries either created by the user or copied from other address book containers. Each session can optionally designate a container to act as the PAB. Personal address book objects are commonly contained in files with the .PAB filename extension.

personal folders
A personal message store as seen through the user interface (UI).

personal form library
A form library that supplies forms to a particular client application. Forms installed in a personal form library reside in a hidden folder of the default message store and are available only to the creating client application. *See also* application form library, folder form library.

personal message store
(PST) A MAPI message store object that is created by a user and is stored in a file with the .PST filename extension.

post
The action of storing a message in a publicly known location such as a public folder. A message is posted for the purpose of making a single copy widely available for reading by human users or processing by an application.

primary identity
An object, commonly an address book entry, that represents the user of a MAPI session. The primary identity is supplied by a service provider and exposed in the properties PR_IDENTITY_ENTRYID, PR_IDENTITY_SEARCH_KEY, and PR_IDENTITY_DISPLAY.

probe
An X.400 messaging term indicating a message that has a specific message class identifier but no content. A probe is used by X.400 message senders to determine whether a message of a particular message class can be sent to a particular recipient, and if so, how.

profile
Configuration information about the set of message services for a session. Profiles are created from information stored in the MAPI configuration file, MAPISVC.INF. A profile can be modified by MAPI, a service provider, or a MAPI configuration interface. *See also* message service.

profile name
A text string that identifies a particular profile.

profile section
A MAPI object providing access to the configuration information of a service provider or message service. A profile section can contain information provided or used by MAPI, a message service, a service provider, or a client application.

profile table
A MAPI table object that provides access to the name of every profile on a particular computer.

property
A data attribute of a MAPI object that implements the **IMAPIProp:IUnknown** interface. Such an object exposes its defined properties to other objects that call **IMAPIProp**. The same property can be defined on more than one type of MAPI object. The information that designates a MAPI property is contained in its property tag. The property's actual data contents are held in its property value. *See also* named property. For more information, see Properties.

property identifier
A unique 16-bit integer value that identifies a particular property. The property identifier is contained in the high-order 16 bits of the property tag. *See also* named property. For more information, see About Property Identifiers.

property page
One section or page of the dialog box of a property sheet, accessed by selecting one of the property sheet tabs.

property set
A group of named properties specified by a **GUID**. Each named property in the set has the **GUID** as part of its name, so the **GUID** identifies a name space. A property set can be user-defined, in which case there is no restriction on the individual property names. In a MAPI-defined name space such as PS_PUBLIC_STRINGS, names must be selected that do not conflict with existing names. For more information, see About Property Names and Property Sets.

property sheet
A common dialog box that is used to display configuration information to the user and to enable the user to modify that information. A property sheet contains a collection of one or more property pages. For more information, see property page.

property tag
A 32-bit unsigned integer value that contains the unique identifier and type of a property. The high-order 16 bits contain the property identifier, and the low-order 16 bits contain the property type. *See also* named property, property value. For more information, see About Property Tags.

property type
A 16-bit integer value that describes the data type for a property and whether it can contain a single value or multiple values. The property type is contained in the low-order 16 bits of the property tag. For more information, see About Property Types.

property value
The current data contents of a property. A property value can consist of a single item of data or multiple items of data. *See also* multivalued property, single-valued property, property tag. For more information, see About Property Values.

provider
See service provider.

provider table
A MAPI table object that provides access to information about every currently loaded service provider.

proxy address
A non-native e-mail address for a recipient, that is, an address meaningful outside of the recipient's messaging domain, such as an Internet or fax address.

PST
See personal message store.

public folder
One of a set of folders made available by a message store and visible to every messaging user that logs on to that message store.

R

read flag
A flag setting, MSGFLAG_READ, that is used in the PR_MESSAGE_FLAGS property of a message to mark it as having been read. The read flag can be set in a variety of situations, including opening, printing, and copying of the message. The read flag being set does not necessarily indicate that the message has been physically read by a human user or acted upon by an application.

read notification
(RN) *See* read report.

read receipt
See read report.

read report
(RN) A report message from a message store provider to a message sender that indicates the read flag has been set for the message by a particular recipient. A read report can be sent in a variety of situations; its being sent does not guarantee the intended recipient has physically read the message.

receive folder
A MAPI folder object that is designated as the destination for any incoming message belonging to a particular message class. Typically, the receive folder is the Inbox in the interpersonal message (IPM) subtree. *See also* IPM subtree.

recipient
A user or group of users designated to receive a particular message. A recipient can be a messaging user, distribution list, or one-off address.

recipient list
A collection of recipients associated with a message. A recipient list can contain zero or more recipients of any type, in any combination, and in any order.

recipient table
A MAPI table object that provides access to information about the recipient list of a message. Each row in the table contains information about one recipient.

record key
A binary value that can be directly compared with other record keys to help find references to a MAPI object. An object's record key is stored in its PR_RECORD_KEY property. *See also* search key, entry identifier.

registration
A request on the part of a client application or MAPI to receive notifications about a specific type of event. *See also* notification.

registry provider
See form library provider.

report

A message sent to a recipient providing the status of a message that recipient had originated and sent. A report can be generated by a service provider, a messaging system component, or the MAPI subsystem. *See also* delivery report, nondelivery report, read report, nonread report.

resolution
See form resolution, name resolution.

resolved recipient
A messaging user or distribution list that has been assigned an entry identifier and an address for a particular messaging system. *See also* name resolution, unresolved recipient.

restriction
A set of criteria imposed against a table to filter the rows of the table, limiting a user's view of the table's data to only those rows that meet the criteria.

root folder
The MAPI folder object that appears at the top of a message store's folder hierarchy. Only one root folder can exist in a message store. Root folders are invisible to the user because of their inability to be moved, copied, renamed, or deleted. Like most folders, a root folder can contain messages and other folders. *See also* IPM subtree.

rule
A specification of an automated response to a particular event. Rules are commonly used to route incoming e-mail; such a rule could, for example, store an incoming message meeting certain conditions in a specified folder. A rule can be implemented by a notification callback function or a messaging hook provider. It can be applied to personal folders and to a public folder.

S

search key
A binary value that can be directly compared with other search keys to help find objects related to a MAPI object. An object's search key is stored in its PR_SEARCH_KEY property. *See also* record key, entry identifier.

search-results folder
A MAPI folder object that contains links to messages that match specified search criteria. A search-results folder can be saved across sessions if desired. It cannot contain other folders or messages, nor can a folder or message be created in it or moved into it.

section
A part of either a session profile or a form configuration file containing interrelated items of information. *See also* profile section.

secure property
A property of a profile section with a property tag in the range 0x6600 through 0x67FF. A secure property is set by a service provider, and is encrypted and invisible to a client application. To view a secure property, it is necessary to ask specifically to view it. Secure properties are typically used for credentials such as passwords.

Sent Items folder
A folder within the interpersonal message (IPM) subtree that is designated to hold copies of messages after they are sent. These copies are saved only if the message store provider supports this functionality and the user of the client application requests it. *See also* IPM subtree.

service provider
A MAPI component that allows a client application to use the services of a messaging system. A service provider is typically part of a message service and offers address book, form management, spooler hook, message store, or transport services. *See also* address book provider, form library provider, messaging hook provider, message store provider, transport provider.

service provider interface
(SPI) The set of interfaces and functions that are implemented or used by service providers. The MAPISPI.H header file contains definitions of all the interface methods and functions in the service provider interface.

session
An active connection between a client application and the MAPI subsystem. As part of the logon procedure, which initiates the session, the client application selects a profile, which identifies the available messaging operations and the service providers available to handle the operations. A session implements the **IMAPISession:IUnknown** interface.

session handle
A handle to a session initiated by a client application using Simple MAPI or Common Messaging Calls. Session handles are returned by the **MAPILogon** or **MAPILogonEx** function at logon time.

shared session
A MAPI session that can be used by multiple client applications on a given computer.

short-term entry identifier
An entry identifier of limited duration and scope. It is typically used only for a single operation and is not stored for later use. It can, however, be converted into a long-term entry identifier.

Simple Mail Transfer Protocol
(SMTP) A protocol designed for reliable and efficient electronic mail transfer that is widely used in government and education facilities and on the Internet.

Simple MAPI
A set of functions that is a component of the MAPI client interface and that enables basic messaging features to be added to a client application. It provides a layer between the client application and the MAPI client interface component. *See also* Common Messaging Calls, OLE Messaging Library.

single-valued property
A property for which the data structure contains one property value of the specified property type. *See also* multivalued property. For more information, see About Property Types.

site
See messaging domain.

SPI
See service provider interface.

spooler
See MAPI spooler.

spooler hook
See messaging hook provider.

standard contents table
A table of the information in a folder that is normally visible to the user. It typically contains messages and subfolders that belong to the folder. *See also* associated contents table, contents table, form, message, view descriptor.

status table
A MAPI table object that provides access to information about each service provider in the active profile, the MAPI spooler and MAPI subsystem, and the address book. The status table describes the state of a MAPI session.

store-and-forward messaging
A messaging model wherein messages are forwarded from a local message store to a component that saves and delivers them if possible. If delivery is not possible with the requested transport provider, this component either forwards messages to another transport provider or holds onto them until the requested transport provider is available. With MAPI, the MAPI spooler performs the store-and-forward function.

subscription
See registration.

support object

A MAPI object presented to a service provider at logon time that furnishes implementations of commonly used methods as well as certain contextual data such as the last error. For more information, see Using Support Objects.

T

table
A MAPI object that provides access to a summary view of object data in row and column format. Tables implement the **IMAPITable:IUnknown** interface. Each row represents an instance of an object; each column represents a property on that object. *See also* associated contents table, attachment table, contents table, display table, hierarchy table, message service table, message store table, one-off table, profile table, provider table, recipient table, standard contents table, status table.

Telephony Application Programming Interface
(TAPI) A set of functions that allows applications to use telephone lines for transporting data across a network without requiring information on details of how the transport process works.

template
A common dialog box that is used for creating new address book entries, such as a one-off address, of a particular type. A template can be viewed as an entire schema that specifies the properties for the dialog box and the code that relates them.

template identifier
An entry identifier for a recipient that enables its address book provider to control its behavior if it is copied into a different address book container. A template identifier can support a messaging user or a distribution list and allows the destination provider to access the original provider's implementation. The template identifier is held in the recipient's PR_TEMPLATEID property. For more information, see About Types of Address Book Identifiers.

tightly coupled
Describes a relationship between two or more providers that enables them to communicate directly with each other without using all of the interface provided by MAPI. A tightly coupled message store provider and transport provider can bypass the MAPI spooler. A tightly coupled address book provider and transport provider can pass a recipient between them using only an entry identifier and without opening a property interface.

TNEF
See Transport-Neutral Encapsulation Format.

transmittable property
A property that is sent along with the message it is associated with. Transmittable properties are those that apply in both the sending and receiving environments, such as PR_IMPORTANCE and PR_RTF_COMPRESSED. *See also* nontransmittable property.

Transport-Neutral Encapsulation Format
(TNEF) A MAPI-defined method of passing MAPI message properties that are not supported by a messaging system. The outbound transport provider invokes the method to bundle all the unsupported properties into a single binary stream. The transport provider may then have to encode the stream into a format its messaging system can handle. The stream accompanies the message through the transport process as an attachment, is decoded if necessary by the inbound transport provider, and is finally passed to TNEF to reconstitute the MAPI properties. *See also* property.

transport provider
A MAPI service provider object that is responsible for transferring messages between a message

store and an underlying messaging system that delivers the messages. Transport providers implement the **IXPProvider:IUnknown** interface.

U

Unicode
A mapping of most known language characters to a set of 16-bit values. Unicode is a worldwide encoding standard and is used exclusively by Windows NT at the system level. *See also* DBCS.

uninitialized state
A condition in which a form's storage object is available only for write access; read access is prohibited. A form is placed in the uninitialized state from the time of its creation until it has been loaded with default or supplied data. *See also* normal state, no-scribble state, hands-off state. For more information, see About Form States.

unresolved recipient
A messaging user or distribution list that has not been assigned an entry identifier and an address for a particular messaging system. *See also* name resolution, resolved recipient.

V

verb
A command appearing in the menu of a form. MAPI defines standard verbs like Compose, Send, and Close.

view
A list of parameters and operations applied to a MAPI table object, commonly the contents table of a folder. A view defines a particular way to display the data in the table, for example which columns, in which display order, and in which sort order. Some applications enable views to be saved.

view context
A MAPI object that supports commands for printing and saving a form and for navigating between forms. View contexts implement the **IMAPIViewContext:IUnknown** interface.

view descriptor
A message of a specific message class which appears in the associated contents table of a folder and specifies a view on that folder. A view descriptor encodes the information in the view for use by a client application.

W

wastebasket
See Deleted Items folder.

Windows Messaging System
(WMS) The MAPI subsystem that is implemented on any of the Microsoft Windows operating systems.

X

X.400
An international message-handling standard for connecting e-mail networks and for connecting users to e-mail networks. X.400 is published by the International Telegraph and Telephone Consultative Committee (CCITT) standards body, now called the International Telecommunications Union (ITU). The X.400 Application Programming Interface Association (XAPIA) defines programming interfaces to X.400. MAPI applications are fully interoperable with X.400 messaging applications.

X.435
An international message-handling standard that is published by the International Telegraph and Telephone Consultative Committee (CCITT) standards body, now called the International Telecommunications Union (ITU), and that implements the Electronic Data Interchange (EDI) standard for integrating data with various native formats into a message.

X.500
An international message-handling standard for directory services, published by the International Telegraph and Telephone Consultative Committee (CCITT) standards body, now called the Internal Telecommunications Union (ITU).

XAPIA
The X.400 Application Programming Interface Association, the standards-setting body for programming interfaces to X.400 components. XAPIA also defines the Common Messaging Calls inteface component.

Legal Information

Microsoft OLE Messaging Library Programmer's Reference

Introduction

The Microsoft® OLE Messaging Library exposes messaging objects for use by Microsoft® Visual Basic® and Microsoft® Visual C++® applications.

The OLE Messaging Library lets you quickly and easily add to your Visual Basic application the ability to send and receive mail messages and to interact with folders and address books. You can create programmable messaging objects, then use their properties and methods to meet the needs of your application.

When you combine messaging objects with other programmable objects exposed by Microsoft Access, Microsoft Excel, and Microsoft Word, you can quickly build custom applications that cover all your business needs. For example, with these powerful building blocks you can build a custom application that allows your users to extract information from a database, copy it to a spreadsheet for analysis, then create a report with the results and mail the report to several people.

The Microsoft OLE Messaging Library does not represent a new messaging model. It represents an additional interface to the Messaging Application Programming Interface (MAPI) model, designed to handle the most common tasks for client developers using Visual Basic and Visual C++.

This guide assumes that you are familiar with the Microsoft Visual Basic programming model. To help you use the OLE Messaging Library, this guide provides a short overview of the MAPI architecture. For complete reference information, see the *MAPI Programmer's Reference*.

The Microsoft OLE Messaging Library requires installation of MAPI and a tool that supports OLE Automation. OLE Automation is supported by the following Microsoft applications:

- Microsoft Visual Basic version 4.0
- Microsoft Visual Basic for Applications
- Microsoft Access version 2.0 or later
- Microsoft Excel version 5.0 or later
- Microsoft Project version 4.0 or later
- Microsoft Visual C++ version 1.5 or later

Note Microsoft Visual Basic version 3.0 does not support multivalued properties.

Quick Start

The following sample program demonstrates how easy it is to add messaging to your applications when you use Visual Basic or Visual Basic for Applications.

In this example, we first create a Session object and log on. We then create a Message object and set its properties to indicate the message recipient, its subject, and the content of the message. We then call the Message object's **Send** method to transmit the message.

```
' You must install the MAPI SDK, registering the
' OLE Messaging Library, to run this sample code
' This sample uses Visual Basic 3.0 error handling.
'
Function QuickStart()
Dim objSession As Object    ' Session object
Dim objMessage As Object    ' Message object
Dim objOneRecip As Object   ' Recipient object

    On Error GoTo error_olemsg

    ' create a session then log on, supplying username and password
    Set objSession = CreateObject("MAPI.Session")
    ' change the parameters to valid values for your configuration
    objSession.Logon 'profileName:="Princess Leia"

    ' create a message and fill in its properties
    Set objMessage = objSession.Outbox.Messages.Add
    objMessage.Subject = "Gift of droids"
    objMessage.Text = "Help us, Obi-wan. You are our only hope."

    ' create the recipient
    Set objOneRecip = objMessage.Recipients.Add
    objOneRecip.Name = "Obi-wan Kenobi"
    objOneRecip.Type = mapiTo
    objOneRecip.Resolve

    ' send the message and log off
    objMessage.Send showDialog:=False
    MsgBox "The message has been sent"
    objSession.Logoff
    Exit Function

error_olemsg:
    MsgBox "Error " & Str(Err) & ": " & Error$(Err)
    Exit Function

End Function
```

The OLE Messaging Library invalidates the Message object after you call its **Send** method. In this example, the developer's code logs off to end the session after sending the message, but if you were to continue the MAPI session, you could avoid potential errors by setting the Message object to **Nothing**.

About Installation

The OLE Messaging Library is installed with the MAPI Software Development Kit (SDK). The MAPI SDK setup program registers the OLE Messaging Library for subsequent use by tools that support OLE Automation.

Note In the current release, the OLE Messaging Library is installed only as part of the MAPI SDK. No separate setup program is provided.

When you use the OLE Messaging Library with a tool that supports OLE Automation, verify that the tool has referenced the OLE Messaging Library. For example, when you are using Microsoft Visual Basic version 4.0, choose the **References** command from the **Tools** menu, and select the check box for **OLE/Messaging 1.0 Object Library**.

When the OLE Messaging Library is available, the following flag is set in the file WIN.INI:

```
[Mail]
OLEMessaging=1
```

The **OLEMsgPersistenceTimeout** registry setting controls how quickly the OLE Messaging Library shuts down and unloads from memory after all messaging objects are released by client applications. On Win32® systems, the setting appears at the following registry location:

```
HKEY_LOCAL_MACHINE\Software\Microsoft\Windows Messaging Subsystem
```

For 16-bit Microsoft® Windows® systems, the **OLEMsgPersistenceTimeout** setting appears within the [MAPI] section of the WIN.INI file.

About This Guide

Overview defines the MAPI terms used in this guide and compares the OLE Messaging Library with the other MAPI programming interfaces. It then describes the design of the OLE Messaging Library, defining the objects and the collections of objects that are available to you with the OLE Messaging Library. This section also explains the relationships between these objects.

Programming Tasks offers sample Visual Basic code for many common programming tasks, such as creating and sending a message, posting a message to a public folder, navigating through folders, searching through address books, and handling errors.

Objects, Properties, and Methods contains comprehensive reference information for the properties and methods of all objects and collection objects.

The appendixes offer additional background information about OLE Automation, the technology used by the OLE Messaging Library.

The best way to learn about the OLE Messaging Library is to alternate your reading with hands-on programming. You can use the sample code that is provided with the OLE Messaging Library. For information about the sample code, see the Release Notes.

Overview

This section offers a brief introduction to MAPI and describes how the OLE Messaging Library fits into the mix of MAPI programming interfaces. It provides a short description of OLE Automation, which is the basis of the design of the OLE Messaging Library. The section concludes with a conceptual overview of the OLE Messaging Library.

Introduction to MAPI

MAPI defines a complete architecture for messaging applications. The architecture specifies several well-defined components. This allows system administrators to mix and match components to support a broad range of vendors, computing devices, and communication protocols.

The MAPI architecture can be used for e-mail, scheduling, personal information managers, bulletin boards, and online services that run on mainframes, personal computers, and hand-held computing devices. The comprehensive architectural design allows MAPI to serve as the basis for a common information exchange.

The MAPI architecture defines messaging *clients* that interact with various messaging *services* through the MAPI programming interfaces, as shown in the following diagram.

{ewc msdncd, EWGraphic, groupx841 0 /a "MAPI.BMP"}

The MAPI architecture

To use the messaging services, a client must first establish a *session*. A session is a specific connection between the client and the MAPI interface based on information provided in a *profile*. The profile contains configuration and user preference information. For example, the profile contains the names of various supporting files, the time interval to check for new messages, and other settings, such as whether to remember the user's password or to prompt the user for the password during each logon. A successful logon is required to enable the client's use of the MAPI system.

After establishing a MAPI session, the client can use the MAPI services. MAPI defines three primary services: Address Books, Message Transports, and Message Stores.

The *Address Book* service is similar to a telephone directory or Yellow Pages. The Address Book can be thought of as a permanent database that contains valid addressing information. An entry in the Address Book is called an *address entry* and consists of a display name, e-mail type, and e-mail address. The display name refers to the name, such as a person's full name, that an application displays to its users. You can provide a display name, and the Address Book service looks up the display name and provides the corresponding messaging system address.

The *Message Transport* supports communication between different devices and different underlying messaging systems.

The *Message Store* stores messages in a hierarchical structure that consists of one or more *folders*. A folder can be a *personal folder* that contains an individual's messages, or a *public folder*, similar to a bulletin board or online forum, that is accessible to many users. Each folder can contain *messages* or other folders. A message represents a communication that is sent from the sender to one or more recipients or that gets posted in a public folder. A message can include an *attachment*, a document that is attached to and sent with the message.

Several properties can be associated with the message: its subject, importance, and delivery properties, such as the time it is sent and received, and whether to notify the sender when the message is delivered and read. Some message properties identify the message as part of a *conversation*. The conversation properties allow you to group related messages and identify the sequence of comments and replies in the thread of the conversation.

The message can have one or more *recipients;* a recipient can be an individual or a *distribution list.* The distribution list can contain individuals and other distribution lists. For messages that are posted to public folders, the recipient can also be the public folder itself. Before sending a message, you can *resolve* each recipient; this means you should check each recipient against the Address Book to make sure the messaging address is valid.

MAPI Programming Interfaces

Microsoft provides several programming interfaces for MAPI, so that developers working in a wide variety of development environments can use this common message exchange.

The following figure shows the OLE Messaging Library as a layer that is built on top of MAPI. This is similar to the way that function calls to the Common Messaging Calls (CMC) interface are mapped to the underlying MAPI interfaces. It also demonstrates that the OLE Messaging Library is available to both Visual Basic/Visual Basic for Applications (VBA) and C/C++ programmers.

{ewc msdncd, EWGraphic, groupx841 1 /a "MAPI.BMP"}

The OLE Messaging Library interface layer

It is important to recognize that the OLE Messaging Library does not offer access to all of the features of MAPI. In particular, it is designed primarily for clients and is not suitable for service providers.

The following table summarizes the programming interfaces that Microsoft provides for MAPI.

Programming interface	Description
MAPI custom controls	User interface elements for Visual Basic version 3.0 developers. (Note: These will be superseded by the OLE Messaging Library.)
Simple MAPI	Functions for C/C++ client developers that allow access to the Inbox (no access to MAPI properties). Most developers should probably use either CMC or MAPI rather than Simple MAPI.
OLE Messaging Library	**Programmable messaging objects for Visual Basic/VBA and C/C++ developers.**
Common Messaging Calls (CMC)	Functions for C/C++ client developers; X.400 API Association (XAPIA) standard.
MAPI	OLE interfaces for C/C++ developers. Full access to all MAPI programming interfaces. Implemented by service providers and called by clients.

MAPI Custom Controls and the OLE Messaging Library

Although both the MAPI custom controls and the OLE Messaging Library are designed for Visual Basic programmers, they represent significantly different capabilities.

A *control* is a user interface element that enables you to display data for the user. The custom controls are usually more convenient to use or offer more specialized capabilities than the standard user interface controls, such as the list box, combo box, command button, and option button.

A programmable object may offer some user interface capabilities, but that is usually not its primary purpose. It offers the very powerful ability to interact with existing OLE objects. For a familiar example, consider the data access objects provided with Microsoft Visual Basic version 3.0 Professional Edition and subsequent versions. The data access library lets you create and use such database objects as tables and queries. As the data access library lets you use database objects, the OLE Messaging Library lets you add messaging to your applications.

The existing MAPI controls for use with Simple MAPI and Visual Basic version 3.0 will be superseded by the OLE Messaging Library.

MAPI Functions and the OLE Messaging Library

Compared to the function-call interfaces of traditional application programming interface (API) libraries, an OLE Automation object library yields faster development and code that is easier to read, debug and maintain.

The OLE Messaging Library also takes care of many programming details for you, such as memory management and keeping count of the number of objects in collections.

The following table compares a traditional function-call interface, such as CMC or Simple MAPI, with the OLE Messaging Library interface.

Task or code	**Function-call interface**	**OLE Messaging Library**
Dim mFiles() As MapiFile Dim mRecips() As MapiRecip	Requires arrays of these structures to be declared, even if the developer does not use them.	Automatically manages these structures as child objects of the parent Message object.
ReDim mRecips(0) ReDim mFiles(0)	Structures are resized by re-dimensioning arrays.	Objects are added to collections with the **Add** method.
mMessage.RecipCount = 1	Requires developer to indicate the number of recipients and attachments.	Automatically determines the number of objects in these collections.
Error handling	Each function call returns an error code.	Integrated with Visual Basic error handling during both design and run time.
Return values	Returned implicitly in the parameters of the function call.	Returned as an explicit result of a method or in object properties.

As programming tasks grow more complex, the function-call approach becomes increasingly unwieldly. In contrast, the OLE Messaging Library expands gracefully to encompass greater complexity. A well-planned, thorough framework of collections, objects, methods, and properties can neatly encompass very complex systems.

Introduction to OLE Automation

The OLE Messaging Library is based on the capabilities provided by OLE Automation. The OLE Messaging Library allows you to create instances of programmable messaging *objects* that you can reference with tools that support OLE Automation, such as Visual Basic.

For the purposes of this documentation, an *object* is an OLE Automation object: a software component that exposes its properties and methods. Such an object follows the Visual Basic programming model and lets you get properties, set properties, and call methods.

You can think of programmable objects as additions or extensions to the programmable objects that are offered as part of Visual Basic, such as forms and controls. Forms and controls expose their properties and methods so that developers can tailor these objects for the needs of their programs. In addition to the forms and controls, Visual Basic allows for the definition of a wide variety of other programmable objects by providing the **CreateObject** and **LoadObject** functions. Note that these functions do not have specialized names, like "CreateSpreadsheet" or "CreateDatabase." They are general-purpose functions that enable an open-ended number of programmable objects, including the OLE Messaging Library.

Throughout this section, Visual Basic will be used as a concrete example of a tool that supports OLE Automation, but the statements about Visual Basic apply to all such tools.

Visual Basic scripts drive the OLE Messaging Library. The scripts can also drive other libraries that support OLE Automation, such as the libraries of programmable objects provided by Microsoft Excel version 5.0 and Microsoft Access version 2.0. Visual Basic can call many different programmable object libraries and can act as the glue that holds all of these objects together.

Each library can create its own objects, set properties, and call methods. The Visual Basic program coordinates the work of all the libraries; for example, it can direct the Microsoft Access object to find data in a specific table, direct the Microsoft Excel object to run calculations using that data, and then direct OLE Messaging Library objects to create a message that contains the results of those calculations and send the message to several recipients.

OLE Messaging Library Object Design

The OLE Messaging Library is designed for ease of use and convenience. It implements the MAPI functions most used by client applications. The OLE Messaging Library is not designed for development of service providers. (For more information about service providers, see Introduction to MAPI.)

This section of the guide describes the design of the OLE Messaging Library.

Note This OLE Messaging Library design does not represent a one-to-one correspondence to MAPI objects. The description of the OLE Messaging Library object design does not always apply to the MAPI programming interface.

The OLE Messaging Library defines the following objects:

- AddressEntry
- Attachment
- Attachments collection
- Field
- Fields collection
- Folder
- Folders collection
- InfoStore
- InfoStores collection
- Message
- Messages collection
- Recipient
- Recipients collection
- Session

The objects supported in the OLE Messaging Library can be grouped into three categories:

- **High-level objects**
- **Child objects** that are created automatically when the high-level objects are created
- **Collections**, or groups of objects of the same type

High-Level Objects

The high-level objects include the Session, Folder, and Message objects. Other objects are accessible only from these high-level objects.

C/C++ programmers can access all high-level objects. Visual Basic programmers can create only the Session object, using the Visual Basic **CreateObject** function with the string "MAPI.Session."

In your Visual Basic application, you must usually use code of the following form to create the high-level session object:

```
Dim objSession As Object
Set objSession = CreateObject("MAPI.Session")
```

C/C++ programmers use the globally unique identifiers (GUIDs) for these objects, defined in the type library for the OLE Messaging Library. The following code fragment demonstrates how to create a Session object and call its **Logon** method:

```
// create a Session object and log on using IDispatch interface
// to the OLE Messaging library
#include <ole2.h>
#include <stdio.h>
#include <stdlib.h>  // for exit
#define dispidM_Logon 119  // get constants for all props, methods
// allows you to save cost of GetIdsFromNames calls
// can generate yourself by calling GetIdsFromNames for all
// properties and methods
// GUID values for Session defined in the type library
static const CLSID GUID_OM_SESSION =
{0x3FA7DEB3, 0x6438, 0x101B, {0xAC, 0xC1, 0, 0xAA, 0, 0x42, 0x33, 0x26}};
void main(void)
{
HRESULT hr;

/* interface pointers */
LPUNKNOWN punk = NULL; // IUnknown *; used to get IDispatch *
DISPPARAMS dispparamsNoArgs = {NULL, NULL, 0, 0};
VARIANT varRetVal;
IDispatch * pSession;

    //Initialize OLE.
    hr = OleInitialize(NULL);
     printf("OleInitialize returned 0x%lx\n", hr);
    VariantInit(&varRetVal);
// Create an instance of the OLE Messaging Library Session object
// Ask for its IDispatch interface.
    hr = CoCreateInstance(GUID_OM_SESSION,
                          NULL,
                          CLSCTX_SERVER,
                          IID_IUnknown,
                          (void FAR* FAR*)&punk);
    printf("CoCreateInstance returned 0x%lx\n", hr);
    if (S_OK != hr)
        exit(1);
    hr = punk->QueryInterface(IID_IDispatch, (void FAR* FAR*)&pSession);
    punk->Release();       // no longer needed; release it
    printf("QI for IID_IDispatch returned 0x%lx\n", hr);
    if (S_OK != hr)
        exit(1);
// Logon using the session object; call its Logon method
    hr = pSession->Invoke(dispidM_Logon, // value = 119
                          IID_NULL,
                          LOCALE_SYSTEM_DEFAULT,
                          DISPATCH_METHOD,
                          &dispparamsNoArgs,
                          &varRetVal,
                          NULL,
                          NULL);
    printf("Invoke returned 0x%lx\n", hr);
    printf("Logon call returned 0x%lx\n", varRetVal.lVal);
// do other things here...
// when done, release the Session dispatch object and shut down OLE
    pSession->Release();
    OleUninitialize();
```

The following table lists the GUIDs for the objects accessible to C/C++ programmers:

OLE Messaging Library Object	**GUID**
Session object	3FA7DEB36438101BACC100AA00423326
Folder object	3FA7DEB56438101BACC100AA00423326
Message object	3FA7DEB46438101BACC100AA00423326

High-Level Objects and Child Objects

All OLE Messaging Library objects can be considered as relative to a Session object. The following diagram shows the logical hierarchy for the OLE Messaging Library.

```
Session
    Folder
        Folders Collection
            Folder...
        Messages Collection
            Message
                Recipients Collection
                    Recipient
                        AddressEntry
                            Fields Collection
                                Field
                Fields Collection
                    Field
                Attachments Collection
                    Attachment
        Fields Collection
            Field
    InfoStores Collection
        InfoStore
            Folder...
```

In addition to the hierarchy of objects, each object has properties and methods. The hierarchy is important because it determines the correct syntax to use in your Visual Basic applications. In your Visual Basic code, the relationship between a parent object and a child object is denoted by the left-to-right sequence of the objects in the Visual Basic statement.

Object Collections

A *collection* is a group of objects of the same type. In the OLE Messaging Library, the name of the collection takes the plural form of the individual OLE Messaging Library object. For example, the Messages collection is the name of the collection that contains Message objects. The OLE Messaging Library supports the following collections:

- Attachments
- Fields
- Folders
- InfoStores
- Messages
- Recipients

There are two kinds of collections: small collections and large collections.

For small collections, the OLE Messaging Library maintains a count of the number of objects in the collection. The Attachments, InfoStores, Recipients, and Fields collections can be characterized this way. You can access individual items using an index into the collection. You can also add and delete items from the collection (except for the InfoStores collection, which is read-only for the OLE Messaging Library).

Small collections, with a known number of member objects, have the **Item** property, the **Count** property, and an implied temporary **Index** property, assigned by the OLE Messaging Library. **Index** properties are valid only during the current MAPI session and can change as your application adds and deletes objects. The first **Index** value is 1. The Visual Basic **For Each** statement operates only on small collections.

For example, in an Attachments collection with three Attachment objects, the first attachment is referred to as Attachments.Item(1), the second as Attachments.Item(2), and the third as Attachments.Item(3). If your application deletes the second attachment, the third attachment becomes the second and Attachments.Item(3) has the value **Nothing**. The **Count** property is always equal to the highest **Index** in the collection.

Other applications can add and delete objects while your application is running. The **Count** property is not updated until you re-create or refresh the collection. For example, you call the Message object's **Update** method to refresh the count in its Attachments and Recipients collections.

For large collections, the OLE Messaging Library does not maintain a count of the number of objects. The Messages and Folders collections are characterized as large collections. Instead of keeping a count, the collections support methods that let you get the first, next, previous, and last item in the collection. The Visual Basic **For Each** statement does not operate on large collections.

For large collections, with an unknown number of member objects, MAPI assigns a permanent, unique string **ID** property when the individual member object is created. These identifiers do not change from one MAPI session to another. You can call the Session object's **GetFolder** or **GetMessage** methods, specifying the unique identifier, to obtain the individual Folder or Message objects. You can also use the **GetFirst** and **GetNext** methods to move from one object to the next in these collections.

Note When you want to use a collection, create a variable that refers to that collection to ensure correct operation of the **GetFirst**, **GetNext**, **GetPrevious**, and **GetLast** methods.

For example, the following two code fragments are not equivalent:

```
' sample 1:  the collection returns the same message both times!
Set objMessage = objInBox.Messages.GetFirst
...
Set objMessage = objInBox.Messages.GetNext
```

```
' sample 2:  use an explicit variable to refer to the collection;
'   the Get methods return two different messages
Set objMsgColl = objSession.Inbox.Messages
Set objMessage = objMsgColl.GetFirst
...
Set objMessage = objMsgColl.GetNext
```

Code sample 1 causes the OLE Messaging Library to create a new Messages collection and to reinitialize the value of the collection's "current message." The **GetFirst** and **GetNext** method calls return the same value for *objMessage*.

Code sample 2 uses the existing collection *objMsgColl*, so the **GetFirst** and **GetNext** calls function as expected for collections with more than one item.

The collections in the OLE Messaging Library are specifically designed for messaging applications. The definition of collections in this document may differ slightly from the definitions of collections in the OLE programming documentation. Where there are differences, the description of the operation of the OLE Messaging Library supersedes the other documentation.

Programming Tasks

This section describes some of the common programming tasks you can perform with the OLE Messaging Library. The first task your application must do is obtain a valid Session object as described in Starting a Session with MAPI.

Category	Programming tasks
General Programming Tasks	Starting a Session with MAPI Handling Errors Improving Application Performance Viewing MAPI Properties
Working with Messages	Adding Attachments to a Message Customizing a Folder or Message Checking for New Mail Creating and Sending a Message Deleting a Message Making Sure the Message Gets There Reading a Message from the Inbox Searching for a Message Securing Messages
Working with Addresses	Changing an Existing Address Entry Selecting Recipients from the Address Book Using Addresses
Working with Folders	Accessing Folders Copying a Message to Another Folder Customizing a Folder or Message Moving a Message to Another Folder Searching for a Folder
Working with Public Folders	Posting Messages to a Public Folder Working with Conversations

Note that you cannot create new distribution lists, new folders, or new address book entries using the OLE Messaging Library. However, you can use other applications or tools, such as the Microsoft Exchange Client, to create these objects. After you create the objects, you can then access them using the OLE Messaging Library.

The following table summarizes the programming procedures that you must use to perform these tasks. Note that all tasks require a Session object and successful logon.

Programming task	Procedure
Accessing Folders	1. Access the Folder object's **Folders** property to obtain its collection of subfolders. 2. Use the Folders collection's **GetFirst**, **GetNext**, **GetPrevious**, and **GetLast** methods to navigate

	through the subfolders.
Adding Attachments to a Message	1. Create or obtain the Message object that is to include the attachment. 2. Call the Message object's Attachments collection's **Add** method.
Changing an Existing Address Entry	1. Obtain a valid AddressEntry object. 2. Update the **Name**, **Type**, or **Address** property. 3. Call the **Update** method.
Checking for New Mail	Maintain a count of the number of messages in the Inbox folder that have the **Unread** property set to TRUE. - or - Sort messages by time and count messages received after a specified time.
Copying a Message to Another Folder	1. Obtain the source message that you want to copy. 2. Call the destination folder's Messages collection's **Add** method, supplying the source message properties as parameters. 3. Copy the source Message object's **Sender** and **Recipients** properties to the new Message object. 4. Call the new Message object's **Update** method.
Creating and Sending a Message	1. Call the Messages collection's **Add** method to create a Message object. 2. Set the Message object's **Text**, **Subject**, and other message properties. 3. Call the message's Recipients collection's **Add** method to add a recipient. 4. Set the Recipient object's **Name**, **Address**, or **AddressEntry** property. 5. Call the Recipient object's **Resolve** method to validate the address information. 6. Call the Message object's **Send** method.
Customizing a Folder or	1. Create or obtain the Folder or

Message	Message object that will have the custom properties. 2. Call the object's Fields collection's **Add** method.
Deleting a Message	1. Select the message you want to delete. 2. Call the Message object's **Delete** method.
Handling Errors	Use the Microsoft Visual Basic **On Error Goto** statement to add exception-handling code just as you would in any Visual Basic application.
Improving Application Performance	Each dot in a Visual Basic statement directs the OLE Messaging Library to create a temporary internal object. Use explicit variables when you reuse messaging objects.
Making Sure the Message Gets There	1. Set the Message object's **DeliveryReceipt** and/or **ReadReceipt** property to TRUE. 2. Call the Message object's **Send** method.
Moving a Message to Another Folder	Use the same procedure as Copying A Message To Another Folder, and then delete the original source message from its folder.
Posting Messages to a Public Folder	1. Use a procedure similar to Creating and Sending a Message, where you specify the name of the public folder as the Recipient name. - or - 1. Call the public folder's Messages collection's **Add** method to create a Message object. 2. Set the Message object's **Text**, **Subject**, **ConversationSubject**, **ConversationIndex**, **TimeSent**, **TimeReceived**, and other message properties. 3. Set the Message object's **Unread**, **Submitted,** and **Sent** properties to TRUE. 4. Call the Message object's **Send** or **Update** method to post the message.
Reading a Message from the Inbox	1. Call the session's Inbox folder's **GetFirst, GetNext, GetPrevious,** and **GetLast** methods to obtain a Message object. 2. Obtain the Message object's **Text**

	property.
Searching for a Folder	Use the Session object's **GetFolder** method to obtain the folder from its known ID value. - or - Call the Folders collection's **Get** methods to get individual folder objects. You can then compare properties of each folder with the desired properties.
Searching for a Message	Use the Session object's **GetMessage** method to obtain the message from its known ID value. - or - Call the Messages collection's **Get** methods to get individual message objects. You can then compare properties of each message with the desired properties.
Securing Messages	1. Set the Message object's **Encrypted** and/or **Signed** properties to TRUE. 2. Perform processing on the message's **Text** property to encrypt or sign the message. 3. Call the Message object's **Send** method.
Selecting Recipients from the Address Book	1. Call the session's **AddressBook** method to use the MAPI AddressBook dialog. 2. Set a Recipients collection object to the Recipients collection returned by the **AddressBook** dialog. 3. Use that Recipients collection or copy individual recipients from it.
Starting a Session with MAPI	1. Create or obtain a Session object. 2. Call the Session object's **Logon** method.
Using Addresses	1. Set the message's Recipient object's **Address** property to a full address. 2. Call the Recipient object's **Resolve** method.
Viewing MAPI Properties	Specify the **Fields** item with a MAPI property tag.
Working with Conversations	1. Set the message's **ConversationTopic** property. 2. Set the message's **ConversationIndex** property. 3. Send the message by calling the

Send method.- or -

3. Post the message in the public folder by setting the **Submitted** property to TRUE.

It is important to understand the hierarchy of the OLE Messaging Library objects, because the hierarchical relationships between objects determine the correct syntax of Visual Basic statements. The relative positions of these objects in the hierarchy indicate how the objects appear from left to right in a Visual Basic statement.

In the sample code that appears in this guide, individual statements are often broken across several lines. The underscore character (_) appears as a line continuation character, indicating that the statement is continued on the next line. This convention is used in an attempt to make the material easy to read.

All sample code that appears in this guide is also available in the form of a Microsoft Excel version 5.0 spreadsheet that contains Visual Basic for Applications modules. For information about the spreadsheet that contains the sample code, see the Release Notes.

Accessing Folders

Folders can be organized in a hierarchy, allowing you to access folders within folders. A child folder within a parent folder is also called a *subfolder*. Subfolders appear within the parent Folder object's Folders collection.

You cannot use the OLE Messaging Library to create new folders. However, after another application, such as the Microsoft Exchange Client, has created a folder, you can use the OLE Messaging Library to access the folder.

There are two general approaches for accessing folders:

- Obtaining the folder directly by calling the Session object's **GetFolder** method.
- Navigating folders using the Folders collection's **Get** methods.

To obtain the folder directly using the **GetFolder** method, you must have the folder's identifier. In the following example, the identifier is stored in the variable *strFolderID*:

```
Function Session_GetFolder()
    On Error GoTo error_olemsg

    If objSession Is Nothing Then
        MsgBox "No active session, must log on"
        Exit Function
    End If
    If strFolderID = "" Then
        MsgBox ("Must first set folder ID variable; see Folder->ID")
        Exit Function
    End If
    Set objFolder = objSession.GetFolder(strFolderID)
    'equivalent to:
    ' Set objFolder = objSession.GetFolder(folderID:=strFolderID)
    If objFolder Is Nothing Then
        Set objMessages = Nothing
        MsgBox "Unable to retrieve folder with specified ID"
        Exit Function
    End If
    MsgBox "Folder set to " & objFolder.Name
    Set objMessages = objFolder.Messages
    Exit Function

error_olemsg:
    MsgBox "Error " & Str(Err) & ": " & Error$(Err)
    Set objFolder = Nothing
    Set objMessages = Nothing
    MsgBox "Folder is no longer available; no active folder"
    Exit Function
End Function
```

To navigate through the hierarchy of folders, start with a known or available folder, such as the Inbox or Outbox, and examine its Folders collection. You can use the Folders collection's **GetFirst** and **GetNext** methods to get each folder in the collection. When you have a subfolder, you can examine its properties, such as its name, to see whether it is the desired folder. The following sample code navigates through all existing subfolders of the Inbox:

```
Function TestDrv_Util_ListFolders()
    On Error GoTo error_olemsg
    If objFolder Is Nothing Then
        MsgBox "must select a folder object; see Session menu"
        Exit Function
    End If
    If 2 = objFolder.Class Then  ' verify Folder object
        x = Util_ListFolders(objFolder)  ' use current global folder
    End If
    Exit Function

error_olemsg:
    MsgBox "Error " & Str(Err) & ": " & Error$(Err)
    Resume Next

End Function

' Function: Util_ListFolders
' Purpose: Recursively list all folders below the current folder
' See documentation topic: Folders collection
Function Util_ListFolders(objParentFolder As Object)
Dim objFoldersColl As Object ' the child Folders collection
Dim objOneSubfolder As Object 'a single Folder object
    On Error GoTo error_olemsg
    If Not objParentFolder Is Nothing Then
        MsgBox ("Folder name = " & objParentFolder.Name)
        Set objFoldersColl = objParentFolder.Folders
        If Not objFoldersColl Is Nothing Then ' loop through all
            Set objOneSubfolder = objFoldersColl.GetFirst
            While Not objOneSubfolder Is Nothing
                x = Util_ListFolders(objOneSubfolder)
                Set objOneSubfolder = objFoldersColl.GetNext
            Wend
        End If
    End If
    Exit Function

error_olemsg:
    MsgBox "Error " & Str(Err) & ": " & Error$(Err)
    Resume Next
End Function
```

See Also

Searching for a Folder

Adding Attachments to a Message

You can add one or more attachments to a message. You add each attachment to the Attachments collection, using the Message object's **Attachments** property. The relationship between the Message object and an attachment is shown here.

Message object
- Attachments collection
  - Attachment object
    - **Type** property
    - **Source** property

The OLE Messaging Library supports three different kinds of attachments: files, links to files, and OLE objects. An attachment's type is specified by its **Type** property. To add an attachment, use the related Attachment object property or method appropriate for that type, as shown in the following table.

Attachment type	Related attachment object property or method
mapiFileData	**ReadFromFile** method
mapiFileLink	**Source** property
mapiOLE	**ReadFromFile** method

The following example demonstrates inserting a file as an attachment. This example assumes that the application has already created the Session object variable *objSession* and successfully called the Session object's **Logon** method, as described in Starting a Session with MAPI.

```
' Function: Attachments_Add_Data
' Purpose: Demonstrate the Add method for type = mapiFileData
' See documentation topic: Adding Attachments To A Message,
'    Add method (Attachments collection)
Function Attachments_Add_Data()
Dim objMessage As Object  ' local
Dim objRecip As Object    ' local

    On Error GoTo error_olemsg
    If objSession Is Nothing Then
        MsgBox ("must first log on; use Session->Logon")
        Exit Function
    End If
    Set objMessage = objSession.Outbox.Messages.Add
    If objMessage Is Nothing Then
        MsgBox "could not create a new message in the Outbox"
        Exit Function
    End If
    With objMessage  ' message object
        .Subject = "attachment test"
        .Text = "Have a nice day."
        Set objAttach = .Attachments.Add         ' add an attachment
        If objAttach Is Nothing Then
            MsgBox "Unable to create new Attachment object"
            Exit Function
        End If
        With objAttach
            .Type = mapiFileData
            .Position = 0  ' Some apps render at start of message
            .Name = "c:\smiley.bmp"
            .ReadFromFile "c:\smiley.bmp"
        End With
        objAttach.Name = "smiley.bmp"
    .Update   ' update the message
    End With
    MsgBox "Created message, added 1 mapiFileData attachment, updated"
    Exit Function

error_olemsg:
    MsgBox "Error " & Str(Err) & ": " & Error$(Err)
    Resume Next

End Function
```

Note that setting a position value within the message can cause some viewers to overwrite the character that appears at that position in the message. You can insert the attachment at various places in the message text.

```
' objMessage and objAttach as defined above
    objMessage.Text = " " & objMessage.Text ' add space for attachment
    objAttach.position = 1
    objMessage.Update
```

The OLE Messaging Library does not actually place the attachment within the message; that is the responsibility of the messaging client application. However, to avoid these display problems with some viewers, you can specify a position value that indicates either the start or the end of the message. You can also use the value -1, which indicates that the attachment should be sent with the message, but should not be rendered by the application.

To insert an attachment of type **mapiOLE**, use code similar to the **mapiFileData** type example. Set the attachment type to **mapiOLE** and make sure that the specified file is a valid OLE *docfile* (a file saved by an OLE-aware application such as Microsoft Word version 6.0 that uses the OLE interfaces **IStorage** and **IStream**).

To add an attachment of type **mapiFileLink**, set the **Type** property to **mapiFileLink** and set the **Source** property to the file name. The following sample code demonstrates this type of attachment:

```
' Function: Attachments_Add
' Purpose: Demonstrate the Add method for type = mapiFileLink
' See documentation topic: Adding Attachments To A Message,
'     Add method (Attachments collection)
Function Attachments_Add()
    On Error GoTo error_olemsg

    If objAttachColl Is Nothing Then
        MsgBox "must first select an attachments collection"
        Exit Function
    End If
    Set objAttach = objAttachColl.Add         ' add an attachment
    With objAttach
        .Type = mapiFileLink
        .Position = 0   ' place at start of message
        .Source = "\\server\bitmaps\honey.bmp"  ' modify UNC name
    End With
    ' must update the message to save the new info
    objOneMsg.Update   ' update the message
    MsgBox "Added an attachment of type mapiFileLink"
    Exit Function

error_olemsg:
    MsgBox "Error " & Str(Err) & ": " & Error$(Err)
    Resume Next

End Function
```

See Also

Position Property (Attachment Object)

Creating and Sending a Message

Changing an Existing Address Entry

The OLE Messaging Library lets you change existing address entries in the personal address book.

To change an existing address entry

1. Select the AddressEntry object to modify. You can obtain the AddressEntry object in several ways, including the following:
   - Call the Session object's **AddressBook** method to let the user select recipients. The method returns a Recipients collection. Examine each Recipient object's **AddressEntry** property to obtain its child AddressEntry object.
   - Use the Message object's **Sender** property to obtain an AddressEntry object.
   - Examine a Message object's Recipients collection to obtain an individual Recipient object, then use its **AddressEntry** property to obtain its child AddressEntry object.
2. Change individual properties of the AddressEntry object, such as the **Name**, **Address**, or **Type** property.
3. Call the AddressEntry object's **Update** method.

Note that the OLE Messaging Library only supports changes to the personal address book. It does not support changes to the global address list.

The following sample code demonstrates this procedure:

```
' Function: AddressEntry_Update
' Purpose: Demonstrate the Update method
'    (Note: OLE Messaging Library only affects the PAB)
' See documentation topic: Update method AddressEntry object
Function AddressEntry_Update()
Dim objRecipColl As Object   ' Recipients collection
Dim objNewRecip As Object    ' New recipient

    On Error GoTo error_olemsg
    If objSession Is Nothing Then
        MsgBox "must log on first"
        Exit Function
    End If
    Set objRecipColl = objSession.AddressBook  ' let user select
    If objRecipColl Is Nothing Then
        MsgBox "must select someone from the address book"
        Exit Function
    End If
    Set objNewRecip = objRecipColl.Item(1)
    With objNewRecip.AddressEntry
        .Name = .Name & " the Magnificent"
        .Type = "X.500"  ' you can update the type, too...
        .Update
    End With
    MsgBox "Updated an address entry name: " & _
            objNewRecip.AddressEntry.Name
    Exit Function

error_olemsg:
    MsgBox "Error " & Str(Err) & ": " & Error$(Err)
    Resume Next

End Function
```

See Also

Using Addresses

Selecting Recipients from the Address Book

Checking for New Mail

The Inbox contains new messages. When users refer to new messages, they can indicate messages that arrive after the last time that they read messages, or they can indicate all unread messages. Depending on the needs of your application users, your applications can check various message properties to determine whether there is new mail.

The following sample code tracks new messages by checking for messages in the Inbox with the **Unread** property value TRUE:

```
' Function: Util_CountUnread
' Purpose:  Count unread messages in a folder
' See documentation topic: Checking For New Mail;
'    Unread property (Message)
Function Util_CountUnread()
Dim cUnread As Integer         ' counter

    On Error GoTo error_olemsg
    If objMessages Is Nothing Then
        MsgBox "must select a messages collection"
        Exit Function
    End If
    Set objMessage = objMessages.GetFirst
    cUnread = 0
    While Not objMessage Is Nothing ' loop through all messages
        If True = objMessage.Unread Then
            cUnread = cUnread + 1
        End If
        Set objMessage = objMessages.GetNext
    Wend
    MsgBox "Number of unread messages = " & cUnread
    Exit Function

error_olemsg:
    MsgBox "Error " & Str(Err) & ": " & Error$(Err)
    Resume Next

End Function
```

You can also check for new messages by counting the messages received after a specified time. For example, your application can maintain a variable that represents the time of the latest message received, based on the Message object's **TimeReceived** property. The application can periodically check for all messages with a **TimeReceived** value greater than the saved value. When new messages are found, the application updates its count of new messages and updates the saved value.

See Also

TimeReceived Property (Message Object)

Reading a Message from the Inbox

Copying a Message to Another Folder

The procedure documented in this section demonstrates a way to copy message properties using the Message object's **Add** method that is supported in the OLE Messaging Library.

Note Using OLE Messaging Library version 1.0, the Message object's **Sender** property and other read-only properties of the Message object are not preserved using the procedure in this section. To preserve these properties using this procedure, you must append their text fields to read-write properties, such as the Message object's **Text** property.

To copy a message from one folder to another folder using the OLE Messaging Library

1. Obtain the source message that you want to copy.
2. Call the destination folder's Messages collection's **Add** method, supplying the source message properties as parameters.

The hierarchy of objects is as follows:

```
Session object
    Folder object (for Inbox, Outbox)
        Messages collection
            Message object
    InfoStores collection
        InfoStore object
            Folder object
                Messages collection
                    Message object
```

To obtain the source message that you want to copy, first obtain its folder, then obtain the message within the folder's Messages collection. For more information about finding messages, see Searching for a Message.

To obtain the destination folder, you can use the following approaches:

- Use the Folders collection's **Get** methods to search for a specific folder.
- Call the Session object's **GetFolder** method with a string parameter that specifies the *FolderID*, a unique identifier for that folder.

For more information about finding folders, see Searching for a Folder.

The following example demonstrates how to copy the first message that appears in the Inbox folder. The message is copied to the Outbox, but could as easily be copied to any folder with a known identifier and therefore accessible using the Session object's **GetFolder** method. This example assumes that the application has already created the Session object variable *objSession* and successfully called the Session object's **Logon** method, as described in Starting a Session with MAPI.

```
'/********************************/
' Function: Util_CopyMessage
' Purpose: Utility functions that demonstrates code to copy a message
' See documentation topic: Copying A Message To Another Folder
Function Util_CopyMessage()
' obtain the source message to copy
' for this sample, just use the first message in the Inbox
' assume session object already created
Dim objDestFolder As Object    ' destination folder
Dim objCopyMsg As Object       ' new message that is the copy
Dim strRecipName As String     ' copy of recipient name from original message
Dim i As Integer              ' loop counter

    On Error GoTo error_olemsg
    If objOneMsg Is Nothing Then
        MsgBox "must first select message"
        Exit Function
    End If
    If objFolder Is Nothing Then
        MsgBox "must first select a folder"
        Exit Function
    End If
    strFolderID = objFolder.Id
    ' Copy to the destination folder
    Set objDestFolder = objSession.GetFolder(strFolderID)
    If objDestFolder Is Nothing Then
        MsgBox "Unable to create destination folder for ID " _
                & strFolderID
        Exit Function
    Else
        MsgBox "Copying message to destination folder " _
                & objDestFolder.Name
    End If
    Set objCopyMsg = objDestFolder.Messages.Add _
        (Subject:=objOneMsg.Subject, _
        Text:=objOneMsg.Text, _
        Type:=objOneMsg.Type, _
        importance:=objOneMsg.importance)
    If objCopyMsg Is Nothing Then
        MsgBox "Unable to create new message in destination folder"
        Exit Function
    End If
    ' copy all the recipients
    For i = 1 To objOneMsg.Recipients.Count Step 1
        strRecipName = objOneMsg.Recipients.Item(i).Name
        If strRecipName <> "" Then
            Set objOneRecip = objCopyMsg.Recipients.Add
            If objOneRecip Is Nothing Then
                MsgBox "unable to create recipient in message copy"
                Exit Function
            End If
            objOneRecip.Name = strRecipName
        End If
    Next i
    ' copy other properties; a few listed here as an example
    objCopyMsg.Sent = objOneMsg.Sent
    objCopyMsg.Text = objOneMsg.Text
    objCopyMsg.Unread = objOneMsg.Unread
    objCopyMsg.Update
    ' if *moving* a message to another folder, delete the original msg:
    '     objOneMsg.Delete
    ' move operation implies that the original message is removed
    Exit Function

error_olemsg:
    MsgBox "Error " & Str(Err) & ": " & Error$(Err)
    Exit Function     ' so many steps to succeed; just exit on error

End Function
```

Note that this procedure does not preserve all message properties. Also note that you cannot copy from the **From** property.

See Also

Moving a Message to Another Folder

Creating and Sending a Message

Creating and sending a message is easy when you use the OLE Messaging Library.

To create and send a message

1. Establish a session with the MAPI system.
2. Call the Messages collection's **Add** method to create a Message object.
3. Supply values for the Message object's **Subject** and **Text** properties.
4. Call the Recipients collection's **Add** method for each recipient.
5. Call the Message object's **Send** method.

The following sample demonstrates each of these steps for a message sent to a single recipient:

```
' This sample also appears as the "Quick Start" sample in the section
"Overview"
Function QuickStart()
Dim objSession As Object    ' Session object
Dim objMessage As Object    ' Message object
Dim objOneRecip As Object   ' Recipient object

    On Error GoTo error_olemsg

    ' create a session then log on, supplying username and password
    Set objSession = CreateObject("MAPI.Session")
    ' change the parameters to valid values for your configuration
    objSession.Logon 'profileName:="Princess Leia", _
                 'profilePassword:="go_rebels"

    ' create a message and fill in its properties
    Set objMessage = objSession.Outbox.Messages.Add
    objMessage.Subject = "Gift of droids"
    objMessage.Text = "Help us, Obi-wan. You are our only hope."

    ' create the recipient
    Set objOneRecip = objMessage.Recipients.Add
    objOneRecip.Name = "Obi-wan Kenobi"
    objOneRecip.Type = mapiTo
    objOneRecip.Resolve

    ' send the message and log off
    objMessage.Update
    objMessage.Send showDialog:=False
    MsgBox "The message has been sent"
    objSession.Logoff
    Exit Function

error_olemsg:
    MsgBox "Error " & Str(Err) & ": " & Error$(Err)
    Resume Next

End Function
```

Note When you edit an object other than the Message object, save your changes using the **Update** method before you clear or reuse the variable that refers to the object. If you do not use the **Update**

method, your changes can be lost without warning.

After calling the Message object's **Send** method, you should not try to access the Message object again. The **Send** method invalidates the Message object.

See Also

Adding Attachments to a Message

Customizing a Folder or Message

Customizing a Folder or Message

The OLE Messaging Library allows customization and extensibility by offering the Field object and Fields collection. A Field object includes a name, a data type, and a value property. An object that supports fields, in effect, lets you add your own custom properties to the object.

The OLE Messaging Library supports the use of fields with the Message and Folder objects.

For example, consider that you want to add a "Keyword" property to messages so that you can associate a string with the message. You may wish to use a self-imposed convention that values of the "Keyword" are restricted to a small set of strings. You can then organize your messages by the "Keyword" property.

The following example shows how to add the field to the Message object:

```
' Function: Fields_Add
' Purpose:  Add a new field object to the Fields collection
' See documentation topic:  Add method (Fields collection)
Function Fields_Add()
Dim cFields As Integer      ' count of Fields in the collection
Dim objNewField As Object   ' new Field object

    On Error GoTo error_olemsg
    If objFieldsColl Is Nothing Then
        MsgBox "must first select Fields collection"
        Exit Function
    End If
    Set objNewField = objFieldsColl.Add( _
                      Name:="Keyword", _
                      Class:=vbString, _
                      Value:="Peru")
    If objNewField Is Nothing Then
        MsgBox "could not create new Field object"
        Exit Function
    End If
    cFields = objFieldsColl.Count
    MsgBox "new Fields collection count = " & cFields
    ' you can now write code that searches for
    ' messages with this "custom property"
    Exit Function

error_olemsg:
    MsgBox "Error " & Str(Err) & ": " & Error$(Err)
    Resume Next

End Function
```

Note that the new field information specified by the **Add** method is not actually saved until you call the Message object's **Update** method.

Note that MAPI stores all custom properties that represent date and time information using Greenwich Mean Time (GMT). The OLE Messaging Library converts these properties so that the values appear to the user in local time.

For a complete list of the valid Field object data types, see the reference documentation for the Fields collection's **Add** method.

See Also

Field Object

Fields Collection Object

Creating and Sending a Message

Deleting a Message

The Message object's **Delete** method deletes the message.

To delete a message

1. Select the message you want to delete.
2. Call the Message object's **Delete** method.
3. Set the Message object to **Nothing**.

You should not try to access the message after deleting it. Doing so can produce unpredictable results.

See Also

Searching for a Message

Handling Errors

The OLE Messaging Library raises exceptions for all errors. When you write Visual Basic applications that use the OLE Messaging Library, use the same run-time error-handling techniques that you use in all your Visual Basic applications: the Visual Basic **On Error Goto** statement.

Note that the error values and error-handling techniques vary slightly depending on whether you are using Visual Basic version 4.0 or older versions of Visual Basic for Applications.

When you use older versions of Visual Basic for Applications, use the **Err** function to obtain the status code and the **Error$** function to obtain a descriptive error message, as in the following example:

```
' Visual Basic for Applications error handling
MsgBox "Error number " & Err & " description. " & Error$(Err)
```

When you use Visual Basic 4.0, use the **Err** object's **Number** property to obtain the status code and use its **Description** property to obtain the error message, as in the following example:

```
'' Visual Basic version 4.0 error handling
MsgBox "Error " & Err.Number & " description. " & Err.Description
```

Depending on your version of Microsoft Visual Basic, the error code will be returned as a long integer or as a short integer, and you should appropriately define the value of the error codes checked by your program.

When you use Visual Basic 4.0, the error value is set to the value of the MAPI HRESULT, a long integer error code. When you use Visual Basic for Applications, the run-time error value is equal to the sum of 1000 and the low-order word of the HRESULT. (This is because Visual Basic 3.0 reserves all run-time error values below 1000 for its own errors.)

The example in this section checks for an error corresponding to the MAPI error code MAPI_E_USER_CANCEL, which has the value 0x80040113. Visual Basic 4.0 users can check directly for this value. Visual Basic for Applications users check for the value of the low-order word plus 1000. The low-order word is 0x0113, or 275, so the value returned by Visual Basic for Applications is 1275.

```
' demonstrates error handling for Logon
' Function: TestDrv_Util_CreateSessionAndLogon
' Purpose: Call the utility function Util_CreateSessionAndLogon
' See documentation topic:  Handling Errors;
'    Creating And Sending A Message
Function TestDrv_Util_CreateSessionAndLogon()
Dim bFlag As Boolean
    On Error GoTo error_olemsg
    bFlag = Util_CreateSessionAndLogon()
    MsgBox "bFlag = " & bFlag
    Exit Function

error_olemsg:
    MsgBox "Error " & Str(Err) & ": " & Error$(Err)
    Resume Next

End Function

' Function: Util_CreateSessionAndLogon
' Purpose: Demonstrate common error handling for Logon
' See documentation topic: Handling Errors
Function Util_CreateSessionAndLogon() As Boolean
    On Error GoTo err_CreateSessionAndLogon
    Set objSession = CreateObject("MAPI.Session")
    objSession.Logon
    Util_CreateSessionAndLogon = True
    Exit Function

err_CreateSessionAndLogon:
    If (Err = 1275) Then
' VB4.0 version:
'   If (Err.Number = MAPI_E_USER_CANCEL) Then
        MsgBox "User pressed Cancel"
    Else
        MsgBox "Unrecoverable Error:" & Err
    End If
    Util_CreateSessionAndLogon = False
    Exit Function

error_olemsg:
    MsgBox "Error " & Str(Err) & ": " & Error$(Err)
    Resume Next

End Function
```

When an error occurs in the MAPI subsystem, the OLE Messaging Library supplies the error value returned by MAPI. However, the value can be returned from any of several different *levels* of software. The lowest level of software is that which interacts directly with hardware, such as a mouse driver or video driver. Higher levels of software move toward greater device independence and greater generality.

The following diagram suggests the different levels of software in Visual Basic applications that use the OLE Messaging Library. Visual Basic applications reside at the highest level and interact with the OLE Messaging Library at the next lower level. The OLE Messaging Library interacts with the MAPI system software, and the MAPI system software interacts with a lower layer of software, the operating system.

{ewc msdncd, EWGraphic, groupx842 0 /a "MAPI.BMP"}

Software components in a Visual Basic application using the OLE Messaging Library

Errors can occur at any level or at the interface between any two levels. For example, a user of your application without security permissions can be denied access to an address book entry. The lowest level in this diagram, the operating system, returns the error to the next higher level, and so on, until the error is returned to the highest level in this diagram, the Visual Basic application.

It is often useful to provide a general error handling capability that will display the complete HRESULT or error code value returned by the OLE Messaging Library.

For more information about run-time error handling and the **Err** object, see your product's Visual Basic documentation. For a listing of the MAPI error values, see Appendix A, Error Codes.

See Also

Error Codes

Starting a Session with MAPI

Improving Application Performance

This section describes how your Visual Basic code can operate most efficiently when you use messaging objects. Note that this section is written primarily for Visual Basic programmers rather than for C programmers.

To access OLE Messaging Library objects, you create Visual Basic statements that concatenate the object names in sequence from left to right, separating objects with a "dot," the period character. For example, consider the following Visual Basic statement.

```
Set objMessage = objSession.Inbox.Messages.GetFirst
```

The OLE Messaging Library creates an internal object for each dot that appears in the statement. For example, the portion of the statement that says "objSession.Inbox" directs the OLE Messaging Library to create an internal Folder object that represents the user's Inbox. The next portion, ".Messages," directs the OLE Messaging Library to create an internal Messages collection object. The final part, ".GetFirst," directs the OLE Messaging Library to create an internal Message object that represents the first message in the user's Inbox. The statement contains three dots; the OLE Messaging Library creates three internal objects.

The best rule of thumb is to remember that dots are expensive. For example, the following two lines of code are very inefficient:

```
' warning: do not code this way, this is inefficient
MsgBox "Text: " & objSession.Inbox.Messages.MoveFirst.Text
MsgBox "Subj: " & objSession.Inbox.Messages.MoveFirst.Subject
```

While this code generates correct results, it is not very efficient. For the first statement, the OLE Messaging Library creates internal objects that represent the Inbox, its Messages collection, and its first message. After the application displays the text, these internal objects are discarded. In the next line, the same internal objects are generated again. A more efficient approach would be to generate the internal objects only once:

```
With objSession.Inbox.Messages.MoveFirst
  MsgBox "Text: " & .Text
  MsgBox "Subj: " & .Subject
End With
```

When your application needs to use an object more than once, define a variable for the object and set its value. The following sample code is very efficient when your application reuses the Folder, Messages collection, or Message objects:

```
' very efficient when the objects will be reused
Set objInboxFolder = objSession.Inbox
Set objInMessages = objInboxFolder.Messages
Set objOneMessage = objInMessages.MoveFirst
With objOneMessage
  MsgBox "The Message Text: " & .Text
  MsgBox "The Message Subject: " & .Subject
End With
```

Now that you understand that a dot in a statement directs the OLE Messaging Library to create a new internal object, it is easy to see that the following sample is not correct:

```
' error: collection returns the same message both times
MsgBox("first msg: " & inBoxObj.Messages.GetFirst)
MsgBox("next msg: " & inBoxObj.Messages.GetNext)
```

The OLE Messaging Library creates a temporary internal object that represents the Messages collection, then discards it after displaying the first message. The second statement directs the OLE Messaging Library to create another new temporary object that represents the Messages collection. This Messages collection is new and has no state information; that is, this new collection has not called **GetFirst**. The **GetNext** statement causes it to display its first message again.

Use the Visual Basic **With** statement or explicit variables to generate the expected results. The following example uses explicit variables:

```
' Use the Visual Basic With statement
With objSession.Inbox.Messages
    Set objMessage = .GetFirst
    '...
    Set objMessage = .GetNext
End With
' Use explicit variables to refer to the collection;
Set objMsgColl = objSession.Inbox.Messages
Set objMessage = myMsgColl.GetFirst
...
Set objMessage = myMsgColl.GetNext
```

For more information about improving the performance of your applications, see your Microsoft Visual Basic programming documentation.

See Also

Handling Errors

Making Sure the Message Gets There

The Message object contains two properties that can direct the underlying MAPI system to report successful receipt of the message: **DeliveryReceipt** and **ReadReceipt**.

When you set these properties to TRUE and send the message, the underlying MAPI system automatically tracks the message for you. When you set the **DeliveryReceipt** property, the MAPI system automatically generates a message to the sender reporting when the recipient receives the message. When you set the **ReadReceipt** property, the MAPI system automatically generates a message to the sender reporting when the recipient reads the message.

See Also

Securing Messages

Moving a Message to Another Folder

The procedure documented in this section demonstrates a way to move message properties using the Message object's **Add** and **Delete** methods supported in the OLE Messaging Library.

Note Using OLE Messaging Library version 1.0, the Message object's **Sender** property and other read-only properties of the Message object are not preserved using the procedure in this section. To preserve these properties using this procedure, you must append their text fields to read-write properties, such as the Message object's **Text** property.

To move a message from one folder to another

1. Obtain the source message that you want to copy.
2. Call the destination folder's Messages collection's **Add** method, supplying source message properties as parameters.
3. Call the source message's **Delete** method to delete the original source message from its folder.

For the complete sample, see Copying a Message to Another Folder. The final lines of code for the procedure should delete the original message:

```
' "Move" implies explicit delete of the initial message
objOneMsg.Delete
```

See Also

Copying a Message to Another Folder

Posting Messages to a Public Folder

To post a message to a public folder, create a message within the public folder by calling the folder's Messages collection **Add** method. Then add your subject and message text as you would for other messages.

Note that for messages in public folders, you must also set a few more message properties than you would when sending a message to a recipient. When you post a message to a public folder, the components of the MAPI architecture that usually handle a message and set its properties do not manage the message. Your application must set the **Unread**, **Submitted**, and **Sent** properties to TRUE, and must set the **TimeSent** and **TimeReceived** properties to the current time.

When you are ready to make the message available, call the **Send** or **Update** method.

Note When posting messages in a public folder, you cannot use the OLE Messaging Library to set the **Sender** property. The **Sender** and related underlying properties are not present for a message created by the OLE Messaging Library.

For more information about the complete procedure for sending messages, see Creating And Sending A Message.

To create a message within a public folder

1. Call the Messages collection's **Add** method to create a Message object.
2. Set the Message object's **Text**, **Subject**, **ConversationSubject**, **ConversationIndex**, **TimeSent**, **TimeReceived**, and other message properties as desired.
3. Set the Message object's **Unread**, **Submitted**, and **Sent** properties to TRUE.
4. Call the Message object's **Send** or **Update** method.

Note that when you post a message, you must explicitly set the **TimeSent** and **TimeReceived** properties. When you send a message using the **Send** method, the MAPI system assigns the values of these properties for you. However, when you post the message by setting the **Submitted** property, your application must set the time properties. Set both time properties to the same value, just before you set the **Submitted** property to TRUE.

```
' Function: Util_New_Conversation
' Purpose: Set properties to start a new conversation in a public folder
' See documentation topic: Working With Conversations;
'     Posting Messages To A Public Folder
Function Util_NewConversation()
Dim objRecipColl As Object
Dim i As Integer
Dim objNewMsg As Object      ' new message object
Dim strNewIndex As String
    On Error GoTo error_olemsg


' objPublicFolder is a global variable that indicates
' the folder in which you want to post the message
    Set objNewMsg = objPublicFolder.Messages.Add
    If objNewMsg Is Nothing Then
        MsgBox "unable to create a new message for the public folder"
        Exit Function
    End If
    strConversationFirstMsgID = objNewMsg.Id   'save for reply
    With objNewMsg
        .Subject = "used space vehicle wanted"
        .ConversationTopic = .Subject
        .ConversationIndex = Util_GetEightByteTimeStamp() ' utility
        .Text = "Wanted: Apollo or Mercury spacecraft with low mileage."
        .TimeSent = Time
        .TimeReceived = .TimeSent
        .Submitted = True
        .Unread = True
        .Sent = True
        .Update
        .Send showDialog:=False
    End With
    Exit Function

error_olemsg:
    MsgBox "Error " & Str(Err) & ": " & Error$(Err)
    Resume Next

End Function
```

For complete information about the **ConversationIndex** property, see Working With Conversations.

See Also

Searching for a Folder

Creating and Sending a Message

Working with Conversations

Reading a Message from the Inbox

After establishing a Session object and successfully logging on to the system, a user can access the *Inbox*. The Inbox is the default folder for mail received by the user.

As described in OLE Messaging Library Object Design, the OLE Messaging Library objects are organized in a hierarchy. The Session object at the topmost level allows access to a Folder. Each Folder contains a Messages collection, and the Messages collection contains individual Message objects. The text of the message appears in the Message object's **Text** property.

```
Session object
    Folder object
        Messages collection
            Message object
                      Text property
```

To obtain an individual message, the application must move down through this object hierarchy to the **Text** property. The following example uses the Session object's **Inbox** property to obtain a Folder object, then uses the Folder object's **Messages** property to obtain a Messages collection object, and calls the Messages collection's methods to get a specific message.

This example assumes that the application has already created the Session object variable *objSession* and successfully called the Session object's **Logon** method, as described in the section, Starting a Session with MAPI:

```
Dim objSession As Object      ' Session object
Dim objInboxFolder As Object  ' Folder object
Dim objInMessages As Object   ' Messages collection
Dim objOneMsg As Object       ' Message object
...
' move down through the hierarchy
Set objInboxFolder = objSession.Inbox
Set objInMessages = objInboxFolder.Messages
Set objOneMsg = objInMessages.GetFirst
MsgBox "The message text: " & objOneMsg.Text
```

Note Use the Visual Basic keyword **Set** whenever you initialize a variable that represents an object. When you set an object variable without using the **Set** keyword, Visual Basic generates an error message.

The example above declares several object variables. However, it is also possible to access the message with fewer variables. The following sample is equivalent to the sample code above:

```
Set objOneMsg = objSession.Inbox.Messages.GetFirst
MsgBox "The message text: " & objOneMsg.Text
```

You should declare an individual variable when the application needs to access an object more than once. When an object is accessed repeatedly, variables can help make your code efficient. For more information, see Improving Application Performance.

See Also

Creating and Sending a Message

Improving Application Performance

Searching for a Message

Searching for a Folder

Two frequently used folders, the Inbox and the Outbox, are available through Session object properties. To access these folders, simply set a Folder object to the corresponding property.

To access other folders, search for the folder using one of the following techniques:

- Call the Session object's **GetFolder** method with a string parameter that specifies the *FolderID*, a unique identifier for the folder.
- Use the **Get** methods to navigate through the Folders collection. Search for a specific folder by comparing the current folder's properties with the desired properties.

Using the Session Object's GetFolder Method

When you know the unique identifier for the folder you are looking for, you can call the Session object's **GetFolder** method.

The unique identifier for the folder, established at the time the folder is created, is stored in its **ID** property. The **ID** property is a string representation of the MAPI entry identifier and its value is determined by the service provider.

The following code fragment contains code that saves the identifier for the folder, then uses it in a subsequent **GetFolder** call:

```
' Function: Session_GetFolder
' Purpose: Demonstrate how to set a folder object
' See documentation topic: Session object GetFolder method
Function Session_GetFolder()
    On Error GoTo error_olemsg

    If objSession Is Nothing Then
        MsgBox "No active session, must log on"
        Exit Function
    End If
    If strFolderID = "" Then
        MsgBox ("Must first set folder ID variable; see Folder->ID")
        Exit Function
    End If
    Set objFolder = objSession.GetFolder(strFolderID)
    'equivalent to:
    ' Set objFolder = objSession.GetFolder(folderID:=strFolderID)
    If objFolder Is Nothing Then
        Set objMessages = Nothing
        MsgBox "Unable to retrieve folder with specified ID"
        Exit Function
    End If
    MsgBox "Folder set to " & objFolder.Name
    Set objMessages = objFolder.Messages
    Exit Function

error_olemsg:
    MsgBox "Error " & Str(Err) & ": " & Error$(Err)
    Set objFolder = Nothing
    Set objMessages = Nothing
    MsgBox "Folder is no longer available; no active folder"
    Exit Function
End Function
```

Using the Get Methods

When you are looking for a folder within a Folders collection, you can navigate down through the collection, examining properties of each Folder object to determine whether it is the folder you want.

The OLE Messaging Library supports the **GetFirst**, **GetLast**, **GetNext**, and **GetPrevious** methods for the Folders collection object.

The following sample demonstrates how to use the **Get** methods to search for the specified folder:

```
' Function: TestDrv_Util_GetFolderByName
' Purpose: Call the utility function Util_GetFolderByName
' See documentation topic: Item property (Folder object)
Function TestDrv_Util_GetFolderByName()
Dim fFound As Boolean
    fFound = Util_GetFolderByName("Junk mail")
    If fFound Then
        MsgBox "Folder named 'Junk mail' found"
    Else
        MsgBox "Folder named 'Junk mail' not found"
    End If
    Exit Function

error_olemsg:
    MsgBox "Error " & Str(Err) & ": " & Error$(Err)
    Resume Next
End Function

' Function: Util_GetFolderByName
' Purpose: Use Get* methods to search for a folder
' See documentation topic:  Searching For a Folder
Function Util_GetFolderByName(strSearchName As String) As Boolean
Dim objOneFolder As Object   ' local; temp version of folder object

    On Error GoTo error_olemsg
    Util_GetFolderByName = False  ' default; assume failure
    If objFolder Is Nothing Then
        MsgBox "must first select a folder; such as Session->Inbox"
        Exit Function
    End If
    Set objFoldersColl = objFolder.Folders  ' Folders collection
    If objFoldersColl Is Nothing Then
        MsgBox "no subfolders; not found"
        Exit Function
    End If
    ' get the first folder in the collection
    Set objOneFolder = objFoldersColl.GetFirst
    ' loop through all the folders in the collection
    Do While Not objOneFolder Is Nothing
        If objOneFolder.Name = strSearchName Then
            Exit Do  ' found it, leave the loop
        Else  ' keep searching
            Set objOneFolder = objFoldersColl.GetNext
        End If
    Loop
    ' exit from the do while loop comes here
    ' if objOneFolder is valid, the folder is found
    If Not objOneFolder Is Nothing Then
        Util_GetFolderByName = True    ' success; set to False above
    End If
    Exit Function

error_olemsg:
    MsgBox "Error " & Str(Err) & ": " & Error$(Err)
    Resume Next
End Function
```

You can also navigate upward by using the **Parent** property.

See Also

Searching for a Message

Searching for a Message

To access a message, search for the message using one of the following techniques:

- Call the Session object's **GetMessage** method with a string parameter that specifies the *MessageID*, a unique identifier for the message.
- Use the **Get** methods to navigate through the folder's Messages collection. Search for a specific message by comparing the current Message object's properties with the desired properties.

Using the Session Object's GetMessage Method

When you know the unique identifier for the message you are looking for, you can call the Session object's **GetMessage** method.

The message identifier specifies a unique identifier that is created for the Message object at the time it is created. The identifier is accessible through the object's **ID** property.

The following code fragment contains code that saves the identifier for the folder, then uses it in a subsequent **GetMessage** call:

```
' Function: Session_GetMessage
' Purpose: Demonstrate how to set a message object using GetMessage
' See documentation topic: GetMessage method (Session object)
Function Session_GetMessage()
    On Error GoTo error_olemsg

    If objSession Is Nothing Then
        MsgBox "No active session, must log on"
        Exit Function
    End If
    If strMessageID = "" Then
        MsgBox ("Must first set message ID variable; see Message->ID")
        Exit Function
    End If
    Set objOneMsg = objSession.GetMessage(strMessageID)
    If objOneMsg Is Nothing Then
        MsgBox "Unable to retrieve message with specified ID"
        Exit Function
    End If
    MsgBox "GetMessage returned msg with subject: " & objOneMsg.Subject
    Exit Function

error_olemsg:
    MsgBox "Error " & Str(Err) & ": " & Error$(Err)
    Set objOneMsg = Nothing
    MsgBox "Message is no longer available; no active message"
    Exit Function
End Function
```

Using the Get Methods

When you are looking for a message within a Messages collection, you can navigate through the collection, examining properties of each Message object to determine if it is the message you want.

The OLE Messaging Library supports the **GetFirst**, **GetLast**, **GetNext**, and **GetPrevious** methods for the Messages collection object.

The following sample demonstrates how to use the **Get** methods to search for the specified message:

```
' Function: TestDrv_Util_GetMessageByName
' Purpose: Call the utility function Util_GetMessageByName
' See documentation topic: Item property (Message object)
Function TestDrv_Util_GetMessageByName()
Dim fFound As Boolean
    On Error GoTo error_olemsg

    fFound = Util_GetMessageByName("Junk mail")
    If fFound Then
        MsgBox "Message named 'Junk mail' found"
    Else
        MsgBox "Message named 'Junk mail' not found"
    End If
    Exit Function

error_olemsg:
    MsgBox "Error " & Str(Err) & ": " & Error$(Err)
    Resume Next
End Function

' Function: Util_GetMessageByName
' Purpose: Use Get* methods to search for a message
' See documentation topic:  Searching for a message
' search through the messages for one with a specific subject
Function Util_GetMessageByName(strSearchName As String) As Boolean
Dim objOneMessage As Object   ' local; temp version of message object

    On Error GoTo error_olemsg
    Util_GetMessageByName = False  ' default; assume failure
    If objFolder Is Nothing Then
        MsgBox "must first select a folder; such as Session->Inbox"
        Exit Function
    End If
    Set objMessages = objFolder.Messages
    Set objOneMessage = objMessages.GetFirst
    If objOneMessage Is Nothing Then
        MsgBox "no messages in the folder"
        Exit Function
    End If
    ' loop through all the messages in the collection
    Do While Not objOneMessage Is Nothing
        If objOneMessage.Subject = strSearchName Then
            Exit Do  ' found it, leave the loop
        Else  ' keep searching
            Set objOneMessage = objMessages.GetNext
        End If
    Loop
    ' exit from the do while loop comes here
    ' if objOneMessage is valid, the message was found
    If Not objOneMessage Is Nothing Then
        Util_GetMessageByName = True    ' success
    End If
    Exit Function

error_olemsg:
    MsgBox "Error " & Str(Err) & ": " & Error$(Err)
    Resume Next

End Function
```

See Also

Searching for a Folder

Securing Messages

The Message object contains two properties that specify security for the message: the **Encrypted** and **Signed** properties. When you want to request that your message be secured, set these flags to TRUE.

Note that these flags simply represent a request to the underlying messaging service. Whether the message is encrypted or signed depends on whether these security measures are implemented by your messaging service.

Neither MAPI nor the OLE Messaging Library performs encryption or digital signing. The OLE Messaging Library simply sets the appropriate MAPI properties so that the proper request for security is delivered to the messaging service. For more information about the capabilities of your messaging service, contact your Microsoft Exchange Server system administrator.

```
Dim objMessage As Object   ' assume valid Message object
'...
objMessage.Encrypted = True
objMessage.Send
```

See Also

Making Sure the Message Gets There

Selecting Recipients from the Address Book

After establishing a Session object and successfully logging on to the system, the user can access the address book to select recipients. You can select recipients from any address book, such as the global address list or the personal address book.

As described in OLE Messaging Library Object Design, the OLE Messaging Library objects are organized in a hierarchy. The Session object at the topmost level contains an **AddressBook** method that lets your application users select recipients from an address book. The method returns a Recipients collection, which contains individual Recipient objects. The Recipient object in turn specifies an AddressEntry object. This hierarchy is shown in the following diagram.

Recipients collection
- Recipient object
  - **Address** property (full address)
  - AddressEntry object
    - **Address** property (e-mail address)
    - **Type** property

To obtain an individual **Address** property that can be used to address and send messages, the application must move down through this object hierarchy. The following code example uses the Recipients collection returned by the Session object's **AddressBook** method.

This example assumes that the application has already created the Session object variable *objSession* and successfully called the Session object's **Logon** method, as described in Starting a Session with MAPI:

```
' Function: Session_AddressBook
' Purpose: Set the global variable that contains the current recipients
'    collection to that returned by the Session AddressBook method
' See documentation topic: AddressBook method (Session object)
Function Session_AddressBook()
    On Error GoTo err_Session_AddressBook

    If objSession Is Nothing Then
        MsgBox "must first create MAPI session and logon"
        Exit Function
    End If
    Set objRecipColl = objSession.AddressBook( _
        Title:="Select Attendees", _
        forceResolution:=True, _
        recipLists:=1, _
        toLabel:="&OLE Messaging")  ' appears on button
    ' Note: initial value not used
    ' parameter not used in call: Recipients:=objInitRecipColl
    MsgBox "Name of first recipient = " & objRecipColl.Item(1).Name
    Exit Function

err_Session_AddressBook:
    If (Err = 91) Then   ' MAPI dlg-related function that sets an object
        MsgBox "No recipients selected"
    Else
        MsgBox "Unrecoverable Error:" & Err
    End If
    Exit Function
End Function
```

See Also

Changing an Existing Address Entry

Using Addresses

Starting a Session with MAPI

As described in OLE Messaging Library Object Design, all messaging objects are relative to the Session object. One of the first tasks of every application is to create a valid Session object and call its **Logon** method.

The Session object is created using the Visual Basic function **CreateObject**. The following code demonstrates how to perform this common startup task:

```
Function Util_CreateSessionAndLogon() As Boolean
    On Error GoTo err_CreateSessionAndLogon

    Set objSession = CreateObject("MAPI.Session")
    objSession.Logon
    Util_CreateSessionAndLogon = True
    Exit Function

err_CreateSessionAndLogon:
    If (Err = 1275) Then  ' VB4.0 uses "Err.Number"
        MsgBox "User pressed Cancel"
    Else
        MsgBox "Unrecoverable Error:" & Err
    End If
    Util_CreateSessionAndLogon = False
    Exit Function

End Function
```

When no parameters are supplied to the **Logon** method, as in the example above, the OLE Messaging Library displays an application-modal logon dialog box that prompts the application user to select a user profile. Based on the characteristics of the selected profile, the underlying MAPI system logs on the user or prompts for password information.

You can also choose to use your own application's dialog box to obtain the parameters needed to log on, rather than using the MAPI logon dialog box. The following example obtains the profile name and password information and directs the **Logon** method not to display a logon dialog box:

```
' Function: Session_Logon_NoDialog
' Purpose: Call the Logon method, set parameter to show no dialog
' See documentation topic: Logon Method (Session object)
Function Session_Logon_NoDialog()
    On Error GoTo error_olemsg
    ' can set strProfileName, strPassword from a custom form
    ' adjust these parameters for your configuration
    If objSession Is Nothing Then
        Set objSession = CreateObject("MAPI.Session")
    End If
    If Not objSession Is Nothing Then
    ' configure these parameters for your needs either here
    ' or in the function Util_Initialize
        objSession.Logon profileName:=strProfileName, _
                         showDialog:=False
    End If
    Exit Function

error_olemsg:
    If 1273 = Err Then
        MsgBox "cannot logon: incorrect profile name or password"
        Exit Function
    End If
    MsgBox "Error " & Str(Err) & ": " & Error$(Err)
    Resume Next
End Function
```

Note Your Visual Basic application should be able to handle cases that occur when a user provides incorrect profile or password information, or that occur when a user cancels from the Logon dialog box. For more information, see Handling Errors.

After establishing a Session object and successfully logging on to the system, the user has access to several default objects provided by the Session object, including the Inbox and Outbox folders. For more information, see Reading a Message from the Inbox.

See Also

Reading a Message from the Inbox

Creating and Sending a Message

Using Addresses

In general, MAPI supports two kinds of addressing:

- Addresses that the MAPI system looks up for you in your address book, based on a display name that you supply.
- Addresses that represent *custom addresses,* that are used as supplied without lookup.

The OLE Messaging Library supports both kinds of addresses with its Recipient object. To look up an address name, you supply the **Name** property only. To use custom addresses, you supply the full address in the **Address** property.

The address book can be thought of as a database in persistent storage, managed by the MAPI system, that contains valid addressing information that is associated with a *display name.* The display name represents the way that a person's name might be displayed for your application users, using that person's full name, rather than the e-mail address that the messaging system uses to transmit the message. For example, the display name "John Doe" could be mapped to the e-mail address "johnd."

In contrast to the address book, the objects that you create with the OLE Messaging Library are temporary objects that reside in memory. When you fill in the Recipient object's **Name** property with a display name, you must then *resolve* the address. To resolve the address means that you ask the MAPI system to look up the display name in the database and supply you with the corresponding address. When the display name is ambiguous, or can match more than one entry in the address book, the MAPI system prompts you to select from a list of possible matching names.

The Recipient object's **Name** property represents the display name. Call the Recipient object's **Resolve** method to resolve the display name.

After the Recipient object is resolved, it has a child AddressEntry object that contains a copy of the valid addressing information from the database. The child AddressEntry object is accessible from the Recipient object's **AddressEntry** property. The Recipient object and AddressEntry object properties are related as follows.

OLE Messaging Library property	MAPI property	Description
Recipient.Address	Combination of PR_ADDR_TYPE and PR_EMAIL_ADDRESS	Full address; AddressEntry object's **Type** and **Address** properties
Recipient.Name	PR_DISPLAY_NAME	Display name
Recipient.AddressEntry.Address	PR_EMAIL_ADDRESS	E-mail address
Recipient.AddressEntry.Type	PR_ADDR_TYPE	E-mail type
Recipient.AddressEntry.Name	PR_DISPLAY_NAME	Display name
Recipient.AddressEntry.ID	PR_ENTRYID	Unique identifier for the address entry

The Recipient object's **Address** property represents a *full address*, that is, the combination of address type and e-mail address that MAPI uses to send a message. The full address represents the same information that appears in the AddressEntry **Address** and **Type** properties.

You can also supply a complete recipient address. By manipulating the address yourself, you direct the MAPI system to send the message to the full address that you supply without using the database. In this case, you must also supply the display name. When you supply a custom address, the Recipient object's **Address** property must use the following syntax.

TypeValue:*AddressValue*

There is also a third method of working with addresses. You can directly obtain and use the Recipient object's child AddressEntry object from messages that have already been successfully sent through the messaging system.

For example, to reply to a message, you can use a Message object's **Sender** property to get a valid AddressEntry object. When you work with valid AddressEntry objects, you do not have to call the **Resolve** method.

Note When you use existing AddressEntry objects, do not try to modify them. In general, do not write directly to the Recipient object's child AddressEntry object properties.

In summary, you can provide addressing information in three different ways:

- Obtain the correct addressing information for a known display name. Set the Recipient object's **Name** property and call the Recipient object's **Resolve** method. Note that the **Resolve** method can display a dialog box.
- Use an existing valid address entry, such as the Message object's **Sender** property, when you are replying to a message. Set the Recipient object's **AddressEntry** property to an existing AddressEntry object that is known to be valid. (You do not need to call the **Resolve** method.)
- Create a custom address. Set the Recipient object's **Address** property, using the correct syntax as described above (use the colon character (**:**) to separate the address type from the address), and call the **Resolve** method.

The following sample code demonstrates these three kinds of addresses:

```
' Function: Util_UsingAddresses
' Purpose:  Set addresses three ways
' See documentation topic: Using Addresses
Function Util_UsingAddresses()
Dim objNewMessage As Object         ' new message object for example
Dim strAddrEntryID As String        ' ID value from AddressEntry object
Dim strName As String               ' Name from AddressEntry object
    On Error GoTo error_olemsg
    If objOneMsg Is Nothing Then
        MsgBox "Must select a message"
        Exit Function
    End If
    With objOneMsg.Recipients.Item(1).AddressEntry
        strAddrEntryID = .Id
        strName = .Name
    End With
    Set objNewMessage = objSession.Outbox.Messages.Add
    If objNewMessage Is Nothing Then
        MsgBox "Unable to add a new message"
        Exit Function
    End If
    ' add three recipients
    ' 1. look up entry in address book specified by profile
    Set objOneRecip = objNewMessage.Recipients.Add( _
        Name:=strName, _
        Type:=mapiTo)
    If objOneRecip Is Nothing Then
```

```
        MsgBox "Unable to add recipient using Display Name"
        Exit Function
    End If
    objOneRecip.Resolve
    ' 2. add a custom recipient
    Set objOneRecip = objNewMessage.Recipients.Add( _
        Address:="SMTP:davidhef@microsoft.com", _
        Type:=mapiTo)
    If objOneRecip Is Nothing Then
        MsgBox "Unable to add recipient using custom addressing"
        Exit Function
    End If
    objOneRecip.Resolve

    ' 3. add a valid address entry object, such as Message.Sender
    Set objOneRecip = objNewMessage.Recipients.Add( _
        entryID:=strAddrEntryID, _
        Name:=strName, _
        Type:=mapiTo)
    If objOneRecip Is Nothing Then
        MsgBox "Unable to add recipient using existing AddressEntry ID"
        Exit Function
    End If

    objNewMessage.Text = "expect 3 different recipients"
    MsgBox ("count = " & objNewMessage.Recipients.Count)
    ' you can also call resolve for the whole collection
    ' objNewMessage.Recipients.Resolve (True) ' resolve all; show dialog

    objNewMessage.Subject = "test"
    objNewMessage.Update           ' update the message
    x = objNewMessage.Send(showDialog:=False)
    Exit Function

error_olemsg:
    MsgBox "Error " & Str(Err) & ": " & Error$(Err)
    Exit Function

End Function
```

See Also

Sender Property (Message Object)

Resolve Method (Recipient Object)

Changing an Existing Address Entry

Viewing MAPI Properties

You can use a feature of the OLE Messaging Library's Fields collection object to view the values of MAPI properties.

The Fields collection's **Item** property allows you to specify the actual *property tag* value as an identifier. A property tag is a 32-bit unsigned integer that contains the property identifier in its high-order 16 bits and the property type (its underlying data type) in the low-order 16 bits. The OLE Messaging Library also supports *multivalued* properties, or properties that represent arrays of values. A multivalued property appears to the Visual Basic application as a variant array; that is, you can use the **For... Next** statement to access individual array entries.

The OLE Messaging Library works with three types of message properties:

- Standard MAPI properties named in the OLE Messaging Library object description language (ODL) file.
- Standard MAPI properties not named in the OLE Messaging Library ODL file.
- Properties named and created by the application.

The Fields collection exposes standard MAPI properties not named in the ODL file and properties named and created by the application. Although the Field object provides a **Delete** method, you should note that some standard MAPI properties, such as those created by MAPI system components, cannot be deleted.

Note that MAPI stores all custom properties that represent date and time information using Greenwich Mean Time (GMT). The OLE Messaging Library converts these properties so that the values appear to the user in local time.

Note   A complete discussion of MAPI properties is beyond the scope of this guide. MAPI properties are defined and covered in detail in the *MAPI Programmer's Reference.*

```
' Function: Fields_Selector
' Purpose: View a MAPI property by supplying a property tag value as
'    the Item value
' See documentation topics: Viewing MAPI Properties;
'    Item property (Fields collection)
Function Fields_Selector()
Dim lValue As Long
Dim strMsg As String

    On Error GoTo error_olemsg

    If objFieldsColl Is Nothing Then
        MsgBox "must first select a Fields collection"
        Exit Function
    End If
    ' you can provide a dialog here so users enter MAPI proptags...
    ' or select property names from a list; for now, hard-coded value
    lValue = &h1a001e
    ' &H1a = PR_MESSAGE_CLASS; &H001e = 30 = PT_STRING8
    ' high-order 16 bits is property id; low-order is property type
    Set objOneField = objFieldsColl.Item(lValue)
    If objOneField Is Nothing Then
        MsgBox "Could not get the Field using the value " & lValue
        Exit Function
    Else
```

```
        strMsg = "Used the value " & lValue & " to access the property"
        strMsg = strMsg & "PR_MESSAGE_CLASS: type = " & objOneField.Type
        strMsg = strMsg & "; value = " & objOneField.Value
        MsgBox strMsg
    End If
    Exit Function

error_olemsg:
    MsgBox "Error " & Str(Err) & ": " & Error$(Err)
    Resume Next

End Function
```

See Also

Field Object

Customizing a Folder or Message

Working with Conversations

Three Message object properties let you show complex relationships among messages by defining them as part of a *conversation*. A conversation is a series of messages, consisting of an initial message and all messages sent in reply to the initial message. When the initial message or a reply elicits additional messages, the resulting messages are called a *conversation thread*. A thread represents a subset of messages in the conversation.

The conversation properties **Conversation**, **ConversationTopic**, and **ConversationIndex** give you another way to organize and display messages. Rather than simply organizing messages by subject, time received, or sender, you can show rich and complex relationships among messages. The **Conversation** property is a binary value that uniquely identifies the conversation. All messages within the same conversation use the same value for the **Conversation** property. The **ConversationTopic** property is a string that describes the overall subject of the conversation. The **ConversationIndex** property is an index that you can use to represent the relationships between messages and replies.

When you start an initial message, set the **Conversation** property to a unique value, such as a globally unique identifier (GUID). Set the **ConversationTopic** property to an appropriate value that will apply to all messages within the conversation. For many applications, the message **Subject** property is appropriate.

You can use your own convention to decide how to use the **ConversationIndex** property. However, it is recommended that you adopt the same convention used by the Microsoft Exchange Client message viewer, so that you can use that viewer's user interface to show the relationships between messages in a conversation.

By convention, Microsoft Exchange Server uses **ConversationIndex** values that represent concatenated time stamp values. The first time stamp in the string represents the original message. When a new message represents a reply to a conversation message, it copies the **ConversationIndex** string of the message it is replying to, and appends a time stamp value to the end of the string. The new string value is used as the **ConversationIndex** value of the new message.

When you use this convention, you can easily see relationships among messages when you sort the messages by **ConversationIndex** values.

The following code sample provides a utility function, **Util_GetEightByteTimeStamp**, which can be used to build Microsoft Exchange Server-compatible **ConversationIndex** values. The utility function calls the OLE function **CoCreateGuid** to obtain the time stamp value from a GUID data structure. The GUID value is composed of a time stamp and a machine identifier; the utility function saves the part that contains the time stamp.

```
' declarations for the Util_GetEightByteTimeStamp function
Type GUID
    Guid1 As Long
    Guid2 As Long
    Guid3 As Long
    Guid4 As Long
End Type
Declare Function CoCreateGuid Lib "COMPOBJ.DLL" (pGuid As GUID) As Long
' Note:  Use "OLE32.DLL" for Windows NT, Win95 platforms
Global Const S_OK = 0
' end declarations section

' Function: Util_GetEightByteTimeStamp
' Purpose: Generate a time stamp for use in conversations
' See documentation topic: Working With Conversations
Function Util_GetEightByteTimeStamp() As String
Dim lResult As Long
Dim lGuid As GUID
    ' Exchange conversation is a unique 8-byte value
    ' Exchange client viewer sorts by concatenated properties
    On Error GoTo error_olemsg

    lResult = CoCreateGuid(lGuid)
    If lResult = S_OK Then
        Util_GetEightByteTimeStamp = _
            Hex$(lGuid.Guid1) & Hex$(lGuid.Guid2)
    Else
        Util_GetEightByteTimeStamp = "00000000"   ' zeroes
    End If
    Exit Function

error_olemsg:
    MsgBox "Error " & Str(Err) & ": " & Error$(Err)
    Util_GetEightByteTimeStamp = "00000000"
    Exit Function

End Function
```

When you start a new conversation, set the **Conversation** property to a complete GUID value. Set the **ConversationIndex** property to the value returned by this function, as follows:

```
' new conversation
o bjMessage.ConversationIndex = Util_GetEightByteTimeStamp()
```

When you are replying to a message in an existing conversation, append the time stamp value to that message's **ConversationIndex** value:

```
' reply within an existing conversation
Dim objOriginalMsg As Object    ' assume valid
Dim strNewIndex As String
'...
' copy the original topic and
' append the current time stamp to the original time stamp
objMessage.ConversationTopic = objOriginalMsg.ConversationTopic
strNewIndex = objOriginalMsg.ConversationIndex _
                & Util_GetEightByteTimeStamp()
objMessage.ConversationIndex = strNewIndex
```

For additional sample code, see Posting Messages To a Public Folder.

See Also

Posting Messages to a Public Folder

Objects, Properties, and Methods

This reference contains property and method information for the OLE Messaging Library objects.

The following table summarizes each object's properties and methods.

Object	Properties	Methods
AddressEntry	Address, Application, Class, DisplayType, Fields, ID, Name, Parent, Session, Type	Delete, Details, Update
Attachment	Application, Class, Index, Name, Parent, Position, Session, Source, Type	Delete, ReadFromFile, WriteToFile
Attachments (collection)	Application, Class, Count, Item, Parent, Session	Add, Delete
Field	Application, Class, ID, Index, Name, Parent, Session, Type, Value	Delete, ReadFromFile, WriteToFile
Fields (collection)	Application, Class, Count, Item, Parent, Session	Add, Delete, SetNamespace
Folder	Application, Class, Fields, FolderID, Folders, ID, MAPIOBJECT*, Messages, Name, Parent, Session, StoreID	Update
Folders (collection)	Application, Class, Parent, Session	GetFirst, GetLast, GetNext, GetPrevious
InfoStore	Application, Class, ID, Index, Name, Parent, ProviderName, RootFolder, Session	(none)
InfoStores Collection	Application, Class, Count, Item, Parent, Session	(none)
Message	Application, Attachments, Class, Conversation, ConversationIndex, ConversationTopic, DeliveryReceipt, Encrypted, Fields, FolderID, ID, Importance, MAPIOBJECT*, Parent, ReadReceipt, Recipients, Sender, Sent, Session, Signed, Size, StoreID, Subject, Submitted, Text, TimeReceived, TimeSent, Type, Unread	Delete, Options, Send, Update

Messages (collection)	Application, Class, Parent, Session	Add, Delete, GetFirst, GetLast, GetNext, GetPrevious, Sort
Recipient	Address, AddressEntry, Application, Class, DisplayType, Index, Name, Parent, Session, Type	Delete, Resolve
Recipients (collection)	Application, Class, Count, Item, Parent, Resolved, Session	Add, Delete, Resolve
Session	Application, Class, CurrentUser, Inbox, InfoStores, MAPIOBJECT*, Name, OperatingSystem, Outbox, Parent, Session, Version	AddressBook, GetAddressEntry, GetFolder, GetInfoStore, GetMessage, Logoff, Logon

* The MAPIOBJECT property is not available to Visual Basic applications. For more information, see the reference for the MAPIOBJECT property.

This reference is organized by object, consisting of a brief summary of each object that lists its properties and methods, followed by reference documentation for the individual properties and methods. The properties and methods are organized alphabetically.

To avoid duplication, the section All OLE Messaging Library Objects describes the properties that have the same meaning for all OLE Messaging Library objects. These are:

- **Application** property
- **Class** property
- **Parent** property
- **Session** property

See Also

Overview

Programming Tasks

All OLE Messaging Library Objects

All OLE Messaging Library objects contain the properties **Application**, **Class**, **Parent**, and **Session**. The **Application** and **Session** properties have the same values for all objects within a given session. The **Parent** property indicates the logical parent of the object. The **Class** property is an integer value that identifies the OLE Messaging Library object.

Note that for the Session object, the **Parent** and **Session** properties are assigned the value **Nothing**. The Session object represents the highest level in the OLE Messaging Library object hierarchy and has no parent.

To reduce duplication, the detailed reference for these common ("superclass") properties appears only once, in this section.

Many objects also have a **Type** property, but the **Type** property is not defined for all objects and its meaning varies depending on the object. The following table summarizes the **Type** property.

Object	**Description of the Type property**
AddressEntry	The messaging system: SMTP, Fax, and so on
Attachment	Attachment type: mapiFileData, mapiFileLink, or mapiOLE
Field	The field data type: vbInteger, vbLong, and so on
Message	The message class: IPM.Note, and so on
Recipient	Recipient type: To, Cc, or Bcc line

For detailed information about the **Type** property, see the reference documentation for each object.

The following table lists the properties that are common to all OLE Messaging Library objects and that have the same meaning for all objects.

Properties

Property name	**Type**	**Access**
Application	String	Read-only
Class	Long	Read-only
Parent	Object	Read-only
Session	Session object	Read-only

Application Property

The **Application** property returns the name of the active application, which is the OLE Messaging Library, "OLE/Messaging." Read-only.

Syntax

object.**Application**

Data Type

String

Remarks

The **Application** property always contains the string "OLE/Messaging."

Note that this behavior for the Microsoft OLE Messaging Library differs from other implementations of OLE Automation servers. Many OLE Automation servers are based on executables (files that take the extension .EXE) and return an object value. The OLE Messaging Library is based on the MAPI subsystem, which is implemented by dynamic link libraries (files that take the extension .DLL).

Example

```
' Function: Session_Application
' Purpose: Display the Application property of the Session object
' See documentation topic: Application property
Function Session_Application()
Dim objSession As Object
    ' error handling
    Set objSession = CreateObject("MAPI.Session")
    If Not objSession Is Nothing Then
        MsgBox "Session's Application property = " _
            & objSession.Application
    End If
    ' error handling
End Function
```

See Also

Version Property (Session Object)

Class Property

The **Class** property returns the OLE Messaging Library object. Read-only.

Syntax

object.**Class**

Data Type

Long

Remarks

The **Class** property contains a numeric constant that identifies the OLE Messaging Library object. The following values are defined.

OLE Messaging Library object	Class ID value	Value
AddressEntry	8	mapiAddressEntry
Attachment	5	mapiAttachment
Attachments (collection)	21	mapiAttachments
Field	6	mapiField
Fields (collection)	22	mapiFields
Folder	2	mapiFolder
Folders (collection)	18	mapiFolders
InfoStore	1	mapiInfoStore
InfoStores (collection)	17	mapiInfoStores
Message	3	mapiMsg
Messages (collection)	19	mapiMessages
Recipient	4	mapiRecipient
Recipients (collection)	20	mapiRecipients
Session	0	mapiSession

Example

```
' Function: Util_DecodeObjectClass
' Purpose: Decode the long integer class value,
'          show the related object name
' See documentation topic: Class property
Function Util_DecodeObjectClass(lClass As Long)
    ' error handling here...
    Select Case (lClass)
        Case mapiSession:
            MsgBox ("Session object; class = " & lClass)
        Case mapiMsg:
            MsgBox ("Message object; class = " & lClass)
    End Select
    ' error handling
End Function
```

```
' Function: TestDrv_Util_DecodeObjectClass
' Purpose: Call the utility function DecodeObjectClass for Class values
' See documentation topic: Class property
Function TestDrv_Util_DecodeObjectClass()
    ' error handling here...
    If objSession Is Nothing Then
        MsgBox "Need to set the Session object: Session->Logon"
        Exit Function
    End If
    ' expect type mapiSession = 0 for Session object
    Util_DecodeObjectClass (objSession.Class)
    Set objMessages = objSession.Inbox.Messages
    Set objOneMsg = objSession.Inbox.Messages.GetFirst
    If objOneMsg Is Nothing Then  ' empty inbox
        Exit Function
    End If
    ' expect type mapiMessage = 3 for Message object
    Util_DecodeObjectClass (objOneMsg.Class)
    ' error handling here...
End Function
```

See Also

All OLE Messaging Library Objects

Parent Property

The **Parent** property returns the parent of the object. Read-only.

Syntax

object.**Parent**

Data Type

Object

Remarks

The **Parent** property in the OLE Messaging Library currently returns the *immediate* parent of an object. For example, the immediate parent for each object is shown in the following table.

OLE Messaging Library object	**Immediate parent in object hierarchy**
AddressEntry (returned by Session.CurrentUser)	Session object
AddressEntry (all others)	Recipient object
Attachment	Attachments collection
Attachments (collection)	Message object
Field	Fields collection
Fields (collection)	Message or Folder object
Folder (Inbox, Outbox, Root folder)	Session object
Folder (all others)	Folders collection
Folders (collection)	Folder object
InfoStore	InfoStores collection
InfoStores (collection)	Session object
Message	Messages collection
Messages (collection)	Folder object
Recipient	Recipients collection
Recipients (collection)	Message object
Session	(not defined)

Note that the **Parent** property represents the *immediate* parent of the object, rather than the *logical* parent of the object. For example, a folder contains a Messages collection, which contains Message objects. The **Parent** property for a message is the immediate parent, the Messages collection, rather than the logical parent, the Folder object.

The Session object represents the highest level in the hierarchy of OLE Messaging Library objects and its **Parent** property is set to **Nothing**.

Example

This example displays the name of the parent Messages collection of a message:

```
' Function: Message_Parent
Function Message_Parent()
    ' error handling here
    If objOneMsg Is Nothing Then
        MsgBox "Need to select a message; see Messages->Get*"
        Exit Function
    End If
    ' Immediate parent of message is the messages collection
    MsgBox "Message immediate parent class = " & objOneMsg.Parent.Class
    ' error handling code
End Function
```

To get to the folder, you have to take the parent of the Messages collection object:

```
' Function: Messages_Parent
' Purpose: Display the Messages collection Parent class value
' See documentation topic: Parent property
Function Messages_Parent()
    ' error handling here...
    If objMessages Is Nothing Then
        MsgBox "No active messages collection"
        Exit Function
    End If
    MsgBox "Messages collection parent has class value: " & _
            objMessages.Parent.Class
    Exit Function
    ' error handling here...
End Function
```

See Also

Class Property

Folder Object

Message Object

Session Object

Session Property

The **Session** property returns the top-level Session object associated with the specified OLE Messaging Library object. Read-only.

Syntax

Set *objSession* = *object*.**Session**

Data Type

Object

Remarks

The Session object represents the highest level in the OLE Messaging Library object hierarchy. Its **Session** property is set to **Nothing**.

Example

```
' Function: Folder_Session
' Purpose: Access the Folder's Session property and display its name
' See documentation topic: Session property
Function Folder_Session()
Dim objSession2 As Object    ' session object to get the property
    ' error handling here...
    If objFolder Is Nothing Then
        MsgBox "No active folder; please select Session->Inbox"
        Exit Function
    End If
    Set objSession2 = objFolder.Session
    If objSession2 Is Nothing Then
        MsgBox "Unable to access Session property"
        Exit Function
    End If
    MsgBox "Folder's Session property name = " & objSession2.Name
    Set objSession2 = Nothing
    ' error handling here...
End Function
```

See Also

Session Object

AddressEntry Object

The AddressEntry object defines valid addressing information for a given messaging system. An address usually represents a person or process to which the messaging system can deliver messages.

The AddressEntry object is often used as a child object of the Recipient object. In this context, the AddressEntry object represents a copy of valid addressing information that is obtained from the address book during a call to the Recipient object's **Resolve** method. When you obtain the AddressEntry object in this context, you should not modify its properties.

Properties

Property name	Type	Access
Address	String	Read/write
Application	String	Read-only
Class	Long	Read-only
DisplayType	Long	Read-only
Fields	Fields collection object	Read-only
ID	String	Read-only
Name	String	Read/write
Parent	Object	Read-only
Session	Session object	Read-only
Type	String	Read/write

Methods

Method name	Parameters
Delete	(none)
Details	(optional) parentWindow as Long
Update	(none)

See Also

Recipient Object

Address Property (AddressEntry Object)

The **Address** property specifies the messaging address of an address list entry or message recipient. Read/write.

Syntax

objAddrEntry.**Address**

Data Type

String

Remarks

The AddressEntry object's **Address** property provides a unique string to identify a message recipient and routing information for messaging systems. The format of the address string is specific to each messaging system.

The AddressEntry object's **Address** and **Type** properties combine to form the *full address*, the complete messaging address that appears in the Recipient object's **Address** property. The Recipient object's **Address** property uses the following syntax:

TypeValue:*AddressValue*

The AddressEntry object's **Address** property corresponds to the MAPI property PR_EMAIL_ADDRESS.

Example

```
' Set up a series of object variables
' Set the Folder and Messages variables; from Session_Inbox
    Set objFolder = objSession.Inbox
    Set objMessages = objFolder.Messages
' Set the Message object variable; from Messages_GetFirst()
    Set objOneMsg = objMessages.GetFirst
' Set the Recipients collection variable; from Message_Recipients()
    Set objRecipColl = objOneMsg.Recipients
' Set the Recipient object variable; from Recipients_FirstItem()
    If 0 = objRecipColl.Count Then
        MsgBox "No recipients in the list"
        Exit Function
    End If
    iRecipCollIndex = 1
    Set objOneRecip = objRecipColl.Item(iRecipCollIndex)
' set the AddressEntry object variable; from Recipient_AddressEntry()
    Set objAddrEntry = objOneRecip.AddressEntry
' from Util_CompareFullAddressParts()
' display the values
    strMsg = "Recipient full address = " & objOneRecip.Address
    strMsg = strMsg & "; AddressEntry type = " & objAddrEntry.Type
    strMsg = strMsg & "; AddressEntry address = " & objAddrEntry.Address
    MsgBox strMsg
```

See Also

Address Property (Recipient Object)

Delete Method (AddressEntry Object)

The **Delete** method deletes the specified address from the address book.

Note The OLE Messaging Library supports the **Delete** method only for the personal address book.

Syntax

objAddressEntry.**Delete()**

Parameters

objAddressEntry
Required. The AddressEntry object.

Remarks

The **Delete** method fails if both the **Address** and **ID** properties are empty.

Example

```
Function AddressEntry_Delete()
    ' error handling here...
    If objAddrEntry Is Nothing Then
        MsgBox "must select an AddressEntry object"
        Exit Function
    End If
    objAddrEntry.Delete
    Set objAddrEntry = Nothing
    Exit Function
    ' error handling
End Function
```

See Also

Add Method (Recipients Collection)

Details Method (AddressEntry Object)

The **Details** method displays a dialog box that provides detailed information about an AddressEntry object.

Syntax

objAddressEntry.**Details([***parentWindow***])**

Parameters

objAddressEntry
    Required. The AddressEntry object.

parentWindow
    Optional. Long. The parent window handle for the details dialog box. A value of **0** (the default, when no value is supplied) specifies a modal dialog box.

Remarks

The dialog box always contains at least the display name and address of the address entry. For AddressEntry objects, the method fails if both the **Address** and **ID** properties are empty.

The *parentWindow* parameter value must be valid or the OLE Messaging Library will not display the dialog box.

The following methods can also invoke MAPI dialog boxes: **Delete** and **Details** methods (AddressEntry object), **Options** and **Send** methods (Message object), **Resolve** method (Recipient object and Recipients collection), **AddressBook** and **Logon** methods (Session object).

See Also

Update Method (AddressEntry Object)

DisplayType Property (AddressEntry Object)

The **DisplayType** property returns the type of the address entry. This property enables special processing based on the type, such as displaying an icon associated with that type. Read-only.

Syntax

objAddressEntry.**DisplayType**

Data Type

Long

Remarks

You can use the display type to sort or to filter address entries.

The following values are defined:

Displaytype value	**Description**
mapiUser	Local user
mapiDistList	Distribution list
mapiForum	Public folder
mapiAgent	Agent
mapiOrganization	Organization
mapiPrivateDistList	Private distribution list
mapiRemoteUser	Remote user

See Also

Add Method (Recipients Collection)

Fields Property (AddressEntry Object)

The **Fields** property returns a single field (a Field object) or a collection of fields (a Fields collection object) of the Folder object. Read-only.

Syntax

objAddressEntry.**Fields**

objAddressEntry.**Fields(***index***)**

objAddressEntry.**Fields(***proptag***)**

objAddressEntry.**Fields(***name***)**

index
  Short integer (less than or equal to 65535). Specifies the index within the collection.

proptag
  Long integer (greater than or equal to 65536). Specifies the property tag value for the MAPI property to be retrieved.

name
  String. Specifies the name of the custom MAPI property.

Data Type

Object

Remarks

Fields provide a generic access mechanism that allows Visual Basic and Visual C++ programmers to retrieve the value of any MAPI property using either a name or a MAPI property tag. To access using the property tag, use Folder.Fields.Item(*proptag*), where *proptag* is the 32-bit MAPI property tag associated with the MAPI property in question, such as PR_MESSAGE_CLASS. To access a Field object using a name, use Folder.Fields.Item(*name*), where *name* is a string that represents the custom property name.

See Also

Fields Collection

ID Property (AddressEntry Object)

The **ID** property returns the unique identifier of the object as a string. Read-only.

Syntax

objAddressEntry.**ID**

Data Type

String

Remarks

You can use the AddressEntry object's **ID** property as a parameter to the Recipient object's **Add** method.

MAPI systems assign a permanent, unique **ID** string when an object is created. These identifiers do not change from one MAPI session to another.

The **ID** property corresponds to the MAPI property PR_ENTRYID, converted to a string of hexadecimal characters.

Example

This example copies information from an AddressEntry object to a Recipient object:

```
' Function: Recipients_Add_EntryID
' Purpose: Add a new recipient to the collection using AddressEntry ID
Function Recipients_Add_EntryID()
Dim strID As String      ' ID from Message.Sender
Dim strName As String    ' name from Message.Sender
Dim objNewMsg As Object  ' new msg; set its recipient using ID
Dim objNewRecip As Object ' Recipient of new message, set from ID, name
    ' error handling
    strID = objOneMsg.Sender.Id   'Address Entry object ID
    strName = objOneMsg.Sender.Name
    Set objNewMsg = objSession.Outbox.Messages.Add
    If objNewMsg Is Nothing Then
        MsgBox "Could not create a new message"
        Exit Function
    End If
    objNewMsg.Subject = "Sample message from OLE Messaging Library"
    objNewMsg.Text = "Called Recipients.Add method w/ entryID parameter"
    Set objNewRecip = objNewMsg.Recipients.Add( _
                      entryID:=strID, _
                      Name:=strName)
    If objNewRecip Is Nothing Then
        MsgBox "Could not create a new recipient"
        Exit Function
    End If
    objNewMsg.Update
    objNewMsg.Send showDialog:=False
    MsgBox "Created a new message in the Outbox and sent it"
    Exit Function
    ' error handling
End Function
```

See Also

Add Method (Recipients Collection)

Name Property (AddressEntry Object)

The **Name** property returns or sets the display name or alias of the AddressEntry object as a string. Read/write.

Syntax

objAddressEntry.**Name**

Data Type

String

Remarks

The AddressEntry object is typically used as a copy of valid addressing information obtained from the address book after you have called the Recipient object's **Resolve** method.

When you obtain the AddressEntry object in this context, you should not modify its properties. To request resolution of a display name, use the Recipient object's **Name** property instead. Set the **Name** property and call the Recipient object's **Resolve** method.

The **Name** property corresponds to the MAPI property PR_DISPLAY_NAME.

Example

```
' for values of variables, see AddressEntry Address property
' Recipient and AddressEntry display names are the same
    strMsg = "Recipient name = " & objOneRecip.Name
    strMsg = strMsg & "; AddressEntry name = " & objAddrEntry.Name
    MsgBox strMsg
```

See Also

Recipient Object

Resolve Method (Recipient Object)

Using Addresses

Type Property (AddressEntry Object)

The **Type** property specifies the address type, such as SMTP, Fax, or X.400. Read/write.

Syntax

objAddressEntry.**Type**

Data Type

String

Remarks

The AddressEntry object's **Type** property specifies the address type. This is usually a tag referring to the messaging system that routes messages to this address, such as SMTP or Fax.

The AddressEntry object's **Address** and **Type** properties combine to form the *full address*, the complete messaging address that appears in the Recipient object's **Address** property. The Recipient **Address** uses the following syntax:

*TypeValue***:***AddressValue*

The **Type** property corresponds to the MAPI property PR_ADDRTYPE.

Example

See the example for the AddressEntry object's **Address** property.

See Also

Address Property (AddressEntry Object)

Address Property (Recipient Object)

Update Method (AddressEntry Object)

The **Update** method saves AddressEntry object changes in the MAPI system.

Syntax

objAddressEntry.**Update([***makePermanent, refreshObject***])**

Parameters

objAddressEntry
Required. The AddressEntry object.

makePermanent
Optional. Boolean. TRUE indicates that the property cache is flushed and all changes are committed in the underlying store. FALSE indicates that the property cache is flushed and not committed to the store. The default value is TRUE.

refreshObject
Optional. Boolean. TRUE indicates that the property cache is reloaded from the values in the underlying store. FALSE indicates that the property cache is not reloaded. The default value is FALSE.

Remarks

Changes to objects are not permanently saved in the MAPI system until you call the **Update** method with the *makePermanent* parameter set to TRUE.

For improved performance, the OLE Messaging Library caches property changes in private storage and updates either the object or the underlying store only when you explicitly request such an update. For efficiency, you should make only one call to **Update** with its *makePermanent* parameter set to TRUE.

The *makePermanent* and *refreshObject* parameters combine to cause the following changes:

	refreshObject = TRUE	**refreshObject = FALSE**
makePermanent = TRUE	Commit all changes, flush the cache, and reload the cache from the store.	Commit all changes and flush the cache.
makePermanent = FALSE	Flush the cache and reload the cache from the store.	Flush the cache.

Call **Update(FALSE, TRUE)** to flush the cache and then reload the values from the store.

Example

The following example changes the display name for the valid AddressEntry address:

```
' Function: AddressEntry_Update
' Purpose: Demonstrate the Update method
'    (Note: OLE Messaging Library only affects the PAB)
Function AddressEntry_Update()
Dim objRecipColl As Object   ' Recipients collection
Dim objNewRecip As Object    ' New recipient

    ' error handling omitted...
    Set objRecipColl = objSession.AddressBook
    If objRecipColl Is Nothing Then
        MsgBox "must select someone from the address book"
        Exit Function
    End If
    Set objNewRecip = objRecipColl.Item(1)
    With objNewRecip.AddressEntry
        .Name = .Name & " the Magnificent"
        .Type = "X.500"   ' you can also change the Type
        .Update
    End With
    MsgBox "Updated an address entry name: " & _
            objNewRecip.AddressEntry.Name
    Exit Function
    ' error handling omitted
End Function
```

See Also

Recipient Object

Attachment Object

The Attachment object represents a document that is an attachment of a message.

Properties

Property name	Type	Access
Application	String	Read-only
Class	Long	Read-only
Index	Long	Read-only
Name	String	Read/write
Parent	Object	Read-only
Position	Long	Read/write
Session	Session object	Read-only
Source	String	Read/write
Type	Long	Read/write

Methods

Method name	Parameters
Delete	(none)
ReadFromFile	fileName as String
WriteToFile	fileName as String

See Also

Attachments Collection

Delete Method (Attachment Object)

The **Delete** method deletes the attachment.

Syntax

objAttachment.**Delete()**

Parameters

objAttachment
Required. The Attachment object.

Remarks

The Attachment object is set to **Nothing** and is removed from memory, but the change is not permanent until you use the **Update**, **Send**, or **Delete** method on the parent Message object.

See Also

Delete Method (Attachments Collection)

Index Property (Attachment Object)

The **Index** property returns the index number for the Attachment object within the Attachments collection. Read-only.

Syntax

objAttachment.**Index**

Data Type

Long

Remarks

The **Index** property indicates the attachment's position within the parent Attachments collection.

An index value should not be considered to be a static value that remains constant for the duration of a session. The index can change whenever an update occurs to a parent object, such as the message or folder.

Example

```
Function Attachments_GetByIndex()
Dim lIndex As Long
Dim objOneAttach As Object  ' assume valid attachment
    ' set error handler here
    If objAttachColl Is Nothing Then
        MsgBox "must select an Attachments collection"
        Exit Function
    End If
    If 0 = objAttachColl.Count Then
        MsgBox "must select collection with 1 or more attachments"
        Exit Function
    End If
    ' prompt user for index; for now, use 1
    Set objOneAttach = objAttachColl.Item(1)
    MsgBox "Selected attachment 1: " & objOneAttach.Name
    lIndex = objOneAttach.Index  ' save index to retrieve this later
    ' ...get same attachment object later
    Set objOneAttach = objAttachColl.Item(lIndex)
    If objOneAttach Is Nothing Then
        MsgBox "Error, could not reselect the attachment"
    Else
        MsgBox "Reselected attachment " & lIndex & _
            " using index: " & objOneAttach.Name
    End If
    Exit Function
```

See Also

Attachments Collection

Item Property (Attachments Collection)

Name Property (Attachment Object)

The **Name** property returns or sets the display name of the Attachment object as a string. Read/write.

Syntax

objAttachment.**Name**

Data Type

String

Remarks

The **Name** property corresponds to the MAPI property PR_ATTACH_FILENAME.

Example

See the example for the Attachment object's **Index** property.

See Also

Attachment Object

Position Property (Attachment Object)

The **Position** property returns or sets the position of the attachment within the body text of the message. Read/write.

Syntax

objAttachment.**Position**

Data Type

Long

Remarks

The **Position** property is a long integer describing where the attachment should be placed in the message body. The attachment overwrites the character present at that position. Applications cannot place two attachments in the same location within a message, and attachments cannot be placed beyond the end of the message body.

The OLE Messaging Library does not manage rendering of the attachment within the message. The **Position** property simply provides directions for the rendering application.

The value **-1** indicates that the attachment is present, but is not rendered. The value **0** and other positive values indicate an index to the text character within the message.

The **Position** property corresponds to the MAPI property PR_RENDERING_POSITION.

Example

```
' from the function Attachments_Add()
    Set objAttach = objAttachColl.Add         ' add an attachment
    With objAttach
        .Type = mapiFileLink
        .Position = 0   ' place at beginning of message
        .Source = "\\server\bitmaps\honey.bmp"  ' UNC name
    End With
    ' must update the message to save the new info
    objOneMsg.Update   ' update the message
    MsgBox "Added an attachment of type mapiFileLink"
```

See Also

Add Method (Attachments Collection)

Text Property (Message Object)

ReadFromFile Method (Attachment Object)

The **ReadFromFile** method loads the contents of an attachment from a file.

Syntax

objAttachment.**ReadFromFile(***fileName***)**

Parameters

objAttachment
Required. The Attachment object.

fileName
Required. The full path and file name to read. For example, C:\DOCUMENT\BUDGET.XLS.

Remarks

The **ReadFromFile** method replaces the existing contents of the Attachment object, if any.

The **ReadFromFile** method operates slightly differently, depending on the value of the Attachment object's **Type** property. The following table describes its operation.

Attachment Type property	**ReadFromFile operation**
mapiFileData	Copies the contents of the specified file to the attachment.
mapiFileLink	Not supported; generates the run-time error MAPI_E_NO_SUPPORT.
mapiOLE	The specified file must be a valid OLE docfile, such as a file previously written by the **WriteToFile** method with a **mapiOLE** type setting.

Note that **ReadFromFile** does not support **mapiFileLink** attachments.

The term "OLE docfile" indicates that the file is written by an application such as Microsoft Word 6.0 or later that writes files using the OLE **IStorage** and **IStream** interfaces.

Note OLE Messaging Library version 1.0 does not support **ReadFromFile** for **mapiFileLink** types. This call generates the run-time error MAPI_E_NO_SUPPORT.

See Also

Add Method (Attachments Collection)

Type Property (Attachment Object)

WriteToFile Method (Attachment Object)

Source Property (Attachment Object)

The **Source** property returns or sets the full path name of the attachment data file for **mapiFileLink** attachments. The **Source** property returns or sets the OLE class name of the attachment for **mapiOLE** attachments. Read/write.

Syntax

objAttachment.**Source**

Data Type

String

Remarks

The OLE Messaging Library does not synchronize the **Source** property and the **ReadFromFile** method. For **mapiFileData** and **mapiOLE** attachments, when you change the **Source** property to indicate a different file, you must also explicitly call the **ReadFromFile** method to update the object data. Similarly, when you call **ReadFromFile** with data from a different file, you must change the **Source** property.

The value of the **Source** property depends on the value of the **Type** property, as described in the table below.

Type property	**Source property**
mapiFileData	Not used; contains an empty string.
mapiFileLink	Specifies a full path name in a universal naming convention (UNC) format, such as \\SALES\INFO\PRODUCTS\NEWS.DOC.
mapiOLE	Specifies the registered OLE class name of the attachment, such as "Word.Document" or "PowerPoint.Show."

The UNC format is suitable for sending attachments to recipients who have access to a common file server.

The **Source** property corresponds to the MAPI property PR_ATTACH_PATHNAME.

Example

```
' from the function Attachments_Add()
    Set objAttach = objAttachColl.Add         ' add an attachment
    With objAttach
        .Type = mapiFileLink
        .Position = 0   ' place at end of message
        .Source = "\\server\bitmaps\honey.bmp"  ' UNC name
    End With
    ' must update the message to save the new info
    objOneMsg.Update   ' update the message
    MsgBox "Added an attachment of type mapiFileLink"
```

See Also

Add Method (Attachments Collection)

Type Property (Attachment Object)

Type Property (Attachment Object)

The **Type** property describes the attachment type. Read/write.

Syntax

objAttachment.**Type**

Data Type

Long

Remarks

Three attachment types are supported:

Type property	Value	Description
mapiFileData	1	Attachment is the contents of a file. (Default value.)
mapiFileLink	2	Attachment is a link to a file.
mapiOLE	3	Attachment is an OLE object.

The value of the **Type** property determines the valid values for the **Source** property.

The Attachment object **Type** property corresponds to the MAPI property PR_ATTACH_METHOD.

Example

```
' from the function Attachments_Add()
    Set objAttach = objAttachColl.Add         ' add an attachment
    With objAttach
        .Type = mapiFileLink
        .Position = 0    ' place at end of message
        .Source = "\\server\bitmaps\honey.bmp"  ' UNC name
    End With
    ' must update the message to save the new info
    objOneMsg.Update    ' update the message
    MsgBox "Added an attachment of type mapiFileLink"
```

See Also

Add Method (Attachments Collection)

ReadFromFile Method (Attachment Object)

Source Property (Attachment Object)

WriteToFile Method (Attachment Object)

WriteToFile Method (Attachment Object)

The **WriteToFile** method saves the attachment to a file in the file system. Note that if the file already exists, this method overwrites it without warning.

Syntax

objAttachment.**WriteToFile(***file name***)**

Parameters

objAttachment
Required. The Attachment object.

file name
Required. String. The full path and file name for the saved attachment. For example, C:\DOCUMENT\BUDGET.XLS.

Remarks

The **WriteToFile** method overwrites the file without warning if a file of that name already exists. Your application should check for the existence of the file before calling **WriteToFile**.

The **WriteToFile** method operates slightly differently, depending on the value of the Attachment object's **Type** property. The following table describes its operation.

Attachment Type property	**WriteToFile operation**
mapiFileData	Copies the contents of the specified file to the attachment.
mapiFileLink	(Not supported)
mapiOLE	Writes the file as an OLE docfile format.

WriteToFile does not support **mapiFileLink** attachments.

See Also

ReadFromFile Method (Attachment Object)

Attachments Collection Object

The Attachments collection contains one or more Attachment objects.

The Attachments collection is considered a *small collection*, which means that it supports count and index values that let you access individual attachment objects through the **Item** property. The Attachments collection supports the Visual Basic **For Each** statement.

Properties

Property name	Type	Access
Application	String	Read-only
Class	Long	Read-only
Count	Long	Read-only
Item	Object	Read-only
Parent	Object	Read-only
Session	Session object	Read-only

Methods

Method name	Parameters
Add	(optional) name as String, (optional) position as Long, (optional) type as Long, (optional) source as String
Delete	(none)

See Also

Attachment Object

Object Collections

Add Method (Attachments Collection)

The **Add** method creates a new Attachment object in the Attachments collection.

Syntax

Set *objAttachment* = *objAttachColl*.**Add([** *name, position, type, source* **])**

Parameters

objAttachment
On successful return, contains the new Attachment object.

objAttachColl
Required. The Attachments collection object.

name
Optional. String. The display name of the attachment. The default value is an empty string. To allow a user to click on the attachment that appears in the message and activate an associated application, supply the full filename, including the file extension.

position
Optional. Long. The position of the attachment within the body text of the message. The default value is **0**.

type
Optional. Long. The type of attachment; either **mapiFileData**, **mapiFileLink**, or **mapiOLE**. The default value is **mapiFileData**.

source
Optional. String. The file name that contains the data for the attachment. The specified file name must be in the appropriate format for the attachment type, specified by the *type* parameter. The default value is an empty string. See the following remarks for a complete description.

Remarks

The **Add** method parameters correspond to the **Name**, **Position**, **Type**, and **Source** properties of the Attachment object. The *source* parameter is also closely related to the **ReadFromFile** method's *filename* parameter.

You can supply the data for the attachment at the same time that you add the attachment to the collection. The **Add** method operates differently, depending on the value of the *type* parameter. The following table describes its operation.

Value of *type* parameter	**Value of *source* parameter**
mapiFileData	Specifies a full path and file name that contains the data for the attachment. For example, C:\DOCUMENT\BUDGET.XLS. The data is read into the attachment.
mapiFileLink	Specifies a full path name in a universal naming convention (UNC) format, such as \\SALES\INFO\PRODUCTS\NEWS.DOC. The attachment is a link, so the **Add** method does not read the data.
mapiOLE	Specifies a full path and file name to a valid OLE docfile. For example, C:\DOCUMENT\BUDGET2.XLS. The data is read into the attachment.

When the *type* parameter has the value **mapiFileLink**, the *source* parameter is a full path name in a UNC format. This is suitable for sending attachments to recipients who have access to a common file server. Note that when you use the *type* **mapiFileLink**, the OLE Messaging Library does not validate the filename.

If you do not specify the *type* and *source* parameters when you call the **Add** method, you must later explicitly set these properties. For **mapiFileData** and **mapiOLE** types, you must also call the **ReadFromFile** method on the new Attachment object to load the attachment's content.

The **Index** of the new Attachment object equals the new **Count** of the Attachments collection. The attachment is saved in the MAPI system when you **Update** or **Send** the parent Message object.

See Also

Count Property (Attachments Collection)

ReadFromFile Method (Attachment Object)

Type Property (Attachment Object)

Count Property (Attachments Collection)

The **Count** property returns the number of Attachment objects in the collection. Read-only.

Syntax

objAttachColl.**Count**

Data Type

Long

Example

This example stores in an array the names of all Attachment objects in the collection:

```
' from the sample function, TstDrv_Util_SmallCollectionCount
' objAttachColl is an Attachments collection
x = Util_SmallCollectionCount(objAttachColl)

Function Util_SmallCollectionCount(objColl As Object)
Dim strItemName(100) As String      ' Names of objects in collection
Dim i As Integer                    ' loop counter
    On Error GoTo error_olemsg
    If objColl Is Nothing Then
        MsgBox "Must supply a valid collection object as a parameter"
        Exit Function
    End If
    If 0 = objColl.Count Then
        MsgBox "No items in the collection"
        Exit Function
    End If
    For i = 1 To objColl.Count Step 1
        strItemName(i) = objColl.Item(i).Name
        If 100 = i Then ' max size of string array
            Exit Function
        End If
    Next i
    ' error handling here...
End Function
```

See Also

Item Property (Attachments Collection)

Delete Method (Attachments Collection)

The **Delete** method deletes the entire Attachments collection.

Syntax

objAttachColl.**Delete()**

Parameters

objAttachColl
Required. The Attachments collection object.

Remarks

The object or collection is set to **Nothing** and it is removed from memory, but the change is not permanent until you use the **Update**, **Send**, or **Delete** method on the parent Message object that contained the deleted Attachments collection.

Be cautious using **Delete** with collections, since the method deletes all objects that are members of the collection.

See Also

Delete Method (Attachment Object)

Message Object

Item Property (Attachments Collection)

The **Item** property works like the accessor property to return a single item from a collection. Read-only.

Syntax

objAttachColl.**Item(***index***)**

index
An integer that ranges from 1 to *object*.**Count**, or a string that specifies the name of the object.

Data Type

Object

Remarks

The **Item** property works like the accessor property for small collections.

Example

```
' from Util_SmallCollectionCount(objColl As Object)
' This sample obtains the collection as a variable
' so it *must* use the Item property
Dim strItemName(100) as String
    ' error handling omitted from this fragment...
    For i = 1 To objColl.Count Step 1
        strItemName(i) = objColl.Item(i).Name
        If 100 = i Then ' max size of string array
            Exit Function
        End If
    Next i
```

See Also

Count Property (Attachments Collection)

Field Object

A Field object represents a property of an object. The Field object gives you the ability to add or access properties of a Folder, Message, or AddressEntry object.

Properties

Property name	Type	Access
Application	String	Read-only
Class	Long	Read-only
ID	Long	Read-only
Index	Long	Read-only
Name	String	Read-only
Parent	Object	Read-only
Session	Session object	Read-only
Type	Integer	Read/write
Value	Variant	Read/write

Methods

Method name	Parameters
Delete	(none)
ReadFromFile	fileName as String
WriteToFile	fileName as String

Remarks

You can add additional properties tailored for your specific application with the Fields collection object. Before adding a field for a Message, Folder, or AddressEntry object, please review the properties that are already provided by the OLE Messaging Library. Many of the most common attributes are already offered. For example, **Subject** and **Priority** are already defined as Message object properties.

Note that the predefined MAPI properties are unnamed when they are accessed in Field objects. For these MAPI properties, the **Name** property is an empty string.

The Field object also supports multivalued MAPI properties. The multivalued property appears to the Visual Basic application as a variant array; that is, you can use the **For... Next** statement to access individual array entries, as shown in the following sample program.

```
    Dim rgstr(0 To 9) As String
    ' Build array of values for MV prop
    For i = 0 To 9
        rgstr(i) = "String" + Str(i)
    Next

    ' Create MV field on the message. Note that we don't specify
    ' the array as third argument to Fields.Add, but add separately.
    Set f = msg.Fields.Add("FancyName", vbString + vbArray)
    f.Value = rgstr     ' Set value of the new field.
    ' Save/send the message, logoff, etc.

... ' code that reads the multivalued properties
    Dim rgstr As Variant
    Set f = msg.Fields.Item("FancyName") ' Get MV Field from the message
    rgstr = f.Value     ' Get array of values into a variant
    For i = LBound(rgret) To UBound(rgret)
        MsgBox rgret(i)
    Next i
```

For more information about MAPI properties, see the reference documentation for the Fields Collection and the *MAPI Programmer's Reference*.

See Also

Fields Collection

Delete Method (Field Object)

The **Delete** method deletes the user-defined or optional Field object.

Syntax

objField.**Delete**

Parameters

objField
Required. The Field object.

Remarks

This method only deletes user-defined fields and fields that represent properties considered optional by the underlying provider.

The object or collection is set to **Nothing** and it is removed from memory, but the change is not permanent until you use the **Update**, **Send**, or **Delete** method on the parent object (either the parent Folder or Message object) that contained the deleted Field object.

See Also

Add Method (Fields Collection)

ID Property (Field Object)

The **ID** property returns the unique identifier of the object as a long integer. Read-only.

Syntax

objField.**ID**

Data Type

Long

Remarks

The Field object **ID** property is unique among identifier properties supported in the OLE Messaging Library. The Field object identifier is a long integer that corresponds to a MAPI property tag value. All other identifier properties are hexadecimal strings.

Example

```
' The ID property is a long value, not a string
' fragment from the function Field_ID()
'   verify that objOneField is valid, then access
    MsgBox "ID is high-order word: 0x" & Hex(objOneField.Id)
```

See Also

Type Property (Field Object)

Value Property (Field Object)

Index Property (Field Object)

The **Index** property returns the index number of this Field object within the Fields collection. Read-only.

Syntax

objField.**Index**

Data Type

Long

Remarks

An index value should not be considered to be a static value that remains constant for the duration of a session. The index can change whenever an update occurs to a parent object, such as the message or folder.

Example

```
' set up a variable as an index to access a small collection
' fragment from the functions Fields_FirstItem, Fields_NextItem
    If objFieldsColl Is Nothing Then
        MsgBox "must first select a Fields collection"
        Exit Function
    End If
    If 0 = objFieldsColl.Count Then
        MsgBox "No fields in the collection"
        Exit Function
    End If
' Fragment from Fields_FirstItem
    iFieldsCollIndex = 1
    Set objOneField = objFieldsColl.Item(iFieldsCollIndex)
    ' verify that the Field object is valid...
' Fragment from Fields_NextItem
    If iFieldsCollIndex >= objFieldsColl.Count Then
        iFieldsCollIndex = objFieldsColl.Count
        MsgBox "Already at end of Fields collection"
        Exit Function
    End If
    iFieldsCollIndex = iFieldsCollIndex + 1
    Set objOneField = objFieldsColl.Item(iFieldsCollIndex)
    ' verify that the Field object is valid...
```

See Also

Count Property (Fields Collection)

Fields Collection

Name Property (Field Object)

The **Name** property returns the name of the field as a string. Read-only.

Syntax

objField.**Name**

Data Type

String

Remarks

The **Name** property is read-only. You set the name of the Field object at the time you create it, when you call the Fields collection's **Add** method.

Note that Field objects used to access MAPI properties do not have names. Names can appear only on the custom properties that you create. For more information, see the **Item** property documentation for the Fields collection.

Example

```
' fragment from Fields_Add
Dim objNewField As Object ' new Field object
    Set objNewField = objFieldsColl.Add( _
                      Name:="Keyword", _
                      Class:=vbString, _
                      Value:="Peru")
    If objNewField Is Nothing Then
        MsgBox "could not create new Field object"
        Exit Function
    End If
    cFields = objFieldsColl.Count
    MsgBox "new Fields collection count = " & cFields
' fragment from Field_Name; modified to use objNewField for active Field
    If "" = objNewField.Name Then
        MsgBox "Field has no name; ID = " & objNewField.Id
    Else
        MsgBox "Field name = " & objNewField.Name
    End If
```

See Also

Add Method (Fields Collection)

ReadFromFile Method (Field Object)

The **ReadFromFile** method loads the value of a string or binary field from the specified file.

Syntax

objField.**ReadFromFile(***fileName***)**

Parameters

objField
Required. The Field object.

fileName
Required. The full path and file name to read. For example, C:\DOCUMENT\BUDGET.XLS.

Remarks

The **ReadFromFile** method reads the string or binary value from the specified file name and stores it as the value of the Field object. It replaces any previously existing value for the field.

Note that **ReadFromFile** is not supported for simple types, such as Integer, Long, and Boolean. Visual Basic provides common functions to read and write these base types to and from files. The **ReadFromFile** method fails if the **Type** property of the Field object is not a string or binary type.

Note that some binary types are converted to a hexadecimal string format when they are stored as Field values. Comparison operations on the **Value** property and the actual contents of the file can return "not equal," even though the values are equivalent.

ReadFromFile returns MAPI_E_INTERFACE_NOT_SUPPORTED for Field objects obtained from a Folder object's Fields collection.

See Also

WriteToFile Method (Field Object)

Type Property (Field Object)

The **Type** property returns or sets the data type of the Field object. Read/write.

Syntax

objField.**Type**

Data Type

Integer

Remarks

The **Type** property specifies the data type of the Field object and determines the range of valid values that can be supplied for the **Value** property. You can set the value of the **Type** property by calling the Fields collection's **Add** method.

Valid data types are described in the following table.

Type	Description	Numeric value	OLE variant type	MAPI property type
vbNull	Null	1	VT_NULL	PT_NULL
vbInteger	Integer	2	VT_I2	PT_I2
vbLong	Long integer	3	VT_I4	PT_LONG
vbSingle	4-byte real (floating point)	4	VT_R4	PT_R4
vbDouble	Double (8-byte real)	5	VT_R8	PT_DOUBLE
vbCurrency	Currency	6	VT_CY	PT_CURRENCY PT_I8
vbDate	Date	7	VT_DATE	PT_APPTIME, PT_SYSTIME
vbString	String	8	VT_BSTR	PT_STRING8, PT_UNICODE, PT_CLSID, PT_BINARY
vbBoolean	Boolean	11	VT_BOOL	PT_BOOLEAN
vbDataObject	Data object	13	VT_UNKNOWN	PT_OBJECT
vbBlob	Blob	65	VT_BLOB	PT_BLOB

Note that the types **vbNull** and **vbDataObject** are not supported in version 1.0.

Note that MAPI stores all custom properties that represent date and time information using Greenwich

Mean Time (GMT). The OLE Messaging Library converts these properties so that the values appear to the user in local time.

Example

```
' Fragment from Fields_Add; uses the type "vbString"
    Set objNewField = objFieldsColl.Add( _
                      Name:="Keyword", _
                      Class:=vbString, _
                      Value:="Peru")
'  verify that objNewField is a valid Field object
' Fragment from Field_Type; display the integer type value
    MsgBox "Field type = " & objOneField.Type
```

See Also

Value Property (Field Object)

Value Property (Field Object)

The **Value** property returns or sets the value of the Field object. Read/write.

Syntax

objField.**Value**

Data Type

Variant

Remarks

The value of the Field object represents a value of the type specified by the **Type** property. For example, when the Field object has the **Type** property **vbBoolean**, the **Value** property can take the value TRUE or FALSE. When the Field object has the **Type** property **vbInteger**, the **Value** property can contain a short integer.

Example

```
' fragment from function Field_Type()
' after validating the Field object objOneField
    MsgBox "Field type = " & objOneField.Type
' fragment from function Field_Value()...
    MsgBox "Field value = " & objOneField.Value
```

See Also

ID Property (Field Object)

Type Property (Field Object)

WriteToFile Method (Field Object)

The **WriteToFile** method saves the field value to a file in the file system.

Syntax

objField.**WriteToFile(***fileName***)**

Parameters

objField
    Required. The Field object.

fileName
    Required. The full path and file name for the saved field; for example, C:
    \DOCUMENT\BUDGET.XLS.

Remarks

The **WriteToFile** method writes the string or binary value of the Field object to the specified file name. It overwrites any existing information in that file.

Note that **WriteToFile** is not supported for simple types, such as Integer, Long, and Boolean. Visual Basic provides common functions to read and write these base types to and from files. The **WriteToFile** method fails if the **Type** property of the Field is not a string or binary type.

Note that some binary types are represented in hexadecimal string format by the OLE Messaging Library and written in binary format. Comparison operations on the **Value** property and the actual contents of the file can return "not equal," even though the values are equivalent.

In addition, support for types can vary among providers. Not all providers support both the String and Binary property types.

See Also

ReadFromFile Method (Field Object)

Fields Collection Object

The Fields collection represents one or more Field objects. Field objects give you the ability to access properties of the object. These include the predefined underlying MAPI properties and your own custom user-defined properties.

Properties

Property name	Type	Access
Application	String	Read-only
Class	Long	Read-only
Count	Long	Read-only
Item	Object	Read-only
Parent	Object	Read-only
Session	Session object	Read-only

Methods

Method name	Parameters
Add	name as String, Class as Long, value as Variant, (optional) PropsetID as String
Delete	(none)
SetNamespace	PropsetID as String

Remarks

MAPI defines a set of properties with identifiers less than the value 0x8000. These are known as *unnamed properties* because they are usually accessed using the identifier rather than a name. You can access these MAPI-defined properties using the Fields collection. All MAPI properties are accessible except those of types PT_OBJECT and PT_CLSID.

You can also extend the properties available through MAPI by defining your own properties. These user-defined properties, defined using a name and automatically assigned an identifier value greater than 0x8000 by the OLE Messaging Library, are known as *named properties*. (C++ programmers can access the property name in the MAPI structure **MAPINAMEID** and convert it to the property tag value.)

All named properties are defined as part of a *property set*, which is also known in the context of the OLE Messaging Library as a *namespace*.

A property set is defined by a GUID, or unique identifier. The OLE Messaging Library represents this GUID as a string of hexadecimal characters. Such identifiers are usually referenced using a constant that starts with the characters PS_, such as PS_PUBLIC_STRINGS, the default property set for all properties created using the OLE Messaging Library.

You can also choose to organize your custom properties within their own semantic space by defining your own property set. The **Add** and **SetNamespace** methods and the **Item** property let you specify the property set identifier to be used for property access. When creating your own property set, you should be aware that MAPI reserves several property set identifiers for specific purposes. The following table lists reserved property sets:

Reserved Property Set	Description
PS_PUBLIC_STRINGS	Default property set for custom properties added using the OLE Messaging Library.

PS_MAPI	Allows providers to supply names for the unnamed properties (properties with identifiers less than 0x8000).
PS_ROUTING_DISPLAY_NAME	Display name properties that are translated between messaging domains.
PS_ROUTING_EMAIL_ADDRESSES	E-mail addresses that are translated between messaging domains.
PS_ROUTING_ADDRTYPE	E-mail address types that are translated between messaging domains.
PS_ROUTING_ENTRYID	Long-term entry identifiers that are translated between messaging domains.
PS_ROUTING_SEARCH_KEY	Search keys that are translated between messaging domains.

To create your own GUID that identifies your property set, you can either use the Win32 command-line utility UUIDGEN or you can call the OLE function **CoCreateGuid** to supply one for you, as demonstrated in the following code:

```
' declarations required for the call to CoCreateGuid
Type GUID
    Guid1 As Long
    Guid2 As Long
    Guid3 As Long
    Guid4 As Long
End Type
Declare Function CoCreateGuid Lib "OLE32.DLL" (pGuid As GUID) As Long
Global Const S_OK = 0
Dim strPropID as String
Dim lResult As Long
Dim lGuid As GUID

' call CoCreateGuid, then convert the result to a hex string
    lResult = CoCreateGuid(lGuid)
    If lResult = S_OK Then
        strPropID =  Hex$(lGuid.Guid1) & Hex$(lGuid.Guid2)
        strPropID = myHexString & Hex$(lGuid.Guid3)
        strPropID = myHexString & Hex$(lGuid.Guid4)
    Else
        ' ... handle error...
    End If
'
```

The Fields collection is considered a *small collection*, which means that it supports count and index values that let you access individual Field objects through the **Item** property. The Fields collection supports the Visual Basic **For Each** statement.

Note that MAPI stores all custom properties that represent date and time information using Greenwich Mean Time (GMT). The OLE Messaging Library converts these properties so that the values appear to the user in local time.

For more information about properties and property sets, see the topic, "About Named Properties," in the *MAPI Programmer's Reference*. For more information about UUIDGEN and **CoCreateGuid**, see the Win32 SDK documentation.

Example

To uniquely identify Field objects in the Fields collection, use the Field object's **Name** property or an index:

```
Set objOneField = objFolder.Fields.Item("BalanceDue")
Set objAnotherField = objMessage.Fields.Item("Keyword")
Set objThirdField = objMessage.Fields.Item(3)
```

See Also

SetNamespace Method (Fields Collection)

Object Collections

Add Method (Fields Collection)

The **Add** method creates a new Field object in the Fields collection.

Syntax

Set *objField* **=** *objFieldsColl*.**Add (***name, Class***[,** *value***] [,** *PropsetID***])**

Parameters

objField
On successful return, contains the new Field object.

objFieldsColl
Required. The Fields collection object.

name
Required. A string that represents the display name of the field.

Class
Required. A constant long integer that represents the data type for the field, such as string or integer. The *Class* parameter represents the same values as the Field object's **Type** property. The following types are allowed:

Type property	**Description**	**Numeric value**	**OLE variant type**
vbNull	Null	1	VT_NULL
vbInteger	Integer	2	VT_I2
vbLong	Long integer	3	VT_I4
vbSingle	4-byte real (floating point)	4	VT_R4
vbDouble	Double (8-byte real)	5	VT_R8
vbCurrency	Currency	6	VT_CY
vbDate	Date	7	VT_DATE
vbString	String	8	VT_BSTR
vbBoolean	Boolean	11	VT_BOOL
vbDataObject	Data object	13	VT_UNKNOWN
vbBlob	Blob	65	VT_BLOB

value
Optional. Variant. The value of the field, of the data type specified in the *type* parameter. When no value is supplied, no data is present for the object. You must make subsequent calls to the Field object's **ReadFromFile** method.

PropsetID
Optional. String. Specifies the identifier of the property set, represented as a string of hexadecimal characters. When the identifier is not present, the property is created within the default property set. The default property set is either the property set specified to the **SetNamespace** method, or the initial default property set value, PS_PUBLIC_STRINGS.

Remarks

Support for the **Add** method is provider-dependent.

The method parameters correspond to the **Name**, **Type**, and **Value** properties of the Field object.

The **Index** of the new Field object equals the new **Count** of the Fields collection. The field is saved in the MAPI system when you **Update** or **Send** the parent object.

When you use the **vbBlob** type, you supply the value in the form of a hexadecimal string that contains the hexadecimal representation of the bytes in the binary object (such as a hexadecimal dump of the object).

Note that MAPI stores all custom properties that represent date and time information using Greenwich Mean Time (GMT). The OLE Messaging Library converts these properties so that the values appear to the user in local time.

The OLE Messaging Library does not support MAPI properties of types PT_OBJECT and PT_CLSID. All others, however, are available through the Fields collection.

Example

```
' Fragment from Fields_Add; uses the type "vbString"
    Set objNewField = objFieldsColl.Add( _
                      Name:="Keyword", _
                      Class:=vbString, _
                      Value:="Peru")
'  verify that objNewField is a valid Field object
' Fragment from Field_Type; display the integer type value
    MsgBox "Field type = " & objOneField.Type
```

See Also

Count Property (Fields Collection)

Field Object

SetNamespace Method (Fields Collection)

Count Property (Fields Collection)

The **Count** property returns the number of Field objects in the collection. Read-only.

Syntax

objFieldsColl.**Count**

Data Type

Long

Example

This example maintains a global variable as an index into the small collection, and uses the **Count** property to check its validity:

```
' from Fields_NextItem
' iFieldsCollIndex is an integer used as an index
'   check for empty collection...
'   check index upper bound
    If iFieldsCollIndex >= objFieldsColl.Count Then
        iFieldsCollIndex = objFieldsColl.Count
        MsgBox "Already at end of Fields collection"
        Exit Function
    End If
    ' index is < count; can be incremented by 1
    iFieldsCollIndex = iFieldsCollIndex + 1
    Set objOneField = objFieldsColl.Item(iFieldsCollIndex)
    If objOneField Is Nothing Then
        MsgBox "Error, cannot get this Field object"
        Exit Function
    Else
        MsgBox "Selected field " & iFieldsCollIndex
    End If
```

See Also

Field Object

Delete Method (Fields Collection)

The **Delete** method deletes all user-defined fields of the Fields collection object.

Syntax

objFieldsColl.**Delete**

Parameters

objFieldsColl
Required. The Fields collection object.

Remarks

This method deletes all user-defined fields and all fields considered optional by the underlying provider.

The object or collection is set to **Nothing** and it is removed from memory, but the change is not permanent until you use the **Update**, **Send**, or **Delete** method on the parent Message object that contained the deleted Fields collection.

Be cautious using **Delete** with collections, since the method deletes all member objects within a collection.

See Also

Field Object

Item Property (Fields Collection)

The **Item** property works like the accessor property to return a single item from a collection. Read-only.

Syntax

objFieldsColl.**Item(***index***)**

objFieldsColl.**Item(***proptag***)**

objFieldsColl.**Item(***name* **[,** *propsetID***])**

objFieldsColl
   Required. Specifies the Fields collection object.

index
   Short integer (less than or equal to 65535; &Hffff). Specifies the index within the collection.

proptag
   Long integer (greater than or equal to 65536). Specifies the property tag value for the MAPI property to be retrieved.

name
   String. Specifies the name of the user-defined property.

propsetID
   Optional. String. Contains the unique identifier for the property set, represented as a string of hexadecimal characters. When *propsetID* is not supplied, the property set used for the access is the default property set value set by this collection's **SetNamespace** method, or the initial default property set value, PS_PUBLIC_STRINGS.

Data Type

Object

Remarks

The **Item** property in the Fields collection object allows access to the predefined MAPI properties and to your own custom user-defined properties.

The long value greater than 65,535 represents a *property tag*. A property tag is a 32-bit unsigned integer that contains the property identifier in its high-order 16 bits and the property type (its underlying data type) in the low-order 16 bits. All MAPI properties are accessible except those of types PT_OBJECT and PT_CLSID.

Several macros for C/C++ programmers are available in the MAPI SDK to help manipulate the property tag data structure. The macros PROP_TYPE and PROP_ID extract the property type and property identifer from the property tag. The macro PROP_TAG builds the property tag from the provided type and identifier components.

For example, you can use the following function to access a custom user-defined property using its property name:

```
' from the function Fields_ItemByName()
    ' error handling here...
    If objFieldsColl Is Nothing Then
        MsgBox "must first select Fields collection"
        Exit Function
    End If
    Set objOneField = objFieldsColl.Item("Keyword")
    If objOneField Is Nothing Then
        MsgBox "could not select Field object"
        Exit Function
    End If
    If "" = objOneField.Name Then
        MsgBox "Field has no name; ID = " & objOneField.Id
    Else
        MsgBox "Field name = " & objOneField.Name
    End If
```

You can also use the **Item** property to access MAPI properties. Note that the built-in MAPI properties are unnamed properties that can only be accessed using the numeric value. They cannot be accessed using a string that represents the name. The following example accesses the MAPI property PR_MESSAGE_CLASS:

```
' from the function Fields_Selector()
    ' ... error handling here
    ' you can provide a dialog to allow entry for MAPI proptags
    ' or select property names from a list; for now, hard-coded
    lValue = &h1a001e ' &H1a = PR_MESSAGE_CLASS;
                      ' &H001e = 30 = PT_STRING8
    ' high-order 16 bits is property id; low-order is property type
    Set objOneField = objFieldsColl.Item(lValue)
    If objOneField Is Nothing Then
        MsgBox "Could not get the Field using the value " & lValue
        Exit Function
    Else
        strMsg = "Used " & lValue & " to access the MAPI property "
        strMsg = strMsg & "PR_MESSAGE_CLASS: type = " & objOneField.Type
        strMsg = strMsg & "; value = " & objOneField.Value
        MsgBox strMsg
    End If
```

The OLE Messaging Library also supports multivalued MAPI properties.

You can also choose to access properties from other property sets, including your own, by either setting the *propsetID* parameter or by calling the **SetNamespace** method to set that property set's unique identifier. For more information, see the reference documentation for the **SetNamespace** method.

See Also

SetNamespace Method (Fields Collection)

Customizing a Folder or Message

Viewing MAPI Properties

Field Object

SetNamespace Method (Fields Collection)

The **SetNamespace** method selects the property set that is to be used for subsequent property accesses using the **Add** method and **Item** property.

Syntax

objFieldsColl.**SetNamespace** *PropsetID*

Parameters

objFieldsColl
: Required. The Fields collection object.

PropsetID
: Required. String. Contains a unique identifier that identifies the property set, represented as a string of hexadecimal characters. The *PropsetID* identifies the property set to be used for subsequent property accesses using the Field object and Fields collection. An empty string resets the default to the property set PS_PUBLIC_STRINGS.

Remarks

The initial default value for the property set is PS_PUBLIC_STRINGS. To create your own property set for your named properties, supply a unique property set identifier to **SetNamespace**. This property set then replaces PS_PUBLIC_STRINGS as the default property set for all subsequent named property accesses using this object. The default property set is used unless explicitly overridden by the optional *PropsetID* parameter. The value is set only for the current object; to continue using the same property set for all objects, you must call **SetNamespace** for each Message object.

To define a new property set, obtain a string that contains hexadecimal characters representing a unique identifier. You can obtain this identifier using either the Win32 command-line utility UUIDGEN or by calling the Win32 function **CoCreateGuid**.

See Also

Fields Collection

Folder Object

The Folder object represents a folder or container within the MAPI system. Folders can contain subfolders and messages.

Properties

Property name	Type	Access
Application	String	Read-only
Class	Long	Read-only
Fields	Fields collection object	Read-only
FolderID	String	Read-only
Folders	Folders collection object	Read-only
ID	String	Read-only
MAPIOBJECT	Object	Read/write (Note: Not available to Visual Basic applications.)
Messages	Messages collection object	Read-only
Name	String	Read/write
Parent	Object	Read-only
Session	Session object	Read-only
StoreID	String	Read-only

Methods

Method name	Parameters
Update	(none)

Remarks

Changes to the folder are not saved by MAPI until you call the **Update** method.

The **ID** property is unique and read-only. MAPI assigns a unique identifier when the Folder object is created. Its value does not change.

Note that the OLE Messaging Library does not support methods to allow you to create new folders.

See Also

Folders Collection

Fields Property (Folder Object)

The **Fields** property returns a single field (a Field object) or a collection of fields (a Fields collection object) of the Folder object. Read-only.

Syntax

objFolder.**Fields**

objFolder.**Fields(***index***)**

objFolder.**Fields(***proptag***)**

objFolder.**Fields(***name***)**

index
Short integer (less than or equal to 65535). Specifies the index within the collection.

proptag
Long integer (greater than or equal to 65536). Specifies the property tag value for the MAPI property to be retrieved.

name
String. Specifies the name of the custom MAPI property.

Data Type

Object

Remarks

Fields provide a generic access mechanism that allows Visual Basic programmers to retrieve the value of any property associated with the Folder object using either a name or a property tag. To access using the property tag, use Folder.Fields.Item(*proptag*), where *proptag* is the 32-bit MAPI property tag associated with the property in question, such as PR_MESSAGE_CLASS. To access a Field object using a name, use Folder.Fields.Item(*name*), where name is a string that represents the custom property name.

Example

This example displays the field name or identifier value of all Field objects within the collection:

```
' many properties are MAPI properties and have no names
' for those properties, display the ID
' fragment from Field_Name
' assume objFieldColl, objOneField are valid objects
For i = 1 to objFieldColl.Count Step 1
    Set objOneField = objFieldColl.Index(i)
    If "" = objOneField.Name Then
        MsgBox "Field has no name; ID = " & objOneField.Id
    Else
        MsgBox "Field name = " & objOneField.Name
    End If
Next i
```

See Also

Field Object

FolderID Property (Folder Object)

The **FolderID** property returns the unique identifier of this subfolder's parent folder as a string. Read-only.

Syntax

objFolder.**FolderID**

Data Type

String

Remarks

MAPI systems assign a permanent, unique identifier string when an object is created. These identifiers do not change from one MAPI session to another.

Note that MAPI systems do not require identifier values to be binary comparable. Accordingly, two identifier values can be different, yet refer to the same object. You can compare identifiers using the MAPI **CompareEntryIDs** method. For more information, see the *MAPI Programmer's Reference*.

The **FolderID** property corresponds to the MAPI property PR_PARENT_ENTRYID, converted to a string of hexadecimal characters.

Example

```
'          fragment from Session_Inbox
    Set objFolder = objSession.Inbox
'          fragment from Folder_FolderID
    strFolderID = objFolder.FolderID
    MsgBox "Parent Folder ID = " & strFolderID
'          can later restore using objSession.GetFolder(strFolderID)
'          fragment from Session_GetFolder
    If "" = strFolderID Then
        MsgBox ("Must first set folder ID variable; see Folder->ID")
        Exit Function
    End If
    Set objFolder = objSession.GetFolder(strFolderID)
    ' error checking here...
```

See Also

ID Property (Folder Object)

Folders Property (Folder Object)

The **Folders** property specifies a collection of subfolders within the parent folder. Read-only.

Syntax

objFolder.**Folders**

Data Type

Object

Example

This example lists all the names of all subfolders of the specified folder:

```
' fragment from Session_Inbox
    Set objFolder = objSession.Inbox
' from TstDrv_Util_ListFolders
   If 2 = objFolder.Class Then  ' verify Folder object
        x = Util_ListFolders(objFolder)  ' use current global folder
   End If

' complete function for Util_ListFolders
Function Util_ListFolders(objParentFolder As Object)
Dim objFoldersColl As Object ' the child Folders collection
Dim objOneSubfolder As Object 'a single Folder object
    ' set up error handler here
    If Not objParentFolder Is Nothing Then
        MsgBox ("Folder name = " & objParentFolder.Name)
        Set objFoldersColl = objParentFolder.Folders
        If Not objFoldersColl Is Nothing Then ' loop through all
            Set objOneSubfolder = objFoldersColl.GetFirst
            While Not objOneSubfolder Is Nothing
                x = Util_ListFolders(objOneSubfolder)
                Set objOneSubfolder = objFoldersColl.GetNext
            Wend
        End If
    End If
    Exit Function
    ' error handler here
End Function
```

See Also

Folders Collection

ID Property (Folder Object)

The **ID** property returns the unique identifier of this Folder object as a string. Read-only.

Syntax

objFolder.**ID**

Data Type

String

Remarks

MAPI systems assign a permanent, unique identifier string when an object is created. These identifiers do not change from one MAPI session to another.

The **ID** property corresponds to the MAPI property PR_ENTRYID, converted to a string of hexadecimal characters.

Example

```
' save the current ID and restore using Session.GetFolder
'         fragment from Session_Inbox
    Set objFolder = objSession.Inbox
'         fragment from Folder_FolderID
    strFolderID = objFolder.ID
    MsgBox "Current Folder ID = " & strFolderID
'         can later restore using objSession.GetFolder(strFolderID)
'         fragment from Session_GetFolder
    If "" = strFolderID Then
        MsgBox ("Must first set folder ID variable; see Folder->ID")
        Exit Function
    End If
    Set objFolder = objSession.GetFolder(strFolderID)
    ' error checking here...
```

See Also

Folders Property (Folder Object)

GetFolder Method (Session Object)

MAPIOBJECT Property (Folder Object)

The **MAPIOBJECT** property returns an **IUnknown** pointer to this Folder object. Not available to Visual Basic applications. Read/write.

Syntax

objFolder.**MAPIOBJECT**

Data Type

Variant (VT_UNKNOWN)

Remarks

The **MAPIOBJECT** property is not available to Visual Basic programs. It is available only to C/C++ programs that use the OLE Messaging Library. The **MAPIOBJECT** property is an **IUnknown** object, which is not supported by Visual Basic. Visual Basic supports **IDispatch** objects. For more information, see the Microsoft *OLE Programmer's Reference*.

See Also

Introduction to OLE Automation

How Programmable Objects Work

Messages Property (Folder Object)

The **Messages** property returns a Messages collection object within the folder. Read-only.

Syntax

objFolder.**Messages**

Data Type

Object

Example

```
' from the QuickStart sample
' use the Messages property of the Outbox folder
    Set objSession = CreateObject("MAPI.Session")
    objSession.Logon
    Set objMessage = objSession.Outbox.Messages.Add
```

See Also

ID Property (Message Object)

Message Object

Messages Collection

Name Property (Folder Object)

The **Name** property returns or sets the name of the Folder object as a string. Read/write.

Syntax

objFolder.**Name**

Data Type

String

Remarks

The **Name** property corresponds to the MAPI property PR_DISPLAY_NAME.

Example

```
Dim objFolder As Object  ' assume valid folder
MsgBox "Folder name = " & objFolder.Name
```

See Also

GetFolder Method (Session Object)

StoreID Property (Folder Object)

The **StoreID** property returns the identifier of the Store object in which this Folder object resides. Read-only.

Syntax

objFolder.**StoreID**

Data Type

String

Remarks

The **StoreID** property corresponds to the MAPI property PR_STORE_ENTRYID, converted to a string of hexadecimal characters.

Note that MAPI systems do not require identifier values to be binary comparable. Accordingly, two identifier values can be different, yet refer to the same object. You can compare identifiers using the MAPI method **CompareEntryIDs**. For more information, see the *MAPI Programmer's Reference*.

Example

```
' from the sample function Folder_ID
    strFolderID = objFolder.ID
' from the sample function Folder_StoreID
    strFolderStoreID = objFolder.storeID
'  can use these IDs with Session.GetFolder()
'  from the sample function Session_GetFolder
    Set objFolder = objSession.GetFolder(folderID:=strFolderID, _
                                   storeID:=strFolderStoreID)
```

See Also

GetFolder Method (Session Object)

Update Method (Folder Object)

The **Update** method saves the folder in the MAPI system.

Syntax

objFolder.**Update([***makePermanent***,** *refreshObject***])**

Parameters

objFolder
Required. The Folder object.

makePermanent
Optional. Boolean. TRUE indicates that the property cache is flushed and all changes are committed in the underlying store. FALSE indicates that the property cache is flushed and not committed to the store. The default value is TRUE.

refreshObject
Optional. Boolean. TRUE indicates that the property cache is reloaded from the values in the underlying store. FALSE indicates that the property cache is not reloaded. The default value is FALSE.

Remarks

Changes to Folder objects are not permanently saved in the MAPI system until you call the **Update** method with the *makePermanent* parameter set to TRUE.

For improved performance, the OLE Messaging Library caches property changes in private storage and updates either the object or the underlying store only when you explicitly request such an update. For efficiency, you should make only one call to **Update** with its *makePermanent* parameter set to TRUE.

The *makePermanent* and *refreshObject* parameters combine to cause the following changes:

	refreshObject = TRUE	**refreshObject = FALSE**
makePermanent = TRUE	Commit all changes, flush the cache, and reload the cache from the store.	Commit all changes and flush the cache.
makePermanent = FALSE	Flush the cache and reload the cache from the store.	Flush the cache.

Call **Update(FALSE, TRUE)** to flush the cache and then reload the values from the store.

See Also

Folders Collection

Folders Collection Object

The Folders collection contains one or more Folder objects.

The Folders collection is considered a *large collection*, which means that you must use a Folder object identifier value or the **Get** methods to access individual Folder objects within the collection.

Properties

Property name	Type	Access
Application	String	Read-only
Class	Long	Read-only
Parent	Object	Read-only
Session	Session object	Read-only

Methods

Method name	Parameters
GetFirst	(none)
GetLast	(none)
GetNext	(none)
GetPrevious	(none)

Remarks

Large collections, such as the Folders collection, do not maintain a count of the number of objects in the collection. Instead you must use the **GetFirst**, **GetNext**, **GetLast**, and **GetPrevious** methods to access individual items in the collection. You can also access a specific folder by using the Session object's **GetFolder** method.

Example

To refer to a unique Folder object within the Folders collection, use the collection's **GetFirst** and **GetNext** methods or use the **FolderID** value as the index.

The following code sample demonstrates the **Get** methods. The sample assumes that you have three subfolders within your Inbox and three subfolders within your Outbox. After this code runs, the three folders in the Inbox are named Blue, Red, and Orange (in that order), and the three folders in the Outbox are named Gold, Purple, and Yellow (in that order).

```
Dim objSession As Object
Dim objMessage As Object
Dim objFolder As Object

Set objSession = CreateObject("MAPI.Session")
objSession.Logon "User", "", True
With objSession.Inbox.Folders
   Set objFolder = .GetFirst
   objFolder.Name = "Blue"
   Set objFolder = .GetNext
   objFolder.Name = "Red"
   Set objFolder = .GetLast
   objFolder.Name = "Orange"
End With
With objSession.Outbox.Folders
   Set objFolder = .GetFirst
   objFolder.Name = "Gold"
   Set objFolder = .GetNext
   objFolder.Name = "Purple"
   Set objFolder = .GetLast
   objFolder.Name = "Yellow"
End With
objSession.Logoff
```

See Also

Object Collections

GetFirst Method (Folders Collection)

The **GetFirst** method returns the first object in the Folders collection. Returns **Nothing** if no first object exists.

Syntax

Set *objFolder* = *objFoldersColl*.**GetFirst()**

Parameters

objFolder
  On successful return, represents the first Folder object in the collection.

objFoldersColl
  Required. The Folders collection object.

Remarks

The **Get** methods are similar to the **Find** and **Move** methods that are used with Microsoft Access databases. The **Get** methods take a different name from these methods because they use a different syntax.

See Also

Type Property (Message Object)

Folder Object

GetLast Method (Folders Collection)

The **GetLast** method returns the last object in the Folders collection. Returns **Nothing** if no last object exists.

Syntax

Set *objFolder* = *objFoldersColl*.**GetLast()**

Parameters

objFolder
 On successful return, represents the last Folder object in the collection.

objFoldersColl
 Required. The Folders collection object.

Remarks

The **Get** methods are similar to the **Find** and **Move** methods that are used with Microsoft Access databases. The **Get** methods take a different name from these methods because they use a different syntax.

See Also

Folder Object

GetNext Method (Folders Collection)

The **GetNext** method returns the next object in the Folders collection. Returns **Nothing** if no next object exists, or when already positioned at the end of the collection.

Syntax

Set *objFolder* = *objFoldersColl*.**GetNext()**

Parameters

objFolder
    On successful return, represents the next Folder object in the collection.
objFoldersColl
    Required. The Folders collection object.

Remarks

The **Get** methods are similar to the **Find** and **Move** methods that are used with Microsoft Access databases. The **Get** methods take a different name from these methods because they use a different syntax.

See Also

Folder Object

GetPrevious Method (Folders Collection)

The **GetPrevious** method returns the previous object in the Folders collection. Returns **Nothing** if no previous object exists, or when already positioned at the first folder in the collection.

Syntax

Set *objFolder* = *objFoldersColl*.**GetPrevious()**

Parameters

objFolder
    On successful return, represents the previous Folder object in the collection.

objFoldersColl
    Required. The Folders collection object.

Remarks

The **Get** methods are similar to the **Find** and **Move** methods that are used with Microsoft Access databases. The **Get** methods take a different name from these methods because they use a different syntax.

See Also

Folder Object

InfoStore Object

The InfoStore object provides access to the root folder for that information store.

Properties

Property name	Type	Access
Application	String	Read-only
Class	Long	Read-only
ID	String	Read-only
Index	Long	Read-only
Name	String	Read-only
Parent	InfoStores collection object	Read-only
ProviderName	String	Read-only
RootFolder	Folder object	Read-only
Session	Session object	Read-only

Methods

(None.)

Remarks

The store provides access to its folder hierarchy through the **RootFolder** property. This property contains the Folder object that represents the root of the IPM tree.

You can also retrieve an InfoStore object with a known identifier by calling the Session object's **GetInfoStore** method.

Example

```
Dim objInfoStore as Object
Set objInfoStore = objSession.InfoStores.Item(1)
Set objFolder = objInfoStore.RootFolder
```

See Also

GetInfoStore Method (Session Object)

InfoStores Collection

ID Property (InfoStore Object)

The **ID** property returns the unique identifier of this InfoStore object as a string. Read-only.

Syntax

objInfoStore.**ID**

Data Type

String

Remarks

MAPI systems assign a permanent, unique identifier string when an object is created. These identifiers do not change from one MAPI session to another. The InfoStore identifier can be used in subsequent calls to the Session object's **GetInfoStore** method.

The **ID** property corresponds to the MAPI property PR_ENTRYID, converted to a string of hexadecimal characters.

Example

```
Dim strInfoStoreID as String  ' hex string version of ID
Dim objInfoStore as Object    ' assume valid
    strInfoStoreID = objInfoStore.Id   ' global variable
    MsgBox "InfoStore ID = " & strInfoStoreID
'...this ID can be used as the parameter to the Session method
    Set objInfoStore = objSession.GetInfoStore(strInfoStoreID)
```

See Also

InfoStores Collection

Index Property (InfoStore Object)

The **Index** property returns the index number for the InfoStore object within the parent InfoStores collection. Read-only.

Syntax

objInfoStore.**Index**

Data Type

Long

Remarks

The **Index** property indicates this object's position within the parent collection.

Example

```
Function InfoStoresGetByIndex()
Dim lIndex As Long
Dim objOneInfoStore As Object  ' assume valid InfoStore
    ' set error handler here
    If objInfoStoreColl Is Nothing Then
        MsgBox "must select an InfoStores collection"
        Exit Function
    End If
    If 0 = objInfoStoreColl.Count Then
        MsgBox "must select collection with 1 or more InfoStores"
        Exit Function
    End If
    ' prompt user for index; for now, use 1
    Set objOneInfoStore = objInfoStoreColl.Item(1)
    MsgBox "Selected InfoStore 1: " & objOneInfoStore.Name
    lIndex = objOneInfoStore.Index  ' save index to retrieve this later
    ' ...get same InfoStore object later
    Set objOneInfoStore = objInfoStoreColl.Item(lIndex)
    If objOneInfoStore Is Nothing Then
        MsgBox "Error, could not reselect the InfoStore"
    Else
        MsgBox "Reselected InfoStore " & lIndex & _
            " using index: " & objOneInfoStore.Name
    End If
    Exit Function
```

See Also

InfoStores Collection

Item Property (InfoStores Collection)

Name Property (InfoStore Object)

The **Name** property returns the name of the InfoStore object as a string. Read-only. The string "Public Folders" is the name of the InfoStore object that contains the public folders. Read-only.

Syntax

objInfoStore.**Name**

Data Type

String

Remarks

The **Name** property corresponds to the MAPI property PR_DISPLAY_NAME.

Example

```
Dim objInfoStore As Object  ' assume valid InfoStore object
MsgBox "Store name = " & objInfoStore.Name
```

See Also

GetInfoStore Method (Session Object)

ProviderName Property (InfoStore Object)

The **ProviderName** property returns the name of the InfoStore provider as a string. Read-only.

Syntax

objInfoStore.**ProviderName**

Data Type

String

Remarks

The **ProviderName** property corresponds to the MAPI property PR_PROVIDER_DISPLAY.

Example

```
Dim objInfoStore As Object  ' assume valid InfoStore object
MsgBox "Store name = " & objInfoStore.Name
```

See Also

GetInfoStore Method (Session Object)

RootFolder Property (InfoStore Object)

The **RootFolder** property returns a folder object representing the root of the IPM tree for this InfoStore object. Read-only.

Syntax

set *objFolder* **=** *objInfoStore*.**RootFolder**

Data Type

Object (Folder object)

Remarks

The **RootFolder** property provides a convenient way to get to this commonly used Folder object.

In addition to the general ability to navigate through the formal collection and object hierarchy, the OLE Messaging Library supports properties that allow your application to directly access the most common folder objects:

- The IPM subtree
- Inbox
- Outbox

Some message stores also support a way to obtain the root folder. For more information, see the documentation for the Session object's **GetFolder** method.

Example

```
' from InfoStores_RootFolder
    If objInfoStore Is Nothing Then
        MsgBox "must first select an InfoStore object"
        Exit Function
    End If
    Set objFolder = objInfoStore.RootFolder
    If objFolder Is Nothing Then
        MsgBox "Unable to retrieve InfoStore root folder"
        Set objMessages = Nothing
        Exit Function
    End If
    If objFolder.Name = "" Then
        MsgBox "Folder set to folder with no name, ID = " & objFolder.Id
    Else
        MsgBox "Folder set to: " & objFolder.Name
    End If
    Set objMessages = objFolder.Messages
    Exit Function
```

See Also

Folder Object

GetFolder Method (Session Object)

Inbox Property (Session Object)

Outbox Property (Session Object)

InfoStores Collection Object

The InfoStores collection provides access to all store objects available to this session. Each store object in turn offers access to the root of the folder hierarchy in that store. This is used primarily to obtain access to the public folders.

Properties

Property name	Type	Access
Application	String	Read-only
Class	Long	Read-only
Count	Long	Read-only
Item	InfoStore object	Read-only
Parent	Session object	Read-only
Session	Session object	Read-only

Methods

(None.)

Remarks

The InfoStores collection is considered a *small collection*, which means that it supports count and index values that let you access individual InfoStore objects through the **Item** property. The InfoStores collection supports the Visual Basic **For Each** statement.

The OLE Messaging Library does not support methods to add or remove store objects from the collection.

In general, you cannot assume that the InfoStore object's **Name** properties are unique. This means that you cannot rely on the name to retrieve the store from the collection. However, you can iterate through all objects in the collection using the InfoStores collection object's **Item** property, and then examine properties of the individual InfoStore objects. You can also rely on the InfoStore object's **ID** property, which is guaranteed to be unique.

See Also

InfoStore Object

Count Property (InfoStores Collection)

The **Count** property returns the number of InfoStore objects in the collection. Read-only.

Syntax

objInfoStoresColl.**Count**

Data Type

Long

Example

This example maintains a global variable as an index into the small collection, and uses the **Count** property to check its validity:

```
' from InfoStores_NextItem
' iInfoStoresCollIndex is an integer used as an index
'   check for empty collection...
'   check index upper bound
    If iInfoStoresCollIndex >= objInfoStoresColl.Count Then
        iInfoStoresCollIndex = objInfoStoresColl.Count
        MsgBox "Already at end of InfoStores collection"
        Exit Function
    End If
    ' index is < count; can be incremented by 1
    iInfoStoresCollIndex = iInfoStoresCollIndex + 1
    Set objInfoStore = objInfoStoresColl.Item(iInfoStoresCollIndex)
    If objInfoStore Is Nothing Then
        MsgBox "Error, cannot get this InfoStore object"
        Exit Function
    Else
        MsgBox "Selected InfoStore " & iInfoStoresCollIndex
    End If
```

See Also

InfoStore Object

Item Property (InfoStores Collection)

The **Item** property works like the accessor property to return a single item from a collection. Read-only.

Syntax

objInfoStoresColl.**Item(***index***)**

objInfoStoresColl
  Required. Specifies the InfoStores collection object.

index
  Contains a long integer.

Data Type

Object

Remarks

The **Item** property returns an InfoStore object.

Example

```
' from InfoStores_NextItem
' iInfoStoresCollIndex is an integer used as an index
'   check for empty collection...
'   check index upper bound
    If iInfoStoresCollIndex >= objInfoStoresColl.Count Then
        iInfoStoresCollIndex = objInfoStoresColl.Count
        MsgBox "Already at end of InfoStores collection"
        Exit Function
    End If
    ' index is < count; can be incremented by 1
    iInfoStoresCollIndex = iInfoStoresCollIndex + 1
    Set objInfoStore = objInfoStoresColl.Item(iInfoStoresCollIndex)
    If objInfoStore Is Nothing Then
        MsgBox "Error, cannot get this InfoStore object"
        Exit Function
    Else
        MsgBox "Selected InfoStore " & iInfoStoresCollIndex
    End If
```

See Also

InfoStore Object

Message Object

The Message object represents a single message, item, document, or form in a folder.

Properties

Property name	Type	Access
Application	String	Read-only
Attachments	Attachments collection object	Read-only
Class	Long	Read-only
Conversation	(Obsolete. Do not use.)	Read/write
ConversationIndex	String	Read/write
ConversationTopic	String	Read/write
DeliveryReceipt	Boolean	Read/write
Encrypted	Boolean	Read/write
Fields	Fields collection object	Read-only
FolderID	String	Read-only
ID	String	Read-only
Importance	Long	Read/write
MAPIOBJECT	(Not for use with Visual Basic.)	Read/write (Note: Not available to Visual Basic applications.)
Parent	Object	Read-only
ReadReceipt	Boolean	Read/write
Recipients	Recipients object	Read-only
Sender	AddressEntry object	Read-only
Sent	Boolean	Read/write
Session	Session object	Read-only
Signed	Boolean	Read/write
Size	Long	Read-only
StoreID	String	Read-only
Subject	String	Read/write
Submitted	Boolean	Read/write
Text	String	Read/write
TimeReceived	Variant (Date/Time)	Read/write
TimeSent	Variant (Date/Time)	Read/write
Type	String	Read/write
Unread	Boolean	Read/write

Methods

Method name	Parameters
Delete	(none)
Options	(optional) parentWindow as Long

Send	(optional) saveCopy as Boolean, (optional) showDialog as Boolean, (optional) parentWindow as Long
Update	(none)

Remarks

The Message object is considered a high-level object.

Visual Basic programmers can create new message objects using the Messages collection's **Add** method.

C/C++ programmers can create new message objects using the OLE function **CoCreateInstance**.

See Also

GetMessage Method (Session Object)

Messages Collection

Messages Property (Folder Object)

Attachments Property (Message Object)

The **Attachments** property returns a single Attachment object or an Attachments collection. Read-only.

Syntax

Set *objAttachColl* **=** *objMessage.***Attachments**

Set *objOneAttach* = *objMessage.***Attachments(***index***)**

objAttachColl
Object. An Attachments collection object.

objMessage
Object. The Message object.

objOneAttach
Object. A single Attachment object.

index
Long. Specifies the number of the attachment within the Attachments collection. Ranges from 1 to the value specified by the Attachments collection's **Count** property.

Example

This example uses the **Attachments** property to retrieve an attachment for the message:

```
' from the sample function Message_Attachments
    Set objAttachColl = objOneMsg.Attachments
    If objAttachColl Is Nothing Then
        MsgBox "unable to set Attachments collection"
        Exit Function
    Else
        MsgBox "Attachments count for this msg: " & objAttachColl.Count
        iAttachCollIndex = 0     ' reset global index variable
    End If
' from the sample function Attachments_FirstItem
    iAttachCollIndex = 1
    Set objAttach = objAttachColl.Item(iAttachCollIndex)
```

See Also

Attachment Object

Attachments Collection

Conversation Property (Message Object)

The **Conversation** property is obsolete. This property has been replaced by the **ConversationIndex** and **ConversationTopic** properties.

See Also

ConversationIndex Property (Message Object)

ConversationTopic Property (Message Object)

Working With Conversations

ConversationIndex Property (Message Object)

The **ConversationIndex** property specifies the index to the conversation thread of the message. Read/write.

Syntax

objMessage.**ConversationIndex**

Data Type

String

Remarks

The **ConversationIndex** property is a string that represents a hexadecimal number. Valid characters within the string include the numbers 0 through 9 and the letters A through F (uppercase or lowercase).

A conversation is a group of related messages that have the same **ConversationTopic** property value. In a discussion application, for example, users can save original messages and response messages. Messages can be tagged with the **ConversationIndex** property so that users can group messages by conversation.

You can use your own convention to decide how this index should be used. However, it is recommended that you adopt the same convention that is used by the Microsoft Exchange Client message viewer, so that you can use that viewer's user interface to show the relationships between messages in a conversation.

By convention, Microsoft Exchange Server uses **ConversationIndex** values that represent concatenated time stamp values. The first time stamp in the string represents the original message. When a new message represents a reply to a conversation message, it copies the **ConversationIndex** string of the message it is replying to, and then appends a time stamp value to the end of the string. The new string value is used as the **ConversationIndex** value of the new message.

When you use this convention, you can see relationships among messages when you sort the messages by **ConversationIndex** values.

The **ConversationIndex** property corresponds to the MAPI property PR_CONVERSATION_INDEX.

Example

The following example takes advantage of an OLE function that is available on computers that run the OLE Messaging Library. The **CoCreateGUID** function returns a value that consists of a time stamp and a machine identifier; this sample code saves the part that contains the time stamp.

```
' declarations section
Type GUID  ' global unique identifier; contains a time stamp
    Guid1 As Long
    Guid2 As Long
    Guid3 As Long
    Guid4 As Long
End Type
' function appears in OLE32.DLL on Windows/NT and Windows 95
Declare Function CoCreateGuid Lib "COMPOBJ.DLL" (pGuid As GUID) As Long
Global Const S_OK = 0   ' return value from CoCreateGuid

Function Util_GetEightByteTimeStamp() As String
Dim lResult As Long
Dim lGuid As GUID
    ' Exchange conversation is a unique 8-byte value
    ' Exchange client viewer sorts by concatenated properties
    On Error GoTo error_olemsg

    lResult = CoCreateGuid(lGuid)
    If lResult = S_OK Then
        Util_GetEightByteTimeStamp = _
            Hex$(lGuid.Guid1) & Hex$(lGuid.Guid2)
    Else
        Util_GetEightByteTimeStamp = "00000000"   ' zeroes
    End If
    Exit Function

error_olemsg:
    MsgBox "Error " & Str(Err) & ": " & Error$(Err)
    Util_GetEightByteTimeStamp = "00000000"
    Exit Function
End Function

Function Util_NewConversation()
Dim i As Integer
Dim objNewMsg As Object      ' new message object
Dim strNewIndex As String    ' value for ConversationIndex
' ... error handling...
    Set objNewMsg = objSession.Outbox.Messages.Add
' ... error handling...
    With objNewMsg
        .Subject = "used space vehicle wanted"
        .ConversationTopic = .Subject
        .ConversationIndex = Util_GetEightByteTimeStamp() ' utility
        .Text = "Wanted: Apollo or Mercury spacecraft with low mileage."
           ' or you could pick the public folder from the address book
        Set objOneRecip = .Recipients.Add(Name:="Car Ads", Type:=mapiTo)
        If objOneRecip Is Nothing Then
            MsgBox "Unable to create the public folder recipient"
            Exit Function
        End If
        .Recipients.Resolve
        .Update
        .Send showDialog:=False
    End With
End Function
```

A subsequent reply to a message should copy the **ConversationTopic** property and append its own time stamp to the original message's time stamp, as shown in the following example:

```
Function Util_ReplyToConversation()
Dim objPublicFolder As Object
Dim i As Integer
Dim objOriginalMsg As Object ' original message in public folder
Dim objNewMsg As Object      ' new message object for reply
Dim strPublicFolderID As String ' ID for public folder

    Set objNewMsg = objSession.Outbox.Messages.Add
'   error checking...obtain objOriginalMsg and check that it is valid
    With objNewMsg
```

```
        .Text = "How about a slightly used Gemini?"   ' new text
        .Subject = objOriginalMsg.Subject   ' copy original properties
        .ConversationTopic = objOriginalMsg.ConversationTopic
        ' append time stamp; compatible with Microsoft Exchange client
        .ConversationIndex = objOriginalMsg.ConversationIndex & _
                              Util_GetEightByteTimeStamp() ' append new
        ' message was sent to a public folder so can copy recipient
        Set objOneRecip = .Recipients.Add( _
                         Name:=objOriginalMsg.Recipients.Item(1).Name, _
                         Type:=mapiTo)
       ' ...more error handling
        .Recipients.Resolve
        .Update
        .Send showDialog:=False
    End With
' ... error handling
End Function
```

See Also

Conversation Property

ConversationTopic Property (Message Object)

Working With Conversations

ConversationTopic Property (Message Object)

The **ConversationTopic** property specifies the name of the conversation thread. Read/write.

Syntax

objMessage.**ConversationTopic**

Data Type

String

Remarks

A conversation is a group of related messages. The **ConversationTopic** property is the string that describes the overall topic of the conversation. To be defined as messages within the same conversation, the messages must have the same value in their **ConversationTopic** property. The **ConversationIndex** property represents an index that indicates a sequence of messages within that conversation.

When you start an initial message, set the **ConversationTopic** property to an appropriate value that will apply to all messages within the conversation. For many applications, the message **Subject** property is appropriate.

Note that the OLE Messaging Library does not automatically copy the **ConversationTopic** property to other messages. When your application manages messages that represent replies to an original message, you should set the **ConversationTopic** property to the same value as the original message.

To change the **ConversationTopic** for all messages in a conversation thread, you must change the property within each message in that thread.

The **ConversationTopic** property corresponds to the MAPI property PR_CONVERSATION_TOPIC.

Example

See the example for the **ConversationIndex** property.

See Also

Conversation Property

ConversationIndex Property (Message Object)

Working With Conversations

Delete Method (Message Object)

The **Delete** method deletes the Message object.

Syntax

objMessage.**Delete**

Parameters

objMessage
Required. The Message object.

Remarks

The **Delete** method permanently deletes the message from the system. Such a deleted message cannot be recovered. Before calling the **Delete** method, the application can prompt the user to verify whether the message should be permanently deleted.

See Also

Delete Method (Messages Collection)

DeliveryReceipt Property (Message Object)

The **DeliveryReceipt** property is TRUE if a delivery-receipt notification message is requested. Read/write.

Syntax

objMessage.**DeliveryReceipt**

Data Type

Boolean

Remarks

Set the **DeliveryReceipt** property to TRUE to obtain a message when the recipients receive the message. The default setting for the OLE Messaging Library is FALSE.

The **DeliveryReceipt** property corresponds to the MAPI property PR_ORIGINATOR_DELIVERY_REPORT_REQUESTED.

See Also

Making Sure The Message Gets There

ReadReceipt Property (Message Object)

Encrypted Property (Message Object)

The **Encrypted** property is TRUE if the message has been encrypted. Read/write.

Syntax

objMessage.**Encrypted**

Data Type

Boolean

Remarks

The OLE Messaging Library does not encrypt or digitally sign the message. The **Encrypted** property is dependent upon the messaging or information store provider.

The **Encrypted** property corresponds to the SECURITY_ENCRYPTED flag of the MAPI property PR_SECURITY.

See Also

Securing Messages

Signed Property (Message Object)

Fields Property (Message Object)

The **Fields** property returns a single field (a Field object) or a collection of fields (a Fields collection object) of the Message object. Read-only.

Syntax

objMessage.**Fields**

objMessage.**Fields(***index***)**

index
 Specifies the name of the field or the number of the field.

Data Type

Field object or Fields collection

Remarks

Field and Fields collection objects give you the ability to add custom fields to a message or to directly examine the underlying MAPI properties of the object. For more information, see the reference topics for the Field object and the Fields collection.

Example

```
' from Message_Fields
    Set objFieldsColl = objOneMsg.Fields
' from Fields_FirstItem
    iFieldsCollIndex = 1
    Set objOneField = objFieldsColl.Item(iFieldsCollIndex)
    If objOneField Is Nothing Then
        MsgBox "error, cannot get this Field object"
    Else
        MsgBox "Selected field " & iFieldsCollIndex
    End If
```

See Also

Field Object

Fields Collection

FolderID Property (Message Object)

The **FolderID** property returns the unique identifier of the folder in which the message resides. Read-only.

Syntax

objMessage.**FolderID**

Data Type

String

Remarks

Save the folder identifier to retrieve the folder at a later time using the Session object's **GetFolder** method.

MAPI systems assign a permanent, unique identifier string when an object is created. These identifiers do not change from one MAPI session to another.

The **FolderID** property corresponds to the MAPI property PR_PARENT_ENTRYID, converted to a string of hexadecimal characters.

See Also

GetFolder Method (Session Object)

ID Property (Message Object)

The **ID** property returns the unique identifier of this message object. Read-only.

Syntax

objMessage.**ID**

Data Type

String

Remarks

MAPI systems assign a permanent, unique identifier string when an object is created. These identifiers do not change from one MAPI session to another.

The **ID** property corresponds to the MAPI property PR_ENTRYID, converted to a string of hexadecimal characters.

Example

```
'     Save id of last message accessed; use at startup
'     from the sample function Message_ID
    strMessageID = objOneMsg.Id

' ... on shutdown, save the ID to storage
' ... on startup, get the ID from storage and restore
'     from the sample function Session_GetMessage
    Set objOneMsg = objSession.GetMessage(strMessageID)
```

See Also

GetMessage Method (Session Object)

Importance Property (Message Object)

The **Importance** property returns or sets the importance of the message as one of **mapiNormal** (the default), **mapiLow**, or **mapiHigh**. Read/write.

Syntax

objMessage.**Importance**

Data Type

Long (Enumeration)

Remarks

The following values are defined:

Constant	Value	Description
mapiLow	0	Low priority
mapiNormal	1	Normal priority (default)
mapiHigh	2	High priority

The **Importance** property corresponds to the MAPI property PR_IMPORTANCE.

Example

This example sets the importance of a message as high:

```
' from the sample function QuickStart:
    Set objMessage = objSession.Outbox.Messages.Add
    '  error checking here to verify the message was created...
    objMessage.Subject = "Gift of droids"
    objMessage.Text = "Help us, Obi-wan. You are our only hope."
    objMessage.Importance = mapiHigh
    objMessage.Send
```

See Also

Send Method (Message Object)

MAPIOBJECT Property (Message Object)

The **MAPIOBJECT** property returns an **IUnknown** pointer to this Message object. Not available to Visual Basic applications. Read/write.

Syntax

objMessage.**MAPIOBJECT**

Data Type

Variant (VT_UNKNOWN)

Remarks

The **MAPIOBJECT** property is not available to Visual Basic programs. It is available only to C/C++ programs that use the OLE Messaging Library. The **MAPIOBJECT** property is an **IUnknown** object, which is not supported by Visual Basic. Visual Basic supports **IDispatch** objects. For more information, see the Microsoft *OLE Programmer's Reference*.

See Also

Introduction to OLE Automation

How Programmable Objects Work

Options Method (Message Object)

The **Options** method displays a message options dialog box where the user can change the submission options for a message.

Syntax

objMessage.**Options([***parentWindow***])**

Parameters

objMessage
: Required. The Message object.

parentWindow
: Optional. Long. The parent window handle for the options dialog box. A value of **0** (the default) specifies an application-modal dialog box.

Remarks

The options are provider-specific and are registered by the provider. Providers are not required to register option sheets. When providers do not register options, the **Options** method returns the error code MAPI_E_NOT_FOUND.

Per-message options are properties of a message that control its behavior after submission. The per-message options are part of the message envelope, not its content.

The following methods can also invoke MAPI dialog boxes: **Delete** and **Details** methods (AddressEntry object), **Send** method (Message object), **Resolve** method (Recipient object and Recipients collection), **AddressBook** and **Logon** methods (Session object).

See Also

Send Method (Message Object)

ReadReceipt Property (Message Object)

The **ReadReceipt** property is TRUE if a read-receipt notification message is requested. Read/write.

Syntax

objMessage.**ReadReceipt**

Data Type

Boolean

Remarks

The **ReadReceipt** property corresponds to the MAPI property PR_READ_RECEIPT_REQUESTED.

See Also

DeliveryReceipt Property (Message Object)

Making Sure The Message Gets There

Recipients Property (Message Object)

The **Recipients** property returns a single Recipient object or a Recipients collection object. Read-only.

Syntax

Set *objRecipColl* **=** *objMessage.***Recipients**

Set *objOneRecip* = *objMessage.***Recipients(***index***)**

objRecipColl
: Object. A Recipients collection object.

objMessage
: Object. The Message object.

objOneRecip
: Object. A single Recipient object.

index
: Long. Specifies the number of the recipient within the Recipients collection. Ranges from 1 to the value specified by the Recipients collection's **Count** property.

Data Type

Object

Example

The **Recipients** property itself is read-only, indicating that you cannot set the value of this property to indicate another collection. However, you can change individual Recipient objects within the collection, add Recipient objects to the collection, and remove Recipient objects from the collection.

This example copies each of the recipients from the original message *objOneMsg* to the copy *objCopyMsg*:

```
' from the sample function Util_CopyMessage
    For i = 1 To objOneMsg.Recipients.Count Step 1
        strRecipName = objOneMsg.Recipients.Item(i).Name
        If strRecipName <> "" Then
            Set objOneRecip = objCopyMsg.Recipients.Add
            If objOneRecip Is Nothing Then
                MsgBox "unable to create recipient in message copy"
                Exit Function
            End If
            objOneRecip.Name = strRecipName
        End If
    Next i
```

See Also

Recipient Object

Send Method (Message Object)

The **Send** method sends the message to the recipients via the MAPI system.

Syntax

objMessage.**Send([***saveCopy, showDialog, parentWindow***])**

Parameters

objMessage
   Required. The Message object.

saveCopy
   Optional. Boolean. If TRUE or omitted, saves a copy of the Message in a user folder, such as the Sent Messages folder.

showDialog
   Optional. Boolean. If TRUE, displays a **Send Message** dialog box where the user can change the message contents or recipients. The default value is FALSE.

parentWindow
   Optional. Long. The parent window handle for the **Send Message** dialog box. A value of **0** (the default) specifies that any dialog box displayed is application-modal. The *parentWindow* parameter is ignored unless *showDialog* is TRUE.

Remarks

The **Send** method is similar to the **Update** method, except **Send** ignores the parent Folder object of the message and saves the message in the current user's default Outbox folder. Messaging systems retrieve messages from the Outbox and transport them to the recipients.

Note that the **Send** method invalidates the Message object. Attempts to access the original Message object result in an error. The original Message object does not have to be set to **Nothing**, but it should not be used for subsequent operations. Use a new Message object to obtain the message from the Outbox or from the Sent Messages folder.

Note that there is one case in which the OLE Messaging Library does not display the dialog box when *showDialog* is set to TRUE: The dialog box is not displayed when the recipient has a null display name. The dialog box is displayed for a null recipient (when the Recipient object is set to **Nothing**).

The following methods can also invoke MAPI dialog boxes: **Delete** and **Details** methods (AddressEntry object), **Options** method (Message object), **Resolve** method (Recipient object and Recipients collection), **AddressBook** and **Logon** methods (Session object).

See Also

Creating and Sending a Message

Posting Messages to a Public Folder

Sent Property (Message Object)

Submitted Property (Message Object)

Sender Property (Message Object)

The **Sender** property returns the originator or original author of a message as an AddressEntry object. Read-only.

Syntax

Set *objAddrEntry* **=** *objMessage*.**Sender**

objAddrEntry
  Object. The returned AddressEntry object that represents the message author.
objMessage
  Object. The Message object.

Data Type

Object

Remarks

The **Sender** property corresponds to the MAPI property PR_SENDER_ENTRYID.

Example

This example displays the name of the sender of a message:

```
' from the sample function Message_Sender
    Set objAddrEntry = objOneMsg.Sender
    If objAddrEntry Is Nothing Then
        MsgBox "Could not set the address entry object from the Sender"
        Exit Function
    End If
    MsgBox "Message was sent by " & objAddrEntry.Name
```

See Also

TimeReceived Property (Message Object)

Sent Property (Message Object)

The **Sent** property is TRUE if the message has been sent through the MAPI system. Read/write.

Syntax

objMessage.**Sent**

Data Type

Boolean

Remarks

In general, there are three different kinds of messages: messages that get *sent,* messages that get *posted,* and messages that get *saved*. Messages that get sent are characterized by traditional e-mail messages sent to a recipient or public folder. Messages that get posted are characterized by messages created in a public folder. Messages that get saved are characterized by messages that are created and saved without either sending or posting.

For all three kinds of messages, you use the **Submitted**, **Sent**, and **Unread** properties and the **Send** or **Update** methods.

The following table summarizes the use of the message properties and methods for these three kinds of messages.

Kind of message	**Method**	**Submitted property**	**Sent property**	**Unread property**
Gets sent	**Send**	**Send** method sets TRUE	Spooler sets TRUE	Spooler sets TRUE
Gets posted	**Update**	Application sets FALSE	Application sets TRUE	Application sets TRUE
Gets saved	**Update**	Application sets FALSE	Application sets FALSE	Application sets TRUE

For messages that get sent, the **Sent** property can be written until the time that you call the **Send** or **Update** method. Note that changing the **Sent** property to TRUE does not cause the message to be sent. Only the **Send** method actually causes the message to be transmitted. After you call the **Send** method, the messaging system controls the **Sent** property and changes it to a read-only property.

A common use for writing a value to the **Sent** property is to set the property to FALSE so that an electronic mail system can save pending, unsent messages in an Outbox folder, or to save work-in-progress messages in a Pending folder before committing the messages to a public information store. Note that you can cause an error if you set the property incorrectly.

The **Sent** property is changed using the following sequence. When you call the **Send** method to send a message to a recipient, the message is moved to the Outbox and the Message object's **Submitted** property is set to TRUE. When the messaging system spooler actually starts transporting the message, the **Sent** property is set to TRUE.

When the message is not sent using the **Send** method, the MAPI system does not change the **Sent** property.   For posted and saved messages that call the **Update** method, you should set the value of the **Sent** property to TRUE just before you post the message.

The **Sent** property corresponds to the MAPI property PR_MESSAGE_FLAGS and the flag MSGFLAG_UNSEND.

See Also

Creating and Sending a Message

Posting Messages to a Public Folder

Submitted Property (Message Object)

Signed Property (Message Object)

The **Signed** property is TRUE if the message has been tagged with a digital signature. Read/write.

Syntax

objMessage.**Signed**

Data Type

Boolean

Remarks

The **Signed** property is dependent upon the messaging or information store provider. The OLE Messaging Library does not actually encrypt or digitally sign the message.

The **Signed** property corresponds to the SECURITY_SIGNED flag of the MAPI property PR_SECURITY.

See Also

Encrypted Property (Message Object)

Securing Messages

Size Property (Message Object)

The **Size** property returns the approximate size in bytes of the message. Read-only.

Syntax

objMessage.**Size**

Data Type

Long

Remarks

The **Size** property is not valid until after the first **Update** or **Send** operation.

The **Size** property corresponds to the MAPI property PR_MESSAGE_SIZE.

See Also

Attachments Property (Message Object)

Text Property (Message Object)

StoreID Property (Message Object)

The **StoreID** property represents the unique identifier for the information store that contains this message. Read-only.

Syntax

objMessage.**StoreID**

Data Type

String

Remarks

The **StoreID** property corresponds to the MAPI property PR_STORE_ENTRYID, converted to a string of hexadecimal characters.

See Also

GetMessage Method (Session Object)

Sent Property (Message Object)

Subject Property (Message Object)

The **Subject** property returns or sets the subject of the message as a string. Read/write.

Syntax

objMessage.**Subject**

Data Type

String

Remarks

The **Subject** property corresponds to the MAPI property PR_SUBJECT.

Example

This example sets the subject of a message:

```
Dim objMessage As Object  ' assume valid message
objMessage.Subject = "Microsoft Bob: Check It Out"
```

See Also

Text Property (Message Object)

Submitted Property (Message Object)

The **Submitted** property is TRUE when the message has been submitted. Read/write.

Syntax

objMessage.**Submitted**

Data Type

Boolean

Remarks

In general, there are three different kinds of messages: messages that get *sent,* messages that get *posted,* and messages that get *saved*. Messages that get sent are characterized by traditional e-mail messages sent to a recipient or public folder. Messages that get posted are characterized by messages created in a public folder. Messages that get saved are characterized by messages that are created and saved without either sending or posting.

For all three kinds of messages, you use the **Submitted**, **Sent**, and **Unread** properties and the **Send** or **Update** methods.

The following table summarizes the use of the message properties and methods for these three kinds of messages.

Kind of message	Method	Submitted property	Sent property	Unread property
Gets sent	**Send**	**Send** method sets TRUE	Spooler sets TRUE	Spooler sets TRUE
Gets posted	**Update**	Application sets FALSE	Application sets TRUE	Application sets TRUE
Gets saved	**Update**	Application sets FALSE	Application sets FALSE	Application sets TRUE

For messages that get sent, you create the message and then call the **Send** method. When the **Send** method is successful, the **Submitted** property is set to TRUE. The value does not change after that point. For messages sent to a public folder, the **Send** method sets the **Submitted** property (rather than the **Sent** property) to TRUE.

For messages that get posted, you create the message directly within a public folder and call **Update**. When you create the message within the public folder, some viewers do not allow the message to become visible to others until you set the **Submitted** property to TRUE.

The **Submitted** property corresponds to the MAPI property PR_MESSAGE_FLAGS.

See Also

Send Method (Message Object)

Sent Property (Message Object)

Text Property (Message Object)

The **Text** property returns or sets the body text of the message as a string. Read/write.

Syntax

objMessage.**Text**

Data Type

String

Remarks

The maximum size of the body text can be limited by the tool that you use to manipulate string variables (such as Microsoft Visual Basic).

Note that the **Text** property is a plain text representation of the message body and does not support formatted text.

The **Text** property corresponds to the MAPI property PR_BODY.

Example

This example sets the body text of a message:

```
Dim objMessage As Object     ' assume valid message
objMessage.Text = "Thank you for buying Microsoft Home(TM) products."
```

See Also

Subject Property (Message Object)

TimeReceived Property (Message Object)

The **TimeReceived** property sets or returns the date and time the message was received as a **vbDate** variant data type. Read/write.

Syntax

objMessage.**TimeReceived**

Data Type

Variant (**vbDate** format)

Remarks

The **TimeReceived** and **TimeSent** properties set and return dates and times as the local time for the user's system.

When you send messages using the Message object's **Send** method, the MAPI system sets the **TimeReceived** and **TimeSent** properties for you. However, when you post messages in a public folder, you must first explicitly set these properties. For a message posted to a public folder, set both properties to the same time value.

Note that the **TimeReceived** and **TimeSent** properties represent local time. However, when you access MAPI time properties through the Fields collection's **Item** property, the time values represent Greenwich Mean Time.

The **TimeReceived** property corresponds to the MAPI Property PR_MESSAGE_DELIVERY_TIME.

Example

This example displays the date and time a message was sent and received:

```
' from the sample function Message_TimeSentAndReceived
   ' verify that objOneMsg is valid, then...
    With objOneMsg
        strMsg = "Message sent " & Format(.TimeSent, "Short Date")
        strMsg = strMsg & ", " & Format(.TimeSent, "Long Time")
        strMsg = strMsg & "; received "
        strMsg = strMsg & Format(.TimeReceived, "Short Date") & ", "
        strMsg = strMsg & Format(.TimeReceived, "Long Time")
        MsgBox strMsg
    End With
```

See Also

Item Property (Fields Collection)

TimeSent Property (Message Object)

TimeSent Property (Message Object)

The **TimeSent** property sets or returns the date and time the message was sent as a **vbDate** variant data type. Read/write.

Syntax

objMessage.**TimeSent**

Data Type

Variant (**vbDate** format)

Remarks

The **TimeReceived** and **TimeSent** properties return dates and times as the local time for the user's system.

When you send messages using the Message object's **Send** method, the MAPI system sets the **TimeReceived** and **TimeSent** properties for you. However, when you post messages in a public folder, you must first explicitly set these properties. For a message posted to a public folder, set both properties to the same time value.

Note that the **TimeReceived** and **TimeSent** properties represent local time. However, when you access MAPI time properties through the Fields collection **Item** property, the time values represent Greenwich Mean Time.

The **TimeSent** property corresponds to the MAPI Property PR_CLIENT_SUBMIT_TIME.

Example

This example displays the date a message was sent and received:

```
' from the sample function Message_TimeSentAndReceived
   ' verify that objOneMsg is valid, then...
    With objOneMsg
        strMsg = "Message sent " & Format(.TimeSent, "Short Date")
        strMsg = strMsg & ", " & Format(.TimeSent, "Long Time")
        strMsg = strMsg & "; received "
        strMsg = strMsg & Format(.TimeReceived, "Short Date") & ", "
        strMsg = strMsg & Format(.TimeReceived, "Long Time")
        MsgBox strMsg
    End With
```

See Also

Item Property (Fields Collection)

TimeReceived Property (Message Object)

Type Property (Message Object)

The **Type** property returns or sets the MAPI message class for the message. Read/write.

Syntax

objMessage.**Type**

Data Type

String

Remarks

The **Type** property returns or sets the MAPI message class for the message. By default, the OLE Messaging Library sets the **Type** value of new messages to the MAPI message class IPM.Note.

The OLE Messaging Library does not impose any restrictions on this value except that it be a valid string value. You can set the value to any string that is meaningful for your application. MAPI uses message class strings in the form IPM.*application.subClass* or IPC.*application.subClass*.

For more information about MAPI message classes, see the *MAPI Programmer's Reference*.

The **Type** property corresponds to the MAPI property PR_MESSAGE_CLASS.

See Also

GetFirst Method (Messages Collection)

Unread Property (Message Object)

The **Unread** property is TRUE if the message has not been read by the current user. Read/write.

Syntax

objMessage.**Unread**

Data Type

Boolean

Remarks

When you post a message to a public folder, you should set the **Unread**, **Submitted**, and **Sent** properties to TRUE before calling the **Send** or **Update** method.

The **Unread** property corresponds to the MAPI property PR_MESSAGE_FLAGS.

See Also

Posting a Message to a Public Folder

Sent Property (Message Object)

Update Method (Message Object)

The **Update** method saves the message in the MAPI system.

Syntax

objMessage.**Update([***makePermanent*, *refreshObject***])**

Parameters

objMessage
Required. The Message object.

makePermanent
Optional. Boolean. TRUE indicates that the property cache is flushed and all changes are committed to the underlying store. FALSE indicates that the property cache is flushed and not committed to the store. The default value is TRUE.

refreshObject
Optional. Boolean. TRUE indicates that the property cache is reloaded from the values in the underlying store. FALSE indicates that the property cache is not reloaded. The default value is FALSE.

Remarks

Changes to Message objects are not permanently saved in the MAPI system until you call the **Update** method with the *makePermanent* parameter set to TRUE.

For improved performance, the OLE Messaging Library caches property changes in private storage and updates either the object or the underlying store only when you explicitly request such an update. For efficiency, you should make only one call to **Update** with its *makePermanent* parameter set to TRUE.

The *makePermanent* and *refreshObject* parameters combine to cause the following changes:

	refreshObject = TRUE	**refreshObject = FALSE**
makePermanent = TRUE	Commit all changes, flush the cache, and reload the cache from the store.	Commit all changes and flush the cache.
makePermanent = FALSE	Flush the cache and reload the cache from the store.	Flush the cache.

Call **Update(FALSE, TRUE)** to flush the cache and then reload the values from the store.

Example

This example changes the subject of the first message in a folder:

```
Set objMessage = objSession.Inbox.GetFirst
' ... verify message
objMessage.Subject = "This is the new subject"
objMessage.Update
```

To add a new Message object, use the **Add** method followed by the **Update** method. This example saves a new message:

```
Dim objMessage As Object   ' message object
' ...
```

```
Set objMessage = objSession.Outbox.Messages.Add
objMessage.Subject = "Microsoft Bob(TM)"
objMessage.Text = "This is incredible, you've got to see it!"
objMessage.Update makePermanent:=True
```

See Also

Send Method (Message Object)

Messages Collection Object

The Messages collection object contains one or more Message objects.

The Messages collection is considered a *large collection*, which means that you must use a Message identifier value or the **Get** methods to access individual Message objects within the collection.

Properties

Property name	Type	Access
Application	String	Read-only
Class	Long	Read-only
Parent	Object	Read-only
Session	Session object	Read-only

Methods

Method name	Parameters
Add	(optional) subject as String, (optional) text as String, (optional) type as String, (optional) importance as Long
Delete	(none)
GetFirst	(optional) filter as Variant
GetLast	(optional) filter as Variant
GetNext	(none)
GetPrevious	(none)
Sort	sortOrder as Long

Remarks

The collection does not maintain a count of the number of Message objects in the collection. Use the **GetFirst**, **GetLast**, **GetNext**, and **GetPrevious** methods to access the Message objects in the collection.

The order in which items are returned by **GetFirst**, **GetNext**, **GetPrevious**, and **GetLast** depends on whether the messages are sorted or not. The Message objects within a collection can be sorted by delivery time, either ascending or descending.

When the items are unsorted, these methods do not return the items in any specified order. The best programming approach to use with unsorted collections is to assume that you can access all items within the collection, but that the order of the objects is undefined.

See Also

Object Collections

Add Method (Messages Collection)

The **Add** method creates and returns a new Message object in the Messages collection.

Syntax

Set *objMessage* = *objMsgColl*.**Add(** **[***subject, text, type, importance***]** *)*

Parameters

objMessage
On successful return, represents the new Message object added to the collection.

objMsgColl
Required. The Messages collection object.

subject
Optional. String. The subject of the message. When this parameter is not supplied, the default value is an empty string.

text
Optional. String. The body text of the message. When this parameter is not supplied, the default value is an empty string.

type
Optional. String. The message class of the message, such as the default, IPM.Note.

importance
Optional. Long. The priority of the message. The following values are defined:

Constant	**Value**	**Description**
mapiLow	0	Low priority
mapiNormal	1	Normal priority (default)
mapiHigh	2	High priority

Remarks

The method parameters correspond to the **Subject**, **Text**, **Type**, and **Importance** properties of the Message object.

You should create new messages in the Outbox folder.

Example

This example adds a new message to a folder:

```
' from the sample function Util_ReplyToConversation
    Set objNewMsg = objSession.Outbox.Messages.Add
    ' verify objNewMsg created successfully...then supply properties
    With objNewMsg
        .Text = "How about a slightly used Gemini?"    ' new text
        .Subject = objOriginalMsg.Subject    ' copy original properties
        .ConversationTopic = objOriginalMsg.ConversationTopic
        ' append time stamp; compatible with Microsoft Exchange client
        Set objOneRecip = .Recipients.Add( _
                      Name:=objOriginalMsg.Recipients.Item(1).Name, _
                      Type:=mapiTo)
        .Recipients.Resolve
        .Update
        .Send showDialog:=False
    End With
```

See Also

Delete Method (Message Object)

Delete Method (Messages Collection)

The **Delete** method deletes all messages in the collection.

Syntax

objMsgColl.**Delete()**

Parameters

objMsgColl
    Required. The Messages collection object.

Remarks

Deletes all member objects within the collection.

Moves the deleted folders and messages to the Deleted Messages folder, if the user has enabled this option. If the folders and messages are already in the Deleted Messages folder, the **Delete** method permanently deletes them, and they cannot be recovered.

Be cautious using **Delete** with collections, since the method deletes all member objects within a collection. For example, this code deletes all of the messages in a folder:

```
objFolder.Messages.Delete
```

See Also

Add Method (Messages Collection)

GetFirst Method (Messages Collection)

The **GetFirst** method returns the first object in the collection. Returns **Nothing** if no first object exists.

Syntax

Set *objMessage* **=** *objMsgColl*.**GetFirst([***filter***])**

Parameters

objMessage
: On successful return, represents the first Message object in the collection.

objMsgColl
: Required. The Messages collection object.

filter
: Optional. String. Specifies the message class of the object, such as the default value, IPM.Note. Corresponds to the **Type** property of the Message object.

Remarks

The **Get** methods are similar to the **Find** and **Move** methods that are used with Microsoft Access databases. The **Get** methods take a different name from these methods because they use a different syntax.

See Also

Message Object

GetLast Method (Messages Collection)

The **GetLast** method returns the last object in the collection. Returns **Nothing** if no last object exists.

Syntax

Set *objMessage* **=** *objMsgColl*.**GetLast(** **[***filter***]** **)**

Parameters

objMessage
: On successful return, represents the last Message object in the collection.

objMsgColl
: Required. The Messages collection object.

filter
: Optional. String. Specifies the message class of the object, such as the default value, IPM.Note. Corresponds to the **Type** property of the Message object.

Remarks

The **Get** methods are similar to the **Find** and **Move** methods that are used with Microsoft Access databases. The **Get** methods take a different name from these methods because they use a different syntax.

See Also

Update Method (Message Object)

GetNext Method (Messages Collection)

The **GetNext** method returns the next object in the Messages collection. Returns **Nothing** if no next object exists.

Syntax

Set *objMessage* **=** *objMsgColl*.**GetNext( )**

Parameters

objMessage
On successful return, represents the next Message object in the collection.

objMsgColl
Required. The Messages collection object.

Remarks

The **Get** methods are similar to the **Find** and **Move** methods that are used with Microsoft Access databases. The **Get** methods take a different name from these methods because they use a different syntax.

See Also

Update Method (Message Object)

GetPrevious Method (Messages Collection)

The **GetPrevious** method returns the previous object in the collection. Returns **Nothing** if no previous object exists.

Syntax

Set *objMessage* **=** *objMsgColl*.**GetPrevious( )**

Parameters

objMessage
On successful return, represents the previous Message object in the collection.

objMsgColl
Required. The Messages collection object.

Remarks

The **Get** methods are similar to the **Find** and **Move** methods that are used with Microsoft Access databases. The **Get** methods take a different name from these methods because they use a different syntax.

See Also

Update Method (Message Object)

Sort Method (Messages Collection)

The **Sort** method sorts the messages in the collection according to the specified sort order.

Syntax

objMsgColl.**Sort(***sortOrder***)**

Parameters

objMsgColl
Required. The Messages collection object.

sortOrder
Required. Long. The specified sort order, one of the following values:

Value	Numeric value	Description
mapiNone	0	No sort
mapiAscending	1	Ascending sort
mapiDescending	2	Descending sort

See Also

Update Method (Message Object)

Recipient Object

Represents a recipient of a message.

Properties

Property name	Type	Access
Address	String	Read/write
AddressEntry	AddressEntry object	Read/write
Application	String	Read-only
Class	Long	Read-only
DisplayType	Long	Read-only
Index	Long	Read-only
Name	String	Read/write
Parent	Object	Read-only
Session	Session object	Read-only
Type	Long	Read/write

Methods

Method name	Parameters
Delete	(none)
Resolve	(optional) showDialog as Boolean

See Also

AddressEntry Object

Recipients Collection

Address Property (Recipient Object)

The **Address** property specifies the *full address* for this recipient. Read/write.

Syntax

objRecipient.**Address**

Data Type

String

Remarks

Sets the value of the Recipient object's **Address** property to specify a custom address. The Recipient **Address** uses the following syntax:

*TypeValue***:***AddressValue*

where *TypeValue* and *AddressValue* correspond to the values of the AddressEntry object's **Type** and **Address** properties.

The Recipient object's **Address** property represents the *full address*, the complete messaging address used by the MAPI system.

The OLE Messaging Library sets the value of the Recipient object's **Address** property for you when you supply the **Name** property and call its **Resolve** method.

The **Address** property corresponds to the MAPI properties PR_EMAIL_ADDRESS and PR_EMAIL_TYPE.

Example

```
' from the sample function Util_CompareAddressParts
    strMsg = "Recipient full address = " & objOneRecip.Address
    strMsg = strMsg & "; AddressEntry type = " & objAddrEntry.Type
    strMsg = strMsg & "; AddressEntry address = " & objAddrEntry.Address
    MsgBox strMsg    ' compare address components
```

See Also

AddressEntry Object

Name Property (Recipient Object)

Resolve Method (Recipient Object)

AddressEntry Property (Recipient Object)

The **AddressEntry** property specifies the AddressEntry object for this recipient. Read/write.

Syntax

objRecipient.**AddressEntry**

Data Type

Object (AddressEntry object)

Remarks

The **AddressEntry** property indicates the AddressEntry object for this recipient. For a complete description of the relationship between the AddressEntry object and the Recipient object, see Using Addresses.

Accessing the **AddressEntry** property forces resolution of an unresolved recipient name. If the name cannot be resolved, the OLE Messaging Library reports an error. For example, when the recipient contains an empty string, the resolve operation returns MAPI_E_AMBIGUOUS_RECIP.

Example

This example compares the **Address** property of the Recipient object with the **Address** and **Type** properties of its child AddressEntry object, accessible through its **AddressEntry** property, to demonstrate the relationships between these properties.

```
' from the sample function Session_AddressEntry
    If objOneRecip Is Nothing Then
        MsgBox "must select a recipient"
        Exit Function
    End If
    Set objAddrEntry = objOneRecip.AddressEntry
    If objAddrEntry Is Nothing Then
        MsgBox "no valid AddressEntry for this recipient"
        Exit Function
    End If
' from the sample function Util_CompareAddressParts
    strMsg = "Recipient full address = " & objOneRecip.Address
    strMsg = strMsg & "; AddressEntry type = " & objAddrEntry.Type
    strMsg = strMsg & "; AddressEntry address = " & objAddrEntry.Address
    MsgBox strMsg    ' compare address components
    strMsg = "Recipient name = " & objOneRecip.Name
    strMsg = strMsg & "; AddressEntry name = " & objAddrEntry.Name
    MsgBox strMsg    ' compare display names (should be same)
```

See Also

AddressEntry Object

Delete Method (Recipient Object)

The **Delete** method deletes the Recipient object.

Syntax

objRecipient.**Delete()**

Parameters

objRecipient
Required. The Recipient object.

Remarks

The object is set to **Nothing** and it is removed from memory. The change is not permanent until you use the **Update**, **Send**, or **Delete** method on the parent Message object.

See Also

Send Method (Message Object)

Update Method (Message Object)

DisplayType Property (Recipient Object)

The **DisplayType** property identifies the recipient type. This property enables special processing based on the type, such as displaying an icon associated with that type. Read-only.

Syntax

objRecipient.**DisplayType**

Parameters

objRecipient
Required. The Recipient object.

Data Type

Long

Remarks

You can use the display type to sort or to filter recipients.

The following values are defined:

DisplayType value	Description
mapiUser	Local user
mapiDistList	Distribution list
mapiForum	Public folder
mapiAgent	Agent
mapiOrganization	Organization
mapiPrivateDistList	Private distribution list
mapiRemoteUser	Remote user

See Also

Item Property (Recipients Collection)

Index Property (Recipient Object)

The **Index** property returns the index number of this Recipient object within the Recipients collection. Read-only.

Syntax

objRecipient.**Index**

Data Type

Long

Remarks

The index number indicates an index within the array of Recipients collection object.

An index value should not be considered to be a static value that remains constant for the duration of a session. The index can change whenever an update occurs to a parent object, such as the message or folder.

Example

```
' from the sample function Recipients_NextItem
'   after some similar validation...
    If iRecipCollIndex >= objRecipColl.Count Then
        iRecipCollIndex = objRecipColl.Count
        MsgBox "Already at end of recipient list"
        Exit Function
    End If
    ' index is < count; can be incremented by 1
    iRecipCollIndex = iRecipCollIndex + 1
    Set objOneRecip = objRecipColl.Item(iRecipCollIndex)
' from the sample function Recipient_Index
    If objOneRecip Is Nothing Then
        MsgBox "must first select a recipient"
        Exit Function
    End If
    MsgBox "Recipient index = " & objOneRecip.Index
```

See Also

Count Property (Recipients Collection)

Item Property (Recipients Collection)

Name Property (Recipient Object)

The **Name** property specifies the name of this Recipient object. Read/write.

Syntax

objRecipient.**Name**

Data Type

String

Remarks

The **Name** property corresponds to the MAPI property PR_DISPLAY_NAME.

Example

```
' from the sample function Util_CompareFullAddressParts()
Dim strMsg As String
    ' validate objects... then display
    strMsg = "Recipient full address = " & objOneRecip.Address
    strMsg = strMsg & "; AddressEntry type = " & objAddrEntry.Type
    strMsg = strMsg & "; AddressEntry address = " & objAddrEntry.Address
    MsgBox strMsg    ' compare address parts

    strMsg = "Recipient name = " & objOneRecip.Name
    strMsg = strMsg & "; AddressEntry name = " & objAddrEntry.Name
    MsgBox strMsg    ' compare display names (should be same)
```

See Also

Recipient Object

Resolve Method (Recipient Object)

The **Resolve** method resolves address information. When the Recipient object's **Name** property is supplied, looks up the corresponding address from the address book. When the Recipient object's **Address** property is supplied, resolves as a custom address.

Syntax

objRecipient.**Resolve([** *showDialog* **])**

Parameters

objRecipient
Required. The Recipient object.

showDialog
Optional. Boolean. If TRUE (the default value), displays a dialog box to prompt the user to resolve ambiguous names.

Remarks

The **Resolve** method operates when the **AddressEntry** property is set to **Nothing**. Its operation depends on whether you supply the Recipient **Name** or **Address** property.

When you supply the **Name** property, **Resolve** looks up the Recipient object's **Name** property in the address book. When a recipient is resolved, the recipient object's **Address** property contains the full address and its **AddressEntry** property contains a reference to an AddressEntry object that represents a copy of information in the address book.

Note that the **Resolve** method does not validate the Recipient object's **Type** property.

When you specify a custom address by supplying the Recipient object's **Address** property, the **Resolve** method does not attempt to compare the address against the address book.

To avoid delivery errors, clients should always resolve recipients before submitting a message to the MAPI system. Resolving the recipient name means either finding a matching address in an address list or having the user select an address from a dialog box.

The **Resolve** method uses the address list specified in the profile, such as the global address book or the personal address book.

The following methods can also invoke MAPI dialog boxes: **Delete** and **Details** methods (AddressEntry object), **Options** and **Send** methods (Message object), **Resolve** method (Recipient object and Recipients collection), **AddressBook** and **Logon** methods (Session object).

See Also

Resolve Method (Recipients Collection)

Resolved Property (Recipients Collection)

Type Property (Recipient Object)

The **Type** property specifies the type of the Recipient object; an integer value that indicates either To, Cc, or Bcc. Read/write.

Syntax

objRecipient.**Type**

Data Type

Long

Remarks

The **Type** property applies to all Recipient objects in the Recipients collection. The property has the following defined values:

Recipient type	Value	Description
mapiTo	1	The recipient is on the To line.
mapiCc	2	The recipient is on the Cc line.
mapiBcc	3	The recipient is on the Bcc line.

The **Type** property corresponds to the MAPI property PR_RECIPIENT_TYPE.

See Also

Address Property (Recipient Object)

Resolve Method (Recipient Object)

Recipients Collection Object

The Recipients collection object specifies the recipients of a message.

Properties

Property name	Type	Access
Application	String	Read-only
Class	Long	Read-only
Count	Long	Read-only
Item	Recipient object	Read-only
Parent	Object	Read-only
Resolved	Boolean	Read-only
Session	Session object	Read-only

Methods

Method name	Parameters
Add	(optional) name as String, (optional) address as String, (optional) type as Long, (optional) entryID as String
Delete	(none)
Resolve	(optional) showDialog as Boolean

Remarks

The Recipients collection is considered a *small collection*, which means that it supports count and index values that let you access individual Recipient objects through the **Item** property. The Recipients collection supports the Visual Basic **For Each** statement.

See Also

Object Collections

Add Method (Recipients Collection)

The **Add** method creates a new Recipient object in the Recipients collection.

Syntax

Set *objRecipient* = *objRecipColl*.**Add([***name, address, type***] | [***entryID***])**

Parameters

objRecipient
: On successful return, represents the new Recipient object added to the collection.

objRecipColl
: Required. The Recipients collection object.

name
: Optional. String. The display name of the recipient. When this parameter is not present, the property of the new object is set to an empty string.

address
: Optional. String. The address of the recipient. When this parameter is not present, the property of the new object is set to an empty string.

type
: Optional. Long. The type of recipient; the initial value for the Recipient object's **Type** property. The following values are valid:

Recipient type	**Value**	**Description**
mapiTo	1	The recipient is on the To line.
mapiCc	2	The recipient is on the Cc line.
mapiBcc	3	The recipient is on the Bcc line.

When this parameter is not present, the object uses the default value **mapiTo**.

entryID
: Optional. String. The identifier of a valid AddressEntry object for this recipient. No default value is supplied for the *entryID* parameter. When present, the other parameters are not used. When not present, the method uses the *name*, *address*, and *type* parameters to determine the recipient.

Remarks

The *name*, *address*, and *type* parameters correspond to the Recipient object's **Name**, **Address**, and **Type** properties, respectively. The *entryID* parameter corresponds to the child AddressEntry object's **ID** property. When the *entryID* parameter is present, the other parameters are not used.

When no parameters are present, an empty Recipient object is created.

You can access the child AddressEntry object through the Recipient object's **AddressEntry** property.

Call the **Resolve** method after you add a recipient.

The **Index** of the new Recipient object equals the new **Count** of the Recipients collection. The recipient is actually saved in the MAPI system when you **Update** or **Send** the parent message object.

Example

This example adds three recipients to a message. The address for the first recipient is resolved using the display name. The second recipient is a custom address, so the resolve operation does not modify it. The third recipient is taken from an existing valid AddressEntry object. The Resolve operation does not affect this recipient.

```
' from the sample function "Using Addresses"

    ' add 3 recipient objects to a valid message object
    ' 1. look up entry in address book
    Set objOneRecip = objNewMessage.Recipients.Add( _
        Name:=strName, _
        Type:=mapiTo)
    ' error handling...verify objOneRecip
    objOneRecip.Resolve

    ' 2. add a custom recipient
    Set objOneRecip = objNewMessage.Recipients.Add( _
        Address:="SMTP:davidhef@microsoft.com", _
        Type:=mapiTo)
    If objOneRecip Is Nothing Then
        MsgBox "Unable to add recipient using custom addressing"
        Exit Function
    End If
    objOneRecip.Resolve

    ' 3. add a valid address entry object, such as Message.Sender
    ' assume valid address entry ID, name from an existing message
    Set objOneRecip = objNewMessage.Recipients.Add( _
        entryID:=strAddrEntryID, _
        Name:=strName, _
        Type:=mapiTo)
    If objOneRecip Is Nothing Then
        MsgBox "Unable to add recipient using existing AddressEntry ID"
        Exit Function
    End If

    objNewMessage.Text = "expect 3 different recipients"
    MsgBox ("count = " & objNewMessage.Recipients.Count)
```

See Also

Resolve Method (Recipients Collection)

Count Property (Recipients Collection)

The **Count** property returns the number of Recipient objects in the collection. Read-only.

Syntax

objRecipColl.**Count**

Data Type

Long

Example

This example uses the **Count** property as a loop counter to copy all recipients from one Recipients collection to another.

```
' from the sample function Util_CopyMessage
' Copy all Recipient objects from one collection to another
'  ... verify valid message object objOneMsg
   For i = 1 To objOneMsg.Recipients.Count Step 1
        strRecipName = objOneMsg.Recipients.Item(i).Name
        If strRecipName <> "" Then
            Set objOneRecip = objCopyMsg.Recipients.Add
            If objOneRecip Is Nothing Then
                MsgBox "unable to create recipient in message copy"
                Exit Function
            End If
            objOneRecip.Name = strRecipName
        End If
    Next i
```

See Also

Item Property (Recipients Collection)

Delete Method (Recipients Collection)

The **Delete** method deletes all Recipients in the collection.

Syntax

objRecipColl.**Delete()**

Parameters

objRecipColl
Required. The Recipients collection object.

Remarks

The object or collection is set to **Nothing** and it is removed from memory, but the change is not permanent until you use the **Update**, **Send**, or **Delete** method on the parent object that contained the deleted object or collection.

Be cautious using **Delete** with collections, since the method deletes all member objects within a collection.

See Also

Delete Method (Recipient Object)

Send Method (Message Object)

Update Method (Message Object)

Item Property (Recipients Collection)

The **Item** property returns a single Recipient from the collection. Read-only.

Syntax

objRecipColl.**Item(***index***)**

objRecipCol
Required. Specifies the Recipients collection object.

index
An integer that ranges from 1 to *objRecipColl*.**Count**, or a string that specifies the name of the object.

Data Type

Object

Remarks

The **Item** property works like the accessor property for a small collection.

Example

```
'  list all recipient names in the collection
    strRecips = ""  ' initialize string
    Set objRecipsColl = objOriginalMsg.Recipients
    Count = objRecipsColl.Count
    For i = 1 To Count Step 1
        Set objOneRecip = objRecipsColl.Item(i)
        strRecips = strRecips & objOneRecip.Name & "; "
    Next i
    MsgBox "Message recipients: " & strRecips
```

See Also

Count Property (Recipients Collection)

Resolve Method (Recipients Collection)

The **Resolve** method searches the Recipients collection to resolve names.

Syntax

objRecipColl.**Resolve([** *showDialog* **])**

Parameters

objRecipColl
: Required. The Recipients collection object.

showDialog
: Optional. Boolean. If TRUE (the default value), displays a dialog box to prompt the user to resolve ambiguous names.

Remarks

Calling the Recipients collection's **Resolve** method is similar to calling the **Resolve** method for each Recipient object in the collection; any ambiguous addresses are resolved as unambiguous addresses. However, calling the **Resolve** method on the Recipients collection also forces an update to the **Count** and **Item** properties and to all Recipient objects in the collection. Any Recipient variable previously set to an object in the collection is invalidated by the **Resolve** call and should be retrieved again from the collection.

The **Resolve** method of the Recipient object itself does not invalidate the object. For more information about the individual **Resolve** operation, see the reference documentation for the Recipient object's **Resolve** method.

The following methods can also invoke MAPI dialog boxes: **Delete** and **Details** methods (AddressEntry object), **Options** and **Send** methods (Message object), **Resolve** method (Recipient object), **AddressBook** and **Logon** methods (Session object).

Example

```
' from the sample function Util_NewConversation
'   create a valid new message object in the Outbox
    With objNewMsg
        .Subject = "used space vehicle wanted"
        ' ... set other properties here...
        Set objOneRecip = .Recipients.Add(Name:="Car Ads", Type:=mapiTo)
        If objOneRecip Is Nothing Then
            MsgBox "Unable to create the public folder recipient"
            Exit Function
        End If
        .Recipients.Resolve
    End With
```

See Also

Resolve Method (Recipient Object)

Resolved Property (Recipients Collection)

Resolved Property (Recipients Collection)

The **Resolved** property is TRUE if all of the recipients in the collection are resolved. Read-only.

Syntax

objRecipColl.**Resolved**

Data Type

Boolean

Remarks

All Recipient objects in the collection are considered resolved when all Recipient objects have a valid AddressEntry object in the **AddressEntry** property.

You should resolve all addresses.   Whenever you supply a display name to obtain an address from the address book or supply a custom address, you should call the **Resolve** method to ensure that the **AddressEntry** property is valid.

When the Recipients collection's **Resolved** property is not TRUE, use either the collection's **Resolve** method or the **Resolve** method for each Recipient object in the collection.

When you use existing valid AddressEntry objects, you do not need to explicitly call the Recipient object's **Resolve** method.

See Also

Resolve Method (Recipient Object)

Session Object

The Session object contains session-wide settings and options. It also contains properties that return top-level objects, such as **CurrentUser**.

Properties

Property name	Type	Access
Application	String	Read-only
Class	Long	Read-only
CurrentUser	AddressEntry object	Read-only
Inbox	Folder object	Read-only
InfoStores	InfoStores object	Read-only
MAPIOBJECT	Object	Read/write (Note: Not available to Visual Basic applications.)
Name	String	Read-only
OperatingSystem	String	Read-only
Outbox	Folder object	Read-only
Parent	Object; set to **Nothing**	(Not applicable)
Session	Object; set to **Nothing**	(Not applicable)
Version	String	Read-only

Methods

Method name	Parameters
AddressBook	(optional) recipients as Object, (optional) title as String, (optional) oneAddress as Boolean, (optional) forceResolution as Boolean, (optional) recipLists as long, (optional) toLabel as String, (optional) ccLabel as String, (optional) bccLabel as String, (optional) parentWindow as Long
GetAddressEntry	entryID as String
GetInfoStore	storeID as String
GetFolder	folderID as String, storeID as String
GetMessage	messageID as String, storeID as String
Logoff	(none)
Logon	(optional) profileName as String, (optional) profilePassword as String, (optional) showDialog as Boolean, (optional) newSession as Boolean, (optional) parentWindow as Long

Remarks

After you create a new Session object, use the **Logon** method to initiate a MAPI session.

See Also

Logon Method (Session Object)

AddressBook Method (Session Object)

The **AddressBook** method displays the MAPI dialog box that allows the user to select entries from the address book. The selections are returned in a Recipients collection object.

Syntax

Set *objRecipients* **=** *objSession*.**AddressBook([** *recipients*, *title, oneAddress, forceResolution, recipLists, toLabel, ccLabel, bccLabel, parentWindow* **])**

Parameters

objRecipients
On successful return, the Recipients collection object. When the user does not select any names from the dialog box, **AddressBook** returns **Nothing**.

objSession
Required. The Session object.

recipients
Optional. Object. A Recipients collection object that provides the initial value for the recipient list boxes in the address book. (Note: This initial Recipient collection is ignored in the OLE Messaging Library.)

title
Optional. String. The title or caption of the address book dialog box. The default value is an empty string.

oneAddress
Optional. Boolean. Allows the user to enter or select only one address. The default value is FALSE.

forceResolution
Optional. Boolean. If TRUE, attempts to resolve all names before closing the address book. Prompts the user to resolve any ambiguous names. The default value is TRUE.

recipLists
Optional. Long. The number of recipient list boxes to display in the address book dialog box:

recipLists	Action
0	Displays no list boxes. The user can interact with the address book dialog box but no recipients are returned by this method.
1	Displays one list box (default) for mapiTo recipients.
2	Displays two list boxes; mapiTo and mapiCc recipients.
3	Displays three list boxes; mapiTo, mapiCc, and mapiBcc recipients.

toLabel
Optional. String. The caption for the button associated with the first list box. Ignored if *recipLists* is less than 1. If omitted, the default value "To:" is displayed.

ccLabel
Optional. String. The caption for the button associated with the second list box. Ignored if *recipLists* is less than 2. If omitted, the default value "CC:" is displayed.

bccLabel
Optional. String. The caption for the button associated with the third list box. Ignored if *recipLists* is less than 3. If omitted, the default value "BCC:" is displayed.

parentWindow
Optional. Long. The parent window handle for the address book dialog box. A value of **0** (the default)

specifies that any dialog box displayed is application modal.

Remarks

The **AddressBook** method returns **Nothing** if the user cancels the dialog box.

To provide an access key for the list boxes, include an ampersand (&) character in the string for the label argument. For example, if *toLabel* is "&Attendees:", users can press ALT+A to move the focus to the first recipient list box.

When you use the **AddressBook** method to let the user select recipients for a new message, you must use two different Recipients collections. This is required because the **Recipients** property of the Message object is read-only. Use the following procedure:

1. Call **Session.AddressBook**, which returns a new Recipients collection.
2. Call **Messages.Add** to create a new message.
3. Loop through the Recipients collection returned by **Session.AddressBook**, adding each recipient to the message's Recipients collection by calling the message's **Recipients.Add** method.

Note that you must update both the **Recipient** and the **AddressEntry** properties of the destination **Recipient** object, as demonstrated in the following example:

```
Dim objNewRecip(MAX_RECIPS) as Object   ' array of new recipients
'...
    Set objNewMessage = objSession.Outbox.Messages.Add
    Set objRecipColl = objSession.AddressBook( _
           Title:="Select Recipients", _
           recipLists:=3) 'use default labels
    ' now add to the new message
    Count = objRecipColl.Count   ' error checking omitted...
    With objNewMessage.Recipients
        For i = 1 To Count Step 1
            Set objRecip = objRecipColl.Item(i)
            Set objNewRecip(i) = .Add( _
                entryID:=objRecip.AddressEntry.ID, _
                type:=objRecip.type)
            objNewRecip(i).Name = objNewRecip(i).AddressEntry.Name
            objNewRecip(i).address = objNewRecip(i).AddressEntry.type _
                & ":" & objNewRecip(i).AddressEntry.address
            objNewRecip(i).Resolve
        Next i
    End With
    objNewMessage.Update  'save it
```

The following methods can also invoke MAPI dialog boxes: **Delete** and **Details** methods (AddressEntry object), **Options** and **Send** methods (Message object), **Resolve** method (Recipient object and Recipients collection), **Logon** method (Session object).

Note The initial Recipients collection, as specified in the *recipients* parameter, is not used in the OLE Messaging Library.

Example

The following example displays an address book dialog box labeled "Select Attendees" with three recipient lists:

```
    If objSession Is Nothing Then
        MsgBox "must first create MAPI session and logon"
        Exit Function
    End If
    Set objRecipColl = objSession.AddressBook( _
        Title:="Select Attendees", _
        forceResolution:=True, _
        recipLists:=3, _
        toLabel:="&Very Important People")  ' on button
        ccLabel:="&Carbon Recipients")
        bccLabel:="&Secret Recipients")
    ' Note: initial value not used in version 1.0
    ' parameter not used in call: Recipients:=objInitRecipColl
    MsgBox "Name of first recipient = " & objRecipColl.Item(1).Name
    Exit Function
```

See Also

AddressEntry Object

Recipients Collection

CurrentUser Property (Session Object)

The **CurrentUser** property returns the active user as an AddressEntry object. Read-only.

Syntax

objSession.**CurrentUser**

Data Type

Object (AddressEntry object)

Remarks

The **CurrentUser** property returns **Nothing** when no user is logged on.

Example

The example logs on if necessary, then creates strings containing information about the current user:

```
If objSession Is Nothing Then
    MsgBox ("Must log on first")
    Exit Function
End If
Set objAddrEntry = objSession.CurrentUser
If objAddrEntry Is Nothing Then
    MsgBox "Could not set the address entry object"
    Exit Function
Else
    MsgBox "full address = " & objAddrEntry.Type & ":" _
             & objAddrEntry.Address
End If
```

See Also

AddressEntry Object

GetAddressEntry Method (Session Object)

The **GetAddressEntry** method returns an AddressEntry object.

Syntax

Set *objAddressEntry* **=** *objSession*.**GetAddressEntry(***entryID***)**

Parameters

objAddressEntry
  On successful return, represents the AddressEntry object specified by *entryID*.

objSession
  Required. The Session object.

entryID
  Required. String that specifies the unique identifier of the address entry.

Example

The following example displays the name of a user from a MAPI address list:

```
' from the function Session_GetAddressEntry
    If objSession Is Nothing Then
        MsgBox "No active session, must log on"
        Exit Function
    End If
    If "" = strAddressEntryID Then
        MsgBox ("Must first set string variable; see AddressEntry->ID")
        Exit Function
    End If
    Set objAddrEntry = objSession.GetAddressEntry(strAddressEntryID)
    MsgBox "full address = " & objAddrEntry.Type & ":" _
            & objAddrEntry.Address
```

See Also

Using Addresses

AddressEntry Object

GetFolder Method (Session Object)

GetMessage Method (Session Object)

GetFolder Method (Session Object)

The **GetFolder** method returns a Folder object from a MAPI information store.

Syntax

Set *objFolder* = *objSession*.**GetFolder(***folderID* **[**, *storeID***])**

Parameters

objFolder
: On successful return, contains the Folder object with the specified identifier. When the folder does not exist, **GetFolder** returns **Nothing**.

objSession
: Required. The Session object.

folderID
: Required. String that specifies the unique identifier of the folder. When you provide an empty string, some providers return the root folder. See the remarks for more information.

storeID
: Optional. String that specifies the unique identifier of the store. The default value is an empty string, which corresponds to the default store.

Remarks

The **GetFolder** method allows you to obtain any folder for which you know the identifier.

For some message stores, you can also obtain the root folder by supplying an empty string as the value for *folderID*. When the message store does not support returning the root folder, the call returns the error value MAPI_E_NOT_FOUND.

The root folder is the parent of the folder that represents the IPM subtree. The IPM subtree is the hierarchy of folders for all messages whose message class starts with IPM, such as IPM.Note.

Note that the root folder differs from the IPM root folder. You can obtain the IPM root folder with the InfoStores object's **RootFolder** property, or when you use the **Parent** property to trace up through the hierarchy of folders. You can only obtain the root folder by calling **GetFolder** with the empty string supplied for the *folderID* parameter.

Example

The following example uses the **GetFolder** method to obtain a specific folder from a MAPI information store:

```
' from the function Session_GetFolder
' requires a global variable that contains the folder ID
' uses a global variable that contains the store ID if present
    If strFolderID = "" Then
        MsgBox ("Must first set folder ID variable; see Folder->ID")
        Exit Function
    End If
    If strFolderStoreID = "" Then ' if it's not there, don't use it
        Set objFolder = objSession.GetFolder(strFolderID)
    Else
        Set objFolder = objSession.GetFolder(folderID:=strFolderID, _
                                    storeID:=strFolderStoreID)
    End If
    If objFolder Is Nothing Then
        Set objMessages = Nothing
        MsgBox "Unable to retrieve folder with specified ID"
        Exit Function
    End If
    MsgBox "Folder set to " & objFolder.Name
    Set objMessages = objFolder.Messages
```

See Also

Folder Object

ID Property (Folder Object)

GetInfoStore Method (Session Object)

The **GetInfoStore** method returns an InfoStore object that can be used to navigate through both public folders and the user's personal folders.

Syntax

Set *objInfoStore* = *objSession*.**GetInfoStore(***storeID***)**

Parameters

objInfoStore
: On successful return, contains the InfoStore object with the specified identifier. When the InfoStore object does not exist, **GetInfoStore** returns **Nothing**.

objSession
: Required. The Session object.

storeID
: Required. String that specifies the unique identifier of the store to retrieve.

Remarks

The **GetInfoStore** method allows you to obtain any information store for which you know the identifier.

Example

The following example uses the **GetInfoStore** method to obtain a specific store:

```
' from the function Session_GetInfoStore
' requires a global variable that contains the info store ID
Dim strInfoStoreID as String ' ID as hex string
Dim objInfoStore As Object  ' InfoStore object
    If strInfoStoreID = "" Then
        MsgBox ("Must first set Store ID variable")
        Exit Function
    End If
    Set objInfoStore = objSession.GetInfoStore(
                          storeID:=strInfoStoreID)
    ' error handling...
    MsgBox "InfoStore set to " & objInfoStore.Name
```

See Also

Folder Object

ID Property (Folder Object)

GetMessage Method (Session Object)

The **GetMessage** method returns a Message object from a MAPI information store.

Syntax

Set *objMessage* = *objSession*.**GetMessage(***messageID* **[**, *storeID***])**

Parameters

objMessage
: On successful return, **GetMessage** returns a Message object. When the specified *messageID* does not exist, **GetMessage** returns **Nothing**.

objSession
: Required. The Session object.

messageID
: Required. String that specifies the unique identifier of the message.

storeID
: Optional. String that specifies the unique identifier of the store. The default value is an empty string, which corresponds to the default store.

Example

The following example displays the subject of a message from a MAPI information store:

```
' fragment from Session_GetMessage
' requires the parameter strMessageID;
' also uses strMessageStoreID if it is defined
    If strMessageID = "" Then
        MsgBox ("Must first set message ID variable; see Message->ID")
        Exit Function
    End If
    If strMessageStoreID = "" Then ' not present
        Set objOneMsg = objSession.GetMessage(strMessageID)
    Else
        Set objOneMsg = objSession.GetMessage(messageID:=strMessageID, _
                                    storeID:=strMessageStoreID)
    End If
```

See Also

ID Property (Message Object)

Message Object

Inbox Property (Session Object)

The **Inbox** property returns a Folder object representing the current user's default Inbox folder. Read-only.

Syntax

objSession.**Inbox**

Data Type

Object (Folder object)

Remarks

This property returns **Nothing** when the current user does not have an Inbox folder.

In addition to the general ability to navigate through the formal collection and object hierarchy, the OLE Messaging Library supports properties that let your application directly access the most common folder objects:

- The IPM subtree
- Inbox
- Outbox

The following sample code uses the Session object's **Inbox** property to initialize a Folder object:

```
' from the function Session_Inbox
    '   make sure the Session object is valid...
    Set objFolder = objSession.Inbox
    If objFolder Is Nothing Then
        MsgBox "Failed to open Inbox"
        Exit Function
    End If
    MsgBox "Folder name = " & objFolder.Name
    Set objMessages = objFolder.Messages
    If objMessages Is Nothing Then
        MsgBox "Failed to open folder's Messages collection"
        Exit Function
    End If
```

See Also

Folder Object

Outbox Property (Session Object)

InfoStores Property (Session Object)

The **InfoStores** property returns an InfoStores object representing a collection of available information stores. Each InfoStore object contains a root folder object. Read-only.

Syntax

objSession.**InfoStores**

Data Type

Object (InfoStores object)

Remarks

You can access public folders through the InfoStores collection. The public folders are maintained in their own InfoStore object that is distinct from the InfoStore object that contains the user's personal messages.

When you know the unique identifier for the InfoStore object, you can also call the Session object's **GetInfoStore** method.

For more information, see the reference documentation for the InfoStores collection and the InfoStore object.

Example

```
' from the functions Session_InfoStores, InfoStores_FirstItem,
' and InfoStore.Name
'   make sure the Session object is valid...
Dim objSession as Object         ' Session object
Dim objInfoStoresColl as Object  ' InfoStores collection
Dim objInfoStore as Object       ' InfoStore object
' assume valid Session object
    Set objInfoStoresColl = objSession.InfoStores
    If objInfoStoresColl Is Nothing Then
        MsgBox "Could not set InfoStores collection"
        Exit Function
    End If
    If 0 = objInfoStoresColl.Count Then
        MsgBox "No InfoStores in the collection"
        Exit Function
    End If
    iInfoStoresCollIndex = 1
    Set objInfoStore = objInfoStoresColl.Item(iInfoStoresCollIndex)
    If objInfoStore Is Nothing Then
        MsgBox "error, cannot get this InfoStore object"
        Exit Function
    Else
        MsgBox "Selected InfoStores " & iInfoStoresCollIndex
    End If
    If "" = objInfoStore.Name Then
        MsgBox "Active InfoStore has no name; ID = " & objInfoStore.Id
    Else
        MsgBox "Active InfoStore has name: " & objInfoStore.Name
    End If
```

See Also

InfoStores Collection

InfoStore Object

Logoff Method (Session Object)

The **Logoff** method logs off from the MAPI system.

Syntax

objSession.**Logoff()**

Parameters

objSession
Required. The Session object.

Example

The following example logs off from the MAPI system:

```
' from the function Session_Logoff
    If Not objSession Is Nothing Then
        objSession.Logoff
        MsgBox "Logged off; reset global variables"
    Else
        MsgBox "No active session"
    End If
```

See Also

Logon Method (Session Object)

Logon Method (Session Object)

The **Logon** method logs on to the MAPI system.

Syntax

objSession.**Logon([** *profileName, profilePassword, showDialog, newSession, parentWindow* **])**

Parameters

objSession
Required. The Session object.

profileName
Optional. A string specifying the user's logon name. To prompt the user to enter a logon name, omit *profileName* and set *showDialog* to TRUE. The default value is an empty string.

profilePassword
Optional. A string specifying the user's logon password. To prompt the user to enter a logon password, omit *profilePassword* and set *showDialog* to TRUE. The default value is an empty string.

showDialog
Optional. Boolean. If TRUE, displays a logon dialog box. The default value is TRUE.

newSession
Optional. Boolean. Determines whether the application opens a new MAPI session or uses the current shared MAPI session. If a shared MAPI session does not exist, *newSession* is ignored and a new session is opened. If the shared MAPI session does exist, this argument takes the following action:

Value	Action
TRUE	Opens an new MAPI session.
FALSE (default)	Uses the current shared MAPI session.

parentWindow
Optional. Long (HWND). Specifies the parent window handle for the logon dialog box. A value of **0** (the default) specifies that any dialog box displayed is application modal. The *parentWindow* parameter is ignored unless *showDialog* is TRUE.

Remarks

The user must log on before your application can use most MAPI objects.

The common MAPI dialog boxes automatically handle many of the error cases that can be encountered during logon. When you call **Logon** and do not supply the optional profile name parameter, the **Choose Profile** dialog box appears, asking the user to select a profile. When the *profileName* parameter is supplied but is not valid, common dialog boxes indicate the error and prompt the user to enter a valid name from the **Choose Profile** dialog box. When no profiles are defined, the Profile Wizard walks the user through the creation of a new profile.

When your application calls the **Logon** method after the user has already successfully logged on, the OLE Messaging Library generates the error MAPI_E_LOGON_FAILURE.

The following methods can also invoke MAPI dialog boxes: **Delete** and **Details** methods (AddressEntry object), **Options** and **Send** methods (Message object), **Resolve** method (Recipient object and Recipients collection), **AddressBook** method (Session object).

Example

The first example displays a logon dialog box that prompts the user to enter a logon password. The second example supplies the *profileName* parameter and does not display the dialog box:

```
' from the function Session_Logon
    Set objSession = CreateObject("MAPI.Session")
    If Not objSession Is Nothing Then
        objSession.Logon showDialog:=True
    End If

' from the function Session_Logon_NoDialog
Function Session_Logon_NoDialog()
    On Error GoTo error_olemsg
    ' can set strProfileName, strPassword from a custom form
    ' adjust these parameters for your configuration
    ' create a Session object if necessary here...
    '
    If Not objSession Is Nothing Then
        ' configure these parameters for your needs...
        objSession.Logon profileName:=strProfileName, showDialog:=False
    End If
    Exit Function

error_olemsg:
    If 1273 = Err Then
        MsgBox "cannot logon: incorrect profile name or password; change
global variable strProfileName in Util_Initialize"
        Exit Function
    End If
    MsgBox "Error " & Str(Err) & ": " & Error$(Err)
    Resume Next
End Function
```

See Also

Starting A Session With MAPI

Logoff Method (Session Object)

MAPIOBJECT Property (Session Object)

The **MAPIOBJECT** property returns an **IUnknown** pointer to this Session object. Not available to Visual Basic applications. Read/write.

Syntax

objSession.**MAPIOBJECT**

Data Type

Variant (VT_UNKNOWN)

Remarks

The **MAPIOBJECT** property is not available to Visual Basic programs. It is available only to C/C++ programs that use the OLE Messaging Library. The **MAPIOBJECT** property is an **IUnknown** object, which is not supported by Visual Basic. Visual Basic supports **IDispatch** objects. For more information, see the Microsoft *OLE Programmer's Reference*.

See Also

Introduction to OLE Automation

Name Property (Session Object)

The **Name** property returns the name of the profile logged on to this session. Read-only.

Syntax

objSession.**Name**

Data Type

String

Remarks

The **Name** property corresponds to the MAPI property PR_DISPLAY_NAME.

Examples

```
' from the function Session_Name
    If objSession Is Nothing Then
        MsgBox "Must log on first: see Session menu"
        Exit Function
    End If
    MsgBox "Session name = " & objSession.Name
```

See Also

Session Object

OperatingSystem Property (Session Object)

The **OperatingSystem** property returns the name and version number of the current operating system. Read-only.

Syntax

objSession.**OperatingSystem**

Data Type

String

Remarks

The OLE Messaging Library returns strings in the following formats:

Operating system	**String value**
Microsoft Windows NT	Microsoft® Windows NT™ *X.xx*
Microsoft Windows for Workgroups	Microsoft® Windows™ *X.xx*

The *X.xx* values are replaced with the actual version numbers. Note that Microsoft Windows for Workgroups version 3.11 returns the string "Microsoft® Windows™ 3.10." This is a feature of that operating system rather than a feature of the OLE Messaging Library.

Example

This example displays the name of the operating system:

```
' from the function Session_OperatingSystem
' assume objSession is a valid Session object
    MsgBox "Operating system = " & objSession.OperatingSystem
```

See Also

Version Property (Session Object)

Outbox Property (Session Object)

The **Outbox** property returns a Folder object representing the current user's default Outbox folder. Read-only.

Syntax

objSession.**Outbox**

Data Type

Object

Remarks

The property returns **Nothing** if the current user does not have or has not enabled the Outbox folder.

In addition to the general ability to navigate through the formal collection and object hierarchy, the OLE Messaging Library supports properties that let your application directly access the most common folder objects:

- The IPM subtree
- Inbox
- Outbox

Example

```
' from the function Session_Outbox
Dim objFolder As Object
' ...
    Set objFolder = objSession.Outbox
    If objFolder Is Nothing Then
        MsgBox "Failed to open Outbox"
        Exit Function
    End If
    MsgBox "Folder name = " & objFolder.Name
    Set objMessages = objFolder.Messages
```

See Also

Folder Object

Inbox Property (Session Object)

Version Property (Session Object)

The **Version** property returns the version number of the OLE Messaging Library as a string, for example, "1.00". Read-only.

Syntax

objSession.**Version**

Data Type

String

Remarks

The version number for the OLE Messaging Library is represented by a string in the form *n.xx*, where *n* represents a major version number and *xx* represents a minor version number.

Example

```
' see the function Session_Version
    Dim objSession As Object
    Set objSession = CreateObject("MAPI.Session")
    ' error handling here...
    MsgBox "Version number is " & objSession.Version
MsgBox "Welcome to OLE Messaging Library version " & objSession.Version
```

See Also

OperatingSystem Property (Session Object)

Error Codes

MAPI can return the following values to the OLE Messaging Library:

MAPI error code	HRESULT (VB4 error value) (hexadecimal)	Low-order word + 1000 (VBA error value) (decimal)
MAPI_E_AMBIGUOUS_RECIP	0x80040700	2792
MAPI_E_BAD_CHARWIDTH	0x80040103	1259
MAPI_E_BAD_COLUMN	0x80040118	1280
MAPI_E_BAD_VALUE	0x80040301	1769
MAPI_E_BUSY	0x8004010b	1267
MAPI_E_CALL_FAILED	0x80004005	17389
MAPI_E_CANCEL	0x80040501	2281
MAPI_E_COLLISION	0x80040604	2540
MAPI_E_COMPUTED	0x8004011a	1282
MAPI_E_CORRUPT_DATA	0x8004011b	1283
MAPI_E_CORRUPT_STORE	0x80040600	2536
MAPI_E_DECLINE_COPY	0x80040306	1774
MAPI_E_DISK_ERROR	0x80040116	1278
MAPI_E_END_OF_SESSION	0x80040200	1512
MAPI_E_EXTENDED_ERROR	0x80040119	1281
MAPI_E_FAILONEPROVIDER	0x8004011d	1285
MAPI_E_FOLDER_CYCLE	0x8004060b	2547
MAPI_E_HAS_FOLDERS	0x80040609	2545
MAPI_E_HAS_MESSAGES	0x8004060a	2546
MAPI_E_INTERFACE_NOT _SUPPORTED	0x80004002	17386
MAPI_E_INVALID_BOOKMARK	0x80040405	2029
MAPI_E_INVALID_ENTRYID	0x80040107	1263
MAPI_E_INVALID_OBJECT	0x80040108	1264
MAPI_E_INVALID_PARAMETE R	0x80070057	1087
MAPI_E_INVALID_TYPE	0x80040302	1770
MAPI_E_LOGON_FAILED	0x80040111	1273
MAPI_E_MISSING_REQUIRED _COLUMN	0x80040202	1514
MAPI_E_NETWORK_ERROR	0x80040115	1277
MAPI_E_NO_ACCESS	0x80070005	1005
MAPI_E_NO_RECIPIENTS	0x80040607	2543
MAPI_E_NO_SUPPORT	0x80040102	1258
MAPI_E_NO_SUPPRESS	0x80040602	2538
MAPI_E_NON_STANDARD	0x80040606	2542
MAPI_E_NOT_ENOUGH_DISK	0x8004010d	1269

MAPI_E_NOT_ENOUGH_MEMORY	0x8007000e	1014
MAPI_E_NOT_ENOUGH_RESOURCES	0x8004010e	1270
MAPI_E_NOT_FOUND	0x8004010f	1271
MAPI_E_NOT_IN_QUEUE	0x80040601	2537
MAPI_E_NOT_INITIALIZED	0x80040605	2541
MAPI_E_NOT_ME	0x80040502	2282
MAPI_E_OBJECT_CHANGED	0x80040109	1265
MAPI_E_OBJECT_DELETED	0x8004010a	1266
MAPI_E_SESSION_LIMIT	0x80040112	1274
MAPI_E_STRING_TOO_LONG	0x80040105	1261
MAPI_E_SUBMITTED	0x80040608	2544
MAPI_E_TABLE_EMPTY	0x80040402	2026
MAPI_E_TABLE_TOO_BIG	0x80040403	2027
MAPI_E_TIMEOUT	0x80040401	2025
MAPI_E_TOO_BIG	0x80040305	1773
MAPI_E_TOO_COMPLEX	0x80040117	1279
MAPI_E_TYPE_NO_SUPPORT	0x80040303	1771
MAPI_E_UNABLE_TO_ABORT	0x80040114	1276
MAPI_E_UNABLE_TO_COMPLETE	0x80040400	2024
MAPI_E_UNCONFIGURED	0x8004011c	1284
MAPI_E_UNEXPECTED_ID	0x80040307	1775
MAPI_E_UNEXPECTED_TYPE	0x80040304	1772
MAPI_E_UNKNOWN_ENTRYID	0x80040201	1513
MAPI_E_UNKNOWN_FLAGS	0x80040106	1262
MAPI_E_USER_CANCEL	0x80040113	1275
MAPI_E_VERSION	0x80040110	1272
MAPI_E_WAIT	0x80040500	2280
MAPI_W_APPROX_COUNT	0x00040482	2154
MAPI_W_CANCEL_MESSAGE	0x00040580	2408
MAPI_W_ERRORS_RETURNED	0x00040380	1896
MAPI_W_NO_SERVICE	0x00040203	1515
MAPI_W_PARTIAL_COMPLETION	0x00040680	2664
MAPI_W_POSITION_CHANGED	0x00040481	2153

The following table lists the MAPI return values in numeric order:

HRESULT (VB4 error value) (hexadecimal)	**Low-order word + 1000 (VBA error value) (decimal)**	**MAPI error code**

0x00040203	1515	MAPI_W_NO_SERVICE
0x00040380	1896	MAPI_W_ERRORS_RETURNED
0x00040481	2153	MAPI_W_POSITION_CHANGED
0x00040482	2154	MAPI_W_APPROX_COUNT
0x00040580	2408	MAPI_W_CANCEL_MESSAGE
0x00040680	2664	MAPI_W_PARTIAL_COMPLETION
0x80004002	17386	MAPI_E_INTERFACE_NOT_SUPPORTED
0x80004005	17389	MAPI_E_CALL_FAILED
0x80040102	1258	MAPI_E_NO_SUPPORT
0x80040103	1259	MAPI_E_BAD_CHARWIDTH
0x80040105	1261	MAPI_E_STRING_TOO_LONG
0x80040106	1262	MAPI_E_UNKNOWN_FLAGS
0x80040107	1263	MAPI_E_INVALID_ENTRYID
0x80040108	1264	MAPI_E_INVALID_OBJECT
0x80040109	1265	MAPI_E_OBJECT_CHANGED
0x8004010a	1266	MAPI_E_OBJECT_DELETED
0x8004010b	1267	MAPI_E_BUSY
0x8004010d	1269	MAPI_E_NOT_ENOUGH_DISK
0x8004010e	1270	MAPI_E_NOT_ENOUGH_RESOURCES
0x8004010f	1271	MAPI_E_NOT_FOUND
0x80040110	1272	MAPI_E_VERSION
0x80040111	1273	MAPI_E_LOGON_FAILED
0x80040112	1274	MAPI_E_SESSION_LIMIT
0x80040113	1275	MAPI_E_USER_CANCEL
0x8004011	1276	MAPI_E_UNABLE_TO_ABORT

4		
0x80040115	1277	MAPI_E_NETWORK_ERROR
0x80040116	1278	MAPI_E_DISK_ERROR
0x80040117	1279	MAPI_E_TOO_COMPLEX
0x80040118	1280	MAPI_E_BAD_COLUMN
0x80040119	1281	MAPI_E_EXTENDED_ERROR
0x8004011a	1282	MAPI_E_COMPUTED
0x8004011b	1283	MAPI_E_CORRUPT_DATA
0x8004011c	1284	MAPI_E_UNCONFIGURED
0x8004011d	1285	MAPI_E_FAILONEPROVIDER
0x80040200	1512	MAPI_E_END_OF_SESSION
0x80040201	1513	MAPI_E_UNKNOWN_ENTRYID
0x80040202	1514	MAPI_E_MISSING_REQUIRED_COLUMN
0x80040301	1769	MAPI_E_BAD_VALUE
0x80040302	1770	MAPI_E_INVALID_TYPE
0x80040303	1771	MAPI_E_TYPE_NO_SUPPORT
0x80040304	1772	MAPI_E_UNEXPECTED_TYPE
0x80040305	1773	MAPI_E_TOO_BIG
0x80040306	1774	MAPI_E_DECLINE_COPY
0x80040307	1775	MAPI_E_UNEXPECTED_ID
0x80040400	2024	MAPI_E_UNABLE_TO_COMPLETE
0x80040401	2025	MAPI_E_TIMEOUT
0x80040402	2026	MAPI_E_TABLE_EMPTY
0x80040403	2027	MAPI_E_TABLE_TOO_BIG
0x80040405	2029	MAPI_E_INVALID_BOOKMARK

0x80040500	2280	MAPI_E_WAIT
0x80040501	2281	MAPI_E_CANCEL
0x80040502	2282	MAPI_E_NOT_ME
0x80040600	2536	MAPI_E_CORRUPT_STORE
0x80040601	2537	MAPI_E_NOT_IN_QUEUE
0x80040602	2538	MAPI_E_NO_SUPPRESS
0x80040604	2540	MAPI_E_COLLISION
0x80040605	2541	MAPI_E_NOT_INITIALIZED
0x80040606	2542	MAPI_E_NON_STANDARD
0x80040607	2543	MAPI_E_NO_RECIPIENTS
0x80040608	2544	MAPI_E_SUBMITTED
0x80040609	2545	MAPI_E_HAS_FOLDERS
0x8004060a	2546	MAPI_E_HAS_MESSAGES
0x8004060b	2547	MAPI_E_FOLDER_CYCLE
0x80040700	2792	MAPI_E_AMBIGUOUS_RECIP
0x80070005	1005	MAPI_E_NO_ACCESS
0x8007000e	1014	MAPI_E_NOT_ENOUGH_MEMORY
0x80070057	1087	MAPI_E_INVALID_PARAMETER

How Programmable Objects Work

How do programmable objects work? How does the OLE Messaging Library offer its powerful ability to create and manage messaging objects?

This appendix provides a very short introduction to the Microsoft Component Object Model, OLE Automation, and the OLE programmability interface **IDispatch**. For complete details, see the *OLE Programmer's Reference*.

You do not need to understand this material in order to use the OLE Messaging Library.

COM Interfaces

With the combination of Microsoft RPC (Remote Procedure Call) and Microsoft OLE technology, Microsoft began to shift the C/C++ programming model from individual API functions, such as those offered in the Windows 3.1 SDK and Win32 SDK, to a distributed object model that is based on *interfaces*. An interface is simply a group of logically-related functions. Note that the interface consists only of functions. There are no facilities for directly accessing data within an interface, except through the functions.

The benefit of such a distributed object model is that it allows developers to create small, independent, self-managing software objects. This modular approach allows software functionality to be developed in small "building blocks" that are then fitted together. Your application no longer has to handle every possible data format or possible application feature, as long as it can be integrated with other objects that can handle the desired formats and features.

The notion of objects is very familiar to Visual Basic developers. Many software industry analysts have noted that the most visible success of object-oriented programming to date is the widespread use of Microsoft Visual Basic custom controls.

One of the benefits of the modular, interface-based approach to software development is that individual interfaces usually contain significantly fewer functions than libraries, with the promise of more efficient use of memory. Whenever you want to use one function in a library, the entire library must be loaded into memory. Splitting function libraries into smaller interfaces makes it more likely that you load only the functions that you actually need. (Or at least that you load fewer that you don't need.)

By convention, interface names start with the letter "I". The functions are given a specific ordering within the interface. Knowing the order of the functions is important for developers who must define their own *vtables*, or function dispatch tables. The C++ compiler creates vtables for you, but if you are writing in C, you must create your own.

The functions of an interface still physically reside in an .EXE or .DLL file, but Microsoft has defined new rules for how these files are registered on the system and how they are loaded and unloaded from memory. Microsoft refers to the new rules as the *Component Object Model,* or *COM*.

According to the rules, the first three functions in all interfaces are always **QueryInterface** (which developers call "QI"), **AddRef**, and **Release**. These functions provide a pointer to the interface when someone asks for it, keep track of the number of programs that are being served by the interface, and control how the physical .DLL or .EXE file gets loaded and unloaded. Any other functions in the interface are defined by the person who creates the interface. The interface that consists of these three common functions, **QueryInterface, AddRef, and Release**, is called **IUnknown**. Developers can always obtain a pointer to an **IUnknown** object.

The component object model, like RPC before it, makes a strong distinction between the definition of the interface and its implementation. The interface functions and the data items that make up the parameters are defined in a very precise way, using a special language designed specifically for defining interfaces. These languages (such as MIDL, the Microsoft Interface Definition Language, and ODL, the Object Definition Language) do not allow you to use indefinite type names, such as **void ***, or types that change from computer to computer, such as **int**. The goal is to force you to specify the exact size of all data. This makes it possible for one person to define an interface, a second person to implement the interface, and a third person to write a program that calls the interface.

Developers who write C and C++ code that use these types of interfaces read the object's interface definition language (IDL) files. They know exactly what functions are present in the interface and what data is required. They can call the interfaces directly.

For developers who are not writing in C and C++, or do not have access to the object's interface definition language files, Microsoft's component object model defines another way to use software components. This is based on an interface named **IDispatch**.

IDispatch

IDispatch is a COM interface that is designed in such a way that it can call virtually any other COM interface. Developers working in Visual Basic often cannot call COM interfaces directly, as they would from C or C++. However, when their tool supports **IDispatch**, as Visual Basic does, and when the object they want to call supports **IDispatch**, they can call its COM interfaces *indirectly*.

The main method offered by **IDispatch** is called **Invoke.** This method adds a level of indirection to the control flow of the Component Object Model. In the standard model, an object obtains a pointer to an interface and then calls a member function of the interface. **IDispatch** adds a level of indirection. Instead of directly calling the member function of the interface, the program calls **IDispatch::Invoke**, and **IDispatch::Invoke** calls the member function for you.

Invoke is a general method-calling machine. Its parameters include a value that identifies the method that is to be called and the parameters that are to be sent to it. In order to be able to handle the wide variety of parameters that other COM methods use, **Invoke** uses a self-describing data structure called a VARIANTARG.

The VARIANTARG structure contains two parts: a type field, which represents the data type, and a data field, which represents the actual value of the data. The values such as VT_I2, VT_I4, and so on, are the constants that define valid values for the data types.

Associated with **IDispatch** is the notion of a *type library*. The type library publishes information about an interface so that it is available to Visual Basic programs. The type library, or *typelib*, contains the same kind of information that C or C++ programmers would obtain from a header file: the name of the method and the sequence and types of its parameters.

An executable or DLL that exposes **IDispatch** and its type library is known as an *OLE Automation server*. The OLE Messaging Library is such a server.

The OLE Messaging Library: An OLE Automation Server

So, let's put it all together, from the bottom up, to see how the OLE Messaging Library works.

- Service providers implement COM interfaces–specifically, the MAPI interfaces–as described in the MAPI documentation.
- The OLE Messaging Library implements several objects (Session, Message, etc.) that act as *clients* to these MAPI interfaces. That is, the OLE Messaging Library objects obtain pointers to the MAPI interfaces and call methods.
- The OLE Messaging Library implements **IDispatch** and acts as an OLE Automation *server* so that it can be called by tools that can use **IDispatch**, such as Visual Basic. That is, the OLE Messaging Library allows other programs to call its **IDispatch** interface. It provides its own registration (.REG) file so that it can be registered on a computer as an OLE Automation server.
- The OLE Messaging Library publishes a type library that contains information about the objects that it makes available through **IDispatch**.
- Your Visual Basic application acts as a client to the OLE Messaging Library. It reads the OLE Messaging Library's type library to obtain information about the objects, methods, and properties. When your Visual Basic application declares a variable as an object (with code such as "Dim objSession as Object") and uses that object's properties and methods (with code such as "MsgBox objSession.Class"), Visual Basic makes calls to **IDispatch** on your behalf.

The relationships between these programs are shown in the following diagram. Visual Basic is a client to the OLE Automation server, the OLE Messaging Library. The OLE Messaging Library, in turn, acts as a client to the MAPI services.

{ewc msdncd, EWGraphic, groupx843 0 /a "MAPI.BMP"}

The OLE Messaging Library and MAPI

The OLE Messaging Library calls Microsoft MAPI interfaces for you. The following table describes the MAPI interfaces that the OLE Messaging Library calls when you manipulate an OLE Messaging Library object.

OLE Messaging Library object	MAPI or OLE interface called by the OLE Messaging Library
AddressEntry	IABContainer, IMAPIProp
Attachment	IAttach
Field	IStream, IMAPIProp
Folder	IMAPIFolder
InfoStore	IMsgStore
Message	IMessage
Recipient	IMAPIProp
Session	IMAPISession

For collection objects, the OLE Messaging Library calls the MAPI interface **IMAPITable.**

The OLE Messaging Library also calls the MAPI interface **IMAPIProp**. Many of the properties exposed by the OLE Messaging Library are based on MAPI properties. The following table describes the mapping between some OLE Messaging Library properties and the underlying MAPI properties.

OLE Messaging Library object	Property	MAPI property	MAPI property type
AddressEntry	Address	PR_EMAIL_ADDRESS	PT_TSTRING
AddressEntry	ID	PR_ENTRYID	PT_BINARY
AddressEntry	Name	PR_DISPLAY_NAME	PT_TSTRING
AddressEntry	Type	PR_ADDRTYPE	PT_TSTRING
Attachment	Index	PR_ATTACH_NUM	PT_LONG
Attachment	Name	PR_ATTACH_FILENAME	PT_TSTRING
Attachment	Position	PR_RENDERING_POSITION	PT_LONG
Attachment	Source	PR_ATTACH_PATHNAME	PT_TSTRING
Attachment	Type	PR_ATTACH_METHOD	PT_LONG
Folder	FolderID	PR_PARENT_ENTRYID	PT_BINARY
Folder	ID	PR_ENTRYID	PT_BINARY
Folder	Name	PR_DISPLAY_NAME	PT_TSTRING
Folder	StoreID	PR_STORE_ENTRYID	PT_BINARY

Message	Conversation	PR_CONVERSATIO N_ KEY	PT_BINARY
Message	ConversationI ndex	PR_CONVERSATIO N_ INDEX	PT_BINARY
Message	Conversation Topic	PR_CONVERSATIO N_ TOPIC	PT_STRIN G
Message	DeliveryRecei pt	PR_ORIGINATOR_ DELIVERY_REPOR T_ REQUESTED	PT_BOOLE AN
Message	Encrypted	PR_SECURITY	PT_LONG
Message	FolderID	PR_PARENT_ENTR YID	PT_BINARY
Message	ID	PR_ENTRYID	PT_BINARY
Message	Importance	PR_IMPORTANCE	PT_LONG
Message	ReadReceipt	PR_READ_RECEIP T_ REQUESTED	PT_BOOLE AN
Message	Sender	PR_SENDER_ ENTRYID	PT_BINARY
Message	Sent	PR_MESSAGE_FLA GS	PT_LONG
Message	Signed	PR_SECURITY	PT_LONG
Message	Size	PR_MESSAGE_SIZ E	PT_LONG
Message	StoreID	PR_STORE_ENTRY ID	PT_BINARY
Message	Subject	PR_SUBJECT	PT_TSTRIN G
Message	Submitted	PR_MESSAGE_FLA GS	PT_LONG
Message	Text	PR_BODY	PT_TSTRIN G
Message	TimeReceive d	PR_MESSAGE_ DELIVERY_TIME	PT_SYSTI ME
Message	TimeSent	PR_CLIENT_SUBMI T_ TIME	PT_SYSTI ME
Message	Type	PR_MESSAGE_CL ASS	PT_TSTRIN G
Message	Unread	PR_MESSAGE_FLA GS	PT_LONG
Recipient	Name	PR_DISPLAY_NAM E	PT_TSTRIN G
Recipient	Type	PR_RECIPIENT_TY PE	PT_LONG

Session	Name	PR_DISPLAY_NAME	PT_TSTRING

For more information about MAPI properties, see the MAPI documentation.

Additional References

The following published references provide additional information about Visual Basic, Visual Basic for Applications, and OLE.

- *Microsoft Visual Basic Programmer's Guide*, Chapter 23, "Programming Other Applications' Objects"
- *Excel Visual Basic for Applications Step by Step*, Microsoft Press
- *Microsoft Excel Visual Basic User's Guide*, Chapter 5, "Working with Objects in Visual Basic" and Chapter 10, "Controlling and Communicating with Other Applications"
- *Microsoft OLE Programmer's Reference*, Microsoft Press
- *Inside OLE*, Microsoft Press

Note that this document contains the latest known information about the Microsoft OLE Messaging Library at the time of publication. Where terms in this document differ from other Visual Basic, OLE, or Component Object Model (COM) terms, this document should be viewed as the definition of the specific implementation represented by the OLE Messaging Library.

Legal Information

Microsoft Exchange Client Extensions Programmer's Reference

About Client Extensions

Using the *Client Extensions Programmer's Reference*, it is possible to develop a wide variety of custom client applications that access Microsoft Exchange services such as the information store. These custom client applications can provide different views of data in the information store, data searching and sorting capabilities, security, custom mail or messaging features, forms, workgroup capabilities, and many other capabilities.

Advantages of Extending Microsoft Exchange

Before developing a completely custom client application, you might want to consider extending the Microsoft Exchange client application. Microsoft Exchange provides a large number of features that enable users to view, search and sort information in the information store. It also provides a development platform for electronic mail, messaging, and workgroup applications that can handle a broad spectrum of information sharing tasks.

Instead of recreating features that are already provided in Microsoft Exchange, you can leverage the existing features and user interface of Microsoft Exchange and add your own custom features (also known as "extensions"). You can even override the default behavior of almost any feature of Microsoft Exchange. This can save a considerable amount of development effort that would otherwise be devoted to creating a custom user interface and custom features.

Additionally, extending the Microsoft Exchange client benefits users by integrating capabilities from many different ISVs into a single user interface, rather than having multiple interfaces through which users accomplish their information sharing tasks.

Microsoft Exchange client extensions can run on any platform that is running the Microsoft Exchange client. This includes Windows 3.1, Windows for Workgroups 3.1, Windows NT, and Windows 95. Since the Microsoft Exchange client is included with Windows 95, any extension you develop can run on any computer running Windows 95 − it is not necessary for the user to purchase a separate product to use your extension on Windows 95.

Extension Possibilities

There are many possible extensions that you can program to work with Microsoft Exchange. These extensions generally fall into four categories, as shown in the following table.

Extension type	Description
Command extensions	Add new commands to the client's menus and toolbar, or replace the behavior of existing commands and toolbar buttons. For example, you can add commands that are specific to certain folders or selected message types.
Event extensions	Invoke custom behavior to handle events such as the arrival of new messages, reading and writing messages, sending messages, reading and writing attached files and selection changes in a window.
Property sheet extensions	Add custom property sheet pages to an object's properties dialog box, enabling users or administrators to view or edit custom message properties.
Advanced criteria extensions	Implement your own custom advanced criteria dialog boxes to be used when searching for messages in the information store. This is useful when you want to enable users to search for custom message properties.

Each of these extension types is described in greater detail in the following sections.

Command Extension Examples

Commands that invoke virtually any type of custom behavior can be added to the menus and toolbars of Microsoft Exchange. For example:

- **Integrated proofing tools.** Custom spelling and grammar tools can be added to the Microsoft Exchange menu. If a user has composed a message, choosing a custom Spelling or Grammar command could invoke an ISV-supplied spell-checker or grammar-checker. It is also possible to add a thesaurus, text formatting tools, and other functionality provided by many word processors.
- **Public folder commands.** When a specific public folder has the focus, menu items or toolbar buttons can be enabled that are specific to that folder. For example, if the public folder is associated with an on-line service, you might want to include a Retrieve Latest command that prompts the on-line service to download the most current information into the selected topic area of the public folder.
- **Message class commands.** When messages of a specific class are selected, new application behavior can be added to handle messages of that class in a specific way. For example, you might want to replace the default behavior of the Microsoft Exchange Delete command when a message of a specific class is selected. The new behavior might simply move the message into a certain folder where it is processed at a later time.

Event Extension Examples

Event extensions enable developers to add or override behavior associated with certain events that occur within Microsoft Exchange. For example:

- **Attachment virus detection.** You can write an extension that transparently scans message attachments for viruses. When a message is received in a user's Inbox, the "message received" event would trigger the virus detection program and the user could be prompted to take action if a virus was found. This is especially useful in environments where users routinely attach executable programs to messages.
- **Message and attachment compression.** You can write an extension that compresses and decompresses messages when certain messaging events are detected. This can be especially useful in an environment where large messages, such as attached documents containing bitmaps, are routinely sent.

Property Sheet Extension Examples

Property sheet extensions are useful for displaying custom properties for messages of a particular message class or for adding your own Microsoft Exchange options. For example:

- **Custom form property sheets.** If your application provides a custom form with custom properties, you can display those properties in one or more property sheet pages. For example, the property sheet of a custom routing form might display the name of the person who initiated the routing process or the routing path of the form.
- **Custom document property sheets.** Your application can supply a custom property sheet that is displayed for certain types of documents. For example, if your extension provides a document archiving system, you might want to provide a property sheet that enables users to select archiving options for documents.

Advanced Criteria Extension Example

Advanced criteria extensions enable you to create your own advanced criteria dialog box that can be used to provide custom search capabilities. For example:

- **Custom property searching.** You can display a custom advanced criteria dialog box that enables users to specify search criteria for custom properties. For example, in a public folder that receives custom messages from an on-line service, there might be custom properties such as "Company" or "Industry" that are defined for each of the messages. With a custom search dialog box, you can supply criteria fields for these custom properties that enable users to perform custom searches. For instance, a user could search for all messages with an Industry property equal to "Biomedical."

Modal and Modeless Extension Windows

You can create extensions that use modal or modeless windows. When a modal window is displayed, users can only interact with the extension and not with Microsoft Exchange. For example, an extension might display a modal Folder Backup dialog box that enables a user to choose a specific folder to back up.

In contrast, some extensions can run modelessly, displaying windows that allow the user to switch their input back and forth between Microsoft Exchange windows and the extension's windows. For example, a Stock Quote extension could display a modeless window that displays dynamically changing stock data while the user interacts with Microsoft Exchange windows.

For simplicity, an extension that displays only modal windows is referred to as a *modal extension* while an extension that displays at least one modeless window is referred to as a *modeless extension*.

Some extensions can involve a combination of modeless and modal windows; that is, modeless extensions might display modal windows at certain times. For example, the Stock Quote extension mentioned previously might display a modal Options dialog box.

When writing modeless extensions, it is important to coordinate the actions of your extension's windows with the windows of Microsoft Exchange. For example, if Microsoft Exchange displays a modal window, modeless extensions should disable their windows. Interfaces that enable this cooperation are implemented by Microsoft Exchange and extensions.

Extension Interfaces and Contexts

To implement any of the extension types supported by Microsoft Exchange, you create a single dynamic link library (DLL) that contains one or more *extension objects*. An extension object is an object that complies with the Microsoft Windows Component Object Model and implements a set of Microsoft Exchange extensibility interfaces. Each extension type has more than one interface associated with it, but some of these interfaces are optional, as the following table shows.

Extension type	Interfaces to implement
Command extensions	**IExchExt** (required). **IExchExtCommands** (required). **IExchExtEvents** (optional). Implement if your command extension's enablement depends on the current selection or a change in the current object such as a folder, store, or message.
Event extensions	**IExchExt** (required). **IExchExtEvents** (optional). Implement if your extension should respond when the object being displayed in a window changes, or when the selection within the window changes. **IExchExtSessionEvents** (optional). Implement if you want to customize the behavior when new messages are delivered. **IExchExtUserEvents** (optional). Implement if you want to handle changes to the currently-selected list box item, text, or object. **IExchExtMessageEvents** (optional). Implement if you want to customize the way Microsoft Exchange manipulates messages. **IExchExtAttachedFileEvents** (optional). Implement if you want to customize the handling of message file attachments.
Property sheet extensions	**IExchExt** (required). **IExchExtPropertySheets** (required).
Advanced criteria extensions	**IExchExt** (required). **IExchExtAdvancedCriteria** (required).

If you implement an interface, you must implement all of its methods, even if your extension does not use them. For those methods that your extension doesn't use, a default response can usually be implemented with little effort. All extension objects need to implement the **IExchExt : IUnknown** interface.

The interaction between Microsoft Exchange and an extension object is bidirectional and involves more than simply calling an extension object's methods. To operate correctly, extension objects must gather information about the version of Microsoft Exchange, the MAPI session, and menu, toolbar and window handles. In most cases, they must also retrieve information from Microsoft Exchange about which objects, such as messages and folders, are currently selected within Microsoft Exchange windows. Retrieving this information is achieved with the **IExchExtCallback : IUnknown** interface, which is passed to many extension object methods.

Before implementing interfaces within an extension object, it is important to know which *contexts* the extension object will be associated with. A context is usually associated with a particular window within the Microsoft Exchange client. Most extension objects are designed to operate only within a particular context or set of contexts, but some can operate in all contexts. The following contexts are defined within the Microsoft Exchange client:

Context	Window
EECONTEXT_TASK	The duration of the Microsoft Exchange task, from program start to program exit. This may span several logons. This context does not correspond to a Microsoft Exchange window.
EECONTEXT_SESSION	The duration of a Microsoft Exchange session from logon to logoff. Multiple logons can occur during a single execution of Microsoft Exchange, but they might not overlap. This context does not correspond to a Microsoft Exchange window.
EECONTEXT_VIEWER	The main Viewer window that displays the folder hierarchy in the left pane and folder contents in the right pane.
EECONTEXT_REMOTEVIEWER	The Remote Mail window that is displayed when the user chooses the Remote Mail command.
EECONTEXT_SEARCHVIEWER	The Find window that is displayed when the user chooses the Find command.
EECONTEXT_ADDRBOOK	The Address Book window that is displayed when the user chooses the Address Book command.
EECONTEXT_SENDNOTEMESSAGE	The standard Compose Note window in which messages of class IPM.Note are composed.
EECONTEXT_READNOTEMESSAGE	The standard read note window in which messages of class IPM.Note are read after they are received.
EECONTEXT_READREPORTMESSAGE	The read report message window in which report messages (Read, Delivery, Non-Read, Non-Delivery) are read after they are

	received.
EECONTEXT_SENDRESENDMESSAGE	The send resend message window that is displayed when the user chooses the Send Again command on the non-delivery report.
EECONTEXT_SENDPOSTMESSAGE	The standard posting window in which existing posting messages are composed.
EECONTEXT_READPOSTMESSAGE	The standard posting window in which existing posting messages are read.
EECONTEXT_PROPERTYSHEETS	A property sheet window.
EECONTEXT_ADVANCEDCRITERIA	The dialog box in which the user specifies advanced search criteria.

Extensions register their "interest" in one or more of these contexts by placing entries in the EXCHNG.INI file on 16-bit versions of Microsoft Windows or in the registry on Windows NT. When an extension registers for a context, it means that events associated with this context will cause the extension to be notified of the event. For example, if an extension registers for the EECONTEXT_VIEWER context, and the user selects an object in the Viewer window, the extension will be called by Microsoft Exchange to notify it of the event.

If an extension does not register interest in specific contexts, it will be loaded and prompted to install itself in all contexts. Since loading extensions may reduce the performance of the Microsoft Exchange client, it is strongly recommended that all extensions specifically include context information as part of their registration.

Note If the standard compose note or post note form in the Microsoft Exchange Viewer window has been replaced by a custom form, extensions that are normally called from the standard compose note and post note windows will not be called unless the replacement IPM.Note form supports extensibility. This means that the form must do all the same work that the Microsoft Exchange client does when activating a form unless the implementor of this custom form explicitly implements this functionality.

How Extensions Work

When Microsoft Exchange is started, it reads its .INI file (on 16-bit versions of Windows) or the registry database (on Windows NT) and activates all the extensions that have registered to participate within the EECONTEXT_TASK context. Extensions that participate in other contexts are activated on-demand when those contexts become current.

After Microsoft Exchange and its initial extensions have been loaded into memory, the interaction between Microsoft Exchange and its extensions is determined mostly by the user's interaction with Microsoft Exchange, but it can also be determined by events such as the arrival of a new message. The following steps give an overview of how Microsoft Exchange interacts with its installed extensions:

1. A context activation occurs.
2. In the order in which they are listed in the .INI file or registry, Microsoft Exchange invokes the **IExchExt::Install** method on each extension object that has registered to participate in the new context. One of the parameters passed to each extension through **Install** is a pointer to a callback object that supports the **IExchExtCallback : IUnknown** interface.
3. Each extension that was called in step 2 uses **IExchExtCallback** to retrieve information about the environment, including the active menu, the active toolbar, the number of objects selected in the current window, and the entry identifier of the selected item. Extensions can also use MAPI and Windows API functions to retrieve information. Extensions use this information to determine if they will participate in the new context.
4. If an extension determines that it will participate in the context, it will return S_OK from **IExchExt::Install**. For example, an extension might need to participate in a context only if a certain folder is open. Otherwise, it returns S_FALSE.
5. In the order in which they are listed in the .INI file or registry, Microsoft Exchange invokes appropriate methods on all extensions that returned S_OK. The methods that are invoked depend on the context. For example, if the context is EECONTEXT_SENDNOTEMESSAGE, Microsoft Exchange first invokes the **IExchExtCommands::InstallCommands** method on all extensions that are registered to participate in this context and which implement the **IExchExtCommands : IUnknown** interface. Extensions can then add menu items to an existing menu and enable or disable them using Windows API calls.

Context Lifetimes

Some contexts last only a short time while others can exist for the entire time that Microsoft Exchange is running. For example, the EECONTEXT_TASK context exists all the time after Microsoft Exchange has logged on because this context refers to the running Microsoft Exchange client application. In contrast, the EECONTEXT_SEARCHVIEWER context exists only while the Find dialog box is displayed.

An extension is loaded into memory the first time a context for which it is registered is created. Once an extension is loaded into memory, it remains in memory for the lifetime of Microsoft Exchange. For example, consider three extensions E1, E2, and E3. If E1 and E2 are both registered for EECONTEXT_SESSION and E3 is registered for EECONTEXT_ADDRBOOK, then E1 and E2 will be loaded when Microsoft Exchange logs on, but E3 will not be loaded until the Address Book is opened. After the Address Book is opened, all three extensions will remain loaded until Microsoft Exchange is closed.

More than one context can exist at the same time. For example, EECONTEXT_VIEWER and EECONTEXT_ADDRBOOK exist simultaneously when the Viewer and the Address Book are active. Additionally, more than one instance of the same context can be active at one time, such as when two Find windows are open or two Compose Note windows are open. In these cases, two instances of the same context are active. When a context is destroyed, all extensions that are instantiated in that context are released but the extension DLLs remain in memory.

Context Interface Mappings

The following table shows which of the event interfaces can be called in each of the Microsoft Exchange contexts. In addition to these, the required interface **IExchExt : IUnknown** is called in all contexts.

Context	Interfaces
EECONTEXT_TASK	None. Only **IExchExt::Install** and **IUnknown::Release** are called from this context.
EECONTEXT_SESSION	**IExchExtSessionEvents**
EECONTEXT_VIEWER, EECONTEXT_REMOTEVIEWER, EECONTEXT_SEARCHVIEWER	**IExchExtCommands,** **IExchExtUserEvents**, **IExchExtPropertySheets**
EECONTEXT_ADDRBOOK	**IExchExtCommands,** **IExchExtUserEvents**, **IExchExtPropertySheets**
EECONTEXT_SENDNOTEMESSAGE, EECONTEXT_READNOTEMESSAGE, EECONTEXT_READREPORTMESSAGE, EECONTEXT_SENDRESENDMESSAGE, EECONTEXT_READPOSTMESSAGE, EECONTEXT_SENDPOSTMESSAGE	**IExchExtCommands,** **IExchExtUserEvents**, **IExchExtMessageEvents**, **IExchExtAttachedFileEvents, IExchExtPropertySheets**
EECONTEXT_PROPERTYSHEETS	**IExchExtPropertySheets**
EECONTEXT_ADVANCEDCRITERIA	**IExchExtAdvancedCriteria**

Using Client Extensions

As described in the previous documentation, Microsoft Exchange client extensions can be grouped into four categories:

- Command extensions
- Event extensions
- Property sheet extensions
- Advanced criteria extensions

You can also develop multi-purpose extensions that span two or more of these categories. For example, you can develop an extension that adds menus to the Microsoft Exchange Viewer window and also provides custom property sheets.

This section provides detailed information on how the Microsoft Exchange client interacts with the major categories of extensions shown in the preceding list. Information is also provided on how to write modeless extensions, how to register extensions within the EXCHEXT.INI file or the registry, and how to optimize performance.

Note For sample code illustrating some of the concepts covered in this section, see the Win32 SDK.

Command Extensions

Command extensions add new commands to the menus or toolbar of the Microsoft Exchange client or replace the behavior of existing menus and toolbar buttons. Consider an extension that adds a new command to the menu bar of the standard send form. To install this new command on the menu bar and handle a user's interaction with it, the extension object implements the **IExchExt : IUnknown** and **IExchExtCommands : IUnknown** interfaces.

The sequence of events that should occur when Microsoft Exchange interacts with your extension object to add new commands to the menu bar or new toolbar buttons in the EECONTEXT_SENDNOTEMESSAGE context are as follows:

1. When the standard send form is activated, but not yet displayed, the Microsoft Exchange client calls the **IExchExt::Install** method on all extensions registered to participate in that context and passes each extension a pointer to an **IExchExtCallback** interface along with the active context, which in this case is EECONTEXT_SENDNOTEMESSAGE.
2. All extensions that are registered to participate in the EECONTEXT_SENDNOTEMESSAGE context and have determined that they will participate return S_OK to Microsoft Exchange.
3. After the New Message window is created, Microsoft Exchange invokes the **IExchExtCommands::InstallCommands** method on all extension objects that returned S_OK to **IExchExt::Install**. These extension objects then add their menu commands or toolbar buttons to the New Message window using Windows API calls for adding menu commands and toolbar buttons. After the commands and buttons are added, Microsoft Exchange displays the New Message window, and the new commands are available to the user.
4. As the user interacts with the New Message window, Microsoft Exchange will frequently receive WM_INITMENU messages. Each time this happens, Microsoft Exchange calls the **IExchExtCommands::InitMenu** method for each extension to give that extension an opportunity to enable, disable, or update its menu items before they are seen by the user. Microsoft Exchange then calls the **IExchExtCommands::QueryButtonInfo** method for both standard Microsoft Exchange toolbar buttons and any buttons installed by extensions.
5. When the user chooses a menu command or a toolbar button, the window receives a WM_COMMAND message with the command identifier of the menu item or toolbar button that was selected. Microsoft Exchange sequentially calls the **IExchExtCommands::DoCommand** method on all extensions that have registered for that context and have implemented the **IExchExtCommands : IUnknown** interface, passing the command identifier as an argument. Even native Microsoft Exchange commands are passed to the extensions, enabling them to replace or enhance these native commands. When an extension is called, it examines the command identifier and determines if it should handle that command. If an extension isn't programmed to handle the command identifier, it should return S_FALSE, and Microsoft Exchange will pass the command identifier to the next extension. If an extension is programmed to handle the command, it should return S_OK. In most cases, extensions will only handle commands that they added to the menu or toolbar with the **IExchExtCommands::InstallCommands** method. If no extension handles the command, Microsoft Exchange will handle the command if it recognizes it. If Microsoft Exchange does not recognize a command, it is ignored.

The following table provides a summary of the interaction between a user, Microsoft Exchange and an extension object following a series of user actions performed by a user with a custom command. It also shows which component − Microsoft Exchange or the client extension − performs the step and in the case of the client extension, what method is invoked.

User	Microsoft Exchange	Extension
		InstallCommands returns S_OK
		QueryButtonInfo

Selects menu command		
	Receives WM_INITMENU message	
		InitMenu
Chooses command		
	Receives WM_COMMAND message	
		DoCommand returns S_OK

For more information on the **IExchExtCommands : IUnknown** and **IExchExt :IUnknown** interfaces, see Interfaces for Extending the Microsoft Exchange Client.

You can install commands on the Microsoft Exchange system menu using the same general process as described in the section called Command Extensions, but instead of using the Windows **GetMenu** function, you'll use **GetSystemMenu**. Once a custom command is installed, Microsoft Exchange passes the command identifier of the system menu command to the extension when the user chooses that command. Handling system commands enables an extension to override default Microsoft Exchange behavior. For example, you might want your extension to override the system menu's Close command and perform a few cleanup operations before terminating the application.

It is also possible to specify menu accelerators that can be used with menu items. These menu accelerators can be invoked by users in the usual way by pressing Alt plus the access key of the menu or command.

Note In accordance with standard user interface conventions, the Services and Options commands on the Microsoft Exchange Tools menu should remain at the bottom of the menu. If your extension adds commands to the Tools menu, it should add them above the Microsoft Exchange Services and Options commands.

Event Extensions

An event extension is an extension that enables you to customize the handling of various events such as new message delivery, reading messages, sending messages, reading and writing attached files, and tracking object selection changes. Extensions that handle events can be installed in the following contexts:

Window	**Context**
Main Viewer window	EECONTEXT_VIEWER
Remote Mail window	EECONTEXT_REMOTEVIEWER
Find window	EECONTEXT_SEARCHVIEWER
Address Book window	EECONTEXT_ADDRBOOK
Session context	EECONTEXT_SESSION

Additionally, user events, such as an object selection in a container, occur in the various message windows.

Four categories of events are defined. These events, which are described in the following sections, are passed to the Microsoft Exchange client when they occur.

User Events

When a Microsoft Exchange user changes the selection within a window or changes which object's contents are being displayed in a window, a user event is sent to Microsoft Exchange extensions. Within the main Viewer window, the selection can be a message store, a folder, or a message. Within the Find window or the Remote Mail window, the selection is always a message. Within the Address Book, it is an address entry.

Extensions that need to be notified whenever a selection or object changes must implement the **IExchExtUserEvents : IUnknown** interface. This interface includes two methods: **IExchExtUserEvents::OnSelectionChange** and **IExchExtUserEvents::OnObjectChange**. Whenever a selection or object changes, Microsoft Exchange calls these methods on all extensions that were installed for that context.

The two **IExchExtUserEvents** methods only pass the extension a pointer to the **IExchExtCallBack : IUnknown** interface, which provides information about the current selection. Using this information, the extension can then act appropriately. For example, it can enable or disable a toolbar button.

Session Events

When a new message arrives in a user's mailbox, a session event is sent to Microsoft Exchange. Extensions can be notified when a new message arrives by installing themselves in the EECONTEXT_SESSION context and by implementing the **IExchExtSessionEvents : IUnknown** interface, which supports only one method: **IExchExtSessionEvents::OnDelivery**. Whenever a session event occurs, Microsoft Exchange calls this method on all extensions that are registered to handle these events.

OnDelivery passes a pointer to the **IExchExtCallBack : IUnknown** interface to the extension, which uses the pointer to obtain details about the current context. The extension returns S_OK from **OnDelivery** if it will handle the event. For example, an extension might use the **IExchExtCallBack** pointer to determine the class of the newly arrived message. If the message class is of a specific type, the extension might want to perform a custom operation on that message, such as extracting its properties and storing them in a database.

Note **IExchExtSessionEvents::OnDelivery** is called after the server has processed the rules attached to the folder. Therefore, **OnDelivery** cannot be called if it was previously handled. Because this is a server process, rules processing is done regardless of any ordering specified in the EXCHNG.INI file or the registry.

Message Events

The Microsoft Exchange client enables extensions to control various details of how a message is handled when certain events occur. This is done with the **IExchExtMessageEvents : IUnknown** interface, which allows extensions to intercept message events:

- Immediately before and immediately after a message's properties are read. For example, when an extension displays a message in a form.
- Immediately before and immediately after a message's properties are written to the message store.
- Immediately before and immediately after Microsoft Exchange resolves unresolved addresses. For example, when a user chooses the Check Names button.
- Immediately before and immediately after Microsoft Exchange submits an open message for delivery.

For example, an extension might want to check spelling or validate data before it is written to the properties of the message in the store and sent to another user.

Each one of the events in the preceeding list is associated with a method that is a member of **IExchExtMessageEvents**. For example, the **IExchExtMessageEvents::OnWrite** method is invoked whenever Microsoft Exchange is about to save the message properties.

Attachment Events

Attachment events occur when an attached file is being read, written, or opened. These events enable extensions to control access to attachments and provide custom handling behavior. Attachment extensions are useful for providing virus detection and compression of attached files. These extensions are called from the context of the message containing the attachment.

Note Not all attachments are files. Attachment events apply only to attached files. Because attachments can themselves be other messages (that is, messages can contain messages), another context corresponding to the attached message might be created when an attached message is opened.

Sending a Note with an Attachment

The following table shows the steps involved in sending a standard message with an attachment. It also shows which component – Microsoft Exchange client or the extension – performs the step and in the case of the client extension, what method is invoked.

User	Microsoft Exchange	Extension
Chooses New Message from the Compose menu		
	Receives WM_COMMAND message, calls command extensions, and creates SENDNOTEMESSAGE context	
		Install(SENDNOTEMESSAGE) returns S_OK
	Installs command extensions	
Types recipient, subject, and text. Drags file attachment to message text.		
		OnReadPattFromSzFile
	Displays attached file in window	
Chooses Send from the File menu		
	Receives WM_COMMAND message, calls command extensions, and handles command itself	
		OnCheckNames

		OnCheckNamesComplete
		OnSubmit
		OnWrite
		OnWriteComplete
		OnSubmitComplete
	Closes send window	
		Release on extension objects
	Destroys SENDNOTEMESSAGE context	

Reading a Message and Opening its Attachment

The following table shows the steps involved in reading a message and opening its attachment. It also shows which component – the Microsoft Exchange client or the extension – performs the step, and in the case of the extension, shows what method is invoked.

User	**Microsoft Exchange**	**Extension**
Chooses Open from the File menu after selecting a message		
	Receives WM_COMMAND message, calls command extensions, and creates READNOTEMESSAGE context	
		OnRead
		OnReadComplete
	Installs command extensions	
	Displays read note	
Double clicks on file attachment		
		QueryDisallowOpenPatt returns S_OK
		OnWritePattToSzFile
		OpenSzFile
	Opens file	
Changes, saves and closes file		
Chooses Close from the File menu		
	Receives WM_COMMAND message, calls command	

	extensions	
Chooses Yes in the Save dialog box		
		OnWrite
		OnReadPattFromSzFile
		OnWriteComplete
	Closes read window	
		Release on extension objects
	Destroys READNOTEMESSAGE context	

Property Sheet Extensions

Property sheet extensions enable you to add custom property sheet pages for information stores, folders, and messages. These pages can be added to property sheets that are displayed in a variety of contexts. For example, you can add a page to the folder property sheet that is displayed when a user chooses the Properties command from the File menu in the main Viewer window when a folder is selected.

Property sheet extensions are useful for displaying custom properties for messages of a particular message class or for adding your own Microsoft Exchange options. For example, if you wrote an application that created and managed a specific public folder, you could add a property sheet page that would enable users to set various application-specific properties for the folder.

The sequence of events that should occur when Microsoft Exchange interacts with your extension object to install and use custom property sheet pages is as follows:

1. When the user opens a property sheet, Microsoft Exchange sequentially calls the **IExchExt::Install** method on all extensions registered to participate in the EECONTEXT_PROPERTYSHEETS context and passes each extension a pointer to an **IExchExtCallback** interface. These extensions, along with extensions that have registered for the context in which the property sheet is displayed will be called to add property sheet pages. For example, if a property sheet is displayed in the Viewer window, extensions registered for the EECONTEXT_PROPERTYSHEETS or EECONTEXT_VIEWER context will be called.
2. Microsoft Exchange calls the **IExchExtPropertySheets::GetMaxPageCount** method on each extension that returned S_OK from **Install**. This enables Microsoft Exchange to allocate sufficient memory for the property sheet page array.
3. When Microsoft Exchange is ready to build the property sheet, it calls the **IExchExtPropertySheets::GetPages** method so that the extension can specify a pointer to the pages it will append. **GetPages** is called immediately before the property sheet is displayed and enables the extension to fill in the pages it wants appended to the Microsoft Exchange Properties dialog box. The standard Microsoft Exchange pages are added first, followed by pages from each extension in the order the extensions are installed. **GetPages** uses the **IExchExtCallback::GetObject** method to retrieve the object for which the information should be displayed and the store which contains that object. The extension must use the standard property sheet structures specified by the Windows API. One of the parameters passed to the extension in **GetPages** is the type of property sheet being displayed − for example, message, folder, or store property sheet.
4. When the user closes the property sheet, Microsoft Exchange calls the **IExchExtPropertySheets::FreePages** method which instructs the extension to free any resources associated with the property sheet pages that were specified in **GetPages**.

The following table summarizes the interaction between a user, Microsoft Exchange and an extension object when a custom property sheet is being added to the Microsoft Exchange client. It also shows which component − the Microsoft Exchange client or the extension − performs the step and in the case of the extension, what method is invoked. To simplify this table, the installation of command extensions is not included. Command extensions are called in the context of a user choosing the Properties command from the File menu.

User	**Microsoft Exchange**	**Extension**
Chooses Properties from the File menu		
	Receives WM_COMMAND message, calls command	

	extensions, and creates a PROPERTYSHEETS context	
		For extensions registered for the PROPERTYSHEETS context, **Install**(PROPERTYSHEETS) is called
		GetMaxPageCount returns S_OK
	Allocates property sheet page array	
		GetPages returns S_OK
	Displays dialog box	
Chooses OK or Cancel button		
	Closes dialog box	
		FreePages
		Release on extensions which were loaded in the PROPERTYSHEETS context

Advanced Criteria Extensions

Advanced criteria extensions enable you to create your own advanced criteria window that can be used when searching for messages. This is especially useful when you want to enable users to search for custom message properties.

Although several extensions that supply advanced criteria dialog boxes might be available, only one advanced criteria window at a time can be displayed from a single context. Multiple advanced criteria contexts can be active at the same time; for example, when more than one Find window is open. It is important to remember that extensions are called in the order in which they were installed.

Because multiple advanced criteria extensions might be installed, it is important to ensure that your advanced criteria extension follows a few guidelines. The primary guideline is to be as selective as possible before choosing to install extensions into an advanced criteria context. For instance, a public folder application should only choose to install its advanced criteria interface when its public folder is open or highlighted. Advanced criteria extensions that are more selective should be listed ahead of more general ones in the EXCHNG.INI file or the registry. Because Microsoft Exchange itself uses very general advanced criteria, most extensions should register themselves before Microsoft Exchange. If two or more such general advanced criteria extensions are present, the first one installed will always be used. The ordering of extensions is determined by their entry order in the EXCHNG.INI file or in the registry. For more information on creating extension entries, see the "Registering Extensions" section later in this chapter.

The sequence of events that should occur when Microsoft Exchange interacts with your extension object to add a custom advanced criteria dialog box to the Microsoft Exchange client is as follows:

1. When the user chooses the command that displays the Microsoft Exchange Find dialog box, Microsoft Exchange calls the **IExchExt::Install** method on all extension objects with the EECONTEXT_ADVANCEDCRITERIA bit set in the context map. During the call, it passes each extension a pointer to an **IExchExtCallback** interface along with the context, which in this case is EECONTEXT_ADVANCEDCRITERIA. The **IExchExtCallBack** pointer enables the extension to retrieve information about the current context. In this case, the current message store and folder will probably be useful.
2. Microsoft Exchange calls the **IExchExtAdvancedCriteria::InstallAdvancedCriteria** method on the first extension to return S_OK from **Install**. Microsoft Exchange uses **InstallAdvancedCriteria** to pass information to the extension, including the window handle of the Find dialog box, the current advanced criteria restriction, and an array of folder entry identifiers to which the criteria will be applied. If the extension returns S_OK from **InstallAdvancedCriteria**, Microsoft Exchange will call the **IExchExtAdvancedCriteria::DoDialog** method when the user selects the Advanced button in the Find dialog box.

   If S_FALSE is returned from **InstallAdvancedCriteria**, remaining extensions will be given the opportunity to examine the current search restriction and display a custom search dialog box. If no extension returns S_OK, Microsoft Exchange will display a default advanced criteria dialog box.
3. If the user chooses the New Search button, changes folders, or closes the Find dialog box after Microsoft Exchange calls **InstallAdvancedCriteria**, Microsoft Exchange calls the **IExchExtAdvancedCriteria::Clear, IExchExtAdvancedCriteria::SetFolders,** or **IExchExtAdvancedCriteria::QueryRestriction** methods respectively to inform the extension of the changes.
4. When the user closes the advanced criteria window, Microsoft Exchange calls the **IExchExtAdvancedCriteria::UninstallAdvancedCriteria** method, which allows the extension to free any resources associated with the advanced criteria window that it displayed.

The following table summarizes the interaction between a user, Microsoft Exchange and an extension object when using a custom advanced criteria dialog box. It also shows which component − Microsoft Exchange or the client extension − performs the step and in the case of the client extension, what method is invoked.

User	Microsoft Exchange	Extension
Chooses Find from the Tools menu		
	Receives WM_COMMAND message, calls command extensions, and creates SEARCHVIEWER and ADVANCEDCRITERIA contexts	
		Install(ADVANCED CRITERIA) returns S_OK
	Installs command extensions	
		InstallAdvancedCriteria returns S_OK
	Displays Find window	
Chooses Folder in the Find dialog box		
	Shows Choose Folder dialog box	
Changes the folders to search in and chooses OK		
	Closes Choose Folder dialog box	
		SetFolders
Chooses New Search in the Find dialog box		
		Clear
Chooses Advanced in the Find dialog box		
		DoDialog
Specifies criteria and chooses OK		
		returning S_OK from **DoDialog**
Chooses Find Now in the Find dialog box		
		QueryRestriction
	Performs search	
Closes Find dialog box		
		UninstallAdvanced Criteria
	Closes Find window	
		Release on extension object
	Destroys	

SEARCHVIEWER and
ADVANCEDCRITERIA
contexts

Task and Session Extensions

In some cases, an extension needs to run when the user starts Microsoft Exchange or right after the user logs on. For example, an extension that integrates with a corporate database may want to establish a connection to the database as part of the logon process.

As shown in the following table, the client extension implements the TASK, SESSION, and VIEWER contexts with only the **IExchExt : IUnknown** interface using the **IExchExt:Install** method (and the **IUnknown::Release** method).

The following table summarizes the interaction between a user, Microsoft Exchange and an extension object when a user launches Microsoft Exchange or logs on to a MAPI session. It also shows which component − Microsoft Exchange or the client extension − performs the step and in the case of the client extension, what method is invoked.

User	Microsoft Exchange	Extension
Launches Microsoft Exchange		
	Creates TASK context	
		Install(TASK) returns S_OK
Logs on with profile and password		
	Creates SESSION context	
		Install(SESSION) returns S_OK
	Creates VIEWER context	
		Install(VIEWER) returns S_OK
Chooses Exit and Logoff from the File menu		
		Release on VIEWER object
	Destroys VIEWER context	
		Release on SESSION object
	Destroys SESSION context	
		Release on TASK object
	Destroys TASK context	
	Shuts down application	

Writing Modeless Extensions

As mentioned previously in this chapter, extension windows can be either modal or modeless. When writing extensions that use modeless windows, it is a good idea to design your code to cooperate with Microsoft Exchange. The most common scenario for this cooperation is enabling or disabling modeless windows when the user switches between Microsoft Exchange windows and the windows of your extension.

Coordination between Microsoft Exchange and modeless extensions is achieved using two interfaces: **IExchExtModeless : IUnknown** and **IExchExtModelessCallback : IUnknown**. Microsoft Exchange implements **IExchExtModelessCallback** and client extensions implement **IExchExtModeless**.

Initializing a Modeless Extension

A modeless extension must first indicate to Microsoft Exchange that it will be displaying modeless. This is done by calling the **IExchExtCallback::RegisterModeless** method. If an extension does not call this method, Microsoft Exchange assumes by default that the extension's window is displayed modally and the extension will not be able to coordinate its actions with those of Microsoft Exchange.

When calling **RegisterModeless**, the extension must pass to Microsoft Exchange a a modeless object that implements the **IExchExtModeless** interface. This interface enables Microsoft Exchange to communicate with the extension object about the state of its windows.

How Modeless Coordination Works

When Microsoft Exchange runs, it creates a modeless callback object with which extensions can communicate. This callback object implements the **IExchExtModelessCallback : IUnknown** interface.

Before displaying a modal window, Microsoft Exchange invokes the **IExchExtModeless::EnableModeless** method on all modeless objects. In this case, the *fEnable* parameter of **IExchExtModeless::EnableModeless** is set to FALSE, indicating that the extension should disable its modeless windows. When Microsoft Exchange removes its modal window, it calls **IExchExtModeless::EnableModeless** with the *fEnable* parameter set to TRUE. The extension can then re-enable its modeless windows.

Similary, if a modeless extension needs to display a modal window, it should call the **IExchExtModelessCallback::EnableModeless** method with the *fEnable* parameter set to FALSE. Microsoft Exchange then disables its modeless windows. When the extension removes its modal window, it should call **IExchExtModeless::EnableModeless** with the *fEnable* parameter set to TRUE, enabling Microsoft Exchange to re-enable its modeless windows.

Additional Programming Considerations

When creating extensions for the Microsoft Exchange client, there are a number of programming considerations you will want to keep in mind beyond the basic technique of implementing extension interfaces. These considerations generally fall into the following categories:

- Performance
- Error handling
- Cooperation with other extensions
- Programming practices

Optimizing Performance

Because the Microsoft Exchange client polls installed extensions every time a context change occurs, Microsoft Exchange can exhibit decreased performance or response time if several extensions are installed. In particular, performance can be affected when displaying menus, sending and receiving messages, or selecting different messages and folders. This performance degradation can be caused by extensions that respond to events that are not applicable to the functionality they provide. It can also be caused by extensions that take a long time to respond when Microsoft Exchange invokes their methods.

When developing an extension, you can reduce the impact on the performance of Microsoft Exchange by keeping the following programming guidelines in mind:

- Register your extension only for contexts that are applicable to it. This can be done by placing the appropriate values in the context map in the EXCHNG.INI file or the registry. The context map designates contexts in which an extension will participate. Although specifying a context map is optional, it is highly recommended. If you don't specify a context map, an attempt will be made to install your extension in all contexts and it will be loaded at all times. For more information about context maps, see the "Registering Extensions" section later in this chapter.
- Register your extension only for interfaces that are applicable to it. This can be done by placing the appropriate values in the interface map in the EXCHNG.INI file or the registry. Although specifying an interface map is optional, it is highly recommended. The interface map prevents Microsoft Exchange from making calls to **IUnknown::QueryInterface** for interfaces that are not supported. This can improve the performance of context creation and application startup.
- Optimize the efficiency of functions that handle various events received by your extension. For example, event-handling functions should minimize the use of any time-consuming calculations, loops, lookups, and file input and output.
- Keep the memory footprint of your extensions to a minimum because once loaded, it remains loaded until Microsoft Exchange is closed. To keep your memory footprint small, you might want to split your extension DLL into two separate DLLs. The first DLL should be a handler that handles most of the extension interaction, especially the default responses that ignore unwanted events. The second DLL should be activated when "real" work must be done, such as executing a custom command. This is especially important for large extensions that are infrequently used.

Handling Error Values

Extensions are responsible for handling and displaying their own errors. Microsoft Exchange handles error return values by not continuing with the current operation. Microsoft Exchange does not display error messages because the extension is usually better aware of what caused the error and what steps should be taken to correct it.

Cooperating with other Extensions

Because extensions that you distribute to customers might be installed alongside extensions developed by other programmers, it is important to design your extensions so they cooperate with other extensions. Cooperation among extensions is important because the order in which extensions are listed in the EXCHNG.INI file or the registry determines the order in which extensions are called to respond to context changes in Microsoft Exchange. Because extensions are called sequentially, extensions that are not designed to operate cooperatively can "block" the execution of other extensions.

For example, suppose a user double clicks on a folder to display its messages. When this happens, Microsoft Exchange will call each extension to determine if it wants to participate in the event. If a "misbehaved" extension handles the event without being selective about the context in which it has been installed, it will block the execution of other extensions that fall after it in the calling sequence. One of these other extensions might be programmed specifically to handle that event, but it will not be given a chance.

To avoid blocking other extensions, your extension should be highly selective in determining when it runs. When Microsoft Exchange passes your extension a pointer to the **IExchExtCallback : IUnknown** interface, your extension should use the methods of this interface to thoroughly examine the current context. An extension should run only if it determines that the current context is specific to it, for example, if the current selection is a custom message type understood only by your extension.

To avoid collisions, consider the following guidelines:

- Avoid programming extensions that provide broad or general behavior for features such as the advanced criteria dialog box that can only have one active extension. Most extensions should operate only in contexts that can be considered extension-specific.
- Extensions that are selective and highly context-specific should be registered at the beginning of the [Extensions] section of the .INI file. Those that exhibit more general behavior should be entered at the end of the [Extensions] section. For example, the standard Microsoft Exchange extension that handles the advanced criteria dialog box always returns S_OK to the **IExchExtAdvancedCriteria::InstallAdvancedCriteria** method. More selective advanced criteria dialog boxes must come before this entry or they will never be called.

Programming Practices

When writing an extension, you can reduce the number of bugs and other problems by observing the following programming practices:

- Do not store pointers to callback functions for later use outside of the current function. The same advice applies for pointers returned from callback functions. The reason for not storing these pointers is that they can become invalid shortly after they are used within the scope of a single function.
- Try not to perform special "tricks" with hMenu or hWnd pointers. Although creative use of these pointer types may not cause problems in the current version of Microsoft Exchange, they may create problems in future versions.
- Avoid adding menus to Microsoft Exchange outside the scope of the **IExchExtCommands::InstallCommands** method.
- When adding menu commands for your provider extension, place these menu commands underneath the Find command on the Tools menu.
- When adding your provider-specific Help command, place it underneath the Microsoft Exchange Help Topics command on the Help menu.

Registering Extensions

Before an extension can be used within the Microsoft Exchange client, it must be *registered*. Registering an extension lets Microsoft Exchange know about its existence and provides other information about the extension so that Microsoft Exchange can work with it in an efficient manner.

On 16-bit versions of Windows, extensions are registered by adding entries to the [Extensions] section of the EXCHNG.INI file. These 16-bit clients include Windows 3.1 and Windows for Workgroups 3.x.

On 32-bit versions of Windows, Microsoft Exchange obtains extension-specific information from HKEY_LOCAL_MACHINE\Software\Microsoft\Exchange\Client\Options in the Windows NT or Windows 95 registry.

Information on shared extensions is read from the [Extensions] section of the SHARED.INI file on 16-bit versions of Windows or from the SHARED32.INI file on 32-bit versions of Windows. The location of the SHARED.INI file is specified by the SharedExtsDir entry in the [Exchange] section of the EXCHNG.INI file. The location of the SHARED32.INI file is specified in the SharedExtsDir entry in HKEY_LOCAL_MACHINE\Software\Microsoft\Exchange\Client\Options.

If the directory is on a network drive, the SharedExtsServer entry can contain the name of the server and the SharedExtsPassword entry contains an unencrypted password. If this is the case, a connection to the server is made without redirecting a local device name.

The 32-bit clients cannot use 16-bit extensions nor can 16-bit clients use 32-bit extensions.

When registering an extension, you need to conform to a specific syntax when placing your entry in the INI file or registry. This syntax is as follows:

Syntax

Tag=Version;**<ExtsDir>**DllName;[Ordinal];[ContextMap];[InterfaceMap];[Provider]

Parameters

The following table shows the parameters used in the syntax line.

Parameter	Description
Tag	An extension identifier that uniquely identifies the .INI entry from other .INI entries.
Version	The version number of the syntax; for example '4.0'.
DllName	The path to the DLL containing the extension.
Ordinal	An optional field that specifies the entry point in the DLL to retrieve the extension object. If this field is empty, the default value is 1.
ContextMap	An optional string made up of '0' and '1' characters which indicate the contexts in which the extension should be loaded. Any unspecified values after the end of the string are assumed to be zero. If no context map is provided, the extension will be loaded in all contexts. For more information on context map bit positions, see the table later in this section.
InterfaceMap	An optional string made up of '0' and '1' characters that indicates the interfaces the extension supports. Although supported interfaces can be obtained through **IUnknown::QueryInterface**, registering which interfaces your extension supports can increase system performance. For more information on interface map bit positions, see the table later in this section.

Provider	An optional string containing the PR_SERVICE_NAME, not the display name, of an ISV-supplied provider that your extension is designed to work with. For example, if your extension is designed to work with a custom address book provider, place the PR_SERVICE_NAME of the address book provider in this parameter. If your extension is not provider-specific, this parameter should be omitted.

Example

```
MyExtension=4.0;<ExtsDir>MYEXT.DLL;2;010001;1100000
```

In this example, EECONTEXT_VIEWER=0x00000002 and EECONTEXT_SENDNOTEMESSAGE =0x00000006 so the ContextMap string '010001' indicates that the extension located at MYEXT.DLL;2 should be loaded only for these two contexts. The interface map, specified by the string 1100000, indicates that the extension is registered only for the **IExchExtCommands : IUnknown** and **IExchExtUserEvents : IUnknown** interfaces.

Throughout the string, all occurrences of the string '<ExtsDir>' are replaced with the value of the SharedExtsDir entry described below.

The following tables indicate which positions in the Context Map and Interface Map correspond to which contexts and interfaces, respectively.

Context Map Bit Positions

Position	Context
1	EECONTEXT_SESSION
2	EECONTEXT_VIEWER
3	EECONTEXT_REMOTEVIEWER
4	EECONTEXT_SEARCHVIEWER
5	EECONTEXT_ADDRBOOK
6	EECONTEXT_SENDNOTEMESSAGE
7	EECONTEXT_READNOTEMESSAGE
8	EECONTEXT_SENDPOSTMESSAGE
9	EECONTEXT_READPOSTMESSAGE
10	EECONTEXT_READREPORTMESSAGE
11	EECONTEXT_SENDRESENDMESSAGE
12	EECONTEXT_PROPERTYSHEETS
13	EECONTEXT_ADVANCEDCRITERIA
14	EECONTEXT_TASK

Interface Map Bit Positions

Position	Interface
1	**IExchExtCommands**
2	**IExchExtUserEvents**
3	**IExchExtSessionEvents**
4	**IExchExtMessageEvents**
5	**IExchExtAttachedFileEvents**
6	**IExchExtPropertySheets**
7	**IExchExtAdvancedCriteria**

Provider Parameter

In some cases, your extension might be designed to work only with a custom provider. For example, your extension might only work with a custom address book provider that replaces the default address book provider of Microsoft Exchange. Under these circumstances, your extension should be prevented from loading if its associated provider is not loaded.

This situation is handled by specifying the name of the extension's associated provider in the [Provider] parameter of your extension's registration line. If you specify the provider, your extension will not be loaded unless the provider has been loaded. This can save some programming effort because it might be difficult for your extension to determine whether a specific provider has been loaded.

When Microsoft Exchange starts, it reads the current profile and attempts to start all providers listed there. After attempting to load all providers specified in the profile, it reads the extension registration lines of its .INI file or registry and omits all extensions that specify a provider that was not in the profile or failed to load.