Legal Notice

About This Book

The *Client Extension Programmer's Guide*, which accompanies the Microsoft Win32® Software Development Kit (SDK), provides information about adding functionality to the Microsoft Exchange client program.

This book should be used in conjunction with the *Messaging Application Programming Interface* (*MAPI) Programmer's Reference, Volumes 1 and 2* and the *MAPI Programmer's Guide* to customize the user interface of the Microsoft Exchange client.

The Microsoft Win32 SDK contains the tools and documentation you need to develop 32-bit applications for the Microsoft Windows®, Microsoft Windows for Workgroups, and Microsoft Windows NT™ family of operating systems. The Win32 SDK also includes the Windows NT Device Driver Kit, which contains the tools, documentation, and sample source you need to write a driver for Windows NT.

Intended Audience

To develop applications for the Microsoft Exchange client environment, your background should include:

- Familiarity with basic features and capabilities of Microsoft Exchange clients.
- Proficiency with C or C++ programming in the Windows environment.
- Strong familiarity with MAPI. Reviewing the *MAPI Programmer's Guide* when you begin application development is highly recommended.
- Familiarity with network environments.

How This Book Is Organized

This book's contents are organized as follows:

- Chapter 1, "About Client Extensions," introduces the four major categories of client extensions and gives examples of their use. It also discusses the four corresponding extension objects and the interfaces they implement. Finally, the chapter explains the mechanism Microsoft Exchange uses to activate client extensions on both 16-bit and 32-bit platforms.
- Chapter 2, "Creating Client Extensions" provides information about creating the various extension types. It also discusses other programming considerations such as error handling, registering extensions, and optimizing performance.
- Chapter 3, "Interfaces for Extending the Microsoft Exchange Client," consists of an alphabetical listing of all interfaces that developers can implement for creating extensions to the Microsoft Exchange client.

How Interface Reference Entries Are Structured

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:	The application or component that must provide an implementation of this interface for other applications to call.
Called by:	The applications that typically call the methods of this interface.

Following the "At a Glance" table is a 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 Microsoft Exchange object. The methods are listed in the same order that they are declared. Methods derived 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 listed in alphabetical order. For each method, the documentation includes a brief purpose statement for the method followed by this information:

Heading	**Content**
Syntax	The syntax for the method.
Parameters	A description of each parameter of the method.
Return Values	A description of each value that the method can return.
Comments	A description of why and how the method is used.
See Also	Cross-references to other topics in the *Client Extension Programmer's Guide* that can help you use this method.

Document Conventions

The following typographical conventions are used throughout this book:

Typographical Convention	Meaning
Bold	Indicates an interface, method, structure, or other keyword in the Client Extension interfaces, MAPI, Microsoft Windows, the OLE application programming interface, C, or C++. For example, **OnSubmit** is a Client Extension 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, *lpeecb* is a Client Extension method parameter. In addition, italics are used to highlight the first use of terms and to emphasize meaning.
UPPERCASE	Indicates flags, return values, and properties. For example, EEAC_INCLUDESUBFOLDERS 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.
monospace	Indicates source code, structure syntax, examples, user input, and program output. For example: `ptbl->SortTable(pSort, TBL_BATCH);`
SMALL CAPITALS	Indicate the names of keys on the keyboard. When you see a plus sign (+) between two key names, you should hold down the first key while pressing the second. For example, CTRL + ALT + DEL. The carriage return key, sometimes marked as a bent arrow on the keyboard, is called ENTER.

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* by Charles Petzold.

For More Information

For information on MAPI, see the *MAPI Programmer's Guide* and *MAPI Programmer's Reference, Volumes 1 and 2*, both contained within the MAPI SDK*.* The *MAPI Programmer's Guide* provides a conceptual overview of the MAPI architecture and describes a wide range of programming tasks illustrated with extensible and reusable sample code. The *MAPI Programmer's Reference, Volume 1* provides a complete reference to the interfaces and methods used with Extended MAPI objects. *Volume 2* provides a complete reference to Extended MAPI functions, macros, structures, data types, properties, and return values and to the Simple MAPI and CMC programming models. It also includes a glossary of MAPI terms.

For more information about OLE programming, see *Inside OLE, Second Edition*, by Kraig Brockschmidt (Redmond, WA: Microsoft Press®, August 1995) and *OLE Programmer's Reference, Volume One* and *Volume Two* (Redmond, WA: Microsoft Press, 1995), and the Microsoft Win32 SDK, published on CD-ROM by the Microsoft Developer Network (MSDN) and by Microsoft Visual C++™.

For more information about the Microsoft Windows programming environment, see the Microsoft Windows SDK for Microsoft Windows*,* Version 3.1. Another good source is *Programming Windows* by Charles Petzold (Redmond, WA: Microsoft Press, 1990).

For more information about Windows 95 programming, see the Microsoft Windows 95 SDK.

For more information about Windows NT programming, see the Microsoft Windows NT SDK.

About Client Extensions

Using the *Client Extension Programmer's Guide*, 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. This chapter provides an overview of some of the programs it is possible to develop with the *Client Extension Programmer's Guide*.

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**

Creating Client Extensions

As described in the previous chapter, 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 chapter provides detailed information on how Microsoft Exchange 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 chapter, 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 before it is displayed), Microsoft Exchange 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 giving the extension the 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 Chapter 3, "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 "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 "housecleaning" 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 Microsoft Exchange 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 do nothing more than 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

Microsoft Exchange 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 or the client 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 body text. Drags file attachment to main body.		
		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 – 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 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 − Microsoft Exchange or the client extension − performs the step and in the case of the client 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

Optimizing Performance

Because Microsoft Exchange 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 Errors

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.

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.

Note that 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 order in which an extension is registered, that is, the order it is listed in the .INI file or registry, might affect the execution of the extension. Given two extensions that are registered within the same context, the first extension that is called to process an event or a command might handle it and not give other extensions a chance to respond.

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.

Interfaces for Extending the Microsoft Exchange Client

The Microsoft Exchange client is an extensible program containing a number of interfaces that independent software vendors can use to extend or enhance its functionality. Two of those interfaces are implemented by Microsoft Exchange and called by extension objects. All the other interfaces for client extensibility are implemented by extension objects and are called by Microsoft Exchange. Both sets of interfaces are described in this chapter.

An extension object is an object that complies with the Microsoft Windows Component Object Model and implements a set of Microsoft Exchange interfaces. Some of the interfaces described in this chapter are optional, but if an extension object implements an interface, it must implement all the functions of that interface.

The **IUnknown** interface provides object management functions from the OLE 2 component object model. Since all extension objects inherit from this interface, all extension interfaces include the three **IUnknown** methods in addition to their own methods.

IExchExt : IUnknown

Microsoft Exchange uses the **IExchExt** interface to load extension objects in all contexts. Most extension objects are designed to operate only within a particular context or set of contexts, but some can operate in all contexts.

At a Glance

Specified in header file:	EXCHEXT.H
Object that supplies this interface:	Extension object
Corresponding pointer type:	LPEXCHEXT
Implemented by:	Extension objects
Called by:	Microsoft Exchange

Vtable Order

Install	Enables an extension object to determine the context into which it is being loaded, along with information about that context.

Note Custom form developers are responsible for supporting extension objects in their custom forms. That is, replacement IPM.Note forms and other custom forms supplied by form developers might choose to load extension objects and call them the same way that Microsoft Exchange does with its native forms. In this documentation, all references to forms refer to the standard forms supplied by Microsoft Exchange, such as the standard IPM.Note.

IExchExt::Install

Enables an extension object to determine the context into which it is being loaded, along with information about that context. The MAPI and user interface information that can be retrieved by the extension includes session, store, viewer, note form, and address book information.

Syntax

HRESULT Install(**LPEXCHEXTCALLBACK** *lpeecb*, **ULONG** *eecontext,* ULONG *ulFlags*)

Parameters

lpeecb
Input parameter pointing to an **IExchExtCallback** interface.

eecontext
Input parameter indicating the currently active Microsoft Exchange context. This parameter can have the following values:

EECONTEXT_TASK
Indicates the context where the Microsoft Exchange client application has started but has not yet logged in. This context begins when the Microsoft Exchange application starts and ends when the Microsoft Exchange application exits. This context does not correspond to a window. This is the first context in which extensions are loaded.

EECONTEXT_SESSION
Indicates the duration of a Microsoft Exchange session. This context and EECONTEXT_TASK are the only contexts that do not correspond to a specific window.

EECONTEXT_VIEWER
Indicates the main Viewer window that displays the folder hierarchy in the left pane and folder contents in the right pane.

EECONTEXT_REMOTEVIEWER
Indicates the remote Viewer window that is displayed when the user chooses the Remote Mail command from the Tools menu in the Microsoft Exchange client.

EECONTEXT_SEARCHVIEWER
Indicates the search Viewer window that is displayed when the user chooses the Find button from the Tools menu in the Microsoft Exchange client.

EECONTEXT_ADDRBOOK
Indicates the Address Book dialog box that is displayed when the user chooses the Address Book button from the Tools menu in the Microsoft Exchange client.

EECONTEXT_SENDNOTEMESSAGE
Indicates the standard send note window in which messages of the IPM.Note class are composed.

EECONTEXT_READNOTEMESSAGE
Indicates the standard read note window in which messages of the IPM.Note class are read after they are received.

EECONTEXT_SENDPOSTMESSAGE
Indicates the standard posting window in which new IPM.Post posting messages are composed.

EECONTEXT_READPOSTMESSAGE
Indicates the standard posting window in which existing posting messages are read.

EECONTEXT_READREPORTMESSAGE
Indicates the read report message window in which the Read, Delivery, Non-Read, and Non-Delivery report messages are read after they are received.

EECONTEXT_SENDRESENDMESSAGE
Indicates the send-resend message window that is displayed when the user chooses the Send

Again button from the nondelivery report.

EECONTEXT_PROPERTYSHEETS
Indicates a property sheet window.

EECONTEXT_ADVANCEDCRITERIA
Indicates any window in which the user specifies advanced search criteria. In Microsoft Exchange, this is a window brought up by the Find button from the Tools menu, or the dialog box brought up by the Filter button.

ulFlags
Input parameter containing a bitmask of flags. The following flag can be set:

EE_MODAL
Specifies that the extension is being installed into a context where modal windows are displayed. Extensions that are loaded into modal contexts should only display modal windows.

Return Values

S_OK
The extension object elects to run in this context.

S_FALSE
The extension object does not elect to run in this context and will be released immediately.

Comments

Microsoft Exchange calls the **IExchExt::Install** method to enable an extension object to determine the context into which it is being loaded, along with information about that context. The context is determined using the callback object returned in the *lpeecb* parameter. The user interface context is described by the value of the *eecontext* parameter.

IExchExtCallback::GetVersion is often called during **Install** to ensure that the version of the Microsoft Exchange client is compatible with your extension object.

IExchExtCallback::GetObject is often called during **Install** to determine whether the currently selected object in the Microsoft Exchange user interface is the type your extension object supports. When **Install** is called with the property sheet's context, your extension object should make sure the selected object matches the type of property sheet that is provided to the extension.

HRESULT return values other than S_OK and S_FALSE indicate that the extension object has encountered an error and Microsoft Exchange will not load the extension object.

See Also

IExchExtCallback::GetVersion method, **IExchExtCallback::GetObject** method

IExchExtAdvancedCriteria : IUnknown

The **IExchExtAdvancedCriteria** interface is used to enable extension objects to replace or enhance the functionality of the Advanced dialog box that appears when the user selects the Advanced button from the Find dialog box.

Only one advanced criteria interface is used by a single context, although different extension objects can implement advanced criteria dialog boxes used by different contexts, potentially at the same time. The first extension object to be called with the **IExchExtAdvancedCriteria::InstallAdvancedCriteria** method is given precedence in determining the situations in which to install its advanced criteria dialog box. This ordering depends on the order in which extensions are listed in the registry database. Advanced criteria implementors can choose whether to take responsibility for advanced criteria based on the selected folder or on the existing restriction.

IExchExtAdvancedCriteria is used by Microsoft Exchange only in the EECONTEXT_VIEWER and EECONTEXT_SEARCHVIEWER contexts.

This interface is optional.

At a Glance

Specified in header file:	EXCHEXT.H
Object that supplies this interface:	Extension object
Corresponding pointer type:	LPEXCHEXTADVANCEDCRITERIA
Implemented by:	Extension objects
Called by:	Microsoft Exchange

Vtable Order

InstallAdvancedCriteria	Enables an extension to install its advanced criteria dialog box.
DoDialog	Displays an extension's advanced criteria dialog box.
Clear	Prompts an extension to clear all of its advanced criteria.
SetFolders	Called when the folders have been changed.
QueryRestriction	Returns a restriction object and a text representation of the restriction.
UninstallAdvancedCriteria	Called when the criteria user interface is closed and enables extensions to release resources.

IExchExtAdvancedCriteria::Clear

Clears the advanced criteria associated with the extension object. This method is called by Microsoft Exchange when the user chooses the Clear All button in the Advanced dialog box.

Syntax

VOID **Clear**()

IExchExtAdvancedCriteria::DoDialog

Displays an extension object's advanced criteria dialog box.

Syntax

HRESULT DoDialog()

Return Values

S_OK
  Advanced criteria, including the *fNot* parameter, were set after the user chose the dialog box's OK button, unless an error occurs.

EXCHEXT_S_NOCHANGE
  No change was made to the dialog box, or the user chose the Cancel button.

EXCHEXT_S_NOCRITERIA
  No criteria were specified in the dialog box. That is, the user changed the criteria to be empty)

Comments

Microsoft Exchange calls the **IExchExtAdvancedCriteria::DoDialog** method when the user selects the Advanced button in the Find dialog box. The extension object should display a dialog box that enables the user to specify search criteria. The dialog box should be modal to the window specified in the *hwnd* parameter of **IExchExtAdvancedCriteria::InstallAdvancedCriteria**.

IExchExtAdvancedCriteria::InstallAdvancedCriteria

Enables an extension object to install its advanced criteria dialog box.

Syntax

HRESULT InstallAdvancedCriteria(**HWND** *hwnd*, **LPSRestriction** *lpres*, **BOOL** *fNot*, **LPENTRYLIST** *lpeidl*, **ULONG** *ulFlags*)

Parameters

hwnd
Input parameter indicating the window handle containing the advanced criteria user interface.

lpres
Input parameter pointing to a MAPI **SRestriction** structure containing the advanced criteria restriction. NULL if none exists. For more information on **SRestriction** structures, see the *MAPI Programmer's Reference*.

fNot
Input parameter indicating TRUE if the user wants only messages that do not meet the criteria specified in the structure pointed to by the *lpres* parameter.

lpeidl
Input parameter pointing to an array of folder entry identifiers to which the criteria will be applied.

ulFlags
Input parameter containing a bitmask of flags. The following flag can be set:

EEAC_INCLUDESUBFOLDERS
Specifies that subfolders of the *lpeidl* parameter are to be included in the search.

Return Values

S_OK
The extension object will display its advanced criteria user interface.

S_FALSE
The extension object will not display its advanced criteria user interface.

Comments

Microsoft Exchange calls the **IExchExtAdvancedCriteria::InstallAdvancedCriteria** method when the user chooses the Advanced button in the Find dialog box. The first extension that returns S_OK is the only **IExchExtAdvancedCriteria** interface implementation used for the criteria dialog box instance. Implementations of this method should examine the *lpres* parameter to see if the restriction's format is recognizable. If not, S_FALSE should be returned and other extensions will be given the opportunity to handle the restriction. Extensions should use comment fields in restrictions to identify their particular formats.

Calls to **InstallAdvancedCriteria** will not be nested within a single context; that is, they will not be prompted to install their advanced criteria dialog box a second time before an existing installation has been uninstalled (with the exception of multiple instances of Microsoft Exchange Find windows and advanced criteria objects). Specifically, separate contexts which use **IExchExtAdvancedCriteria** can have overlapping lifespans.

To determine whether your extension should participate in the current advanced criteria context, you can obtain information about the context during the last call to the **IExchExt::Install** method. You might want to save information about the advanced criteria context for later use. You cannot save the pointer to the callback object during the call to **Install**.

If S_FALSE is returned, remaining extension objects will be given the opportunity to recognize the restriction and display the user interface. If no extension object returns S_OK, the Advanced button will be disabled. Microsoft Exchange will provide default advanced criteria functionality.

See Also

IExchExtCallback::GetObject method

IExchExtAdvancedCriteria::QueryRestriction

Returns a restriction object and a text version of the restriction. The text version is a user-readable representation of the query that the user composed in the Advanced dialog box. The format of the text version depends on the complexity of the criteria. For example, it could be a description in plain text English or in SQL language.

Syntax

HRESULT QueryRestriction(**LPVOID** *lpvAllocBase*, **LPSRestriction FAR** ******lppres*, LPSPropTagArray FAR * lppPropTags, LPMAPINAMEID FAR * FAR * lpppPropNames, **BOOL *** *lpfNot*, **LPTSTR** *lpszDesc*, **ULONG** *cchDesc*, **ULONG** *ulFlags*)

Parameters

lpvAllocBase
: Input parameter pointing to the memory allocation base to be used when allocating memory for the restriction data. For more information on allocating memory, see the *MAPI Programmer's Guide*.

lppres
: Output parameter pointing to where to return a pointer to the advanced criteria restriction.

lppPropTags
: Output parameter pointing to where to return a pointer to a MAPI property tag array containing the property tags of any named properties involved in the restriction. This parameter is used by Microsoft Exchange to find property tags which need to be remapped should the restriction be applied to a different store.

lppPropNames
: Output parameter pointing to where to return an array of MAPI property **NAMEID** structures that define the properties whose tags are given in the *lppPropTags* parameter. This value is used by Microsoft Exchange to obtain the appropriate new property tags should the restriction be applied to a different store.

lpfNot
: Output parameter pointing to a variable that returns TRUE if the user only wants messages that do not meet the criteria specified in the structure pointed to by the *lpres* parameter.

lpszDesc
: Output parameter pointing to a buffer which the extension should fill with a string containing a text description of the restriction.

cchDesc
: Input parameter indicating the maximum length of the *lpszDesc* parameter.

ulFlags
: Input parameter containing a bitmask of flags.

  EXCHEXT_UNICODE
  : Microsoft Exchange sets the *ulFlags* parameter to this value if it wants the extension to fill the *lpszDesc* parameter with a unicode string.

Comments

Microsoft Exchange calls the **IExchExtAdvancedCriteria::QueryRestriction** method when the criteria which the user has specified must be instantiated as a restriction. For example, when the user closes the Advanced dialog box, Microsoft Exchange prompts the extension object to return the current restriction. The restriction is a MAPI construct specifying a query. Memory for the returned restriction object must be allocated with the **MAPIAllocateMore** function using the *pvAllocBase* parameter as the base. **QueryRestriction** can be called zero, one, or multiple times depending on the context and the user's choices.

The Advanced Criteria restriction returned from this method must start with a Comment restriction that includes *lpProp[0].ulPropTag* == PR_COMCRIT_COMMENT_ID and *lpProp[0].Value.ul* == *ulComCritCmtAdvanced*. The actual restriction is included afterwards.

IExchExtAdvancedCriteria::SetFolders

Notifies an extension object when the folders to which the search criteria apply have been changed.

Syntax

VOID **SetFolders**(**LPENTRYLIST** *lpeidl*, **ULONG** *ulFlags*)

Parameters

lpeidl
: Input parameter pointing to an array of folder entry identifiers to which the search criteria will be applied.

ulFlags
: Input parameter containing a bitmask of flags. The following flag can be set:

  EEAC_INCLUDESUBFOLDERS
  : Microsoft Exchange sets the *ulFlags* parameter to this value to specify that subfolders of the *lpeidl* parameter are to be included in the search.

Comments

Microsoft Exchange calls the **IExchExtAdvancedCriteria::SetFolders** method when the user changes the folders to which the search criteria will be applied. This enables extensions to take additional actions depending on the newly selected folder.

IExchExtAdvancedCriteria::UninstallAdvancedCriteria

Notifies an extension object when the search criteria dialog box is closed.

Syntax

VOID **UninstallAdvancedCriteria**()

Comments

Microsoft Exchange calls the **IExchExtAdvancedCriteria::UninstallAdvancedCriteria** method when the search criteria dialog box is closed. The effect of this method is similar to what is expected during the **IExchExtMessageEvents::OnReadComplete**, **OnWriteComplete**, **OnSubmitComplete**, and **OnCheckNamesComplete** methods. When **UninstallAdvancedCriteria** is invoked, the extension object can release interfaces previously claimed with the **AddRef** method and release any other resources used while the criteria dialog box was displayed.

The extension object can discard any state information it is maintaining, such as memory allocations and the current state of the dialog box. The extension should not free any memory returned by the **IExchExtAdvancedCriteria::QueryRestriction** method.

IExchExtAttachedFileEvents : IUnknown

The **IExchExtAttachedFileEvents** interface is used to enable extension objects to replace or enhance the default attachment-handling behavior of Microsoft Exchange.

The **IExchExtAttachedFileEvents** interface is used only in the following contexts: EECONTEXT_SENDNOTEMESSAGE, EECONTEXT_READNOTEMESSAGE, EECONTEXT_SENDPOSTMESSAGE, EECONTEXT_READPOSTMESSAGE, EECONTEXT_READREPORTMESSAGE, EECONTEXT_SENDRESENDMESSAGE.

This interface is optional.

At a Glance

Specified in header file:	EXCHEXT.H
Object that supplies this interface:	Extension object
Corresponding pointer type:	LPEXCHEXTATTACHEDFILEEVENTS
Implemented by:	Extension objects
Called by:	Microsoft Exchange

Vtable Order

OnReadPattFromSzFile	Replaces or enhances the behavior of Microsoft Exchange when adding attachments to a message from a file.
OnWritePattToSzFile	Replaces or enhances the behavior of Microsoft Exchange when writing attachments from a message to a file (such as when responding to the Save command.)
QueryDisallowOpenPatt	Prevents a document file attachment from being opened directly without writing it first to an intermediate file.
OnOpenPatt	Replaces or enhances the behavior of Microsoft Exchange when opening a file attachment directly from the message.
OnOpenSzFile	Replaces or enhances the behavior of Microsoft Exchange when opening an attachment from a temporary file.

IExchExtAttachedFileEvents::OnReadPattFromSzFile

Replaces or enhances the behavior of Microsoft Exchange when adding attachments to a message from a file.

Syntax

HRESULT OnReadPattFromSzFile(**LPATTACH** *lpAtt*, **LPTSTR** *lpszFile*, **ULONG** *ulFlags*)

Parameters

lpAtt
: Input parameter pointing to the MAPI **IAttach : IMAPIProp** interface on the attachment. For more information on **IAttach**, see the *MAPI Programmer's Reference*.

lpszFile
: Input parameter pointing to a string containing the name and path of the file being attached.

ulFlags
: Input parameter containing a bitmask of flags. The following flag can be set:

  EXCHEXT_UNICODE
  : Specifies that the *lpszFile* parameter is a unicode string.

Return Values

S_OK
: The extension object replaced Microsoft Exchange behavior and handled attaching the file on its own. Microsoft Exchange will consider the task handled.

S_FALSE
: The extension object did nothing or added additional behavior. Microsoft Exchange will continue to call extensions or complete the work itself.

Comments

Microsoft Exchange calls the **IExchExtAttachedFileEvents::OnReadPattFromSzFile** method when it is about to read an attached file's data. This can occur on initial attachment and when the contents of the attachment are being updated from a temporary file. If an error occurs, **OnReadPattFromSzFile** should display an error message and return an error. Microsoft Exchange will not continue to attach the file, nor will it display an additional error message, but will stop the user action that requested the attachment.

IExchExtAttachedFileEvents::OnWritePattToSzFile

Replaces or enhances the behavior of Microsoft Exchange when writing attachments from a message to a file.

Syntax

HRESULT OnWritePattToSzFile(**LPATTACH** *lpAtt*, **LPTSTR** *lpszFile*, **ULONG** *ulFlags*)

Parameters

lpAtt
Input parameter pointing to the MAPI **IAttach : IMAPIProp** interface on the attachment. For more information on **IAttach**, see the *MAPI Programmer's Reference*.

lpszFile
Input parameter pointing to a string containing the name and path of the intermediate file being written.

ulFlags
Input parameter containing a bitmask of flags. The following flag can be set:

EXCHEXT_UNICODE
Specifies that the *lpszFile* parameter is a unicode string.

Return Values

S_OK
The extension object replaced Microsoft Exchange behavior and wrote out the file on its own. Microsoft Exchange will consider the task handled.

S_FALSE
The extension object did nothing or added additional behavior. Microsoft Exchange will continue to call extension objects or complete the work itself.

Comments

Microsoft Exchange calls the **IExchExtAttachedFileEvents::OnWritePattToSzFile** method when it is about to write an attachment to a file, such as when the user chooses the Save or Open button. If an error occurs, **OnWritePattToSzFile** should display an error message and return an error. Microsoft Exchange will not continue to try and write out the file, nor will it display an error message, but will stop the user action that requested the file be written.

IExchExtAttachedFileEvents::OnOpenPatt

Replaces or enhances the behavior of Microsoft Exchange when opening a document file attachment directly from a message.

Syntax

HRESULT OnOpenPatt(**LPATTACH** *lpAtt*)

Parameters

lpAtt
Input parameter pointing to the MAPI **IAttach : IMAPIProp** interface on the attachment. For more information on **IAttach**, see the *MAPI Programmer's Reference*.

Return Values

S_OK
The extension object replaced Microsoft Exchange behavior and opened the file on its own. Microsoft Exchange will consider the task handled.

S_FALSE
The extension object did nothing or added additional behavior. Microsoft Exchange will continue to call extensions or complete the work itself.

Comments

Microsoft Exchange calls the **IExchExtAttachedFileEvents::OnOpenPatt** method when it is about to open an attachment directly from the message. Before writing it to an intermediate file, Microsoft Exchange allows extension objects to open directly from the attachment. File viewers might use this method to display the content of the attachment without writing an intermediate file to the file system.

Microsoft Exchange attempts to find an application that is registered to handle the attached file's class. If no such application can be found, Microsoft Exchange writes the attachment to a temporary file and prompts Microsoft Windows to start the application that is registered to handle files with the given file extension (such as .XLS for Microsoft Excel).

It then calls the **IExchExtAttachedFileEvents::OnOpenSzFile** method on extension objects instantiated in the context that supports the **IExchExtAttachedFileEvents : IUnknown** interface. Microsoft Exchange may try to open an attachment from a file after learning from **IExchExtAttachedFileEvents::QueryDisallowOpenPatt** that it is allowed to do so. At this point, Microsoft Exchange prompts the extension object if it wants to do special handling of the file such as replacing the open-attachment behavior or by doing other work such as detecting viruses.

If an error occurs, **OnOpenPatt** should display an error message and return an error. Microsoft Exchange will not continue to try and open the attachment, nor will it display an error message, but will stop the user action that requested the attachment be opened.

IExchExtAttachedFileEvents::OnOpenSzFile

Replaces or enhances the behavior of Microsoft Exchange when opening an attachment from a temporary file.

Syntax

HRESULT OnOpenSzFile(**LPTSTR** *lpszFile*, **ULONG** *ulFlags*)

Parameters

lpszFile
: Input parameter pointing to a string containing the name and path of the temporary file being opened.

ulFlags
: Input parameter containing a bitmask of flags. Set this parameter to one of the following values:

  EXCHEXT_UNICODE
  : Indicates that the *lpszFile* parameter is a unicode string.

  EEAFE_OPEN
  : Indicates that the attachment is being opened.

  EEAFE_PRINT
  : Indicates that the attachment is being printed.

  EEAFE_QUICKPREVIEW
  : Indicates that the attachment is being quickviewed.

Return Values

S_OK
: The extension object replaced Microsoft Exchange behavior and opened the file on its own. Microsoft Exchange will consider the task handled.

S_FALSE
: The extension object did nothing or added additional behavior. Microsoft Exchange will continue to call extensions or, if necessary, open the file attachment itself.

Comments

Microsoft Exchange calls the **IExchExtAttachedFileEvents::OnOpenSzFile** method when it is about to open an attachment, but was not able to open it directly from the data in the attachment in the message. When this occurs, the attachment data is written to an intermediate file. Microsoft Exchange attempts to find an application that is registered to handle the attached file's class. If it finds it, the application is started and prompted to open the attachment.

If an error occurs, **OnOpenSzFile** should display an error message and return an error. Microsoft Exchange will not continue to try and open the attachment, nor will it display an error message, but will stop the user action that requested the attachment be opened.

IExchExtAttachedFileEvents::QueryDisallowOpenPatt

Prevents a file attachment from being opened directly without writing it first to an intermediate file.

Syntax

HRESULT QueryDisallowOpenPatt(**LPATTACH** *lpAtt*)

Parameters

lpAtt
: Input parameter pointing to the MAPI **IAttach : IMAPIProp** interface on the attachment. For more information on **IAttach**, see the *MAPI Programmer's Reference*.

Return Values

S_OK
: The extension object will prevent other extensions or Microsoft Exchange from opening the attachment directly without writing it to a file first.

S_FALSE
: The extension object did nothing. Microsoft Exchange will call other extension objects if opening the attachment directly ought to be disallowed.

Comments

Microsoft Exchange calls the **IExchExtAttachedFileEvents::QueryDisallowOpenPatt** method when it is about to open the attachment directly without writing it first to an intermediate file. An extension object might elect to disallow opening attachments directly if it has encrypted or modified the attachment and should decrypt or restore it to an intermediate file.

If an error occurs, **QueryDisallowOpenPatt** should display an error message and return an error. Microsoft Exchange will not continue to try to open the file, nor will it display an error message, but will stop the user action that requested the attachment be opened.

IExchExtCallback : IUnknown

The **IExchExtCallback** interface enables extension objects to retrieve information about the current context, including the objects that are currently selected in the Microsoft Exchange window. **IExchExtCallback** uses the methods of the **IUnknown** interface for reference management.

IExchExtCallback is implemented by a callback object supplied by Microsoft Exchange. A pointer to this interface is passed as a parameter to many methods of the Microsoft Exchange extensibility API. This interface and any interfaces it might return are valid only for the time of the call to one of its methods and might not be retained when the extension object's method returns. The called method does not need to release the callback object unless it has completed a corresponding call to **AddRef**.

At a Glance

Specified in header file:	EXCHEXT.H
Object that supplies this interface:	Extension object
Corresponding pointer type:	LPEXCHEXTCALLBACK
Implemented by:	Microsoft Exchange
Called by:	Extension objects

Vtable Order

GetVersion	Returns the version number of the Microsoft Exchange application.
GetWindow	Returns a window handle corresponding to the specified flag.
GetMenu	Returns the Microsoft Exchange menu handle for the current window.
GetToolbar	Returns a toolbar's window handle.
GetSession	Returns an interface to the current open MAPI session and associated address book.
GetObject	Returns an interface and store for a particular object.
GetSelectionCount	Returns the number of objects selected in the window.
GetSelectionItem	Returns the entry identifier of a selected item in a Microsoft Exchange window.
GetMenuPos	Returns the position of a command or set of commands on the Microsoft Exchange menu.
GetSharedExtsDir	Returns the Microsoft Exchange shared-extensions directory.
GetRecipients	Returns a pointer to the recipient list of the currently selected item.
SetRecipients	Sets the recipient list for the currently selected item.
GetNewMessageSite	Returns interface pointers to the message site and view context of the selected message.
RegisterModeless	Enables extension objects that display modeless windows to coordinate with windows displayed by the Microsoft Exchange client.
ChooseFolder	Displays a dialog box that enables users to choose a specific message store and folder.

IExchExtCallback::ChooseFolder

Displays a dialog box that enables users to choose a specific message store and folder.

Syntax

HRESULT ChooseFolder(**LPEXCHEXTCHOOSEFOLDER** *peecf*)

Parameters

peecf
Input parameter pointing to a structure that describes the Choose Folder dialog box. The structure also includes output members that provide information about the selections made by the user when the user chooses the OK button. The structure has the following members:

UINT cbLength
Input member indicating the size of the structure.

HWND hwnd
Input member indicating the parent window of the Choose Folder dialog box.

LPTSTR szCaption
Input member indicating the caption that is displayed in the dialog box's title bar.

LPTSTR szLabel
Input member indicating a label that is displayed at the top of the list box of folders.

LPTSTR szHelpFile
Input member specifying which Help file should be launched when the user chooses the Help button.

ULONG ulHelpID
Input member indicating the topic that should be referenced in the Help file.

HINSTANCE hinst
Input member indicating the instance handle containing the resource of a custom dialog box. Set this value only if you want to supply your own dialog box template.

UINT uiDlgID
Input member indicating the resource identifier of the instance of the custom dialog box template. Set this value only if you set the *hinst* parameter.

LPEECFHOOKPROC lpeecfhp
Input member indicating a hook proc pointer. Set this value only if you want to do special processing of messages within the dialog box.

DWORD dwHookData
Input member used to cache additional information about events that occur while the dialog box is displayed. Use this member if you are providing a hook process for handling dialog box messages.

ULONG ulFlags
Input member containing a bitwise combination of the following flags:

EECBCF_GETNAME	Determines whether the name of the folder selected by the user is returned. This name must be freed using the **MAPIFreeBuffer function**. For more information on **MAPIFreeBuffer**, see the *MAPI Programmer's Reference*.
EECBCF_HIDENEW	Hides the New button on the Choose Folder dialog box. This button enables users to create a new folder.
EECBCF_PREVENTROOT	Disables the OK button when the root is the

currently selected folder. In most cases, this flag should be specified because the root folder is not a valid container of messages.

LPMDB pmdb
Output member that is an interface pointer to the message store object selected in the Choose Folder dialog box.

LPMAPIFOLDER pfld
Output member that is an interface pointer to the folder object selected in the Choose Folder dialog box.

LPTSTR szName
Output member indicating the name of the folder the user selected. This value enables you to get the name of the folder without using a MAPI **GetProps** method. This value is only provided if the EECBCF_GETNAME flag is set.

DWORD dwReserved1 - dwReserved3
These three members are reserved for future use and should be set to NULL by extensions.

Return Values

S_OK
No errors occurred.

Comments

Extension objects call the **IExchExtCallback::ChooseFolder** method to enable users to select a specific message store or folder. For example, if your application provides a backup command, you might want to prompt the user to specify which folder in which message store should be backed up.

IExchExtCallback::GetMenu

Returns the menu handle for the current Microsoft Exchange window.

Syntax

HRESULT GetMenu(**HMENU FAR *** *lphMenu*)

Parameters

lphMenu
: Output parameter pointing to the Microsoft Exchange menu handle.

Return Values

S_OK
: The *lphMenu* parameter returned a non-NULL HMENU.

S_FALSE
: The *lphMenu* parameter returned a NULL HMENU.

E_INVALIDARG
: The *lphMenu* parameter is NULL or no HWND is associated with the context where the extension is installed.

IExchExtCallback::GetMenuPos

Returns the position of a command, menu, or set of commands on the Microsoft Exchange menu for the current context.

Syntax

HRESULT GetMenuPos(**UINT** *cmdid*, **HMENU FAR** * *lphmenu*, **ULONG FAR** * *lpmposMin*, **ULONG FAR** * *lpmposMax*, **ULONG** *ulFlags*)

Parameters

cmdid
Input parameter indicating the command identifier of interest.

lphMenu
Output parameter pointing to the menu handle where the *cmdid* parameter can be found.

lpmposMin
Output parameter pointing to the menu position above the command.

lpmposMax
Output parameter pointing to the menu position below the command.

ulFlags
Input parameter containing a bitmask of flags. The following flag can be set:

EECBGMP_RANGE
Specifies that the returned positions should correspond to the separator-delimited range of the menu containing the command.

Return Values

S_OK
No errors occurred.

E_FAIL
Microsoft Exchange cannot find the menu associated with the *cmdid* parameter.

E_INVALIDARG
The command identifier does not correspond to a menu item or the *lphmenu* parameter is NULL, or there is no HMENU associated with the context where the extension is installed.

IExchExtCallback::GetNewMessageSite

Returns interface pointers to the message, message site, and view context for a new message that is being composed. Using these interface pointers, an extension object can display a form for the new message.

Syntax

HRESULT GetNewMessageSite(**ULONG** *fComposeInFolder*, **LPMAPIFOLDER** *pfldFocus*, **LPPERSISTMESSAGE** *ppermsg*, **LPMESSAGE FAR *** *ppmsg*, **LPMAPIMESSAGESITE FAR *** *ppmms*, **LPMAPIVIEWCONTEXT FAR *** *ppmvc*, **ULONG** *ulFlags*)

Parameters

fComposeInFolder
Input parameter indicating whether the message is being composed in the same folder from which it was activated. If this flag is TRUE, the message site supports default behavior for posting messages, which means that the message always remains in the same folder in which it was created. If this flag is FALSE, the message site supports default behavior for sending messages, which means that messages are automatically moved to the Outbox or Inbox when the user sends or saves the message.

pfldFocus
Input parameter pointing to the folder in which the message is to be composed.

ppermsg
Input parameter pointing to a variable that points to the **IPersistMessage** interface on the form for which the new message site is being obtained.

ppmsg
Output parameter pointing to a variable where the pointer to the message being composed is stored.

ppmms
Output parameter pointing to a variable where the pointer to the MAPI message site for the new message is stored.

ppmvc
Output parameter pointing to a variable where the pointer to the MAPI view context for the new message is stored.

ulFlags
Input parameter containing a bitmask of flags. The following flag can be set:

EECBGNMS_MODAL
Indicates that the new form should be modal.

Return Values

S_OK
No error occurred.

E_INVALIDARG
No session is associated with the context where the extension is installed or the *ulFlags* parameter has EECBGNMS_MODAL set.

Comments

Extension objects call the **IExchExtCallback::GetNewMessageSite** method to display a form for a new message the user has created. For example, an extension that performs scheduling functions might have a New Meeting menu command which results in the creation of a new message and the display of a custom form that enables users to fill in the fields of the message.

IExchExtCallback::GetObject

Returns interfaces for a particular object and the object's containing store. The interfaces returned depend on the context of the caller. For a list of possible interfaces returned, see the "Comments" section.

Syntax

HRESULT GetObject(**LPMDB FAR *** *lppmdb*, **LPMAPIPROP FAR *** *lppmp*)

Parameters

lppmdb
: Output parameter pointing to a variable where the pointer to the store object containing the *lppmp* parameter is stored.

lppmp
: Output parameter pointing to a variable where the pointer to a MAPI object is stored.

Return Values

S_OK
: Either the *lppmdb* or *lppmp* parameter returned non-NULL values.

S_FALSE
: Both the *lppmdb* and *lppmp* parameters returned NULL values.

E_INVALIDARG
: Both the *lppmdb* and *lppmp* parameters are NULL.

Comments

The **IExchExtCallback::GetObject** method is used to return an interface and message store for a particular object. The interface returned depends on the context or the caller. When called by the **IExchExtPropertySheets::GetPages** method, the object is the object for which information should be displayed, and the store is the store containing that object.

The interfaces must be released before the call which was given the *lpeecb* parameter returns. The following list describes the interfaces that are returned when **GetObject** is called from various contexts.

EECONTEXT_SESSION, EECONTEXT_TASK
: Both interfaces returned are NULL.

EECONTEXT_VIEWER
: The interfaces returned by **GetObject** are NULL when called from the **IExchExt::Install** or **IExchExtCommands::InstallCommands** methods because the Viewer is not necessarily permanently associated with the same store or object.

  When called from the **IExchExtCommands::InitMenu** or **IExchExtCommands::DoCommand** methods, the object is the open folder being displayed and the store is the store that contains that folder. When the root of a store is displayed, the object is the IPM_SUBTREE folder. Both pointers are NULL when the list of stores is displayed in the content pane.

EECONTEXT_REMOTEVIEWER
: Both interfaces are NULL.

EECONTEXT_SEARCHVIEWER
: The object is the search folder being displayed, and the store is the store which contains that search folder.

EECONTEXT_ADDRBOOK
: The interfaces are generally NULL when called from **Install** or **InstallCommands** because the address book is not always necessarily associated with the same container.

When called from **InitMenu** and **DoCommand**, the object is the ABContainer being displayed and the store is NULL. **GetObject** is called in the following contexts:

EECONTEXT_SENDNOTEMESSAGE,
EECONTEXT_READNOTEMESSAGE,
EECONTEXT_SENDPOSTMESSAGE,
EECONTEXT_READPOSTMESSAGE,
EECONTEXT_READREPORTMESSAGE,
EECONTEXT_SENDRESENDMESSAGE,

The object is the message being displayed and the store is the store which contains that message.

The object interface returned in the *lppmp* parameter supports the **IMAPIProp : IUnknown** interface as well as the additional interface methods specific to its type.

Object	**Interface type**
Folder	**IMAPIFolder**
Message	**IMessage**
Store	**IMsgStore**
Address Book	**IABContainer**

See Also

IExchExt::Install method, **IExchExtAdvancedCriteria::InstallAdvancedCriteria** method

IExchExtCallback::GetRecipients

Returns a pointer to the recipient list for the currently open item.

Syntax

HRESULT GetRecipients(**LPADRLIST FAR *** *lppal*)

Parameters

lppal
  Output parameter pointing to the location where the address list is returned.

Return Values

S_OK
  No error occurred.

E_INVALIDARG
  The function was called from a context that does not support recipients or there is no recipients list.

Comments

The recipient list is returned in a MAPI **ADRLIST** structure. For more information on the MAPI **ADRLIST** structure, see the *MAPI Programmer's Reference*.

This method is called in the following contexts:

EECONTEXT_SENDNOTEMESSAGE,
EECONTEXT_READNOTEMESSAGE,
EECONTEXT_SENDPOSTMESSAGE,
EECONTEXT_READPOSTMESSAGE,
EECONTEXT_READREPORTMESSAGE,
EECONTEXT_SENDRESENDMESSAGE

See Also

IExchExtCallback::SetRecipients method

IExchExtCallback::GetSelectionCount

Returns the number of objects selected in the active window. Selections can only be made in windows that display containers.

Syntax

HRESULT GetSelectionCount(ULONG FAR * *lpceid***)**

Parameters

lpceid
Output parameter pointing to a variable containing the number of objects selected in the window.

Return Values

S_OK
The *lpceid* parameter returned the count of the current selection.

E_INVALIDARG
The *lpceid* parameter is NULL or **GetSelectionCount** was called from an invalid context (EECONTEXT_SESSION, EECONTEXT_TASK, EECONTEXT_MESSAGE).

Comments

The **IExchExtCallback::GetSelectionCount** method is used to return the number of objects selected in the active window. The window is determined by the context in which the extension object is installed.

See Also

IExchExtCommands::InitMenu method

IExchExtCallback::GetSelectionItem

Returns the entry identifier and other information of the currently selected item in a Microsoft Exchange window.

Syntax

HRESULT GetSelectionItem(**ULONG** *ieid*, **ULONG FAR *** *lpcbeid*, **LPENTRYID FAR *** *lppeid*, **ULONG FAR *** *lpulType*, **LPTSTR** *lpszMsgClass*, **ULONG** *cbMsgClass*, **ULONG FAR *** *lpulMsgFlags*, **ULONG** *ulFlags*)

Parameters

ieid
Input parameter indicating the index of the entry identifier to be retrieved. Valid values for this parameter are 0 to *lpceid* -1 where *lpceid* is returned in the **IExchExtCallback::GetSelectionCount** method.

lpcbeid
Output parameter pointing to the number of bytes returned in the *lppeid* parameter.

lppeid
Output parameter pointing to a variable where the pointer to the entry identifier of the selection is stored.

lpulType
Output parameter pointing to the type of entry identifier returned in the *lppeid* parameter. This parameter can have the following values:

MAPI_STORE = 1, if *lppeid* refers to a message store.

MAPI_ADDRBOOK = 2, if *lppeid* refers to an address book.

MAPI_FOLDER = 3, if *lppeid* refers to a folder.

MAPI_MESSAGE = 5, if *lppeid* refers to a message.

lpszMsgClass
Input-output parameter pointing to a buffer into which Microsoft Exchange will copy the message class of the selected message.

cbMsgClass
Output parameter indicating the size of the message class.

lpulMsgFlags
Output parameter pointing to the message flags returned. For more information on PR_MESSAGE_FLAGS, see the *MAPI Programmer's Reference*.

ulFlags
Reserved; must be zero.

Return Values

S_OK
No errors occurred.

S_FAIL
Microsoft Exchange failed to access the selected item.

E_INVALIDARG
The *ulFlags* parameter is not fMapiUnicode or it is called from an invalid context that does not support the selected item.

Comments

The **IExchExtCallback::GetSelectionItem** method is used to return the entry identifier of the currently selected item in a Microsoft Exchange Window. For example, if a custom form is installed and the user selects this form, extension objects that implement the **IExchExtUserEvents::OnSelectionChange** method can use **GetSelectionItem** to determine the message class of the selected message. If the extension object is programmed to handle messages of this class, it might display or enable menu items specific to the message class.

If you do not use a parameter in this method, pass NULL for that parameter.

See Also

IExchExtCommands::InitMenu method

IExchExtCallback::GetSession

Returns interfaces to the current open MAPI session and associated address book.

Syntax

HRESULT GetSession(**LPMAPISESSION FAR *** *lppses*, **LPADRBOOK FAR *** *lppab*)

Parameters

lppses
: Output parameter pointing to a variable where the pointer to the current MAPI session object is stored.

lppab
: Output parameter pointing to a variable where the pointer to the address book object associated with the *lppses* parameter is stored.

Return Values

S_OK
: Either the *lppses* or *lppab* parameter returned a non-NULL value.

S_FALSE
: Both the *lppses* and *lppab* parameters returned NULL values.

E_INVALIDARG
: Both the *lppses* and *lppab* parameters are NULL.

Comments

The *lppses* and *lppab* parameters might be NULL. Both interfaces must be released before the method which was passed the *lpeecb* parameter returns a value.

IExchExtCallback::GetSharedExtsDir

Returns the location of the Microsoft Exchange shared extensions directory.

Syntax

HRESULT GetSharedExtsDir(**LPTSTR** *lpszDir*, **ULONG** *cchDir*, **ULONG** *ulFlags*)

Parameters

lpszDir
: Input-output parameter pointing to a buffer to be filled with the location of the shared directory.

cchDir
: Input parameter indicating the size of the *lpszDir* parameter.

ulFlags
: Input parameter containing a bitmask of flags. The following flag can be set:

  EXCHEXT_UNICODE
  : Specifies that the *lpszDir* parameter is a unicode string.

Return Values

S_OK
: No error occurred.

S_FALSE
: There is no SharedExtsDir.

E_INVALIDARG
: The *lpszDir* parameter is NULL, the *cchDir* parameter is zero, or the *ulFlags* parameter is not fMapiUnicode.

IExchExtCallback::GetToolbar

Returns a toolbar window handle.

Syntax

HRESULT GetToolbar(**ULONG** *tbid*, **HWND FAR *** *lphwndTb*)

Parameters

tbid
: Input parameter indicating the toolbar identifier.

lphwndTb
: Output parameter pointing to the Microsoft Exchange toolbar window handle.

Return Values

S_OK
: The *lphWnd* parameter returned a non-NULL HWND.

S_FALSE
: The *lphWnd* parameter returned a NULL HWND.

E_INVALIDARD
: The *tbid* parameter is not EETB_STANDARD or the *lphWnd* parameter is NULL.

Comments

Currently, only the standard toolbar defined by EETB_STANDARD is supported. It is therefore not possible to add toolbar buttons to other toolbars such as the formatting toolbar.

See Also

IExchExtCommands::ResetToolbar method

IExchExtCallback::GetVersion

Returns the version number of the Microsoft Exchange client.

Syntax

HRESULT GetVersion(**ULONG FAR** * *lpulVersion*, **ULONG** *ulFlags*)

Parameters

lpulVersion
Output parameter pointing to the location where the version of Microsoft Exchange is returned. The format of the returned version is determined by the value of the *ulFlags* parameter.

ulFlags
Input parameter containing a bitmask of flags used to specify the type and format of the version information that is returned. These flags can have the following values:

EECBGV_GETBUILDVERSION
Requests that the major version is returned in the HIWORD and the minor version in the LOWORD. Extension objects should verify that the major version matches the EECBGV_BUILDVERSION_MAJOR with which they were compiled, and that the minor version is at least the minor version with which the extension object was tested.

EECBGV_GETACTUALVERSION
Requests that the version returned corresponds to the actual version of the calling client.

EECBGV_GETVIRTUALVERSION
Requests that the version returned is the version which the client wants the extension to base its decisions on. For example, if an application named Zippypost 5.0 implements Microsoft Exchange extensibility to exactly match that implemented in Microsoft Exchange, the actual version would be Zippypost 5.0 but the virtual version would be Microsoft Exchange.

Return Values

S_OK
No error occurred.

Comments

Extension objects should test against the virtual version and should reserve actual version checking for determination of additional features specific to a particular kind of client or version.

The actual and virtual versions are made up of four bytes, which represent in order from most significant to least the product, the platform, the major version, and the minor version. For example, Microsoft Exchange 4.0 for Windows NT is 0x01030400. Microsoft Exchange for the Macintosh® would be 0x0104052A. Zippypost 3.2 for Windows 3.1 might be 0x13010320.

See Also

IExchExt::Install method

IExchExtCallback::GetWindow

Returns the window handle for the current Microsoft Exchange window.

Syntax

HRESULT GetWindow(**HWND FAR *** *lphwnd*)

Parameters

lphwnd
Output parameter pointing to the Microsoft Exchange window handle.

Return Values

S_OK
The *lphwnd* parameter returned a non-NULL HWND.

S_FALSE
The *lphwnd* parameter returned a NULL HWND.

E_INVALIDARG
The *lphwnd* parameter is NULL.

IExchExtCallback::RegisterModeless

Ensures coordination between the modeless windows and modal windows displayed by the Microsoft Exchange client.

Syntax

HRESULT RegisterModeless (LPEXCHEXTMODELESS *peem*,
LPEXCHEXTMODELESSCALLBACK FAR * *ppeemcb***)**

Parameters

peem
Input parameter pointing to an **IExchExtModeless** interface implemented by the extension. This interface should be implemented as another object with separate reference counts from the main extension object.

ppeemcb
Output parameter pointing to an **IExchExtModelessCallback** interface implemented by Microsoft Exchange.

Return Values

S_OK
No errors occurred.

S_FALSE
It was not possible to register a modeless window for the extension.

E_INVALIDARG
The *peemObj* or *ppeemcb* parameter is NULL.

Comments

Extension objects that run in the process space of the Microsoft Exchange client and display modeless windows use the **IExchExtCallback::RegisterModeless** method primarily to enable and disable Microsoft Exchange windows and extension object windows in a coordinated manner. If an extension object has registered for modeless behavior by calling this method and has called **IExchExtModelessCallBack::AddWindow**, Microsoft Exchange invokes **IExchExtModeless::EnableModeless** (with its *fEnable* parameter set to FALSE) before displaying a modal window. The extension object can then disable its modeless windows. When Microsoft Exchange removes its modal window, it invokes **EnableModeless** (with its *fEnable* parameter set to TRUE). The extension object can then re-enable its modeless windows.

Similary, if a modeless extension object needs to display a modal window, it should call **IExchExtModelessCallback::EnableModeless** (with its *fEnable* parameter set to FALSE). Microsoft Exchange will then disable its modeless windows. When the extension object removes its modal window, it should invoke **IExchExtModeless::EnableModeless** (with its *fEnable* parameter set to TRUE), enabling Microsoft Exchange to re-enable its modeless windows.

When an extension registers itself as modeless, it supplies a modeless object that enables Microsoft Exchange to communicate with the extension object about the state of its windows.

See Also

IExchExtModeless interface

IExchExtCallback::SetRecipients

Sets the recipient list for the currently selected item.

Syntax

HRESULT SetRecipients(LPADRLIST *lpal*)

Parameters

lpal
Input parameter pointing to the location of the new address list.

Return Values

S_OK
No error occurred.

E_INVALIDARG
The method was called from a context that does not support recipients.

Comments

Use MAPI functions to fill a MAPI **ADRLIST** structure with the appropriate information.

This method is called in the following contexts:

EECONTEXT_SENDNOTEMESSAGE,
EECONTEXT_READNOTEMESSAGE,
EECONTEXT_SENDPOSTMESSAGE,
EECONTEXT_READPOSTMESSAGE,
EECONTEXT_READREPORTMESSAGE,
EECONTEXT_SENDRESENDMESSAGE

See Also

IExchExtCallback::GetRecipients

IExchExtCommands : IUnknown

The **IExchExtCommands** interface is implemented by extension objects to add and execute custom menu or toolbar command buttons. Extension objects can also replace existing Microsoft Exchange commands or enhance their behavior before Microsoft Exchange carries them out.

Use the operating system's native API to implement the methods given in the following table. For example, when implementing the **IExchExtCommands::InstallCommands** method for an extension object that will run in the Microsoft Windows 95 environment, the extension object must use the Win32 structures, messages, and function calls for Windows 95 to manipulate menu or toolbar commands. **IExchExtCommands** is used in all contexts except EECONTEXT_SESSION, EECONTEXT_TASK, and EECONTEXT_PROPERTYSHEETS.

This interface is optional.

At a Glance

Specified in header file:	EXCHEXT.H
Object that supplies this interface:	Extension object
Corresponding pointer type:	LPEXCHEXTCOMMANDS
Implemented by:	Extension objects
Called by:	Microsoft Exchange

Vtable Order

InstallCommands	Enables an extension to install its menu commands or toolbar buttons.
InitMenu	Enables an extension to update its menu items when the user begins using the menus.
DoCommand	Carries out a menu or toolbar command chosen by the user.
Help	Provides user Help for a command.
QueryHelpText	Provides status bar or tool tip Help text for a command.
QueryButtonInfo	Provides information about the extension's toolbar buttons.
ResetToolbar	Enables an extension to restore its toolbar buttons to their default positions.

IExchExtCommands::DoCommand

Carries out a menu or toolbar command chosen by the user.

Syntax

HRESULT DoCommand(**LPEXCHEXTCALLBACK** *lpeecb*, **UINT** *cmdid*)

Parameters

lpeecb
: Input parameter pointing to an **IExchExtCallback** interface.

cmdid
: Input parameter indicating the command identifier of the command to be carried out.

Return Values

S_OK
: The extension object recognized the *cmdid* parameter and carried out the command. Microsoft Exchange will consider the task handled.

S_FALSE
: The extension object did not recognize the *cmdid* parameter. Microsoft Exchange will continue prompting extension objects to carry out the command and, if necessary, carry out the command itself. In some cases, an extension object might perform some action but still return S_FALSE to give other extensions the opportunity to handle the command.

Comments

Microsoft Exchange calls the **IExchExtCommands::DoCommand** method when it receives a WM_COMMAND or WM_SYSCOMMAND message corresponding to a menu command, system menu command, or toolbar button so that the extension object can carry out any commands it implements or supersede the Microsoft Exchange implementation of a command. The command identifier passed to WM_COMMAND is mapped to the published set of Microsoft Exchange commands. It is this published command identifier that is passed to **DoCommand** in the *cmdid* parameter. The command identifier in WM_COMMAND maps to, but is different than, the command identifier specified in the *cmdid* parameter. Extension objects should therefore not rely on this mapping.

If an error occurs, **DoCommand** should display a message and return an error value. Microsoft Exchange will not continue trying to perform the command nor will it display a message.

The system command identifiers that extensions can override include:

SC_CLOSE
SC_MAXIMIZE
SC_MINIMIZE
SC_MOVE
SC_RESTORE
SC_SIZE
SC_TASKLIST

These commands are standard wparam values for the WM_SYSCOMMAND message.

IExchExtCommands::Help

Provides user Help for a command.

Syntax

HRESULT Help(**LPEXCHEXTCALLBACK** *lpeecb*, **UINT** *cmdid*)

Parameters

lpeecb
: Input parameter pointing to an **IExchExtCallback** interface.

cmdid
: Input parameter indicating the command identifier selected by the user.

Return Values

S_OK
: The extension object recognized the *cmdid* parameter and provided Help. Microsoft Exchange will consider the task handled.

S_FALSE
: The extension object did not recognize the *cmdid* parameter. Microsoft Exchange will continue prompting extension objects and, if necessary, provide Help itself.

Comments

Microsoft Exchange calls the **IExchExtCommands::Help** method when the user requests Help on a command. If the extension object has added a menu, the *cmdid* parameter corresponding to that menu is the *cmdid* parameter of the first command on the menu minus one.

The extension object can provide Help for any command it implements or supersede Microsoft Exchange Help text for an existing command.

IExchExtCommands::InitMenu

Enables an extension object to update its menu items when the user begins to use the menu.

Syntax

VOID **InitMenu** (**LPEXCHEXTCALLBACK** *lpeecb*)

Parameters

lpeecb

Input parameter pointing to an **IExchExtCallback** interface. The *lpeech* parameter is used to determine information about the selection if this is appropriate to the context.

Comments

Microsoft Exchange calls the **IExchExtCommands::InitMenu** method when it receives a WM_INITMENU message, allowing the extension object to enable, disable, or update its menu commands before the user sees them.This method is called frequently and should be written in a very efficient manner.

If your command extension needs to be enabled when specific message classes or folders are selected, use the **IExchExtCallback::GetSelectionCount** method to determine whether there currently is a selection, and use the **GetSelectionItem** method to traverse the selection. You would traverse the selection to see whether the selected objects are of interest to your command extension.

Microsoft Exchange intercepts the **EnableMenuItem** message and applies the same enabled or disabled state to the associated toolbar button. These menu items and toolbar buttons do not have to be set independently.

See Also

IExchExtCallback::GetSelectionCount method, **IExchExtCallback::GetSelectionItem** method

IExchExtCommands::InstallCommands

Enables an extension object to install its menu commands and toolbar command buttons into a Microsoft Exchange window.

Syntax

HRESULT InstallCommands(**LPEXCHEXTCALLBACK** *lpeecb*, **HWND** *hwnd*, **HMENU** *hmenu*; **UINT FAR *** *lpcmdidBase*, **LPTBENTRY** *lptbeArray*, **UINT** *ctbe*, **ULONG** *ulFlags*)

Parameters

lpeecb
Input parameter pointing to an **IExchExtCallback** interface.

hwnd
Input parameter indicating the handle of the top-level Microsoft Exchange window.

hmenu
Input parameter indicating the Microsoft Exchange menu handle.

lpcmdidBase
Input-output parameter pointing to the first command identifier the extension object can use. This is also the command identifier for a menu item or toolbar button.

lptbeArray
Input parameter pointing to an array of toolbars.

ctbe
Input parameter indicating the number of toolbars in the *lptbeArray* parameter.

ulFlags
Reserved; must be zero.

The following members are used with the **TBENTRY** structure, which is defined in the EXCHEXT.H header file:

hwnd
Input member indicating the window handle of the toolbar.

tbid
Input member indicating the toolbar identifier. A toolbar identifier distinguishes between multiple toolbars, such as the standard toolbar and the format toolbar. Currently, only the standard toolbar is defined with the EETB_STANDARD value identifier.

ulFlags
Reserved; must be zero.

itbbBase
Input-output member indicating the index of the extension's first toolbar button in the set of available buttons.

Return Values

S_OK
No error occurred.

Comments

Microsoft Exchange calls the **IExchExtCommands::InstallCommands** method to enable an extension object to install its menu commands and toolbar command buttons into a Microsoft Exchange window. Microsoft Exchange invokes **InstallCommands** before a top level Microsoft Exchange window, such as the Viewer, is made visible.

To prevent identifier collisions, the *lpcmdidBase* parameter points to the first identifier the extension object can use for a command identifier. The extension must update the contents of the *lpcmdidBase* parameter so that on return it contains the first identifier the next extension can use. The *lptbeArray* parameter contains a list of toolbars with the number of toolbars in the *ctbe* parameter.

To prevent identifier collisions, the *itbbBase* parameter contains the index of the extension's first toolbar button in the set of available buttons. This index is not the same as the position of the extension's first button displayed in the toolbar.

The extension object must use the Windows API to add menus, popup menus, and top-level menus. Windows 95 toolbar messages are used on all platforms to register and add toolbar buttons. If an extension object elects to place its toolbar buttons on the toolbar by default, the extension object must add them to the toolbar's set of available buttons when **InstallCommands** is called and then position those buttons on the toolbar when the **IExchExtCommands::ResetToolbar** method is called.

If an error is returned, Microsoft Exchange will not try to install another extension and will fail to create the window associated with the context.

IExchExtCommands::QueryButtonInfo

Provides information about the extension object's toolbar buttons.

Syntax

HRESULT QueryButtonInfo(**ULONG** *tbid*, **UINT** *itbb*, **LPTBBUTTON** *ptbb*, **LPTSTR** *lpsz*, **UINT** *cch*, **ULONG** *ulFlags*)

Parameters

tbid
Input parameter indicating the toolbar identifier.

itbb
Input parameter indicating an index into the list of available buttons.

ptbb
Input-output parameter pointing to a **TBBUTTON** toolbar button structure, which can be found in the EXCHEXT.H header file.

lpsz
Input parameter pointing to a buffer in which text describing the button should be returned. This description text appears in the Customize Toolbar dialog box. This parameter can be NULL to indicate that text is not required.

cch
Input parameter indicating the size of the buffer pointed to by the *lpsz* parameter.

ulFlags
Input parameter containing a bitmask of flags. The following flag can be set:

EXCHEXT_UNICODE
Specifies that the *lpsz* parameter is a unicode string.

Return Values

S_OK
The extension object recognized the *itbb* parameter and provided information.

S_FALSE
The extension object did not recognize the *itbb* parameter. Microsoft Exchange will continue to call extension objects and if necessary provide information itself.

Comments

Microsoft Exchange calls the **IExchExtCommands::QueryButtonInfo** method when it needs information about a toolbar button. Currently only one toolbar is defined, EETB_STANDARD. For information on manipulating toolbars, see the Microsoft Windows 95 Software Developer's Kit.

If an error is returned, Microsoft Exchange will not call other extensions and will use default or null information.

IExchExtCommands::QueryHelpText

Provides status bar or tool tip Help text for a command.

Syntax

HRESULT QueryHelpText(**UINT** *cmdid*, **ULONG** *ulFlags*, **LPTSTR** *lpsz*, **UINT** *cch*)

Parameters

cmdid
Input parameter indicating the command identifier selected by the user.

ulFlags
Input parameter containing a bitmask of flags. The following flags can be set:

EECQHT_STATUS
Microsoft Exchange requests a string to be displayed in the status bar.

EECQHT_TOOLTIP
Microsoft Exchange requests a string to be displayed in a tool tip.

lpsz
Input-output parameter indicating the buffer to be filled with Help text.

cch
Input parameter indicating the size of the *lpsz* parameter's buffer.

Return Values

S_OK
The extension object recognized the *cmdid* parameter and returned text in the *lpsz* parameter.

S_FALSE
The extension object did not recognize the *cmdid* parameter. Microsoft Exchange will continue to call extension objects and if necessary provide text itself.

Comments

Microsoft Exchange calls the **IExchExtCommands::QueryHelpText** method when the user chooses a menu command or selects and holds the mouse down over a toolbar button. The extension object can provide Help text for any commands it implements or supersede Microsoft Exchange Help text for an existing command.

If an error is returned, Microsoft Exchange will not call other extensions and will use a default or blank string.

IExchExtCommands::ResetToolbar

Enables an extension to restore its toolbar buttons to their default positions.

Syntax

HRESULT ResetToolbar(**UINT** *tbid*, **ULONG** *ulFlags*)

Parameters

tbid
: Input-output parameter indicating the toolbar identifier. Currently, only the EETB_STANDARD toolbar is defined.

ulFlags
: Reserved; must be zero.

Return Values

S_OK
: No error occurred.

Comments

Microsoft Exchange calls the **IExchExtCommands::ResetToolbar** method when the user chooses the Reset button from the Customize Toolbar dialog box, which means that the toolbar has been reset to its default state. The extension object should place its default toolbar buttons into their default locations.

See Also

IExchExtCallback::GetToolbar method

IExchExtMessageEvents : IUnknown

The **IExchExtMessageEvents** interface is implemented by extension objects and gives them the ability to replace or enhance the default message-handling behavior of Microsoft Exchange. Microsoft Exchange always calls these methods in pairs. For example, if it calls the **IExchExtMessageEvents::OnRead** method, it also calls the **IExchExtMessageEvents::OnReadComplete** method; if it calls the **IExchExtMessageEvents::OnWrite** method, it also calls the **IExchExtMessageEvents::OnWriteComplete** method, and so forth. The **OnReadComplete**, **OnWriteComplete**, **IExchExtMessageEvents::OnSubmitComplete**, and **IExchExtMessageEvents::OnCheckNamesComplete** methods are used to undo operations whenever an error occurs. They are also used to release resources that are allocated during calls to the **OnRead**, **OnWrite**, **IExchExtMessageEvents::OnSubmit**, and **IExchExtMessageEvents::OnCheckNames** methods. For example, an extension object can allocate buffers to validate text during a call to the **OnRead** method, and deallocate these buffers during a call to the **OnReadComplete** method.

There is a specific set of steps followed by Microsoft Exhange to handle errors during a call to these functions. This is best illustrated with a specific example that applies to all other methods in this interface. Suppose Microsoft Exchange calls **OnCheckNames** to notify extensions that it is about to check names and give extensions an opportunity to resolve names. If no extensions return S_OK, Microsoft Exchange performs the name resolution itself. After the resolution is completed or if it failed, Microsoft Exchange calls the **OnCheckNamesComplete** method of all the extensions whose **OnCheckNames** method it called, passing in EEME_FAILED if the operation failed. If any extension returns an error from **OnCheckNamesComplete**, Microsoft Exchange calls each of those extensions again, passing in EEME_COMPLETE_FAILED as the value for the *ulFlags* parameter. The same series of steps applies to other methods in this interface.

IExchExtMessageEvents is used only in the following contexts:
EECONTEXT_SENDNOTEMESSAGE, EECONTEXT_READNOTEMESSAGE,
EECONTEXT_SENDPOSTMESSAGE, EECONTEXT_READPOSTMESSAGE,
EECONTEXT_READREPORTMESSAGE, EECONTEXT_SENDRESENDMESSAGE

This interface is optional.

At a Glance

Specified in header file:	EXCHEXT.H
Object that supplies this interface:	Extension object
Corresponding pointer type:	LPEXCHEXTMESSAGEEVENTS
Implemented by:	Extension objects
Called by:	Microsoft Exchange

Vtable Order

OnRead	Replaces or enhances the behavior of Microsoft Exchange when reading information from a message.
OnReadComplete	Enables extension objects to roll back their implementation of the **OnRead** method in case of an error or to release resources allocated by **OnRead**.
OnWrite	Replaces or enhances the behavior of Microsoft Exchange when writing information to a

	message.
OnWriteComplete	Enables extension objects to roll back their implementation of the **OnWrite** method in case of an error or to release resources allocated by **OnWrite**.
OnCheckNames	Replaces or enhances the behavior of Microsoft Exchange when recipient names typed by the user are being resolved to their address book entries.
OnCheckNamesComplete	Enables extension objects to roll back their implementation of the **OnCheckNames** method in case of an error or to release resources allocated by **OnCheckNames**.
OnSubmit	Replaces or enhances the behavior of Microsoft Exchange when a message has been submitted.
OnSubmitComplete	Enables extension objects to roll back their implementation of the **OnSubmit** method in case of an error or to release resources allocated by **OnSubmit**.

IExchExtMessageEvents::OnCheckNames

Replaces or enhances the behavior of Microsoft Exchange when recipient names typed by the user are being resolved to their address book entries.

Syntax

HRESULT OnCheckNames(**LPEXCHEXTCALLBACK** *lpeecb*)

Parameters

lpeecb
Input parameter pointing to the **IExchExtCallback** interface. The *lpeecb* parameter is used to determine information about the message and the window containing the unresolved names.

Return Values

S_OK
The extension object replaced Microsoft Exchange behavior and handled the name resolution on its own. Microsoft Exchange will consider the task handled.

S_FALSE
The extension object did nothing or added additional behavior. See the text provided in the definition of the **IExchExtMessageEvents** interface in this reference.

Comments

Microsoft Exchange calls the **IExchExtMessageEvents::OnCheckNames** method when it is about to resolve unresolved names.

If an error occurs, **OnCheckNames** should display an error message and return an error. Microsoft Exchange will not continue to check names, nor will it display an error message, but will stop the user action that caused the check names operation. This method should call the **IExchExtCallback::GetObject** method to determine the message identifier of the item being displayed and its container. The container is usually a folder, but it can also be a message or an information store.

See Also

IExchExtCallback::GetObject method, **IExchExtMessageEvents : IUnknown** interface

IExchExtMessageEvents::OnCheckNamesComplete

Enables extension objects to roll back their implementation of the **IExchExtMessageEvents::OnCheckNames** method in case of an error or to release resources allocated by **OnCheckNames**.

Syntax

HRESULT OnCheckNamesComplete(**LPEXCHEXTCALLBACK** *lpeecb*, **ULONG** *ulFlags*)

Parameters

lpeecb
: Input parameter pointing to the **IExchExtCallback** interface. The *lpeecb* parameter is used to determine information about the message and the window containing the unresolved names.

ulFlags
: Input parameter containing a bitmask of flags. The following flags can be set:

  EEME_FAILED
  : Indicates that the check names operation could not be completed successfully. See the text provided in the definition of the **IExchExtMessageEvents** interface in this reference.

  EEME_COMPLETE_FAILED
  : See the text provided in the definition of the **IExchExtMessageEvents** interface in this reference.

Return Values

S_OK
: The extension object handled the event. Microsoft Exchange will consider the task handled.

S_FALSE
: The extension object did nothing or added additional behavior. Microsoft Exchange will continue to call extensions or complete the work itself.

Comments

Microsoft Exchange calls the **IExchExtMessageEvents::OnCheckNamesComplete** method after it has resolved the addresses. This method should call the **IExchExtCallback::GetObject** method to determine the message identifier of the item being displayed and its container. The container is usually a folder, but it can also be a message or an information store.

See Also

IExchExtCallback::GetObject method, **IExchExtMessageEvents : IUnknown** interface

IExchExtMessageEvents::OnRead

Replaces or enhances the behavior of Microsoft Exchange when reading information from a message.

Syntax

HRESULT OnRead(**LPEXCHEXTCALLBACK** *lpeecb*)

Parameters

lpeecb
: Input parameter pointing to the **IExchExtCallback** interface. The *lpeecb* parameter is used to determine information about the message being read and the window whose controls are to be populated.

Return Values

S_OK
: The extension object replaced Microsoft Exchange behavior and populated the window used to display the message. Microsoft Exchange will consider the task handled.

S_FALSE
: The extension object did nothing or added additional behavior. Microsoft Exchange will continue to call extension objects or complete the work itself.

Comments

Microsoft Exchange calls the **IExchExtMessageEvents::OnRead** method when it is about to read the properties of a message so that they can be displayed to the user in a standard form. If an error occurs, **OnRead** should display an error message and return an error. Microsoft Exchange will not continue to read the message, nor will it display an error message, but will stop the user action that caused the read operation to occur.

This method should call the **IExchExtCallback::GetObject** method to determine the message identifier of the item being read and its container. The container is usually a folder, but it can also be a message or an information store.

See Also

IExchExtCallback::GetObject method, **IExchExtMessageEvents : IUnknown** interface

IExchExtMessageEvents::OnReadComplete

Enables extension objects to roll back their implementation of the **IExchExtMessageEvents::OnRead** method in case of an error or to release resources allocated by the **OnRead** method.

Syntax

HRESULT OnReadComplete(**LPEXCHEXTCALLBACK** *lpeecb*, **ULONG** *ulFlags*)

Parameters

lpeecb
: Input parameter pointing to the **IExchExtCallback** interface. The *lpeecb* parameter is used to determine information about the message being read and the window whose controls are to be populated.

ulFlags
: EEME_FAILED
: Indicates that the check names operation could not be completed successfully. See the text provided in the definition of the **IExchExtMessageEvents** interface in this reference.
: EEME_COMPLETE_FAILED
: See the text provided in the definition of the **IExchExtMessageEvents** interface in this reference.

Return Values

S_OK
: The extension object handled the event. Microsoft Exchange will consider the task handled.

S_FALSE
: The extension object did nothing or added additional behavior. Microsoft Exchange will continue to call extensions or complete the work itself.

Comments

Microsoft Exchange calls the **IExchExtMessageEvents::OnReadComplete** method when it has finished reading the properties of a message for display to the user in a standard form.

If an error occurs, **OnReadComplete** should display an error message and return an error. Microsoft Exchange will then call each extension object again with **OnReadComplete** with *ulFlags* = EEME_FAILED to indicate that the read operation failed. Microsoft Exchange will not continue to read the message, nor will it display an error message, but will stop the user action that caused the read operation.

OnReadComplete should call the **IExchExtCallback::GetObject** method to determine the message identifier of the item being read and its container. The container is usually a folder, but it can also be a message or an information store.

See Also

IExchExtCallback::GetObject method, **IExchExtMessageEvents : IUnknown** interface

IExchExtMessageEvents::OnSubmit

Replaces or enhances the behavior of Microsoft Exchange when a message has been submitted.

Syntax

HRESULT OnSubmit(**LPEXCHEXTCALLBACK** *lpeecb*)

Parameters

lpeecb
Input parameter pointing to the **IExchExtCallback** interface. The *lpeecb* parameter is used to determine information about the message being submitted and the window whose controls were saved.

Return Values

S_OK
The extension object replaced Microsoft Exchange behavior and handled the submission on its own. Microsoft Exchange will consider the task handled.

S_FALSE
The extension object did nothing or added additional behavior. Microsoft Exchange will continue to call extensions or complete the work itself.

Comments

Microsoft Exchange calls the **IExchExtMessageEvents::OnSubmit** method when it is about to submit an open message. At this point, the information will have been written to the message.

If an error occurs, **OnSubmit** should display an error message and return an error. Microsoft Exchange will not submit the message, nor will it display an error message, but will stop the user action that caused the submit.

This method should call the **IExchExtCallback::GetObject** method to determine the message identifier of the item being submitted and its container. The container is usually a folder, but it can also be a message or an information store.

See Also

IExchExtCallback::GetObject method, **IExchExtMessageEvents : IUnknown** interface

IExchExtMessageEvents::OnSubmitComplete

Enables extension objects to roll back their implementation of the **IExchExtMessageEvents::OnSubmit** method in case of an error or to release resources allocated by **OnSubmit**.

Syntax

VOID **OnSubmitComplete**(**LPEXCHEXTCALLBACK** *lpeecb*, **ULONG** *ulFlags*)

Parameters

lpeecb
: Input parameter pointing to the **IExchExtCallback** interface. The *lpeecb* parameter is used to determine information about the message and the window containing the displayed information.

ulFlags
: EEME_FAILED
  : Indicates that the check names operation could not be completed successfully. See the text provided in the definition of the **IExchExtMessageEvents** interface in this reference.

  EEME_COMPLETE_FAILED
  : See the text provided in the definition of the **IExchExtMessageEvents** interface in this reference.

Comments

The **IExchExtMessageEvents::OnSubmitComplete** method is used to enable extension objects to roll back the Microsoft Exchange default implementation of the **IExchExtMessageEvents::OnSubmit** method. **OnSubmitComplete** should call the **IExchExtCallback::GetObject** method to determine the message identifier of the item being submitted and its container. The container is usually a folder, but it can also be a message or an information store.

This method returns VOID because at this point the message should have already been submitted, so it is too late to cancel the operation.

See Also

IExchExtCallback::GetObject method, **IExchExtMessageEvents : IUnknown** interface

IExchExtMessageEvents::OnWrite

Replaces or enhances the behavior of Microsoft Exchange when writing information to a message.

Syntax

HRESULT OnWrite(**LPEXCHEXTCALLBACK** *lpeecb*)

Parameters

lpeecb
: Input parameter pointing to the **IExchExtCallback** interface. The *lpeecb* parameter is used to determine information about the message being written and the window whose controls are to be saved.

Return Values

S_OK
: The extension object replaced Microsoft Exchange behavior and handled writing out the information on its own. Microsoft Exchange will consider the task handled.

S_FALSE
: The extension object did nothing or added additional behavior. Microsoft Exchange will continue to call extension objects or complete the work itself.

Comments

Microsoft Exchange calls the **IExchExtMessageEvents::OnWrite** method when it is about to write the contents of a standard form to the properties of a message. If an error occurs, the extension object should display an error message and return an error. Microsoft Exchange will not continue to write the message, call another extension object's **OnWrite** method, or display an error message. However, it will stop the user action that caused the write operation.

This method should call the **IExchExtCallback::GetObject** method to determine the message identifier of the item being written and its container. The container is usually a folder, but it can also be a message or an information store.

See Also

IExchExtCallback::GetObject method, **IExchExtMessageEvents : IUnknown** interface

IExchExtMessageEvents::OnWriteComplete

Enables extension objects to roll back the Microsoft Exchange default implementation of the **IExchExtMessageEvents::OnWrite** method in case of an error or to release resources allocated by **OnWrite**.

Syntax

HRESULT OnWriteComplete(**LPEXCHEXTCALLBACK** *lpeecb*, **ULONG ulFlags**)

Parameters

lpeecb
: Input parameter pointing to the **IExchExtCallback** interface. The *lpeecb* parameter is used to determine information about the message that was written and the window whose controls are to be saved.

ulFlags
: EEME_FAILED
: Indicates that the check names operation could not be completed successfully. See the text provided in the definition of the **IExchExtMessageEvents** interface in this reference.
: EEME_COMPLETE_FAILED
: See the text provided in the definition of the **IExchExtMessageEvents** interface in this reference.

Return Values

S_OK
: The extension object successfully rolled back the write operation and freed resources.

S_FALSE
: The extension object did nothing or added additional behavior. Microsoft Exchange will continue to call extensions or complete the work itself.

Comments

Microsoft Exchange calls the **IExchExtMessageEvents::OnWriteComplete** method after it has finished writing the contents of a standard form to the properties of a message. If an error occurs, the extension object should display an error message and return an error. Microsoft Exchange will then call each extension object again with **OnWriteComplete** with *ulFlags* = EEME_FAILED to indicate that the write operation failed.

The extension object should free resources used for the write operation.

This method should call the **IExchExtCallback::GetObject** method to determine the message identifier of the item being written and its container. The container is usually a folder, but it can also be a message or an information store.

See Also

IExchExtCallback::GetObject method, **IExchExtMessageEvents : IUnknown** interface

IExchExtModeless : IUnknown

The **IExchExtModeless** interface enables modeless extensions to coordinate their actions with Microsoft Exchange.

IExchExtModeless should not be implemented by extensions as another interface on their main extension interface, but should instead be implemented on a separate object with its own independent set of reference counts.

This interface is optional.

At a Glance

Specified in header file:	EXCHEXT.H
Object that supplies this interface:	Extension object
Corresponding pointer type:	LPEXCHEXTMODELESS
Implemented by:	Extension objects
Called by:	Microsoft Exchange

Vtable Order

TranslateAccelerator	Enables an extension object to intercept messages before they are processed by the Microsoft Exchange client.
EnableModeless	Notifies an extension object to disable its modeless windows.

IExchExtModeless::EnableModeless

Enables extension objects to enable or disable their modeless windows.

Syntax

HRESULT EnableModeless(**HWND** *hwnd*, **BOOL** *fEnable*)

Parameters

hwnd
Input parameter pointing to the handle of the parent window of the modal window being displayed or removed.

fEnable
Input parameter set to TRUE when Microsoft Exchange should enable its modeless windows (that is, when the extension has removed a modal window). Set to FALSE when Microsoft Exchange should disable its modeless windows (that is, when the extension is about to display a modal window).

Return Values

S_OK
No error occurred.

Comments

Microsoft Exchange uses the **IExchExtModeless::EnableModeless** method to notify extension objects that they should enable or disable their modeless windows. If an extension has registered for modeless behavior using the **IExchExtCallback::RegisterModeless** method and has called the **IExchExtModelessCallBack::AddWindow** method, Microsoft Exchange calls the extension's **EnableModeless** method when it displays or removes a modal window. In the case of creation, the extension object should disable its modeless windows; in the case of removal, the extension object should enable its modeless windows.

IExchExtModeless::TranslateAccelerator

Enables modeless extensions to intercept and handle messages.

Syntax

HRESULT TranslateAccelerator(**LPMSG** *pmsg*)

Parameters

pmsg
: An input parameter pointing to the message.

Return Values

S_OK
: The message was handled by the extension object.

S_FALSE
: The message was not handled by the extension object.

Comments

The **IExchExtModeless::TranslateAccelerator** method is used to enable modeless extension objects to intercept and handle messages. While running in-process, an extension object might need to intercept and handle various messages that would otherwise be handled by Microsoft Exchange. If the extension object has registered for modeless behavior using the **IExchExtCallback::RegisterModeless** method and has called the **IExchExtModelessCallBack::AddWindow** method, the Microsoft Exchange client will call **TranslateAccelerator** whenever it receives a Windows message. If the extension object handles the message, it should return S_OK to prevent Microsoft Exchange from handling the same message.

IExchExtModelessCallback : IUnknown

The **IExchExtModelessCallback** interface is implemented by the Microsoft Exchange client to coordinate with extension objects using a modeless user interface.

At a Glance

Specified in header file:	EXCHEXT.H
Object that supplies this interface:	Extension object
Corresponding pointer type:	LPEXCHEXTMODELESSCALLBACK
Implemented by:	Microsoft Exchange
Called by:	Extension objects

Vtable Order

EnableModeless	Enables the Microsoft Exchange client to enable or disable its modeless windows.
AddWindow	Enables Microsoft Exchange to keep track of modeless windows displayed by extension objects.
ReleaseWindow	Enables the Microsoft Exchange client to keep track of modeless windows displayed by extension objects.

IExchExtModelessCallback::AddWindow

Enables Microsoft Exchange to keep track of modeless windows that are displayed by extensions.

Syntax

HRESULT AddWindow()

Return Values

S_OK
    No error occurred.

Comments

Extension objects should call the **IExchExtModelessCallback::AddWindow** method when they display a modeless window. This method, along with the **IExchExtModelessCallback::ReleaseWindow** method, enables Microsoft Exchange to keep track of modeless windows that are displayed by extensions. Microsoft Exchange will not terminate until all extension windows have been closed. It is not necessary to call this method multiple times when multiple modeless windows are displayed.

To enable users to exit Microsoft Exchange with one command, modeless extensions should intercept the Microsoft Exchange Exit and Exit and Logoff commands on the File menu. By doing so, extensions can prompt the user to save any changes that have been made and then close the user's modeless windows, returning S_FALSE to enable other extensions to close their windows as well. In this way, extensions can cooperate with Microsoft Exchange when a user wants to close all windows and exit.

See Also

IExchExtModelessCallback::ReleaseWindow method

IExchExtModelessCallback::EnableModeless

Extension objects should call this method just before they display or remove a modal dialog box, allowing the Microsoft Exchange client to enable or disable its modeless windows.

Syntax

HRESULT EnableModeless(**HWND** *hwnd*, **BOOL** *fEnable*)

Parameters

hwnd
Input parameter containing the handle of the parent window of the modal window being displayed or removed.

fEnable
Input parameter. Set to FALSE when your extension is going to display a modal window. Set to TRUE when your extension has removed its modal window.

Return Values

S_OK
No error occurred.

Comments

If an extension object calls the **IExchExtModelessCallback::EnableModeless** method when it creates or removes a modal window, the Microsoft Exchange client will disable or enable its modeless windows as appropriate.

IExchExtModelessCallback::ReleaseWindow

Enables the Microsoft Exchange client to be notified when an extension object's modeless windows have been removed.

Syntax

HRESULT ReleaseWindow()

Return Values

S_OK
    No error occurred.

Comments

Extension objects should call the **IExchExtModelessCallback::ReleaseWindow** method when they release all their modeless windows. This method, along with the **IExchExtModelessCallback::AddWindow** method, enables the Microsoft Exchange client to keep track of modeless windows displayed by extensions. Microsoft Exchange cannot be closed until all extension windows have been closed.

Extensions are expected to make an equal number of calls to **AddWindow** and **ReleaseWindow**. For example, if an extension calls **AddWindow** for every modeless window it displays, it should call **ReleaseWindow** for each modeless window it closes. Similarly, if an extension calls **AddWindow** only once for its modeless windows, it should call **ReleaseWindow** only once when the last of the extension's modeless windows is closed.

To enable users to exit Microsoft Exchange with one command, modeless extensions should intercept the Exit and Exit and Logoff commands on the Microsoft Exchange File menu. By doing so, extensions can prompt the user to save any changes that have been made and then close the user's modeless windows, returning S_FALSE to enable other extensions to close their windows as well. In this way, extensions can cooperate with Microsoft Exchange when a user wants to close all windows and exit.

See Also

IExchExtModelessCallback::AddWindow method

IExchExtPropertySheets : IUnknown

The **IExchExtPropertySheets** interface is used to enable extension objects to append pages to Microsoft Exchange property sheets. **IExchExtPropertySheets** is used only in the EECONTEXT_PROPERTYSHEETS context.

For more information on property sheets, see the Win32 documentation.

This interface is optional.

At a Glance

Specified in header file:	EXCHEXT.H
Object that supplies this interface:	Extension object
Corresponding pointer type:	LPEXCHEXTPROPERTYSHEETS
Implemented by:	Extension objects
Called by:	Microsoft Exchange

Vtable Order

GetMaxPageCount	Returns the maximum number of pages an extension will add to the property sheet.
GetPages	Adds property sheet pages to the current list of pages.
FreePages	Frees any resources allocated by the **GetPages** method.

IExchExtPropertySheets::FreePages

Frees all resources allocated by the **IExchExtPropertSheets::GetPages** method.

Syntax

VOID **FreePages**(**LPPROPSHEETPAGE** *lppsp*, **ULONG** *ulFlags*, **ULONG** *cpsp*)

Parameters

lppsp
: Input parameter pointing to a variable where the pointer to the first page of a list of property sheet pages previously added is stored. The extension object should not free the memory associated with the *lppsp* parameter because this parameter points to an element of an array allocated by Microsoft Exchange.

cpsp
: Input parameter containing the number of property sheet pages added to the *lppsp* parameter by the extension object.

ulFlags
: Input parameter containing a bitmask of flags used to indicate the type of property sheet whose pages are being freed. The following flags can be set:

  EEPS_MESSAGE
  : Indicates a message's property sheet.

  EEPS_FOLDER
  : Indicates a folder's property sheet.

  EEPS_STORE
  : Indicates a store's property sheet.

  EEPS_TOOLSOPTIONS
  : Indicates the Tools.Options property sheet.

Comments

Microsoft Exchange calls the **IExchExtPropertySheets::FreePages** method when the user closes the properties dialog box and Microsoft Exchange prompts the extension objects to free their pages and other associated resources and interfaces.

IExchExtPropertySheets::GetMaxPageCount

Returns a value for the maximum number of pages an extension object will add to the property sheet.

Syntax

ULONG **GetMaxPageCount**(**ULONG** *ulFlags*)

Parameters

ulFlags
Input parameter containing a bitmask of flags used to indicate the type of property sheet being displayed. The following flags can be set:

EEPS_MESSAGE
Indicates a message's property sheet.

EEPS_FOLDER
Indicates a folder's property sheet.

EEPS_STORE
Indicates a store's property sheet.

EEPS_TOOLSOPTIONS
Indicates the Tools.Options property sheet.

Comments

Microsoft Exchange calls the **IExchExtPropertySheets::GetMaxPageCount** method when it is allocating memory for the property sheet page array. The value returned by this method represents the maximum number of pages to be shown at any one time not the sum of all property sheet pages for all objects.

IExchExtPropertySheets::GetPages

Adds property sheet pages to the current list of pages.

Syntax

HRESULT GetPages(**LPEXCHEXTCALLBACK** *lpeecb*, **ULONG** *ulFlags*, **LPPROPSHEETPAGE** *lppsp*, **ULONG FAR *** *lpcpsp*)

Parameters

lpeecb
Input parameter pointing to the **IExchExtCallback** interface.

ulFlags
Input parameter containing a bitmask of flags used to indicate the type of property sheet being displayed. The following flags can be set:

EEPS_MESSAGE
Indicates a message's property sheet.

EEPS_FOLDER
Indicates a folder's property sheet.

EEPS_STORE
Indicates a store's property sheet.

EEPS_TOOLSOPTIONS
Indicates the Tools.Options property sheet.

lppsp
Output parameter pointing to a variable that stores the pointer to a list of property sheet pages the extension will fill.

lpcpsp
Output parameter pointing to a variable containing the number of property sheet pages actually filled in by the extension object. The number must be less than or equal to the maximum requested by the **IExchExtPropertySheets::GetMaxPageCount** method.

Return Values

S_OK
No error occurred.

Comments

Microsoft Exchange calls the **IExchExtPropertySheets::GetPages** method when it is building a property sheet and needs each extension object to fill in any pages that will be appended. Microsoft Exchange, not the extension object, handles the allocation of the page array pointed to by the *lppsp* parameter. Microsoft Exchange allocates this array after it calls the extension object and gets the value returned by the **IExchExtPropertySheets::GetMaxPageCount** method. The extension object should not allocate its own pages in the *lppsp* parameter, nor should it free them when the **IExchExtPropertySheets::FreePages** method is called. Instead, it should free the additional resources added during the call to **GetPages**.

The Windows 95 property sheet structures are used to describe the pages.

IExchExtSessionEvents : IUnknown

The **IExchExtSessionEvents** interface is used to enable an extension object to respond to the arrival of new messages. This interface can be used to create rules-based inbox processing and other custom handling of new messages. **IExchExtSessionEvents** is used only in the EECONTEXT_SESSION context.

This interface is optional.

At a Glance

Specified in header file:	EXCHEXT.H
Object that supplies this interface:	Extension object
Corresponding pointer type:	LPEXCHEXTSESSIONEVENTS
Implemented by:	Extension objects
Called by:	Microsoft Exchange

Vtable Order

OnDelivery	Replaces or enhances the behavior of Microsoft Exchange when a new message arrives.

IExchExtSessionEvents::OnDelivery

Replaces or enhances the behavior of Microsoft Exchange when a new message arrives.

Syntax

HRESULT OnDelivery(**LPEXCHEXTCALLBACK** *lpeecb*)

Parameters

lpeecb
: Input parameter pointing to the **IExchExtCallback** interface. The *lpeecb* parameter is used to determine information about the message or messages which have arrived.

Return Values

S_OK
: The extension object replaced Microsoft Exchange default behavior with its own behavior. Microsoft Exchange will consider the task handled.

S_FALSE
: The extension object did nothing or added additional behavior. Microsoft Exchange will continue to call extension objects or complete the work itself.

Comments

The **IExchExtSessionEvents::OnDelivery** method is used to replace or enhance the behavior of Microsoft Exchange when a new message arrives. If an error occurs, **OnDelivery** should display an error message and return an error value. Microsoft Exchange will not notify the user about the new message, nor will it display an error message or continue to call extension objects in response to the new message.

When a new message arrives, the extension object obtains its message identifier by calling the **IExchExtCallback::GetObject** method.

See Also

IExchExtCallback::GetObject method

IExchExtUserEvents : IUnknown

The **IExchExtUserEvents** interface is used to enable an extension object to handle changes to the currently-selected list box item, text, or object. **IExchExtUserEvents** is used in all contexts except EECONTEXT_TASK and EECONTEXT_SESSION.

This interface is optional.

At a Glance

Specified in header file:	EXCHEXT.H
Object that supplies this interface:	Extension object
Corresponding pointer type:	LPEXCHEXTUSEREVENTS
Implemented by:	Extension objects
Called by:	Microsoft Exchange

Vtable Order

OnSelectionChange	Enables the extension object to enable, disable, or update visual elements.
OnObjectChange	Enables the extension object to enable, disable, or update visual elements in an object.

IExchExtUserEvents::OnObjectChange

Enables the extension object to enable, disable, or update visual elements in an object displayed in the window.

Syntax

VOID **OnObjectChange** (**LPEXCHEXTCALLBACK** *lpeecb*)

Parameters

lpeecb
Input parameter pointing to the **IExchExtCallback** interface. The *lpeech* parameter is used to determine information about the selection.

Comments

Microsoft Exchange calls the **IExchExtUserEvents::OnObjectChange** method when the object within the container displayed by the Microsoft Exchange window changes. For example, this method is called when a user changes a folder whose contents are displayed in the Viewer, or when the user moves to the next message in cases where the message's class permits the same window to be used.

OnObjectChange should call the **IExchExtCallback::GetObject** method to determine the identifier of the changed object and its container. The container is usually a folder, but it can also be a message or an information store.

See Also

IExchExtCallback::GetObject method

IExchExtUserEvents::OnSelectionChange

Enables the extension object to enable, disable, or update visual elements.

Syntax

VOID **OnSelectionChange**(**LPEXCHEXTCALLBACK** *lpeecb*)

Parameters

lpeecb
  Input parameter pointing to an **IExchExtCallback** interface that is used to determine information about the currently selected item.

Comments

Microsoft Exchange calls the **IExchExtUserEvents::OnSelectionChange** method when the currently selected message, folder or store changes in a Microsoft Exchange window. This method is also called when a selection in a list box changes or when the current selection of text in a form changes. For example, this method is called when an item is selected in a Viewer list box, or when text is selected in a note form. It is also called when the user changes the currently selected message or folder.

This method might be called very frequently and should therefore be implemented efficiently.

This method should call the **IExchExtCallback::GetObject** method to determine the identifier of the newly selected object and its container.

See Also

IExchExtCallback::GetObject method

Introduction to MAPI Programming

This chapter presents some of the issues that are important to consider before beginning serious development work. The chapter is divided into three sections, starting with information about installing and using the MAPI Software Development Kit (SDK). The next section presents information of interest primarily to client application developers: how to choose between the four client APIs (Extended or Simple MAPI, CMC, and OLE Messaging) and how to create a client application that takes advantage of more than one client API. The last section describes the logon process and how profiles and message services are created and configured.

Installing the MAPI SDK

The MAPI Software Development Kit (SDK) includes tools to help you develop your providers and your message service. 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 Windows 95. Before installing the MAPI SDK, you must verify that the computer contains a Microsoft Windows SDK and either a Microsoft Windows-compatible C or C++ compiler.

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

Using the MAPI Setup Program

The setup program, SETUP.EXE, installs the MAPI SDK. SETUP.EXE decompresses and copies the SDK software from the SDK CD to your hard disk and adds a MAPI program group to your Program Manager window.

1. Insert Disk 1 or the CD-ROM disc into the appropriate drive.
2. Using the Windows File Manager, select the CD drive and double-click the SETUP.EXE file in the root directory. SETUP.EXE will automatically perform the rest of the installation.

SETUP.EXE decompresses and copies the SDK software from the SDK disks or CD-ROM to your local disk and adds a MAPI program group to your Program Manager window.

SETUP.EXE asks whether you want a complete or a custom installation. Selecting a complete installation copies all of the MAPI files to the appropriate directories. Selecting a custom installation allows you to select specific files. SETUP.EXE displays information about each individual component and prompts you 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 each type of client application and service provider, including several different transport providers and a hook provider. There are tools for accessing properties and tables. For configuration, MAPI provides two applications: a Control Panel applet for full configuration support and the Profile Wizard.

The following table lists the MAPI SDK samples and tools by name and provides a brief description:

MAPI SDK Component	**Description**
PROPVU.DLL	Tool for displaying and changing properties.
SAMPLE.AB	Sample address book provider written in C.
SAMPLE.MS	Sample message store provider written in C.
SAMPLE.XP	Sample transport provider written in C.
ABVIEW.DLL	Tool for displaying and changing address book objects.
MDBVIEW.DLL	Tool for displaying and changing message store objects.

Issues Affecting Client Applications

These next two sections apply to developers of new or existing client applications. As a client application developer, you have a choice between one or more sets of APIs. The first section will help you to decide which of the sets is right for your application. The second section discusses how to merge Extended MAPI with another API in your application.

Selecting a Client Programming Interface

Choosing between one of the four client APIs might be an easy decision for some client developers. Maybe you only know one language and you have no time to learn another. Depending on the language, your choice may be automatically narrowed further. Maybe you have a time limit for adding messaging capabilities to your existing application. A time limit for implementation suggests a simple API, such as Simple MAPI or CMC. Maybe you need to write a complex workgroup application. Extended MAPI is the API of choice for complex applications.

Each of these situations are fairly straightforward. In fact, deciding between Extended MAPI and one of the layered APIs should be an easy choice. When your application heavily depends on messaging, use Extended MAPI. Extended MAPI applications have more direct contact with and control over the underlying messaging system services like address books, message stores, and transports. Applications that can benefit from Extended MAPI functionality include shared group applications like schedulers and calendars, work flow and message management applications, electronic mail clients, rules-based inbox managers, or Electronic Data Interchange (EDI) clients.

Simple MAPI and CMC offer programmers a fast, easy way to build basic messaging applications from scratch or add messaging functionality to existing applications. The features provided by the two client interfaces is nearly identical, but CMC is bit 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 of the functions in the CMC and Simple MAPI APIs together with the features that 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 and underlying messaging system hardware. CMC was developed in conjunction with the X.400 Application Programming Interface Association (XAPIA) standards organization and electronic mail vendors and users. Where Simple MAPI works on Windows-based platforms only, CMC is available for Windows, MS-DOS, Macintosh, and Unix systems. Therefore, CMC is a good choice for client applications that need to use multiple messaging systems on multiple platforms. Simple MAPI 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.

The remaining choice, OLE Messaging, is an object oriented API used primarily by Visual Basic and Visual C/C++ client application developers. OLE Messaging provides programmable objects that publish properties and methods which can then be managed by Visual Basic and Visual Basic for Applications programs. OLE Messaging 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 what Extended MAPI offers. OLE Messaging is a good choice for Visual Basic applications or applications that require a moderate amount of messaging support.

To summarize, the following table shows how each of the client APIs differ:

Issues	**CMC**	**Simple MAPI**	**Extended MAPI**	**OLE Messaging**
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	cross-platform

Using Multiple Client Programming Interfaces

Most client applications will use one client API - either OLE Messaging, Simple MAPI, CMC or Extended 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 Extended 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. Extended MAPI uses a structure called an entry identifier; OLE Messaging and Simple MAPI use a string called a message identifier. Message identifiers can be simple or compound.

The following API functions are available for translating between the different identifiers used by the different client API sets:

- HrSzFromEntryID
- HrComposeMessageID
- HrEntryIDFromSz
- HrComposeEID
- HrDecomposeEID
- HrComposeMsgID
- HrDecomposeMsgID

HrSzFromEntryID turns an entry identifier into a string that can be used with either OLE Messaging or Simple MAPI. The **HrComposeMessageID** function, provides the same functionality in a more general way.

HrEntryIDFromSz translates a string into an entry identifier. The string may an either a simple or compound OLE Messaging or Simple MAPI identifier. If a compound message identifier is to be converted, the **HrComposeEID** function works better than **HrEntryIDFromSz**. **HrEntryIDFromSz** will succeed, but subsequent uses of the resulting entry identifier will fail.

For translating between CMC and Extended MAPI identifiers, use **HrComposeEID** and **HrDecomposeEID**. These functions can handle binary message identifiers that require string versions. Use **HrDecomposeEID** with care; it is an expensive call.

The **HrComposeMsgID** and **HrDecomposeMsgID** functions translate between the entry identifier that is created by **HrComposeEID** and a string suitable for OLE Messaging and Simple MAPI. These functions require an Extended MAPI session handle. Therefore, if your application was not started with the Extended MAPI logon API function, **MAPILogonEx**, you will need to translate your current session handle into an Extended MAPI session handle.

It is possible to convert CMC and Simple MAPI sessions into Extended MAPI sessions by calling one of two API functions: **ScMAPIXFromCMC** or **ScMAPIXFromSMAPI.** To convert CMC sessions into Extended MAPI sessions, use **ScMAPIXFromCMC**. To convert Simple MAPI sessions, use **ScMAPIXFromSMAPI**. Both functions take a your current session handle as input and return a pointer to an Extended MAPI session object. Session objects are discussed in more detail later in the chapter called "Using Session Objects." Note that there is no way to convert an Extended MAPI session into a CMC or Simple MAPI session. There is also no way to convert a CMC session into a Simple MAPI session.

When you are using multiple client APIs, exercise caution when 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 functions return unsigned long values while Extended MAPI functions and methods return data structures known as HRESULTs. Both values are based on a numeric code which, in many cases, is the same for both caller types. However, there are a few cases where they are different. The following table lists the differences; refer to the section on error handling in the "Introduction to Extended MAPI Programming" and the *MAPI Programmer's Reference* for more information.

Simple MAPI Return Value	**Extended 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

About Message Services and Profiles

Some users require the services of several messaging systems, each with three or more service providers. Because it is cumbersome to have to install and configure each of these service providers individually, MAPI has created 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 single setup program to install the related service providers and write to MAPI's configuration file, MAPISVC.INF, and a message service entry point program to handle the configuration of each provider.

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 may 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 whose goal is to make adding a message service to a profile as easy as possible. The Profile Wizard consists of a series of dialogs, 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 an unique identifier known as a MAPIUID. The MAPI sections contain information internal to MAPI, such 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.

Figure 1 below illustrates 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, group10830 0 /a "SDKEX.bmp"}

Figure 1   Relating Profiles to Message Services and Service Providers.

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

Figure 2 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 file. 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, group10830 1 /a "SDKEX.bmp"}

Figure 2   Message Services, the Profile, and Service Providers.

Establishing a Session

Before any calls can be made to an underlying messaging system, client applications 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 credentials, and insuring that all of the profile's message services are properly configured.

Some of the message services in the profile may require credentials, such as passwords, to be entered. Passwords can either be saved in the profile or entered by the user with every logon. Passwords are optional features in MAPI. Whether or not they are required and how they are implemented depends on the messaging system and the platform. Some platforms support security based on user operating system accounts; others do not.

MAPI also supports unified logon, in which a single initial logon to the messaging system gains a user 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.

The following table lists the MAPI platforms 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
MS-DOS	no
Macintosh	no

One of the most important parts of the logon process is determining if all of the message services in the profile are properly configured. The profile is the initial source for configuration information. If information for a particular message service is missing, the logon process tries 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. Another cause for failure is when the user cancels this user interface before entering the needed information.

When a logon process fails once, the user is informed 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 necessary to complete the process. If this second attempt proves unsuccessful, MAPI disables all providers in that message service for the duration of the session. In effect, the whole service is disabled. This is done because if one provider fails logon, typically other providers will also fail.

Sometimes the logon process will 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. Either an individual session or a shared session can be established. Individual sessions are private connections; there is a one-to-one relationship between a client application and the profile it is using. A shared session allows multiple client applications on a computer workstation to work from the same profile. Shared sessions are established once but can be "used" by other client applications who ask to use them. The profile and credentials are specified only with the initial logon.

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 may be required to enforce a session limit.

To summarize, this flowchart illustrates how the logon process works with the profile to establish a session.

{ewc msdncd, EWGraphic, group10830 2 /a "SDKEX_09.bmp"}

Figure 3   Flow of logon process.

Platform Issues

When using the MAPI SDK to create a client application or a service provider, there are platform issues to keep in mind. The degree of interoperability between clients and message services depends on the platforms the clients and message services used for development. Also, the platform that holds the MAPI SDK installation is also a factor.

Message services support clients that are written for one of the following platforms:

- Windows NT (32-bit)
- Windows 95 (32-bit)
- Win16 (16-bit compatibility mode running on Windows NT or Windows 95)

Selecting a development platform affects writers of both message services and clients. Message service writers should develop in an environment so that they will be able to be used by a wide variety of clients. Client implementors should choose an environment that will allow them to use the message services they want.

If you have installed the MAPI SDK under Windows NT, you will be able to create message services that will run on Windows NT, Windows 95, or in a 16-bit compatibility mode environment. Client written in either the Windows NT or 16-bit environments will be able to access your service.

If you have installed the MAPI SDK under Windows 95, your future message service will run on either Windows 95 or in 16-bit compatibility mode. Client written in either the Windows 95 or 16-bit environments will be able to access your service.

Clients written for either of the 32-bit platforms or for the 16-bit compatibility mode environment can access two types of message services: those written for the 32-bit platforms and single source services. Single source services are built with a single set of source files from which binaries can be built to run on any of the platforms: Windows NT, Windows 95, or Win16.

If you target all three platforms using the single-source approach, you can take advantage of 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, you can write your code as if you were writing with the standard Win32 library of API functions. However, you compile 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. Refer to the MAPI Programmer's Reference for complete descriptions of the MAPI Win32 functions.

If you are planning to write for either Windows NT or Windows 95, it makes sense to write a message service that will operate on both 32 bit platforms, thereby increasing its usability. Targeting both 32 bit platforms is easy to do because the additional code required for the second platform is minimal.

If you have installed the MAPI SDK in the 16-bit environment, your message service will only be able to run in that environment. You will only be able to support the 16 bit environment on Windows 3.1+ because MAPI does not support the 32-bit subset, Win32s.

About This Book

The MAPI Programmer's Guide is one volume in the set of documentation that accompanies the MAPI Software Development Kit (SDK). The MAPI SDK documentation will be included with the BackOffice Software Development Kit (SDK) documentation.

Intended Audience

The MAPI Programmer's Guide is written for C and C++ developers with a wide range of needs and experience. The MAPI Programmer's Guide targets developers with limited messaging knowledge as well as developers of workgroup applications or specialized messaging system services. Because the intended audience is so varied, some chapters may apply to some developers and not others. An attempt was made to organize the book in such a way as to make it easy for readers to locate relevant material and avoid what was not applicable.

For developers intending to write basic messaging applications or enhance an application with messaging functionality, no prerequisite knowledge is necessary. For developers intending to create full-scale workgroup applications or specialized messaging system services, a background in messaging and a familiarity with the Component Object Model (COM) is recommended.

How This Book is Organized

The MAPI Programmer's Guide is organized in chapters, beginning with a discussion of key concepts and architecture. Chapter 2 is an introduction to MAPI programming, including information that is applicable to developers of client applications and service providers. The next two chapters are tutorials for application developers writing with the Simple MAPI and CMC set of API functions. Chapters 5, 6, and 7 describe the basic building blocks of the Extended MAPI environment: objects, interfaces, and properties. Chapter 8 is an introduction to programming with Extended MAPI, presenting topics that apply to all Extended MAPI developers. The next several chapters are devoted to using specific objects. Each chapter contains a description of the interfaces and properties the object supports and information about how the object is typically used.

Chapter 16 teaches developers how to program Extended MAPI client applications. After this chapter, the material focuses on service providers. There is one chapter that applies to all developers of service providers and then individual chapters for each specific type of service provider. The last chapter discusses how to build custom forms.

Document Conventions

This manual uses the following typographic conventions:

Convention	Description
`monospaced text`	Examples and program code appear in a non proportional typeface.
Bold text	Literal expressions that must be typed exactly as shown, such as function names, structure names, and keywords, appear in this bold typeface. Visual Basic custom control error messages appear in bold typeface.
Italic text	Italic text indicates new terms and variable expressions, such as parameters, where you fill in the actual value.
CAPITAL LETTERS	Filenames, directory names, paths, and MAPI function flags and error messages appear in capital letters.
( )	In syntax statements, parentheses enclose one or more parameters that you pass to a function.
[ ]	Brackets enclose optional syntax items. Type only the syntax within the brackets, not the brackets themselves.
	The line-continuation character indicates that code continued from one line to the next should be typed all on one line. For example: `SharedExtensionsDir= \` `\SERVER1\SHARE1\MAILEXTS`

For More Information

For more information about OLE 2.0 programming, see *Inside OLE 2*, by Kraig Brockschmidt, and the *OLE 2.0 Programmer's Reference, Volumes 1 and 2,* published by Microsoft Press. For more information about Visual Basic programming, see your Microsoft Visual Basic documentation.

Programming with CMC

This chapter contains conceptual information for programming with the CMC application programming interface (API). You should understand the material in this chapter before writing applications that use the CMC APIs. This chapter covers concepts needed for sending and receiving messages as well as dealing with addresses and message attachments.

About CMC

The Common Messaging Calls (CMC) client interface is a set of ten functions that enables developers to add simple messaging capabilities to their applications quickly. For example, an application 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 MAPI advantage of messaging service independence. The CMC API is especially valuable because it is also independent of the operating system and the underlying hardware used by the messaging service, 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 the client application must run on multiple platforms and provide simple messaging services on each of these platforms.

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 may not be. In this document, "CMC" and "CMC implementation" should be taken to mean "Microsoft's implementation of the CMC API using MAPI APIs."

The CMC API was developed in conjunction with the X.400 API Association (XAPIA) standards organization and with 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, an application written to the CMC standard can be ported to other platforms. In contrast, MAPI is a Windows-only standard.

The CMC API works as a layer between a messaging-enabled application and a messaging service. The messaging service in turn can support multiple messaging protocol services, 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 protocol services. However, the API does enable developers to use extensions to invoke protocol-specific functions. For more information, see Using CMC Extensions.A directory, a submission queue, and a receiving mailbox are the three messaging service components in the CMC API model.

{ewc msdncd, EWGraphic, group10822 0 /a "SDKEX_08.bmp"}

Using the directory, the messaging-enabled application can look up information about users of connected messaging services and can resolve users' names to actual addresses. Some services may 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 service; 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 service 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, the application can retrieve summaries of the contents of a mailbox, along with identifiers for the particular messages summarized. The application can use these identifiers to select and retrieve individual messages.

CMC Programming Basics

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

- To send a message, a messaging-enabled application

1. Establishes a session with the messaging service either through the **cmc_logon** function or interactively by sending the CMC_LOGON_UI_ALLOWED flag value with the **cmc_send** function.
2. Submits a message to the submission queue. Usually, a messaging-enabled application does so through a **cmc_send** function. If a messaging-enabled application uses **cmc_send**, it must first create a **CMC_message** structure to pass to the **cmc_send** function. The messaging-enabled application can also use 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. Closes the session by using the **cmc_logoff** function.

- To retrieve a message, a messaging-enabled application

1. Establishes a session by using the **cmc_logon** function.
2. Retrieves a summary of mailbox information by using the **cmc_list** function.
3. Retrieves an individual message by using the **cmc_read** function.
4. Optionally, enables a user to act on the message in the mailbox (for example, delete it or move it to another folder) using the **cmc_act_on** function.
5. Releases memory allocated by CMC by using the **cmc_free** function.
6. Closes the session by using the **cmc_logoff** function.

- To look up names and address information in the directory, a messaging-enabled application

1. Establishes 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. Translates a user's display name (what the user sees) into a messaging address (what the underlying message system uses) by using **cmc_look_up**. With this function, the application can also request that the standard CMC addressing dialog box be used to view recipient-specific details or create addressing lists.
3. Releases memory allocated by CMC by using the **cmc_free** function.
4. Closes the session by using the **cmc_logoff** function.

Note that these tasks do not have to be performed in isolation from each other. For example, a messaging-enabled 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, the application can make many CMC calls to perform messaging tasks such as send messages, receive them, make directory queries.

The CMC API uses a fixed set of API functions, data structures, and data types. The *API functions* are the functions that your application calls to carry out messaging tasks. The *data structures* are the groupings of information that your application 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 (Boolean, floating point, and so on) and can have specific operations performed on them.

The CMC API functions listed in alphabetic order:

Action	Function	Description
Sending messages	**cmc_send**	Sends a message.

	cmc_send_documents	Sends a message. This function is string based and is usually used in macro language calls.
Receiving messages	cmc_act_on	Performs an action on a specified message.
	cmc_list	Lists summary information about messages meeting specified criteria.
	cmc_read	Returns a specified message.
Looking up names	cmc_look_up	Looks up addressing information.
Administration	cmc_free	Frees memory allocated by the messaging service.
	cmc_logoff	Terminates a session with the messaging service.
	cmc_logon	Establishes a session with the messaging service.
	cmc_query_configuration	Determines information about the installed CMC service.

The CMC data structures listed in alphabetic order:

Data structure name	**Description**
CMC_attachment	Message attachment structure
CMC_counted_string	String with an explicit length designation
CMC_extension	Extension structure
CMC_message	Message structure
CMC_message_reference	Message-reference structure
CMC_message_summary	Message-summary structure
CMC_object_identifier	Object-identifier structure
CMC_recipient	Originator or recipient structure
CMC_time	Time structure

The CMC data types listed in alphabetic order:

Data type name	**Description**
CMC_boolean	Value indicating logical true or false
CMC_buffer	Pointer to a data item
CMC_enum	Data type containing a value from an enumeration
CMC_flags	Container for flag bit values
CMC_return_code	Return value indicating either that a function succeeded or why it failed
CMC_session_id	Unique identifier for a session
CMC_string	Character-string pointer

CMC_ui_id	User-interface handle

Complete reference information for all CMC API functions, structures, and data types is provided in the CMC API Function Referencetopic.

Starting a CMC Session

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

CMC session identifiers can be interchanged through function calls among CMC, MAPI, Simple MAPI, and Extended MAPI. For example, you can cast the session identifier returned from **cmc_logon** to the LPMAPISESSION value and use it to access the methods in the **IMAPISession** interface. Likewise, you can cast the LPMAPISESSION value returned by the **MAPILogon** function to a **CMC_session_id** (or LHANDLE), used in CMC (or Simple MAPI) function calls. This allows client applications to use CMC for the bulk of their messaging tasks without sacrificing access to the MAPI functions should they be needed.

Client applications can also use the **cmc_query_configuration** function (described following) to determine logon identity and messaging options before calling the **cmc_logon** function.

Addressing Messages

Addressing messages with CMC is a matter of using the **cmc_look_up** function to find addresses that your application can then use with the **cmc_send** function. Using **cmc_look_up**, your application 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 you are 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.

Sending Messages

Sending messages with CMC involves building a **CMC_message** structure and passing it to the **cmc_send** function. Your application fills out different fields in the **CMC_message** depending on the parameters you use to invoke **cmc_send**. For example, if you pass the CMC_LOGON_UI_ALLOWED flag to **cmc_send**, then your application does not need to provide a list of recipients, a subject line, or a message body because CMC will query the user for that information.

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

Receiving Messages

Receiving messages with CMC is slightly more complex than sending messages:

- To receive a message

1. Call the **cmc_list** function to obtain an array of message summaries.
2. Choose a specific message reference from that array and pass it to the **cmc_read** function to retrieve the corresponding message.
3. Call the **cmc_free** function to release the storage allocated for the array when your application no longer needs it.

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

Using CMC Extensions

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

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

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 extensions without obtaining identifiers from the X.400 API Association. If this is done, applications which use those extensions may not be portable to other CMC implementations.

The CMC *common extension set* contains those function and data extensions that are common to most messaging services 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 whose names start with CMC_X_COM. For a full list of common extension declarations, see the CMC API Function Referencetopic

A generic data structure, **CMC_extension**, has been defined as a basis from which to create these extensions. A **CMC_extension** structure consists of an item code identifying the extension, an item data type holding the length of extension data or the data itself, an item reference that points to where the extension value is stored or that is NULL if there is no related item storage, and flag values for the extension. The item code field identifies a particular extension and so determines the meanings of values in the other fields.

The syntax for the **CMC_extension** structure is as follows:

```
typedef struct {
     CMC_uint32  item_code;
     CMC_uint32  item_data;
     CMC_buffer  item_reference;
     CMC_flags   extension_flags;
} CMC_extension
```

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 a messaging application to CMC or as an output parameter from CMC to an application. If an extension is an input parameter, the application in question allocates memory for the extension structure and any other structures associated with the extension. If an extension is an output parameter, CMC allocates the storage for the extension result, if necessary, and the application must free the allocated storage with a call to the **cmc_free** function.

The general procedure for using extensions is:

1. Create a **CMC_extension** structure and fill in the fields in the structure 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 documented 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 fields in the extension structure if the CMC function uses the extension structure to pass values back to your application.

Administering with CMC

Your application needs to perform a small number of administrative tasks when using CMC. These tasks are

- Querying the CMC implementation to determine what features it supports.
- Releasing memory that was allocated by CMC.
- Terminating CMC sessions when the application no longer needs them.

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

In some circumstances, the 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. That memory is typically the result of a CMC function. For example, the **cmc_look_up** function allocates an array of **CMC_recipient** structures to hold the results of an address book search. The application is responsible for calling the **cmc_free** function on that array after it is no longer needed. To free memory allocated by CMC, the application only needs to pass a pointer to the memory to the **cmc_free** function.

The application 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 the application does this. The **cmc_free** function will make all the necessary memory management calls to properly free the memory CMC has allocated.

When your application no longer needs any messaging services, it should use the **cmc_logoff** function to close the CMC session that it obtained with the **cmc_logon** function. Depending on the nature of your application and whether it is possible to establish a CMC session without the user's assistance, it may make sense to close the session as soon as a group of messaging calls are finished. For other applications it may make more sense to establish one CMC session when the application starts and retain it until the application exits.

CMC Programming Examples

This topic 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

The following code example demonstrates how an application logs on and off and uses **cmc_query_configuration** to get system information:

```
/* Local variables used */

CMC_return_code Status;
CMC_boolean     UI_available;
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 */
```

The following code example demonstrates how an application sends 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 */
```

The following code example demonstrates how an 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;

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);
```

The following code example demonstrates how an application looks up a specific recipient and gets 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);
```

The following code example demonstrates how an application specifies 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;    /* Common extensions I need 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;    /* Common extensions I need 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 */
```

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 system 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. Several 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 have different programming interfaces, making it difficult for one application to interact with several of them at once.

To solve this problem, Microsoft, along with more than 100 independent software vendors (ISVs), messaging system providers, 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 yet separate 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. First, MAPI is an open programming interface. An open programming interface provides services in a generic way, allowing its users to add any necessary customization. MAPI is open so that the widest variety of applications and service providers now and in the future can take advantage of this set of common interfaces. MAPI can serve all levels and types of applications, from the word processing application with minimal messaging needs to the complex workgroup application with more demanding needs. In fact, any application that has to either exchange information in a particular format, store information, or retrieve information using a set of criteria or restrictions, can benefit from using the MAPI programming interface.

Second, MAPI provides separation between the programming interface used by the messaging applications and the programming interface used by the back-end messaging system service providers. MAPI has been implemented as the messaging component of an open architecture known as the Microsoft® Windows Open Services Architecture (WOSA). WOSA standards are being developed in the areas of database, directory, security, and messaging technology. The WOSA programming interfaces allow developers to write applications and back-end services with the confidence that these products can be easily connected in a distributed computing environment.

Separating the interfaces enables a single application to use multiple messaging systems using a common, Windows-based user interface. This is a great benefit to users; they can select from a variety of systems, depending on their needs at any one time. For example, the same messaging application can 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 arrive in one universal inbox. Having a single application handling all of these systems not only reduces development costs but also reduces costs for application purchases, user training, and system administration.

Programmers also benefit from this separation because it removes dependencies placed on the application by the messaging system and vice versa. Programmers no longer have to take into account the idiosyncrasies of another component. They focus only on their component, whether it be a messaging system service provider or a messaging application, and MAPI handles the rest, thereby reducing development time and costs.

The third feature in the set involves cross platform support, which allows one version of an application to be run on a variety of machines. MAPI applications can be run on the 16-bit Windows 3.x platforms, the 16 and 32-bit platforms available on Windows 95 and Windows NT, and the Macintosh. Because communication in an organization typically includes fax, host-based messaging services such as DEC All-In-1, voice mail, public communications services such as AT&T EasyLink, CompuServe, MCI MAIL, and other types of services, MAPI supports more than local area network (LAN) - based messaging systems. Drivers can be written for all of these services and can be installed by users according to their needs, providing true independence from specific transport providers.

The fourth feature focuses on the depth and usability of the programming interface. MAPI is aimed at the powerful, new market of workgroup applications - applications that demand more of their messaging systems. To satisfy this market, MAPI offers much more than basic messaging in the programming interface. For example, with MAPI's programming interface, applications can format messages using fonts and color, preprocess messages according to predefined rules, and present customized views of messages and other information.

Last, MAPI is integrated with the operating system. Integration with the Windows family of operating systems enables all developers of Windows-based applications to have access to a consistent and available 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 drivers, or service providers, are installed.

About the MAPI Architecture

MAPI defines a modular architecture, as is shown in the figure below. At one end are the messaging, or client, applications. These applications are known as client applications because they are clients of the MAPI subsystem. There are basically three types of client applications; their differences are discussed in the next section. Underneath the client applications is the MAPI subsystem and related component, the MAPI spooler. The MAPI subsystem is made up of the programming interface and the common user interface. The MAPI spooler is a separate process responsible for sending messages to the appropriate messaging system.

Between the MAPI components and the underlying messaging systems sit service providers. Service providers are the drivers that sit between the world of unique messaging systems and the MAPI standard. Service providers translate requests from MAPI-complaint client applications into tasks a specific messaging system can understand. When a task is complete, the service providers translate again, converting status and information that is messaging system-specific to 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. Service providers are discussed more in later sections.

{ewc msdncd, EWGraphic, group10823 0 /a "SDKEX.bmp"}

Figure 1. MAPI Architecture.

About Client Applications

MAPI client applications are divided into 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 can add a "Send" command to its File menu, enabling the user to send a document to another user.

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, or workflow, 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.

MAPI supports each of these types of client applications by providing in its subsystem different levels of programming interfaces. This multi-level programming interface is described in the following section.

About the MAPI Subsystem

The MAPI subsystem contains two main parts: a part that responds to messaging calls from client applications and a part that provides an implementation of common user interface components. Within the first part is a comprehensive applications programming interface (API) called Extended MAPI. In spite of its name, Extended MAPI is not an extension of another API set. Extended MAPI is a powerful, object-oriented interface, a low level abstraction based on the OLE Component Object Model. Extended MAPI defines a set of objects that share structure and behavior, enabling developers to work with these objects in a consistent manner.

Extended MAPI has a large feature set. Application developers can use Extended MAPI to, for example, provide access to message or recipient properties, customize messages and address books, receive or send event notifications, or search through message databases for messages of a particular type. Although all types of client applications can use Extended MAPI, typically only workgroup applications and service providers need and want its power and complexity. Messaging-aware and messaging-enabled applications can get by on a simpler, more restrictive API.

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

Figure 2 illustrates how Simple MAPI, OLE Messaging, and CMC are layered between Extended MAPI and client applications. Some client applications call directly into Extended MAPI while others call into Simple MAPI, OLE Messaging, or CMC. Calls to these higher level APIs are forwarded to Extended MAPI.

{ewc msdncd, EWGraphic, group10823 1 /a "SDKEX.bmp"}

Figure 2. MAPI's Messaging APIs.

About the MAPI Spooler

The MAPI spooler is a separate process, responsible for sending messages to the appropriate messaging system, that is built with the MAPI subsystem DLL. The MAPI spooler plays a vital role in message receipt and delivery. When the messaging system is unavailable, the 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 spooler runs as a background process, doing much of its work when client applications are idle, thus improving performance.

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

- Keeping track of the recipient types that are handled by specific transport providers.
- Sending new message notifications to client applications.
- Invoking message preprocessing and postprocessing.
- Generating message delivery reports.
- Maintaining status on processed recipients.

About Service Providers

Between the MAPI subsystem and the messaging systems are the various service providers. Service providers are links from 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, address book or directory, and message transport. MAPI supports each type of service independently, allowing a vendor to offer one or 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 would write 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 the Personal Message Store (PST) and an address book provider known as the Personal Address Book (PAB). These providers can be used in isolation, as the only message store or address book providers, or in combination with other service providers.

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. Most messages are created with the standard form that comes with the MAPI subsystem. Users address a message to one or more users or group of users, also called recipients, and enter the message's text, or body. Each recipient might be associated with an installed address book provider or have no known association. This latter type of recipient is known as a custom recipient. Custom recipients might be either temporary entries, lasting only until the message is submitted, or they might be permanent entries if they are saved in the PAB.

When the message is ready for submission, the MAPI subsystem checks to make sure each recipient represents a valid address for one of the installed transport providers. If there is a question, any ambiguous names are assigned to the appropriate address book provider to resolve. The message is placed in the outbound message store queue and the MAPI spooler takes over. The spooler finds an appropriate transport provider to handle the transmission. If the transport is available, the spooler transfers the message and the transport delivers it. If the transport is unavailable, the spooler either holds on to the message until the transport provider becomes available or sends it to another appropriate provider.

The following sections go into more detail about the different types of service providers.

About Address Book Providers

Address book providers handle access to address list and directory information. Address list information is made up of address entries called recipients, which are users who receive messages. A recipient may be an individual messaging user or a group of users who are commonly addressed together, called a distribution list. MAPI provides containers for organizing this information. Depending on the approach of the address book provider, the data can be held in either a single container or multiple containers and the containers can be organized as a flat list or hierarchically.

Although multiple address book providers can be installed simultaneously, MAPI integrates all the information supplied by the 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 will expose 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.

When you use Extended MAPI, you can view the contents of address book containers and, in some cases depending on the address book provider, add new entries and modify existing entries. The PAB is a special address book container because it allows users to store copies of frequently used addresses. The PAB also maintains entries for recipients who are not in the main address book of the underlying messaging system. These custom recipients may be created for the duration of the session only or be stored in the PAB for greater longevity.

Figure 3 shows typical MAPI address book organization.

{ewc msdncd, EWGraphic, group10823 2 /a "SDKEX_04.bmp"}

About Message Store Providers

In a typical messaging environment, users and applications can generate a large number of messages that must be saved and organized in some manner. Each message store provider implements a hierarchical system for saving and organizing messages known as a message store. Like address book providers, message store providers also use MAPI containers for organization. Message stores containers, known as folders, can contain messages and other folders. Folders provide the means for sorting and filtering messages and for customizing their view in the user interface display. Special folders called search-results folders hold links to messages in other folders that satisfy search criteria entered by a user. Search-results folders allow users to look for messages that arrived before the first of May or were sent by John Smith, for example.

MAPI messages contain the text that makes up the body of the message, information that allows the message to be sent, and possibly one or more attachments. The body of the message can be in plain text or formatted, using special fonts, colors, or styles. An attachment is additional data related to and transported with a message in the form of a file, another message, or an OLE object.

Messages that can be stored in the message store include those received from other users or applications as well as those currently being composed by the user. If a message is copied into multiple folders, each copy appears as a separate message and can be copied, deleted, or modified individually. Some message store providers allow messages, once received, to be changed and stored back in their folders. For example, a user could rotate a fax message that arrived upside down and store it right side up for later viewing.

The hierarchical message store architecture is shown in the figure below.

{ewc msdncd, EWGraphic, group10823 3 /a "SDKEX_03.bmp"}

Figure 4. Message Store Hierarchy.

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 pre-processing and post-processing tasks that are necessary. Multiple transport providers may be installed for a single session; there is typically one transport provider for every active messaging system.

Clients communicate with the transport provider via the MAPI spooler. The spooler sends messages to and accepts messages from the transport provider. Incoming messages are delivered to the appropriate message store 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 asked by the client application to flush the transmit and receive queues.

Transport providers register with MAPI to handle one or more particular types of addresses. 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 spooler's first choice is unavailable, and another transport provider has also registered to handle the specific recipient type, the spooler sends the message to the alternate provider. If the unavailable transport 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 the 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 client applications are presented with these dialogs automatically. Extended MAPI clients and service providers have the choice between using the MAPI dialog boxes or implementing their own. Unless there are very customized needs, it makes sense to take advantage of the MAPI user interface. Developers will save time and give their users tools that are consistent across applications and messaging systems.

The MAPI user interface provides many different dialog boxes, some for session start up, some for addressing, some for message composition. The message composition dialog box is used to create interpersonal messages; to create messages of other types you can use special dialog boxes, or forms, that may be provided by the originator of the message type. Implementing and displaying these special forms is discussed in detail in the chapter on building custom forms.

Programming with MAPI

This chapter discusses topics that apply to client and service provider writers using the full set of MAPI interfaces and API functions. The topics are not arranged in any particular order; they are simply tips and information that may be of interest to anyone programming with Extended MAPI.

Using Entry Identifiers

A service provider uses a piece of binary data known as an entry identifier to refer to one of its objects. In the Address Book, for instance, entry identifiers are used to identify messaging users, distribution lists, and containers. To clients and other service providers, entry identifiers are simply binary data. Only the service provider that has assigned the identifier understands what is inside.

Entry identifiers come in two types: *short-term* and *long-term*. Short-term entry identifiers are fast to construct, but their uniqueness is guaranteed only over the life of the current session on the current workstation. Short-term entry identifiers are used for display in tables, where it is necessary to provide data quickly for browsing.

Long term entry identifiers are slower to construct, but they are long lasting. There are two types of long term identifiers: temporaland positional. Entry identifiers that last over time are known as temporal identifiers. Entry identifiers that last across computer workstations are known as positional. The only entry identifiers that are both temporary and positional are the identifiers created for custom recipients.

Validating Data

As implementors and users of interface methods, you 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. This section describes those guidelines for validating both data passed to your methods and data that is returned from methods that you call.

Validating Parameters

When a method that you have implemented is called, there are two types of validation you can perform: debug and full. If you perform debug validation, you display the values of the parameters given to you and provide verbose information should there be errors. If you perform full validation, you call either the **ValidateParameters** or **UlValidateParameters** API function. The level you choose to perform depends on who you expect to call your interface method. When the expected caller is MAPI or a service provider, you can feel safe with debug validation. When the expected caller is a client, full validation is recommended. The model imposed is that within a session, service providers are dependable components and that the parameters passed by these components can be assumed to be valid. Clients are not always as dependable.

MAPI follows these guidelines as well in its interface implementations, providing debug validation in those interface implementations that are called internally or by service providers and full validation in interface implementations that are called by clients. The validation in API functions and the utility interfaces (**ITableData** and **IPropData**) is more lenient; only debug validation is provided.

The two API functions that MAPI provides for full validation implement the same functionality. The function you use in your validation code depends on the return type of the method you are validating. **ValidateParameters** is used for parameters passed to methods that return HRESULT values; **UlValidateParameters** is used for parameters passed to methods that return ULONG values. Both functions examine the validity of read and write pointers, structures, and reserved flags. If there is a size limit on parameters, the incoming size is also inspected.

The parameters to these functions differ slightly depending on whether you are making the calls using C or C++. This is because of the existence of the extra first parameter added to all method calls made in C: a pointer to the current object. In C++, this is the implicit *this* pointer. The first parameter in both languages is a constant that represents the method to be validated; valid values are defined in the MAPIVAL.H header file. The second parameter is the address of the beginning of the parameter list to be validated. If you are making this call in C, this is the address of the object pointer. In C++, this is the first argument.

For example, **IMAPIProp::SetProps** is a method that returns an HRESULT value and requires three parameters: a count of property value structures, a pointer to a property value array, and the address of a pointer to a property problem array. Because this method returns an HRESULT value, **ValidateParameters** is the appropriate function to call. If you are validating these parameters in a C implementation of **SetProps**, your parameter list would contain the extra first parameter and your call to **ValidateParameters** would be made as follows:

```
STDMETHODIMP Message_SetProps(LPMESSAGE lpMessage, ULONG cValues,
     LPSPropValue lpPropArray, LPSPropProblemArray FAR *lppProblems)
{
     /* Validate the Message object */

     /* Validate remaining parameters, using standard MAPI validation. */
     ValidateParameters (IMAPIProp_SetProps, &lpMessage)
}
```

In C++, this example would be written as follows. The parameter list has the standard three parameters beginning with *cValues* which becomes the second parameter passed to **ValidateParameters**.

```
STDMETHODIMP CMessage::SetProps(ULONG cValues,
     LPSPropValue lpPropArray, LPSPropProblemArray FAR *lppProblems)
{
     /* Validate the CMessage object */

     /* Validate remaining parameters, using standard MAPI validation. */
```

```
    ValidateParameters (IMAPIProp_SetProps, cValues)
}
```

The following table lists the interfaces that can be validated by **ValidateParameters** and **UIValidateParameters**. Assume that all of the methods for these interfaces can be validated.

IABContainer	IABLogon
IABProvider	IDistList
IMAPIAdviseSink	IMAPIContainer
IMAPIControl	IMAPIFolder
IMAPIProps	IMAPIStatus
IMAPITable	IMessage
IMsgStore	IMSLogon
IMSProvider	IStream
IXPLogonr	IXPProvider
IUnknown	

Validating Object Pointers

The need to validate object pointers arises when you make calls to retrieve MAPI objects or when you make private method calls 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 that you can choose from to ensure that the object pointers you are given are accurate. 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 non-zero.

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.

Threading in MAPI

MAPI is designed to be thread-safe. To ensure safety with objects that you as a service provider get back, you must follow these rules:

Objects that will be used on a different thread from the one on which they were created must use **CoMarshallInterace** to move between threads.

When using asynchronous notification, wrap your advise sink objects using the API function **HrWrapAdviseSink**. Wrapping your advise sinks in this way guarantees that your advise method will be called on the same thread that was used to call **MAPIInitialize**. This is an optional procedure, but recommended procedure. Avoiding it will cause you to never know on which thread you will be called.

About Event Notification

Event notification is the communication of information between two MAPI objects. One object registers an interest in learning of a change or error, called an event, that may take place in another 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*. MAPI advise sink objects implement the **IMAPIAdviseSink** interface, which contains a single method, **OnNotify**. The advise source notifies the advise sink by calling **OnNotify** and passing a notification data structure that contains information about the event. **OnNotify** can perform tasks in response to the notification, such as updating data in memory or refreshing a screen display.

Typically client applications implement advise sink objects and service providers implement the advise source. Unlike the advise sink, which is a particular type of MAPI object, the advise source may be one of the many different types of MAPI objects that supports event notification. To name a few, advise source objects include sessions, tables, message stores, address book objects, and folders. All advise source objects contain two methods to handle notification registration: **Advise** and **Unadvise**. Clients call **Advise** when they want to register for a notification and **Unadvise** when they want to cancel the registration. **Advise** returns a non-zero connection number to represent the registration.

There are nine types of notifications that can be sent to an advise sink. Some of these are specific to a particular advise source; others are sent by most or all advise source objects. Clients can register for one or more of these types using the constant that MAPI has defined for the specific type. 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 set of events and lists the advise source objects that can support them. The event constant is included with the event type.

Event Type	**Description**	**Supporting Objects**
Critical error (*fnevCriticalError*)	A global error or event has occurred, such as a session shut down in progress.	Logon, 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.	Tables, folders

Status object modified *(fnevStatusObjectModified)*	Information in a MAPI status object has changed.	Status
New mail (*fnevNewMail*)	A message has been delivered and is waiting to be processed.	Message store, folders

Supporting event notification can be complicated, depending on the number of events supported and the number of registered advise sinks. Service providers can handle it themselves or accept help from MAPI. There are three methods in the **IMAPISupport** interface that keep track of all the registrations for a particular advise source and send the appropriate notifications when necessary. To use **IMAPISupport** to manage registration, service providers forward their **Advise** and **Unadvise** calls to the **IMAPISupport::Subscribe** and **IMAPISupport::Unsubscribe** methods**.** When an event occurs, service providers call **IMAPISupport::Notify** to ask MAPI to notify all registered advise sinks.

Figure 1 illustrates the flow of calls between clients, service providers, and MAPI that constitute the event notification process. Dotted lines are used to indicate optional lines of communication.

{ewc msdncd, EWGraphic, group10824 0 /a "SDKEX.bmp"}

Figure 1. Event notification.

When registering for notifications, clients have a few options. For objects that are related hierarchically, such as message stores and folders, they can register with the child object or the parent object. A registration on a child object causes a notification to be sent only when the event occurs within that object while a registration on a parent object causes a notification when the event occurs on the parent object and/or any of its child objects. Address books and message stores accept registration for events affecting their child objects. For example, clients can register with a message store object to receive notifications about changes affecting any or all of its folders.

Clients can register directly with a service provider or indirectly through MAPI, using the session object. Direct registration is when the client calls the **Advise** method in the object that will experience the event and send the notification. Indirect registration is when the client calls the **Advise** method in the session object, passing in the entry identifier of the real advise source object. **IMAPISession::Advise** can accept registration for events that affect the session object itself or another object. When an entry identifier is specified, **IMAPISession::Advise** forwards the call to the **Advise** method in the appropriate advise source object. The resulting notifications are generated by the service provider rather than by the MAPI session object.

Managing Memory

Knowing how and when to allocate and free memory is an important part of programming with Extended MAPI. MAPI provides three API functions that manage memory in a consistent way. When both 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 at will, independent of the service provider.

MAPIAllocateBuffer, **MAPIAllocateMore**, and **MAPIFreeBuffer** are the three functions that enable client applications and service providers to exchange blocks of memory. These functions manage memory using double linked lists. **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 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 a call, **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 you do use **MAPIAllocateMore** to allocate additional memory for a buffer, always free it by passing the buffer allocated with **MAPIAllocateBuffer** to **MAPIFreeBuffer**. Ignor any errors returned from **MAPIFreeBuffer**. The function almost always succeeds, and in the rare case that an error is returned, there is little the caller can do about it.

Note: Clients call the **MAPIAllocateBuffer** and **MAPIAllocateMore** functions directly, while providers must make an indirect call, retrieving the function pointers through a call to **IMAPISupport::GetMemAllocRoutines**.

MAPI also supports the use of the OLE interface, **IMalloc**, for memory management. The **MAPIGetDefaultMalloc** function returns an OLE allocator object which allows you to call **IMalloc** interface methods. The main advantage to using the **IMalloc** methods for managing memory is that you can reallocate an existing buffer. The MAPI memory functions do not support reallocation.

When allocating a a property value array, allocate the structures using **MAPIAllocateBuffer** for the main part and either have enough extra in the original allocation for all the members, or use **MAPIAllocateMore** to allocate them. For the SRowList and ADRLIST structures, the requirement is that the property value array be allocated separately.

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 entry 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**.

Figure 2 illustrates the layout of an **ADRLIST** data structure, indicating the separate memory allocations required. The dotted lines show memory that can be allocated and released with one call.

{ewc msdncd, EWGraphic, group10824 1 /a "SDKEX.bmp"}

Figure 2. Memory management for ADRLIST structures.

MAPI provides two API functions, **FreePadrlist** and **FreePRows**, that can save you some work freeing 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.

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. When allocating and freeing non-MAPI related memory internally within your client application or service provider, you can use whatever mechanism makes sense.

The model defines parameters as one of three types. They may 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. The follow table explains the rules for allocating and releasing memory for these types of parameters:

Type	**Memory Allocation**	**Memory Release**
input	Caller is responsible and may use any mechanism.	Caller is responsible and may use any mechanism.
output	Called function is responsible and must use **MAPIAllocateBuffer**.	Called function is responsible and must use **MAPIFreeBuffer**.
input-output	Caller is responsible for the initial allocation and callee may reallocate if necessary using **MAPIAllocateMore.**	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 you return an error, 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**).

Handling Errors

The MAPICODE.H header file contains many of the values that you might want to return from an interface method or might see when you call an interface method. Success is defined as both S_OK and SUCCESS_SUCCESS, where S_OK is typically used with OLE and Extended MAPI and SUCCESS_SUCCESS is used with Simple MAPI. All of the warning and error values consist of three parts separated by underscores. The first part is the prefix MAPI, the second part is W for warning and E for error, and the third part is a string that describes the actual condition. For example, MAPI_E_TOO_COMPLEX indicates an error that occurred because the implementation could not handle the request whereas MAPI_W_PARTIAL_COMPLETION is a warning generated when a method has succeeded in doing some of the work requested, but not all.

Success, warning, and error values are returned using a 32-bit number known as a result handle, or HRESULT. HRESULTs are really not handles to anything; they are merely a 32-bit value with several fields encoded in the value. A zero result indicates success and a non-zero result indicates failure.

HRESULTs work differently depending on the platform you are 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. In fact, the SCODE is no longer used. MAPI on 32-bit platforms works solely with HRESULTs.

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, group10824 2 /a "SDKEX.bmp"}

HRESULTs on 32-bit platforms have the following format:

{ewc msdncd, EWGraphic, group10824 3 /a "SDKEX.bmp"}

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 currently five facilities: 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. Most SCODEs and HRESULTs set the facility field to FACILITY_ITF, indicating an interface method error. The following table describes the various facility fields:

Facility	**Description**
FACILITY_NULL	For broadly applicable common status codes such as. S_OK; value is zero
FACILITY_ITF	For most status codes returned from interface methods; value is defined by the interface. That is, two SCODEs or HRESULTs 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 whose code (lower 16 bits) value is in the range of DOS error codes (that is, less than 256) have the same meaning as the corresponding DOS error.

The code field is a unique number that is assigned to represent the error or warning.

Using Macros for Error Handling

MAPI defines several macros for making it easier to work with SCODEs on 16-bit platforms and HRESULTs on both platforms. Some of the macros and functions below provide conversion between the two datatypes and are quite useful in code that runs only on 16-bit platforms, codes that runs on both 16-bit and 32-bit platforms, and 16-bit code that is being ported to a 32-bit platform. These same macros are meaningless in 32-bit environments and are available to provide compatibility and make porting easier.

The set of macros are listed below; refer to *The MAPI Programmer's Reference* for complete descriptions of each.

Macro	**Description**
GetScode	Returns an SCODE given an HRESULT
ResultFromScode	Returns an HRESULT given an SCODE
MakeResult	Returns an HRESULT given an SCODE that represents an error
MAKE_SCODE	Returns an SCODE given an HRESULT
HrFailed	Tests an HRESULT for a warning or 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 the severity of the SCODE or HRESULT - returns TRUE if the severity is zero and FALSE if it is one
FAILED	Tests the severity of the SCODE or HRESULT - returns TRUE if the severity is one and FALSE if it is zero

Note Calling **MakeResult** for S_OK verification carries a performance penalty. You should not routinely use **MakeResult** for successful results. **GetScode** and **MakeResult** are externally defined and not MAPI support functions. You will not find them in the table of support functions passed to your provider, but in a separate DLL that you can link with your provider.

Strategies for Error Handling

Because interface methods are virtual, it is not possible to know, as a caller, the full set of values that may be returned from any one call. One implementation of a method may return five values; another may return eight. The *MAPI Programmer's Reference* lists a few values that may be returned for each method; these are the values that you must check for and handle in your code because they have special meanings. Other values may be returned, but since they are not meaningful, you do not need to write special code to handle those. A simple check for zero or not zero is adequate.

A few of the interface methods return warnings. If a method that you call can return a warning, use the **HrFailed** macro to test the return value rather than a check for zero or not zero. Warnings, like success codes, equate to zero. If you do not use the macro, you will miss these codes.

Although most interface methods and functions return HRESULTs, 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 SCODEs or HRESULTs.
- No Extended MAPI call ever returns a Simple MAPI error.

If you work with both clients that use both Simple and Extended MAPI API calls, you should 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 you make a call to an unsupported feature in Simple MAPI, you receive the error MAPI_E_NOT_SUPPORTED. When you make the same type of call in Extended MAPI, you receive one of four possible errors: MAPI_E_NO_SUPPORT, MAPI_E_INTERFACE_NOT_SUPPORTED, MAPI_E_INVALID_PARAMETER, or MAPI_E_VERSION. The table below points out these similarities and differences.

Simple MAPI Return Value	**Extended 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

As an implementor of interface methods, you 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. If your method requires a return value that is not defined by MAPI in the MAPICODE.H header file, you can use a special value - MAPI_E_EXTENDED_ERROR. MAPI_E_EXTENDED_ERROR indicates to the caller that more information about the error is available; the caller retrieves that information by calling the **GetLastError** method on the same object that returned 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 the MAPI world, **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 you make a session call, and MAPI forwarded the call to a service provider to do the work, you may be tempted to call GetLastError on the service provider object. This is wrong; you must call **IMAPISession::GetLastError**, the session object's method.

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 makes implementation easier and processing faster. Rather than handling many requests and returning a value for each, deferring errors allows the calls to be bundled. Processing many requests at once cuts down on network traffic, therely improving performance. Deferring errors is especially useful in calls to delete or copy properties, which can be quite lengthy operations.

Because some operations can take time, service providers may choose to process certain calls in a deferred manner regardless of whether the MAPI_DEFERRED_ERRORS flag was set on the call. Deferring errors in these situations is preferable to rejecting the call and returning MAPI_E_TOO_COMPLEX because of time or resource constraints.

When you have set the MAPI_DEFERRED_ERRORS flag, you must be prepared to receive the return information at any time, possibly when it is too late to do anything about it or after you no longer have any data about the original request.

Session Identity

Every Extended MAPI session has the concept of a primary identity which describes who it is. The definition of primary identity is different depending on the message service. For example, many services designate an address book recipient as the identity for the session.

There are a set of properties associated with supplying an identity:PR_IDENTITY_DISPLAY_NAME, PR_IDENTITY_SEARCH_KEY, and PR_IDENTITY_ENTRYID. These properties are essential for configuration when the provider has an identity.

Not every service provider can supply an identity; service providers that do not have an identity set their PR_RESOURCE_FLAGS property in the MAPISVC.INF file to STATUS_NO_PRIMARY_IDENTITY. Service providers that can supply the primary identity set their PR_RESOURCE_FLAGS property to STATUS_PRIMARY_IDENTITY. When a profile is built and MAPI determines identity for the session, the first service provider that has claimed primary identity is granted it. The STATUS_PRIMARY_IDENTITY bit for the PR_RESOURCE_FLAGS property for the other service providers is turned off. If the service provider with primary identity is deleted from the profile, the next provider in line assumes the role. cannot be used to supply an identity.

Administering Profiles and Message Services

Profile administration is an important part of MAPI, for without valid profiles, logon cannot occur and a session cannot be established. Profile administration ensures the validity of each profile on the workstation and provides a means for updating obsolete information. Client applications can either avoid supporting administration by relying on the applications that MAPI supplies, the Control Panel applet and the Profile Wizard, or provide some level of support. To enable client applications to implement administrative support for their users, MAPI has implemented four interfaces:

- IProfAdmin
- IProfSect
- IMsgServiceAdmin
- IProviderAdmin

IProfAdmin and **IProfSect** work with profiles and are used by both clients and service providers. **IProfAdmin** is used to create, copy, or delete profiles while **IProfSect** is used to access and modify profile properties.

IMsgServiceAdmin and **IProviderAdmin** are for clients only; they deal with message services and the service providers within those services. **IMsgServiceAdmin** enables clients to modify their profile's message service sections; **IProviderAdmin** enables clients to add or delete service provider sections.

Clients and service providers access pointers to these interfaces in a variety of ways. The **OpenProfileSection** method implemented in the **IMAPISupport**, **IMAPISession**, and **IProviderAdmin** interfaces returns a pointer to an **IProfSect** implementation. **IMAPISession** also includes a method, **AdminServices,** for providing clients with access to an **IMsgServiceAdmin** pointer.

The API function **MAPIAdminProfiles,** callable by both clients and service providers, returns an **IProfAdmin** pointer. To access the **IProviderAdmin** interface, clients must call **IMsgServiceAdmin::AdminProviders** method pointer. Message services receive an **IProviderAdmin** pointer when their entry point function is called; it is one of the input parameters for the prototype.

Figure 3 illustrates 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, group10824 4 /a "SDKEX.bmp"}

Figure 3. Interface Methods and Functions for Profile Administration.

These next sections describe how to use the administration interfaces to manage profiles and message services.

Using IProfAdmin

The methods of **IProfAdmin** allow users to create, copy, and delete profiles, and to change passwords. Call **IProfAdmin::CreateProfile** to create a new profile. **CreateProfile** accepts a set of option flags that provide MAPI with information. For example, 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. Set the MAPI_DIALOG flag to allow the display of a user interface so that the user can enter configuration information if necessary. **CreateProfile** calls the entry point function for each message service to be added to the profile, asking that each service's configuration property sheets be displayed. If a user interface is disallowed, the message services remain unconfigured.

The **IProfAdmin::ChangeProfilePassword** method allows you to change a profile's password. Keep in mind that this method is unsupported on 32-bit platforms; you will receive the error MAPI_E_NO_SUPPORT.

When you call **IProfAdmin::DeleteProfile** to delete a profile, MAPI calls the entry point function of every message service in the profile with the *ulContext* parameter set to MSGSERVICE_DELETE as the context. 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 **IProfAdmin::SetDefaultProfile** method marks a profile as the default. 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 you from adversely modifying a profile that is in use. MAPI will prevent you from deleting the profile, but will not prevent you from deleting every message service in it, causing every message service provider to stop and unpredictable results to occur.

Using IMsgServiceAdmin

The methods of **IMsgServiceAdmin** allow clients to access other objects, such as the message services table and a provider administration object. Clients retrieve **IMsgServiceAdmin** pointers either through the session object with a call to **IMAPISession::AdminServices** or by calling **IProfAdmin::AdminServices**. Most messaging clients use the session object; special configuration clients, such as the Control Panel applet, use the profile administration object.

Clients can use the **GetMsgServiceTable** method to access a MAPI table object containing properties of all of the message services in the current profile. One of the properties included in the table is the PR_SERVICE_UID property, the unique identifier for the service. Another property, PR_SERVICE_NAME, comes from the [Services] section in MAPISVC.INF. The value for this property should never be changed or localized. PR_SERVICE_NAME can be used to programmatically identify the type of message service. The PR_RESOURCE_FLAGS property describes the message service's capabilities. Once a client calls GetMsgServiceTable to access a view of the message services table, any changes that occur to message services within the profile will not be reflected in the view. Refer to the Chapter 9, "Using Tables," for more information about the message service table.

Another table object, the providers table, is accessible with the **GetProviderTable** method. The providers 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

Clients can use the information in the providers table to change the transport order, the order in which transports are called to deliver messages. When you change the transport order with a call to **MsgServiceTransportOrder**, you must pass in a list of all of the transports in the profile. It is not possible to switch the position of a few transports by passing in only those few. The **MsgServiceTransportOrder** call overrides the transport provider preferences as indicated by the STATUS_XP_PREFER_LAST bit in the provider's PR_RESOURCE_FLAGS property.

Whether or not clients are allowed to change anything about the service providers that belong to the service is up to the service itself. Typically, clients are not allowed to add service providers to or delete service providers from a service. Also, clients are not allowed 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.

When **IMsgServiceAdmin::CreateMsgService** is called to add a message service to the profile, 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 MSGSERVICE_CREATE. **CreateMsgService** creates a provider section in the profile for every provider section in MAPISVC.INF. Once the message service has been added, clients must call **GetMsgServiceTable** to access its unique identifier.

Call **IMsgServiceAdmin::ConfigureMsgService** to configure a newly added message service. 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, call **IMAPIStatus::SettingsDialog** instead.

IMsgServiceAdmin::CopyMsgService lets clients copy a message service to the same profile or a different profile. 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.

Clients that want to delete a message service from a profile must first locate its unique identifier in the message services table and then call **IMsgServiceAdmin::DeleteMsgService**. Before removing information from the profile, **DeleteMsgService** calls the message service's entry point function to give the service an opportunity to clean up.

Message services can designate a service provider to be the supplier of the primary identity for a profile by calling **IMAPISupport::SetPrimaryIdentity**. Not every service provider can become the supplier of a profile's identity because some providers have no notion of identity. Such service providers set the STATUS_NO_PRIMARY_IDENTITY bit in the PR_RESOURCE_FLAGS property.

Usually within a message service, every provider that can supply an identity has the same notion of it. Typically, an address book recipient entry identifier serves as the primary identity. Service providers that can supply an identity set the STATUS_PRIMARY_IDENTITY bit in the PR_RESOURCE_FLAGS property along with setting the PR_IDENTITY_DISPLAY_NAME, PR_IDENTITY_SEARCH_KEY, PR_IDENTITY_ENTRYID properties. Providers that supply identity set these properties and include them when they are building a row entry for the status table.

About the MAPI Samples

About the Sample Client

The MAPI SDK includes a sample MAPI client application written in C++ called CLIENT. The CLIENT was written to illustrate most of the typical tasks MAPI clients perform.

This sample client allows users to submit and retrieve messages from any underlying messaging system and includes the following features:

- Manipulation of multiple message stores
- Reply, reply to all, and forwarding features
- Full configuration support
- OLE style drag and drop

The user interface is implemented with the common controls provided with Windows 95. The hierarchy table, for example, is implemented with a tree view control. This implementation enables each item in the table to be associated with a label and a bitmap.

To help with debugging, the CLIENT has a built-in tracing mechanism called **TraceResult**. You'll notice its use throughout the code fragments.

The CLIENT defines several object classes to wrap calls to the MAPI interfaces. There are two objects that work with message stores: CStoreClient and CStoreView. For each message store provider in the user's profile, the CLIENT instantiates one CStoreClient and one CStoreView. The CStoreClient object handles the management of the message store by wrapping calls made to the various interfaces: **IMAPISession**, **IMsgStore**, **IMessage**, and **IFolder**. The CStoreView object handles the user interface for the message store.

The code excerpts from the CLIENT are not exact copies due to space limitations. Logic that did not directly relate to the topic being discussed was not included in the documentation. Also not included were lengthy comments, debug statements, error handling blocks, and code that related more to user interface programming than MAPI programming. The goal of the MAPI Programmer's Guide is to focus on individual MAPI-related topics and to provide a task-oriented approach to MAPI programming. Using the sample code verbatim would lessen readability and introduce a level of unneeded complexity. The entire sample client, including all source and header files, is included with the MAPI SDK.

About the Sample Message Store Providers

About the Sample Address Book Providers

About the Sample Transport Providers

MAPI Form Architecture

This chapter provides an overview of the MAPI form architecture.

Note Because the MAPI forms architecture is based on the OLE component object model, developing form container applications or form server applications requires knowledge of OLE programming. For more information on OLE, see the *OLE Programmer's Reference*.

MAPI Form Components

A typical MAPI environment in which forms are used consists of the following components:

Component	Description
MAPI	Enables messaging in a networked environment.
MAPI service providers	Provide basic messaging services to MAPI clients.
MAPI client applications	Provide the user interface that enables users to interact with forms. Microsoft Exchange is one example of a MAPI client.
MAPI form interfaces	Used exclusively by form objects and form registry providers.
Form registry providers	Manage the storage and activation of form objects.
Form objects	Display and handle form interaction.

Each component is described in further detail following.

MAPI

Within the MAPI messaging system, a form object is treated no differently than any other MAPI object. MAPI enables client applications to work with form registry providers in the same manner that client applications work with other service providers.

This 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.

Because forms are types of messages, they exhibit properties and behavior that is consistent with MAPI message objects. Because forms can have properties (also called "fields"), controls, and a display rendering that is application-specific, forms are manipulated through a set of MAPI interfaces that are specific for forms. The actual definition of a form is stored in a *form registry*, which is discussed later in this chapter.

To incorporate forms within your application, you use MAPI form functions to create and access forms and form properties. For more information on these interfaces, see the Forms API section later in this chapter. Like basic messages, MAPI forms contain some predefined properties such as the form's sender, the form's intended recipient, and when it 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 fields for Bug Type, Bug Severity, and Product Version.

MAPI interfaces are separate from the MAPI form interfaces. Although the two sets of interfaces are part of the same specification, there is very little interaction or dependence between them. To a MAPI client, calling MAPI form interfaces is the same as calling any MAPI functions. You still need to establish a MAPI session, identify the correct providers in the MAPI profile, and so on.

MAPI Service Providers

Form objects have the same access to the services of MAPI address book providers, message store providers, and transport providers, as any other client application in a MAPI messaging environment. Additionally, service providers can be developed to provide special handling for the properties and message classes of particular form objects.

MAPI Client Applications

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 registry providers and form objects.

MAPI Form Interfaces

MAPI defines the following interfaces for use in developing form objects and form registry providers:

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 objects against a specific form container.
IMAPIFormFactory	Supports the use of configurable run-time form objects in distributed computing environments.
IMAPIFormInfo	Enables client applications to work with properties that are specific to a message class.
IMAPIFormMgr	Enables client applications to get information about form objects, activates form objects, and installs form objects 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 Programmer's Reference, Volume 1*.

Form Registry Providers

To help organize form objects and make them easily accessible to client applications throughout the organization, the MAPI form architecture includes form registries, which support the installation, administration, replication, and use of form objects.

Although developers can implement their own form registry providers, most environments will use the form registry provider implemented by Microsoft.

Forms Objects

- **Form Objects**. Instantiated objects created by form servers or form clients. These can include objects that represent the forms themselves, or they can be "support" objects such as view context objects, message site objects, and form info objects. In most cases, form clients and form servers communicate through objects rather than through direct communication.
- **Form Servers**. Form-specific code that manages a user's interaction with a form. Form servers are sometimes referred to as "message class handlers" or "form handlers." Every form has at least one form server associated with it.

A MAPI form is simply a type of MAPI message. That is, it belongs to a particular message class with a Class ID. From a programmer's perspective, most forms have two components:

- A user interface that enables users to interact with form objects.
- A form server that manages the user's interaction with the message.

Most users know only about the form's user interface – they never see the form server. Usually, a form's user interface appears as a frame window or popup window that renders the property values of the message.

Developing MAPI Form Objects

Developing a form object includes the following steps:

1. Determine the properties that constitute the form object's property set.
2. Name each property.
3. Name the new property set.
4. Generate a unique message class identifier (CLSID) for the form object.
5. Create a configuration file for the form object.
6. Write code to handle the user's interaction with the form object.

Common Messaging Calls (CMC)

This chapter describes the functionality architecture of the Common Messaging Calls (CMC) application programming interface (API). It provides a model of the CMC interfaces and describes the the relationship in that interface between an application and the various components of a messaging service.

CMC Architecture

The CMC API works as a layer between a messaging-enabled application and a messaging service. The messaging service in turn can support multiple messaging protocol services, each using different messaging formats and protocols, for example, X.400, RFC 822, and SMTP. The design of the CMC interface specifies that all functions in it be independent of the messaging protocol services. However, the API does enable developers to use extensions to invoke protocol-specific functions. For more information on using extensions, see "Extensions." later in this chapter.

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

{ewc msdncd, EWGraphic, group10827 0 /a "SDKEX_08.bmp"}

Figure 1 Model of the Common Messaging Calls API

Using the directory, the messaging-enabled application can look up information about users of connected messaging services and can resolve addressing information provided with user names to useful addresses. Some services might also provide an interface enabling users to create recipient lists for messages or find out details about a specific recipient.

CMC assigns a submission queue for each messaging-enabled application. By doing so, the CMC model provides each application with synchronous submission of messages to the messaging service. Once the application has completed a call to send a message to the messaging service, the application transfers all further responsibility for sending the message to the CMC implementation.

On the receiving side, a mailbox receives all messages for a user. The messaging service 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, the application can retrieve selective summaries of the contents of a mailbox, along with identifiers for the particular messages summarized. The application can use these identifiers to select and read individual messages.

CMC Programming Basics

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

To send a message, a messaging-enabled application:

1. Establishes a session with the messaging service either through the **cmc_logon** function or interactively by sending the CMC_LOGON_UI_ALLOWED flag value with the **cmc_send** function.
2. Submits a message to the submission queue. Usually, a messaging-enabled application does so through a **cmc_send** function. If a messaging-enabled application uses **cmc_send**, it must first generate the CMC message structure used in the **cmc_send** function. The messaging-enabled application can also use the more limited **cmc_send_documents** function to send a message; this function is primarily used to call from macro languages.
3. Uses the **cmc_logoff** function to close the session.

To retrieve a message, a messaging-enabled application:

1. Establishes a session through the **cmc_logon** function.
2. Retrieves a summary of mailbox information through the **cmc_list** function.
3. Retrieves an individual message by using the **cmc_read** function.
4. If necessary, enables a user to act on the message in the mailbox (for example, delete it) using the **cmc_act_on** function.
5. Releases memory allocated by the system for structures by passing the returned pointer to the **cmc_free** function.
6. Closes the session by using the **cmc_logoff** function.

To look up names and address information in the directory, a messaging-enabled application:

1. Establishes 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. Uses **cmc_look_up** to translate a user's display name into a messaging address. With this function, the application can also request a dialog box to create addressing lists or recipient-specific details.
3. Releases memory allocated by the system for structures by passing the returned pointer to the **cmc_free** function.
4. Closes a session by using the **cmc_logoff** function.

Sessions

CMC function calls occur within the context of a session. Your application establishes a session with a **cmc_logon** call and terminates a session with a **cmc_logoff** call. The **cmc_logon** call also checks the user's credentials for the messaging service and sets session attributes. Session attributes include character set and version number. Currently, there is no support for sharing sessions among applications in the CMC API.

Service Configuration Queries

The established configuration of the messaging service is available for query by messaging-enabled applications. An application may query the service to determine the service's support for different versions of the CMC API and for different extensions and to receive information on the environmental parameters that comprise the configuration. The CMC API does not define a function for the modification of service configuration information and does not define the form of the stored information (for example, file format).

CMC Functions, Data Structures, and Data Types

The CMC API uses a fixed set of API functions, data structures, and data types. *Data structures* are those records or arrays that can be applied to data so the data can be interpreted and specific operations can be performed on it. *Data types* are data elements that have a specific range of values and specific memory storage characteristics (Boolean, floating point, and so on) and can have specific operations performed on them,.

The CMC API functions listed in alphabetic order:

Action	Function	Description
Sending messages	**cmc_send**	Sends a message.
	cmc_send_documents	Sends a message; this function is string based and is usually used in macro language calls.
Receiving messages	**cmc_act_on**	Performs an action on a specified message.
	cmc_list	Lists summary information about messages meeting specified criteria.
	cmc_read	Reads and returns a specified message.
Looking up names	**cmc_look_up**	Looks up addressing information.
Administration	**cmc_free**	Frees memory allocated by the messaging service.
	cmc_logoff	Terminates a session with the messaging service.
	cmc_logon	Establishes a session with the messaging service.
	cmc_query_configuration	Determines information about the installed CMC service.

The CMC data structures listed in alphabetic order:

CMC_attachment	Message-attachment structure
CMC_counted_string	String with an explicit length designation
CMC_extension	Extension structure
CMC_message	Message structure
CMC_message_reference	Message-reference structure
CMC_message_summary	Message-summary structure
CMC_object_identifier	Object-identifier structure

CMC_recipient
Originator or recipient structure

CMC_time
Time structure

The CMC data types listed in alphabetic order:

CMC_boolean
Value indicating logical true or false

CMC_buffer
Pointer to a data item

CMC_enum
Data type containing a value from an enumeration

CMC_flags
Container for flag bit values

CMC_return_code
Return value indicating either that a function succeeded or why it failed

CMC_session_id
Unique identifier for a session

CMC_string
Character-string pointer

CMC_ui_id
User-interface handle

Complete reference information for all CMC data structures and types is provided in "CMC Data Structure Reference.

Extensions

The data structures and functions defined by CMC can be added to through the use of extensions. Extensions are used to add fields to data structures and parameters to function calls.

A generic data structure, **CMC_extension**, has been defined as a basis from which to create these extensions. **CMC_extension** consists of an item code identifying the extension, an item data type holding the length of extension data or the data itself, an item reference that points to where the extension value is stored or that is NULL if there is no related item storage, and flags for the extension.

The syntax for **CMC_extension** is as follows:

```
typedef struct {
     CMC_uint32  item_code;
     CMC_uint32  item_data;
     CMC_buffer  item_reference;
     CMC_flags   extension_flags;
} CMC_extension
```

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 a messaging application to the CMC service or as an output parameter from CMC to an application. If an extension is an input parameter, the application in question allocates memory for the extension structure and any other structures associated with the extension. If an extension is an output parameter, CMC allocates the storage for the extension result, if necessary and the application must free the allocated storage with a **cmc_free** call.

Extensions play a dual role in CMC messaging. First, they provide a mechanism to accommodate features not common across all messaging services. Second, they enable extension of CMC in the future, minimizing backward-compatibility issues. When using extensions to accommodate features specific to a messaging service, use caution. Reliance on specific features limits the portability of applications across messaging services; also, such features may not survive a journey through multiple gateways in a mixed messaging network.

The Common Messaging Calls common extension set contains those function and data extensions that are common to most messaging services but are not in the CMC base specification. For a full list of common extension declarations, see the CMC header file.

CMC Programming Examples

This section includes example code that illustrates various common messaging operations. Included are the following:

- 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

The following code example demonstrates how an application logs on and off and uses **cmc_query_configuration** to get system information:

```
/* Local variables used */

CMC_return_code Status;
CMC_boolean     UI_available;
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 */
```

The following code example demonstrates how an application sends 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 */
```

The following code example demonstrates how an application lists, reads, and deletes 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;

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);
```

The following code example demonstrates how an application looks up a specific recipient and gets 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);
```

The following code example demonstrates how an application specifies 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;    /* Common extensions I need 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;    /* Common extensions I need 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 */
```

Simple MAPI

Microsoft's Simple Messaging Application Programming Interface (MAPI) is a set of functions that applications use to create, manipulate, transfer, and store messages. Simple MAPI gives application developers freedom to define the purpose and content of messages, and flexibility in their management of stored messages. MAPI provides a common interface that application developers use to create messaging-enabled and messaging-aware applications independent of the underlying messaging system.

Simple MAPI includes routines intended for use by applications, such as spreadsheet programs and word processors that must become messaging-aware. You can also use Simple MAPI to build custom messaging applications such as a calendaring and scheduling program. These applications are typically concerned with the easy exchange of datafiles and simple messages generated by these applications.

This small, easy-to-use set of APIs allows applications to access inbound and outbound messages. Although designed to be called from C programs, you can call the functions with little or no parameter modification from application-specific and standalone scripting packages such as Visual Basic, Actor, Smalltalk, and Object Vision. The target audience is application developers who have documents or other application data within their applications that they want to send or receive as messages. Because of its ease of use, Simple MAPI allows applications to become messaging-aware. Consider the following example:

A network monitoring application generates a network usage report. Instead of printing the report or requiring a user to cut/paste it into a message, the application can mail the report directly to a Microsoft Excel or Toolbook application that consolidates the information and forwards it on to a network operations center.

The Simple MAPI functions include a user interface, but you can call the APIs non-interactively (not generating any user interface (UI)). The implementation provides the user interface is provided to allow the user to address the message, enter a note, and send the message. MAPI does not define the exact style of the user interface, and it can be different on each platform that implements Simple MAPI.

If the user has not yet logged-in to the messaging system, the calls prompt, as appropriate, for the required session parameters, for example, user name and password.

MAPI Sessions

Both Simple and Extended MAPI calls are made in the context of a client *session* that identifies messaging operations and the providers that handle those operations. Most Simple MAPI calls include a session handle acquired by a MAPI call, **MAPILogon**. Acquiring a session handle is called "Logging on" to the messaging system. While MAPI returns one session handle to the client, several provider sessions may be established on behalf of the client by the messaging subsystem.

Some Simple MAPI functions support two modes of operation, each with its own way to log onto the messaging system: implicitly and explicitly. Only explicit logon (described below) requires use of the **MAPILogon** call.

Implicit Logon

Some Simple MAPI functions generate a logon dialog when called outside of an established MAPI session. They substitute a NULL for the required MAPI session handle and log onto the messaging system. If required, the calling application can suppress the MAPI dialogs. When using implicit logon, MAPI restores the state of the messaging system prior to the call after the call completes − that is, if no session existed before the call, then the session opened by the MAPI call is closed by the time the call returns. Implicit logon can only be used with certain Simple MAPI calls.

Implicit logon is most useful for applications that can only make a single call or cannot maintain state information across calls. Applications should not use implicit logon that make multiple MAPI calls.

Explicit Logon

Any Simple or Extended MAPI application can call **MAPILogon** and provide credentials that establish a session with the messaging subsystem. When using **MAPILogon**, MAPI returns a session handle that you can used in future MAPI calls. This session must be explicitly closed using **MAPILogoff**.

Logon Credentials

MAPI applications must provide profile credentials in order to log onto the messaging system. The WMS organizes different sets of providers and provider identities into profiles so that starting a MAPI application starts all the appropriate store, address book, and transport providers. The messaging subsystem stores this information about which providers to start and associated them with a profile name. You can protect a profile with a password.

When an application establishes a session with the messaging subsystem, it must provide the profile name and (if required) the correct password. MAPI starts the appropriate providers and return a session handle which represents the session.

Shared Session

Workgroup applications should all be able to work from the same profile without presenting profile-selection dialogs when each application begins. To make this possible, MAPI introduces the idea of a shared session that is established once but can be "used" when subsequent applications ask to use the shared session. On a given computer, only one shared session can exist at a time. If an application does not specify a profile name or password on the **MAPILogon** call, the messaging subsystem attempts to acquire the shared session if one exists. Applications can suppress this behavior by setting the MAPI_NEW_SESSION flag, which instructs the messaging subsystem to ignore the shared session if one exists.

An application that sets its session to be the shared session can set the MAPI_ALLOW_OTHERS flag on the **MAPILogon** call. The most recent session established with the MAPI_ALLOW_OTHERS flag becomes the shared session. If that application (and all applications that subsequently acquire the shared session) log off, then the session is closed, and, if one exists, the previously specified shared session becomes the shared session.

Note Simply because an application acquires the shared session does not mean that its session handle matches that of other shared session applications. Session handles are unique to the applications that generate them and are not valid across tasks.

Default Profile

Workgroup applications that do not know which profile to used for a given invocation can specify the default profile – the default profile differs from the shared session, which is a particular running MAPI session that can be of any profile. The default profile, by comparison, is always the same profile, and has nothing to do with any running MAPI session. An application that provides a null profile name for **MAPILogon** is first checked for use of the shared session. If the shared session is not suppressed and you can acquire it, MAPI returns the shared session to the calling application. If use of the shared session is suppressed or impossible, then **MAPILogon** proceeds as if the name of the default profile had been passed in as the *lpszProfileName* parameter. You can suppress this behavior by using the MAPI_EXPLICIT_PROFILE flag. If you set this flag, **MAPILogon** proceeds in the situation described above as if you passed a null string in as *lpszProfileName*.

Session Limits

Some systems may allow a limited number of sessions − applications must be prepared to handle returns of MAPI_E_TOO_MANY_SESSIONS. Typically, situations in which a user is logged into the system (via an email application) and a messaging-enabled application attempts to log on with a different identity may fail with this error.

Simple MAPI Logon UI

Applications may choose to generate their own logon (i.e. profile selection) UI or have MAPI generate common UI for them. A flag in the **MAPILogon** call, MAPI_LOGON_UI, controls whether or not the messaging subsystem is allowed to prompt the user interactively with a profile selection dialog. Keeping this flag clear is very useful when writing non-interactive applications that provide credentials programmatically − If the MAPI_LOGON_UI flag is clear, profile selection UI, which might block an unattended application, is never presented. Even if the MAPI_LOGON_UI flag is set, UI is not always generated. If a valid profile and password are included on the **MAPILogon** call, a session is generated and no UI is presented − a profile selection dialog is only presented if a logon with the supplied credentials should fail.

About MAPI Interfaces

MAPI interfaces, as stated earlier, are collections of related functions. Interfaces provide a binary standard for object interaction and are what is known in the C++ world as abstract base classes. This means that only the prototypes of the methods are defined; the interface name, method names, and method parameters are specified, but there are no actual implementations. It is up to the object inheriting the interface to provide its own implementation. Sometimes this object is a service provider object; sometimes it is a client object; often it is an object provided with the MAPI subsystem.

MAPI interfaces define a contract between the object implementing the interface and the object using the implementation. This contract can be loose or tight, depending on the method. A loose contract allows a method to return MAPI_E_NO_SUPPORT, indicating no functionality. A tight contract disallows the MAPI_E_NO_SUPPORT return value; all implementations of the method must provide some functionality. For example, the **IMAPIStatus::FlushQueues** method makes sense only for transport providers. However, message store providers also implement status objects and the **IMAPIStatus** interface, using a loose contract. Message store providers return MAPI_E_NO_SUPPORT in their **FlushQueues** implementation.

About Interface Relationships

As mentioned in the discussion about objects, all MAPI interfaces ultimately derive from the base COM interface, **IUnknown.** The MAPI inheritance hierarchy is fairly straightforward. There are only three base interfaces, or interfaces that other interfaces can inherit: **IUnknown**, **IMAPIProp**, and **IMAPIContainer**. Most interfaces are direct descendants of **IUnknown**, some inherit from **IMAPIProp**, and a few inherit from **IMAPIContainer**.

{ewc msdncd, EWGraphic, group10829 0 /a "SDKEX_06.bmp"}

Figure 1. MAPI Interface Hierarchy

About IUnknown

IUnknown, as the top of the interface hierarchy, is implemented in all MAPI objects. **IUnknown** provides mechanisms for obtaining an interface implementation and controlling its lifetime with three methods: **QueryInterface**, **AddRef**, and **Release**. **QueryInterface** enables potential users of an object to find out if the object supports a particular interface. If the object does support the interface, **QueryInterface** returns a pointer to it. **AddRef** and **Release** make sure that an object is not freed while it is still being used, keeping a tally known as a reference count.

The following illustration shows how **QueryInterface** works. The person on the left is the caller; the person on the right is the object with the implementation of **QueryInterface**. The caller asks for an interface that supports specific functionality, in this case, baseball. Because the object doesn't support the interface, the caller receives nothing in return. However, when the caller asks a second time for a supported interface (that is, football), the object returns an interface pointer that can be the basis for further interaction between the caller and the object. In this example, the ball represents the interface pointer.

{ewc msdncd, EWGraphic, group10829 1 /a "SDKEX.bmp"}

Figure 2. How the QueryInterface method works.

Now, to translate the conceptual into code. The following example shows how **QueryInterface** would be implemented in C++ for an object that supports the fictional MAPI Football interface. To provide a more interesting example, the IFootball interface inherits from the ISports interface which inherits from **IUnknown**, rather than directly inheriting from **IUnknown**. Notice that the same pointer can be returned for any of the interfaces.

```
HRESULT CFootball::QueryInterface (REFIID   riid,
                                   LPVOID * ppvObj)
{
    // Always set out parameter to NULL
    *ppvObj = NULL;

    // Is the caller asking for one of my interfaces?
    if (riid == IID_IFootball || riid == IID_ISports ||
        riid == IID_IUnknown)
    {
        // Yes - return pointer and increment reference count
        *ppvObj = (LPVOID)this;
        AddRef();
        return NOERROR;
    }

    // No - return error.
    return E_NOINTERFACE;
}
```

The **AddRef** and **Release** methods control the reference count of an object. The reference count defines the object's lifetime. Without reference counting, the object would never know when it was safe to free its memory. As long as the reference count is greater than zero, the object's memory will not be freed.

There are a few basic reference counting rules. All methods or API functions that return interface pointers 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. The following code samples show how to implement **AddRef** and

Release for the CFootball object. For a detailed description of COM reference counting, refer to the OLE 2.0 Programmer's Reference.

```
//
ULONG CFootball::AddRef()
{
    // Increment the object's internal counter
    ++m_cRef;
    return m_cRef;
}

ULONG CFootball::Release()
{
    // Decrement the object's internal counter
    ULONG ulRefCountT = --m_cRef;
    // Deallocate if ref count is 0. Destructor frees memory
    if (0 == m_cRef)
    {
        delete this;
    }
    return ulRefCountT;
}
```

Selecting Interfaces to Implement

Because MAPI provides so many interfaces, it can be difficult to determine exactly which interfaces to implement and which interfaces to use. Typically the type of MAPI component you are writing and the feature set it provides determines the set of interfaces you will need to implement and use.

Service providers implement many more interfaces than clients, who are primarily users of interfaces, rather than implementors. The typical client application will only have to implement one interface: **IMAPIAdviseSink**. **IMAPIAdviseSink** is used to send event notifications. If your client also works with custom forms, you'll have to implement these additional interfaces: **IMAPIFormAdviseSink**, **IMAPIMessageSite**, **IMAPIViewAdviseSink**, and **IMAPIViewContext**.

Address book, message store, and transport providers all implement their own flavor of provider interface and logon interface. The provider interfaces are called **IABProvider**, **IMSProvider**, and **IXPProvider**, respectively; the logon interfaces are called **IABLogon**, **IMSLogon**, and **IXPLogon**. All of the provider interfaces are used in the same way as the logon interfaces. MAPI uses the provider interfaces to start up and shut down a service provider and the logon interfaces to handle requests from clients. However, because there are tasks that are specific to the type of service provider, the provider and logon interfaces cannot be identical. Refer to the discussion in "Considerations for Service Providers" or to the *MAPI Programmer's Reference* for details about the implementation of these interfaces.

Address book providers also implement **IABContainer**, **IDistList**, **IMAPITable**, and **IMailUser** and the interfaces a few of these interfaces derive from: **IMAPIContainer** and **IMAPIProp**.

Message store providers also implement the base interfaces **IMAPIContainer** and **IMAPIProp** as well as **IMsgStore, IMAPIFolder, IMessage, IAttach, IMAPITable,** and **IMAPIStatus.**

Transport providers typically implement only two additional interfaces: **IMAPIProp,** and **IMAPIStatus.** Remote transport providers also implement **IMAPIFolder** and **IMAPITable.**

A few miscellaneous interfaces exist to support custom components such as form servers, message class handlers, and message hook providers. **IMAPIForm** and **IPersistMessage** are implemented by form servers; **ISpoolerHook** is implemented by hook providers.

Selecting Interfaces to Use

The group of MAPI interfaces that you should use depends on the type of component that you are writing and the features you are including. Most clients and service providers frequently use **IMAPITable** and **IMAPIProp** and occasionally use **IMAPIProgress.** This is where the commonality between the two component types ends. Clients use a great many different interfaces, some implemented by MAPI and others implemented by service providers. MAPI provides clients with the **IAddrBook** interfaces for integrated address book access, the **IMAPISession** interface for general session access, and the **IMAPIStatus** interface for monitoring session status.

Clients can interact with service providers either indirectly, through **IMAPISession**, or directly through a variety of interfaces implemented by particular service providers. To make direct contact with address book providers, clients call the **IABContainer**, **IDistList**, and **IMailUser** interfaces. To access a message store provider directly, clients call **IAttach**, **IMAPIFolder**, **IMessage**, and **IMsgStore**. Clients also interact directly with service providers through their implementation of **IMAPIStatus.**

Clients that implement special features use other interfaces. For example, clients that perform profile and message service configuration use **IProfAdmin**, **IMsgServiceAdmin**, and **IProviderAdmin** and clients that display custom forms use **IMAPIForm**, **IPersistMessage**, **IMAPIFormContainer**, **IMAPIFormInfo**, and **IMAPIFormMgr**.

Service providers typically use only a few interfaces. They use **IMAPISupport**, a large interface implemented by MAPI to help service providers handle requests from clients. **IMAPISupport** has many methods, some of which are applicable to all service providers and others that apply only to specific types. See the *MAPI Programmer's Reference* for detailed information about the methods in **IMAPISupport**. Service providers that send event notifications also use **IMAPIAdviseSink**.

Some service providers use **ITableData** and **IPropData**, two utility interfaces implemented by MAPI. **ITableData** allows service providers to manage the underlying data of a table while **IPropData** allows service providers to set object and property access. Both of these interfaces are optional; service providers are not required to use them.

Transport providers that support the Transmission-Neutral Encapsulation Format (TNEF) for transferring properties use the **ITnef** interface supplied by MAPI.

About MAPI Objects

Now it is time to move into the object oriented world of programming with Extended MAPI. Programming with Extended MAPI is similar to programming with any other object oriented API. With Extended MAPI, you will implement some objects and use other objects. Where Extended MAPI differs from the standard object oriented methodology is in its special use of objects.

Extended MAPI, as mentioned previously, is based on the Component Object Model (COM) introduced with OLE. The Component Object Model is a specification that defines an environment for sharing code and data across processes. COM *objects* are unique because their public methods belong to one or more MAPI *interfaces*, or collections of related functions. Outsiders obtain access to COM objects only through pointers to these interfaces. MAPI interfaces define how objects behave and operate within the COM world.

MAPI defines many types, or classes, of objects, each type characterized by the interfaces it supports. Depending on whether you are creating a client application or a service provider, you will implement some objects and use others. When you implement an object, you provide the code for its interface methods. When you use an object, you make calls to the methods.

The following table defines the types of MAPI objects that clients and service providers work with in a typical session. You may recognize some of these types from the architectural discussion in Chapter 1.

MAPI Object Type	Description
Property	An object that describes a set of attributes for a MAPI object, these are the object's private data members
Table	An object that provides access to an object's private data in row and column format, similar to a database table
Container	An object that provides access to a collection of objects
Message	An object that can be sent to one or more recipients with a messaging system
Folder	An object that acts as a container for messages
Attachment	An object that contains additional data, such as a file, to be associated with a message
Message Store	An object that acts as a database of messages, organized hierarchically
Mail User	An object that describes an individual recipient of a message.
Distribution List	An object that describes a grouping of individual message recipients
Address Book	An object that acts as a database of recipients
Profile Section	An object that describes a particular message service or service provider
Session	An object created by the MAPI subsystem and passed to clients to represent a connection to underlying messaging systems
Support	An object created by the MAPI subsystem and passed to service providers to help them take care of client requests
Status	An object that is used to obtain access to the state of a session resource, such as the MAPI spooler or

	a service provider
Advise Sink	An object that provides a callback to be invoked when a particular object experiences a change or an error occurs
Logon	An object created by a service provider and passed to the MAPI subsystem for handling event notification and client requests
Provider	An object created by a service provider and passed to the MAPI subsystem for handling provider logon and shut down.
Form Server	An object for managing a user's interaction with a custom dialog box for entering data
Form Container	An object that provides access to information about a custom dialog box

Some of these types of objects have single implementations while other types have multiple implementations. For example, when a user logs on to MAPI, a session object is created that represents a connection with the messaging systems accessed through the message services listed in the profile. Each session has only one session object.

Property objects, on the other hand, are instantiated many times during the course of a session. Property objects describe the private data of other MAPI objects. For example, every message object includes in its private data one or more recipients. This data is accessible only through the interface methods implemented in the property object. Properties are explained in more detail in Chapter 7, "About MAPI Properties".

Table objects are another example of frequently implemented MAPI objects. Table objects make it possible to view properties in row and column format. Each row in the table represents a MAPI object and each column represents the value of one of the object's properties. There are many types of tables, named for the type of objects that are represented in the tables' rows. For example, the service providers table lists properties for all of the service providers in the current session. The status table lists the properties for status objects. Status objects are implemented by MAPI and service providers to provide access to status information. For more detail about MAPI tables, see Chapter 9, "Using Tables."

Some objects have the concept of access, meaning that when you open an object, you are granted a particular level of access. Only if you are granted read-write access will you be allowed to make changes and save those changes permanently. The level of access that you are given when you open an object depends on the specific object being opened, the access level you request, the service provider processing the open call, and your own individual access rights. MAPI allows you to request a type of access by defining the MAPI_MODIFY for read-write access and the MAPI_BEST_ACCESS flag for the highest level of access that the service provider can allow. Remember that you are only requesting a particular level of access; the final say is up to the service provider. That is, if a service provider does not allow changes to be made to its objects, setting the MAPI_MODIFY flag will never have an effect.

About Object Relationships

MAPI objects are related in several different ways. Some objects have a hierarchical relationship. The association between message stores, folders, messages, and attachments is an example of this type of relationship. A message store contains folders which contain messages which contain attachments. Other objects have a sibling relationship, where the objects share properties and behavior. For example, there are two types of container objects: folders and address book containers. Both of these objects allow users to set search criteria, to open sub-objects (that is, objects contained within them), and to retrieve two types of tables: a contents table and a hierarchical table.

Figure 1 below shows hierarchical and sibling relationships of some of the MAPI objects from the client's perspective. The session object is at the top of the tree because it is through the session that the client gains access to all other objects. The next level includes the message stores table, a table object that lists properties for all of the message store providers in the current session, and the MAPI address book, a composite object that supplies access to all of the address book providers. The message stores table and address book are used to access the objects implemented by particular service providers, shown next, in hierarchical order.

{ewc msdncd, EWGraphic, group10831 0 /a "SDKEX.bmp"}

Figure 1. Sibling and Hierarchical Relationships Between Objects.

Another kind of relationship involves inheritance, the process by which one object acquires the data and methods of another object, enabling objects to be built as a collection of standard components. All COM objects support inheritance to some degree. Interfaces implemented by COM objects inherit from a single interface called **IUnknown**. Some interfaces inherit directly from **IUnknown**; other interfaces inherit from **IUnknown** through another interface. Inheritance enables an object's user to call methods in both the interfaces the object supports directly and the interfaces it inherits as if they belonged to the same interface. To the user, the difference is transparent.

For example, Figure 2 below shows a client application making two calls to a MAPI folder object, an object that supports the **IMAPIFolder** interface. **IMAPIFolder** inherits from the property object interface, **IMAPIProp**, which inherits from **IUnknown**. One of the calls, **GetProps**, belongs to the **IMAPIProp** interface. The other call, **CreateMessage**, belongs to **IMAPIFolder**. However, the client application doesn't know or care that these calls belong to different interfaces. The client simply directs the calls to the folder object.

Figure 2. Inheritance Between Objects.

Implementing MAPI Objects

MAPI objects are implemented using C data structures or C++ classes, depending on the language and the API set you are using for your client application or service provider. Service providers are written with C or C++ and the Extended MAPI API; client applications can use any of the three supported languages and the four supported API sets. If you are creating a new client application or service provider with Extended MAPI and you are vacillating between languages, use C++. Why? C++ is better because Extended MAPI is an object oriented technology and C++ lends itself more readily to object oriented development. Your code will be simpler and more straightforward making maintenance easier. Also, the syntax descriptions in *The MAPI Programmer's Reference* and the sample code in this volume are in C++. However, if you are proficient only in C and lack the time and resources to learn C+ +, don't worry. C clients and service providers work just fine.

When you define a MAPI object in C, you need to explicitly include a pointer to a virtual table, or Vtable, as the first member of your object. The Vtable is an array of ordered function pointers; there is one pointer for every method in each interface supported by your object. The order of the pointers must follow the order of the methods in the interface specification. 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 for you by the compiler. In C, it is not.

Figure 3 illustrates how this works. An object wanting to use an interface retrieves a pointer to the interface. This interface pointer points to the Vtable pointer stored as the first member of the object implementing the interface. The Vtable pointer points to the actual Vtable and each member of the Vtable pointers to code implementing one of the methods in the interface..

{ewc msdncd, EWGraphic, group10831 1 /a "SDKEX.bmp"}

Figure 3. Internal Representation of a MAPI Object.

Now for an example to help clarify. Suppose you're defining an object that supports the fictional IFootBall interface. IFootBall, like all good MAPI interfaces, derives from **IUnknown**. IFootBall has three methods that are defined in the following order: Fumble, Pass, GetTouchDown.

You would define this object in C++ as follows. The **IUnknown** methods appear first followed by the IFootBall methods. Data and methods specific to the object, such as the reference count, follow.

```
class   CFootBall : public IFootBall
{
public:
// IUnknown virtual methods
    HRESULT QueryInterface
                (REFIID                   riid,
                 LPVOID *                 ppvObj);
    ULONG    AddRef();
    ULONG    Release();

// IFootBall virtual methods
    HRESULT    Fumble();
    HRESULT    Pass();
    HRESULT    GetTouchDown();

// Other methods specific to this object
    BOOL IsGameOver();
    ULONG GetScore();

// Constructors and destructors
public :
    CFootBall ();
    ~CFootBall ();
// Data members
private :
    LONG                m_cRef;
    CSportObj *         m_pSportObj ;
};
```

The C implementation is more complex because of the added Vtable layer. The FOOTBALL object contains a pointer to its Vtable and private data and methods. The Vtable is initialized with the addresses of the implementation of these methods in the order specified by the interface.

```
typedef struct _FOOTBALL
{
    FOOTBALL_Vtbl FAR *lpVtbl;

//  Other methods specific to this object
    BOOL IsGameOver();
    ULONG GetScore();

//  Private data members
    LONG                m_cRef;
    CSportObj *         m_pSportObj ;
} FOOTBALL;

//  Vtable data structure
FOOTBALL_Vtbl vtblFTBALL =
{
     IFootBall_QueryInterface,
     IFootBall_AddRef,
     IFootBall_Release,
     IFootBall_Fumble
     IFootBall_Pass,
     IFootBall_GetTouchDown,
}

//  Code to initialize Vtable and data members
     FOOTBALL * lpFootBall;
     lpFootBall->lpVtbl = &vtblFTBALL;
     lpFootBall->m_cRef = 1;
     lpFootBall->m_pSportObj = lpSport;
```

Using MAPI Objects

Using an object means calling methods in one of the interfaces implemented by the object. You make interface method calls indirectly through a pointer to the interface implementation. Interface pointers are returned by the **QueryInterface** method in **IUnknown**, by a method provided by a different object, or by an API function provided by MAPI. Many of the methods and API functions for instantiating objects return an interface pointer as an output parameter.

If you are using objects written in C++, you can use the pointer directly to make a method call as follows:

```
lpObj->Method(parm1, parm2, ...parmX);
```

However, in C, you would need to call the method indirectly, through the Vtable pointer, as follows:

```
lpObj->lpVtbl->Method(lpObj, parm1, parm2, ...parmX);
```

Notice the addition of the object pointer as the first parameter to the method call. The C++ compiler implicitly adds a pointer to the object you are calling as the first parameter to every method call. This pointer is known as the *this* pointer. Since in C there is no such pointer; you must add it as the first parameter in the parameter list for every method that you call.

Identifying MAPI Objects

Many MAPI objects are identified by what is known as an *entry identifier*. Entry identifiers are data structures that contain two values: one that uniquely identifies the object instance itself and one that uniquely identifies the service provider that owns the object. The value that identifies the provider is known as a UID, or unique identifier. Service providers inform MAPI of their UID when they are initially loaded.

When clients make a call to MAPI or a service provider to request an operation on an object, they pass the entry identifier of the object as one of the call's parameters. The recipient of the call uses the entry identifier to determine who is responsible for the object and can handle the request.

Entry identifiers may be short-term or long-term. Short-term entry identifiers are valid only for the life of the session or until the memory for them is explicitly freed. Long-term entry identifiers are stored on disk, enabling them to last from session to session.

About MAPI Objects

Now it is time to move into the object oriented world of programming with Extended MAPI. Programming with Extended MAPI is similar to programming with any other object oriented API. With Extended MAPI, you will implement some objects and use other objects. Where Extended MAPI differs from the standard object oriented methodology is in its special use of objects.

Extended MAPI, as mentioned previously, is based on the Component Object Model (COM) introduced with OLE. The Component Object Model is a specification that defines an environment for sharing code and data across processes. COM *objects* are unique because their public methods belong to one or more MAPI *interfaces*, or collections of related functions. Outsiders obtain access to COM objects only through pointers to these interfaces. MAPI interfaces define how objects behave and operate within the COM world.

MAPI defines many types, or classes, of objects, each type characterized by the interfaces it supports. Depending on whether you are creating a client application or a service provider, you will implement some objects and use others. When you implement an object, you provide the code for its interface methods. When you use an object, you make calls to the methods.

The following table defines the types of MAPI objects that clients and service providers work with in a typical session. You may recognize some of these types from the architectural discussion in Chapter 1.

MAPI Object Type	**Description**
Property	An object that describes a set of attributes for a MAPI object, these are the object's private data members
Table	An object that provides access to an object's private data in row and column format, similar to a database table
Container	An object that provides access to a collection of objects
Message	An object that can be sent to one or more recipients with a messaging system
Folder	An object that acts as a container for messages
Attachment	An object that contains additional data, such as a file, to be associated with a message
Message Store	An object that acts as a database of messages, organized hierarchically
Mail User	An object that describes an individual recipient of a message.
Distribution List	An object that describes a grouping of individual message recipients
Address Book	An object that acts as a database of recipients
Profile Section	An object that describes a particular message service or service provider
Session	An object created by the MAPI subsystem and passed to clients to represent a connection to underlying messaging systems
Support	An object created by the MAPI subsystem and passed to service providers to help them take care of client requests
Status	An object that is used to obtain access to the state of a session resource, such as the MAPI spooler or

	a service provider
Advise Sink	An object that provides a callback to be invoked when a particular object experiences a change or an error occurs
Logon	An object created by a service provider and passed to the MAPI subsystem for handling event notification and client requests
Provider	An object created by a service provider and passed to the MAPI subsystem for handling provider logon and shut down.
Form Server	An object for managing a user's interaction with a custom dialog box for entering data
Form Container	An object that provides access to information about a custom dialog box

Some of these types of objects have single implementations while other types have multiple implementations. For example, when a user logs on to MAPI, a session object is created that represents a connection with the messaging systems accessed through the message services listed in the profile. Each session has only one session object.

Property objects, on the other hand, are instantiated many times during the course of a session. Property objects describe the private data of other MAPI objects. For example, every message object includes in its private data one or more recipients. This data is accessible only through the interface methods implemented in the property object. Properties are explained in more detail in Chapter 7, "About MAPI Properties".

Table objects are another example of frequently implemented MAPI objects. Table objects make it possible to view properties in row and column format. Each row in the table represents a MAPI object and each column represents the value of one of the object's properties. There are many types of tables, named for the type of objects that are represented in the tables' rows. For example, the service providers table lists properties for all of the service providers in the current session. The status table lists the properties for status objects. Status objects are implemented by MAPI and service providers to provide access to status information. For more detail about MAPI tables, see Chapter 9, "Using Tables."

Some objects have the concept of access, meaning that when you open an object, you are granted a particular level of access. Only if you are granted read-write access will you be allowed to make changes and save those changes permanently. The level of access that you are given when you open an object depends on the specific object being opened, the access level you request, the service provider processing the open call, and your own individual access rights. MAPI allows you to request a type of access by defining the MAPI_MODIFY for read-write access and the MAPI_BEST_ACCESS flag for the highest level of access that the service provider can allow. Remember that you are only requesting a particular level of access; the final say is up to the service provider. That is, if a service provider does not allow changes to be made to its objects, setting the MAPI_MODIFY flag will never have an effect.

About Object Relationships

MAPI objects are related in several different ways. Some objects have a hierarchical relationship. The association between message stores, folders, messages, and attachments is an example of this type of relationship. A message store contains folders which contain messages which contain attachments. Other objects have a sibling relationship, where the objects share properties and behavior. For example, there are two types of container objects: folders and address book containers. Both of these objects allow users to set search criteria, to open sub-objects (that is, objects contained within them), and to retrieve two types of tables: a contents table and a hierarchical table.

Figure 1 below shows hierarchical and sibling relationships of some of the MAPI objects from the client's perspective. The session object is at the top of the tree because it is through the session that the client gains access to all other objects. The next level includes the message stores table, a table object that lists properties for all of the message store providers in the current session, and the MAPI address book, a composite object that supplies access to all of the address book providers. The message stores table and address book are used to access the objects implemented by particular service providers, shown next, in hierarchical order.

{ewc msdncd, EWGraphic, group10832 0 /a "SDKEX.bmp"}

Figure 1. Sibling and Hierarchical Relationships Between Objects.

Another kind of relationship involves inheritance, the process by which one object acquires the data and methods of another object, enabling objects to be built as a collection of standard components. All COM objects support inheritance to some degree. Interfaces implemented by COM objects inherit from a single interface called **IUnknown**. Some interfaces inherit directly from **IUnknown**; other interfaces inherit from **IUnknown** through another interface. Inheritance enables an object's user to call methods in both the interfaces the object supports directly and the interfaces it inherits as if they belonged to the same interface. To the user, the difference is transparent.

For example, Figure 2 below shows a client application making two calls to a MAPI folder object, an object that supports the **IMAPIFolder** interface. **IMAPIFolder** inherits from the property object interface, **IMAPIProp**, which inherits from **IUnknown**. One of the calls, **GetProps**, belongs to the **IMAPIProp** interface. The other call, **CreateMessage**, belongs to **IMAPIFolder**. However, the client application doesn't know or care that these calls belong to different interfaces. The client simply directs the calls to the folder object.

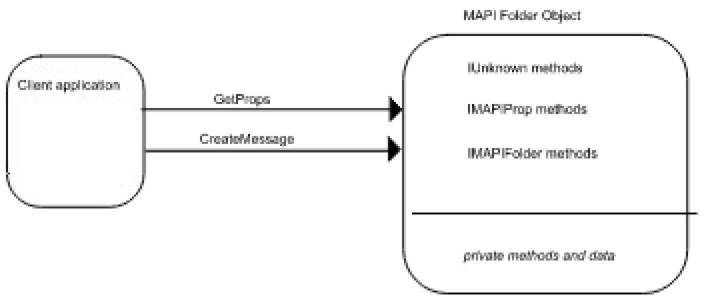

Figure 2. Inheritance Between Objects.

Implementing MAPI Objects

MAPI objects are implemented using C data structures or C++ classes, depending on the language and the API set you are using for your client application or service provider. Service providers are written with C or C++ and the Extended MAPI API; client applications can use any of the three supported languages and the four supported API sets. If you are creating a new client application or service provider with Extended MAPI and you are vacillating between languages, use C++. Why? C++ is better because Extended MAPI is an object oriented technology and C++ lends itself more readily to object oriented development. Your code will be simpler and more straightforward making maintenance easier. Also, the syntax descriptions in *The MAPI Programmer's Reference* and the sample code in this volume are in C++. However, if you are proficient only in C and lack the time and resources to learn C++, don't worry. C clients and service providers work just fine.

When you define a MAPI object in C, you need to explicitly include a pointer to a virtual table, or Vtable, as the first member of your object. The Vtable is an array of ordered function pointers; there is one pointer for every method in each interface supported by your object. The order of the pointers must follow the order of the methods in the interface specification. 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 for you by the compiler. In C, it is not.

Figure 3 illustrates how this works. An object wanting to use an interface retrieves a pointer to the interface. This interface pointer points to the Vtable pointer stored as the first member of the object implementing the interface. The Vtable pointer points to the actual Vtable and each member of the Vtable pointers to code implementing one of the methods in the interface..

{ewc msdncd, EWGraphic, group10832 1 /a "SDKEX.bmp"}

Figure 3. Internal Representation of a MAPI Object.

Now for an example to help clarify. Suppose you're defining an object that supports the fictional IFootBall interface. IFootBall, like all good MAPI interfaces, derives from **IUnknown**. IFootBall has three methods that are defined in the following order: Fumble, Pass, GetTouchDown.

You would define this object in C++ as follows. The **IUnknown** methods appear first followed by the IFootBall methods. Data and methods specific to the object, such as the reference count, follow.

```
class   CFootBall : public IFootBall
{
public:
// IUnknown virtual methods
    HRESULT QueryInterface
                (REFIID                 riid,
                 LPVOID *               ppvObj);
    ULONG    AddRef();
    ULONG    Release();

// IFootBall virtual methods
    HRESULT    Fumble();
    HRESULT    Pass();
    HRESULT    GetTouchDown();

// Other methods specific to this object
    BOOL IsGameOver();
    ULONG GetScore();

// Constructors and destructors
public :
    CFootBall ();
    ~CFootBall ();
// Data members
private :
    LONG                    m_cRef;
    CSportObj *             m_pSportObj ;
};
```

The C implementation is more complex because of the added Vtable layer. The FOOTBALL object contains a pointer to its Vtable and private data and methods. The Vtable is initialized with the addresses of the implementation of these methods in the order specified by the interface.

```
typedef struct _FOOTBALL
{
    FOOTBALL_Vtbl FAR *lpVtbl;

//  Other methods specific to this object
    BOOL IsGameOver();
    ULONG GetScore();

//  Private data members
    LONG                m_cRef;
    CSportObj *         m_pSportObj ;
} FOOTBALL;

//  Vtable data structure
FOOTBALL_Vtbl vtblFTBALL =
{
     IFootBall_QueryInterface,
     IFootBall_AddRef,
     IFootBall_Release,
     IFootBall_Fumble
     IFootBall_Pass,
     IFootBall_GetTouchDown,
}

//  Code to initialize Vtable and data members
     FOOTBALL * lpFootBall;
     lpFootBall->lpVtbl = &vtblFTBALL;
     lpFootBall->m_cRef = 1;
     lpFootBall->m_pSportObj = lpSport;
```

Using MAPI Objects

Using an object means calling methods in one of the interfaces implemented by the object. You make interface method calls indirectly through a pointer to the interface implementation. Interface pointers are returned by the **QueryInterface** method in **IUnknown**, by a method provided by a different object, or by an API function provided by MAPI. Many of the methods and API functions for instantiating objects return an interface pointer as an output parameter.

If you are using objects written in C++, you can use the pointer directly to make a method call as follows:

```
lpObj->Method(parm1, parm2, ...parmX);
```

However, in C, you would need to call the method indirectly, through the Vtable pointer, as follows:

```
lpObj->lpVtbl->Method(lpObj, parm1, parm2, ...parmX);
```

Notice the addition of the object pointer as the first parameter to the method call. The C++ compiler implicitly adds a pointer to the object you are calling as the first parameter to every method call. This pointer is known as the *this* pointer. Since in C there is no such pointer; you must add it as the first parameter in the parameter list for every method that you call.

Identifying MAPI Objects

Many MAPI objects are identified by what is known as an *entry identifier*. Entry identifiers are data structures that contain two values: one that uniquely identifies the object instance itself and one that uniquely identifies the service provider that owns the object. The value that identifies the provider is known as a UID, or unique identifier. Service providers inform MAPI of their UID when they are initially loaded.

When clients make a call to MAPI or a service provider to request an operation on an object, they pass the entry identifier of the object as one of the call's parameters. The recipient of the call uses the entry identifier to determine who is responsible for the object and can handle the request.

Entry identifiers may be short-term or long-term. Short-term entry identifiers are valid only for the life of the session or until the memory for them is explicitly freed. Long-term entry identifiers are stored on disk, enabling them to last from session to session.

About MAPI Interfaces

MAPI interfaces, as stated earlier, are collections of related functions. Interfaces provide a binary standard for object interaction and are what is known in the C++ world as abstract base classes. This means that only the prototypes of the methods are defined; the interface name, method names, and method parameters are specified, but there are no actual implementations. It is up to the object inheriting the interface to provide its own implementation. Sometimes this object is a service provider object; sometimes it is a client object; often it is an object provided with the MAPI subsystem.

MAPI interfaces define a contract between the object implementing the interface and the object using the implementation. This contract can be loose or tight, depending on the method. A loose contract allows a method to return MAPI_E_NO_SUPPORT, indicating no functionality. A tight contract disallows the MAPI_E_NO_SUPPORT return value; all implementations of the method must provide some functionality. For example, the **IMAPIStatus::FlushQueues** method makes sense only for transport providers. However, message store providers also implement status objects and the **IMAPIStatus** interface, using a loose contract. Message store providers return MAPI_E_NO_SUPPORT in their **FlushQueues** implementation.

About Interface Relationships

As mentioned in the discussion about objects, all MAPI interfaces ultimately derive from the base COM interface, **IUnknown.** The MAPI inheritance hierarchy is fairly straightforward. There are only three base interfaces, or interfaces that other interfaces can inherit: **IUnknown**, **IMAPIProp**, and **IMAPIContainer**. Most interfaces are direct descendants of **IUnknown**, some inherit from **IMAPIProp**, and a few inherit from **IMAPIContainer**.

{ewc msdncd, EWGraphic, group10832 2 /a "SDKEX_06.bmp"}

Figure 1. MAPI Interface Hierarchy

About IUnknown

IUnknown, as the top of the interface hierarchy, is implemented in all MAPI objects. **IUnknown** provides mechanisms for obtaining an interface implementation and controlling its lifetime with three methods: **QueryInterface**, **AddRef**, and **Release**. **QueryInterface** enables potential users of an object to find out if the object supports a particular interface. If the object does support the interface, **QueryInterface** returns a pointer to it. **AddRef** and **Release** make sure that an object is not freed while it is still being used, keeping a tally known as a reference count.

The following illustration shows how **QueryInterface** works. The person on the left is the caller; the person on the right is the object with the implementation of **QueryInterface**. The caller asks for an interface that supports specific functionality, in this case, baseball. Because the object doesn't support the interface, the caller receives nothing in return. However, when the caller asks a second time for a supported interface (that is, football), the object returns an interface pointer that can be the basis for further interaction between the caller and the object. In this example, the ball represents the interface pointer.

{ewc msdncd, EWGraphic, group10832 3 /a "SDKEX.bmp"}

Figure 2. How the QueryInterface method works.

Now, to translate the conceptual into code. The following example shows how **QueryInterface** would be implemented in C++ for an object that supports the fictional MAPI Football interface. To provide a more interesting example, the IFootball interface inherits from the ISports interface which inherits from **IUnknown**, rather than directly inheriting from **IUnknown**. Notice that the same pointer can be returned for any of the interfaces.

```
HRESULT CFootball::QueryInterface (REFIID   riid,
                                   LPVOID * ppvObj)
{
    // Always set out parameter to NULL
    *ppvObj = NULL;

    // Is the caller asking for one of my interfaces?
    if (riid == IID_IFootball || riid == IID_ISports ||
        riid == IID_IUnknown)
    {
        // Yes - return pointer and increment reference count
        *ppvObj = (LPVOID)this;
        AddRef();
        return NOERROR;
    }

    // No - return error.
    return E_NOINTERFACE;
}
```

The **AddRef** and **Release** methods control the reference count of an object. The reference count defines the object's lifetime. Without reference counting, the object would never know when it was safe to free its memory. As long as the reference count is greater than zero, the object's memory will not be freed.

There are a few basic reference counting rules. All methods or API functions that return interface pointers 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. The following code samples show how to implement **AddRef** and

Release for the CFootball object. For a detailed description of COM reference counting, refer to the OLE 2.0 Programmer's Reference.

```
//
ULONG CFootball::AddRef()
{
    // Increment the object's internal counter
    ++m_cRef;
    return m_cRef;
}

ULONG CFootball::Release()
{
    // Decrement the object's internal counter
    ULONG ulRefCountT = --m_cRef;
    // Deallocate if ref count is 0. Destructor frees memory
    if (0 == m_cRef)
    {
        delete this;
    }
    return ulRefCountT;
}
```

Selecting Interfaces to Implement

Because MAPI provides so many interfaces, it can be difficult to determine exactly which interfaces to implement and which interfaces to use. Typically the type of MAPI component you are writing and the feature set it provides determines the set of interfaces you will need to implement and use.

Service providers implement many more interfaces than clients, who are primarily users of interfaces, rather than implementors. The typical client application will only have to implement one interface: **IMAPIAdviseSink**. **IMAPIAdviseSink** is used to send event notifications. If your client also works with custom forms, you'll have to implement these additional interfaces: **IMAPIFormAdviseSink**, **IMAPIMessageSite**, **IMAPIViewAdviseSink**, and **IMAPIViewContext**.

Address book, message store, and transport providers all implement their own flavor of provider interface and logon interface. The provider interfaces are called **IABProvider**, **IMSProvider**, and **IXPProvider**, respectively; the logon interfaces are called **IABLogon**, **IMSLogon**, and **IXPLogon**. All of the provider interfaces are used in the same way as the logon interfaces. MAPI uses the provider interfaces to start up and shut down a service provider and the logon interfaces to handle requests from clients. However, because there are tasks that are specific to the type of service provider, the provider and logon interfaces cannot be identical. Refer to the discussion in "Considerations for Service Providers" or to the *MAPI Programmer's Reference* for details about the implementation of these interfaces.

Address book providers also implement **IABContainer**, **IDistList**, **IMAPITable**, and **IMailUser** and the interfaces a few of these interfaces derive from: **IMAPIContainer** and **IMAPIProp**.

Message store providers also implement the base interfaces **IMAPIContainer** and **IMAPIProp** as well as **IMsgStore, IMAPIFolder, IMessage, IAttach, IMAPITable,** and **IMAPIStatus.**

Transport providers typically implement only two additional interfaces: **IMAPIProp,** and **IMAPIStatus.** Remote transport providers also implement **IMAPIFolder** and **IMAPITable.**

A few miscellaneous interfaces exist to support custom components such as form servers, message class handlers, and message hook providers. **IMAPIForm** and **IPersistMessage** are implemented by form servers; **ISpoolerHook** is implemented by hook providers.

Selecting Interfaces to Use

The group of MAPI interfaces that you should use depends on the type of component that you are writing and the features you are including. Most clients and service providers frequently use **IMAPITable** and **IMAPIProp** and occasionally use **IMAPIProgress.** This is where the commonality between the two component types ends. Clients use a great many different interfaces, some implemented by MAPI and others implemented by service providers. MAPI provides clients with the **IAddrBook** interfaces for integrated address book access, the **IMAPISession** interface for general session access, and the **IMAPIStatus** interface for monitoring session status.

Clients can interact with service providers either indirectly, through **IMAPISession**, or directly through a variety of interfaces implemented by particular service providers. To make direct contact with address book providers, clients call the **IABContainer**, **IDistList**, and **IMailUser** interfaces. To access a message store provider directly, clients call **IAttach**, **IMAPIFolder**, **IMessage**, and **IMsgStore**. Clients also interact directly with service providers through their implementation of **IMAPIStatus**.

Clients that implement special features use other interfaces. For example, clients that perform profile and message service configuration use **IProfAdmin**, **IMsgServiceAdmin**, and **IProviderAdmin** and clients that display custom forms use **IMAPIForm**, **IPersistMessage**, **IMAPIFormContainer**, **IMAPIFormInfo**, and **IMAPIFormMgr**.

Service providers typically use only a few interfaces. They use **IMAPISupport**, a large interface implemented by MAPI to help service providers handle requests from clients. **IMAPISupport** has many methods, some of which are applicable to all service providers and others that apply only to specific types. See the *MAPI Programmer's Reference* for detailed information about the methods in **IMAPISupport**. Service providers that send event notifications also use **IMAPIAdviseSink**.

Some service providers use **ITableData** and **IPropData**, two utility interfaces implemented by MAPI. **ITableData** allows service providers to manage the underlying data of a table while **IPropData** allows service providers to set object and property access. Both of these interfaces are optional; service providers are not required to use them.

Transport providers that support the Transmission-Neutral Encapsulation Format (TNEF) for transferring properties use the **ITnef** interface supplied by MAPI.

Preliminary Outline

The guide to writing Extended MAPI clients and providers is in its preliminary state of conception. Many of the topics have not yet been written; other topics have not yet been reviewed for technical accuracy. This outline is included to show readers how the MAPI Programmer's Guide will be organized and to give some idea of future content. The chapters that are presented in this release are not the final versions of this material; there may be inaccuracies and missing information. Future builds of the MAPI SDK Help will include up-to-date and technically correct Extended MAPI guide information as it becomes available.

About MAPI Properties

There are two ways to define MAPI properties. First, there's the universal way as specified in Webster's Dictionary and introduced in the previous chapter "About Objects." Webster's Dictionary defines a property as an attribute common to all members of a class. In the MAPI environment, "members of a class" translates to instances of a MAPI object type. A property is used to describe a MAPI object of a particular type. For example, the message body, list of recipients, and message subject are properties of every MAPI message object.

The other definition is the object oriented COM way of looking at MAPI properties. MAPI properties are implemented as MAPI objects themselves. Like all MAPI objects, property objects implement an interface that derives from the **IUnknown** interface, in this case the **IMAPIProp** interface. However, MAPI property objects are special because they are never used by themselves as stand alone objects. Instead they are included in the objects they describe. That is, every MAPI message object includes a property object to provide access to its properties; included among them are the message body, list of recipients, and message subject.

Objects that include property objects implement interfaces that inherit from **IMAPIProp**. Most of the interfaces that inherit from **IMAPIProp** have their own set of unique methods like the **IMessage** interface, the interface implemented by message objects. However, some interfaces do not have any additional methods. An object that implements one of these interfaces is basically just a property object that offers access to its own set of properties. The only methods available to you with this type of object are the methods of **IMAPIProp** and **IUnknown**. An example of this special type of property object is the attachment object. Attachment objects implement the **IAttach** interface, which has no unique methods of its own. Client applications use attachment objects to access the properties of message attachments, such as the actual data and filename of the attachment.

The methods of **IMAPIProp** provide access not only to the property's value, but also to data that describes the value. Every property has an associated type and identifier, a combination known as the property's tag.

If you are familiar with C++, you could equate a C++ object's private data to a set of properties. An external user of a C++ object can only access the private data through one or more methods that the object defines as public. An external user of a MAPI object that supports a set of MAPI properties can only access those properties through one or more methods in its **IMAPIProp** implementation.

Figure 1 illustrates the structure of a MAPI message object, a typical example of an object that supports properties. Message objects implement the methods of the **IMessage**, **IMAPIProp,** and **IUnknown** interfaces. These methods are all public; message objects may or may not also have private method implementations. The properties and associated property tag information is kept private. In spite of the fact that there are three separate interfaces, from the client's perspective, there is only one set of methods. The pointer that the client gets back after making a call to instantiate a message object can be used to call any of the methods in any of the interfaces.

{ewc msdncd, EWGraphic, group10834 0 /a "SDKEX.bmp"}

Figure 1 Layout of an object that supports properties.

Properties are used extensively throughout the MAPI architecture. MAPI defines a large set of properties, assigning each property a name, an identifier, and a type. Setting the value in some cases is up to the service provider, in some cases is up to the client, and in some cases it is up to MAPI. Some properties are read-only; once they have an initial value they cannot change. Other properties are read-write. The ability to change a read-write property may be based on your access to the property. That is, some properties are read-only to clients and read-write to service providers. Still other properties are what is known as computed; service providers set their value from the value of another property. Clients, or other users of the property, have read-only access to computed properties.

Because the MAPI architecture is open and extensible, it is possible to define new, custom properties that are accessed in the same way that the standard, predefined properties are accessed. It may be desirable to define new properties to support a custom message class or to supplement the standard set for an existing message class.

Property data is displayed using another type of MAPI object, the table. MAPI tables consist of rows that represent individual objects supporting the properties and columns that contain property values. For example, the status table is used to present session status information to clients. Each service provider contributes a row to the table. Each column in the row contains a value for a property that the service provider supports that is relevant to the status of the session.

Some properties apply to a variety of MAPI objects. In fact, any mail user or distribution list property can also appear on a message. To provide an alternate example, MAPI defines a property known as a display name which is a string that is used to represent an object in a dialog box. Folders, message stores, service providers, forms, and mail user and distribution list recipients all support the display name property, PR_DISPLAY_NAME. Other properties are specific to one type of object, such as the attachment method property, PR_ATTACH_METHOD. Attachment method is a property that describes the method by which data is associated with a message. This property applies only to MAPI attachment objects.

Properties can be short term, existing only for the duration of the current session, or long term, stored persistently and available throughout multiple sessions. MAPI objects can store their properties either with the data of the object or in the profile.

This chapter is divided into two main sections. The first section focuses on the data and methods of MAPI property objects. The second section goes into detail about implementing and using MAPI properties in your client or service provider.

Property Value Data Structures

The property value, or **SPropValue**, data structure is used to describe each MAPI property. The **SPropValue** structure is made up of three pieces: the property tag, some bytes for alignment, and the property value.

The property tag 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 alignment field is designed for computers that require 8-byte alignment for 8-byte values. You can allocate an array of property values on an 8-byte boundary and cause the array to automatically have the proper byte alignment. If you write code on such computers, you should use memory allocation routines that allocate the **SPropValue** arrays on 8-byte boundaries.

The property value is a union of type PV. Depending on the type of property, its value varies. Some properties are single integers, some are character strings, some can be MAPI objects themselves.

The **SPropValue** data structure is defined as follows.

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

This section discusses each of the fields in the **SPropValue** structure in detail, describing their purpose and organization. More information about how the structure is used and implemented is covered in the "Working with Properties" sections, later in this chapter.As mentioned above, each property has a type, an identifier, and a value. These pieces of data are stored in an **SPropValue** data structure, which is defined as follows:

About Property Tags

Property tags, as mentioned above, they contain a unique identifier and a type. They are the first member of the **SPropValue** data structure and are structured as follows:

{ewc msdncd, EWGraphic, group10834 1 /a "SDKEX_10.bmp"}

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. MAPI's 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.

These next sections describe property identifiers and types in detail.

About Property Identifers

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.

The range of property identifiers runs from 0001 through 3FFF. 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. Beyond 8000 is the range for what is known as named properties, or properties that have a name associated with their identifier. Service providers use named properties to customize their property set.

Service providers can use the range 0x67F0 thru 0x67FF to define secure properties. Secure properties have values that are hidden and encrypted, a useful way for storing passwords and other information that requires additional protection.

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 local value for the current session. Another messaging system would not be able to use the non-transmittable properties. For example, the property that says when a message has been received is specific to the message. The concept of transmittable properties applies primarily to transport providers. To easily 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 who is responsible for the properties in each range.

Range	Owner	Type of Property
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 providers	Recipient properties
6000 - 65FF	User	Non-transmittable message properties
6600 - 67FF	Any service provider	Internal non-transmittable properties invisible to clients
6800 - 7BFF	Creators of custom message classes	Message content properties
7C00 - 7FFF	Creators of custom message classes	Non-transmittable message properties.
8000 - FFFE	User	Properties identified only by name, through the methods in **IMAPIProp** for mapping property names to identifiers

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	**Contents**
3000 - 33FF	Common properties, such as display name and entry identifier.
3400 - 35FF	Message store properties
3600 - 36FF	Container properties
3700 - 38FF	Attachment properties
3900 - 39FF	Address book properties
3A00 - 3BFF	Mail user properties
3C00 - 3CFF	Distribution list properties
3D00 - 3DFF	Profile properties
3E00 - 3FFF	Status object properties

About Property Types

Property types represent the format of the property value, the underlying data type. The set of property types defined by MAPI is included in the MAPIDEFS.H header file. These types are represented by constants that follow a similar naming convention as the one for property tags. All property types have two parts; the first part is the prefix "PT_" and the second part is a string describing the actual type, such as LONG or STRING8.

Unlike property identifiers, new property types cannot be created. New properties must be defined using the existing types.

Property types may be single or multi-valued. A single value property contains one value of its type such as a single integer, Boolean, character string, sized array of bytes, or data structure.

A multi-value property contains multiple values of the same type. To represent a multi-value property, the flag MV_FLAG is combined with the property type in a logical OR operation. The following diagram illustrates the structure of a multi-valued property of type 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, group10834 2 /a "SDKEX_12.bmp"}

Figure 2 A Multi-valued property

MAPI recommends that service providers and clients support both single and multi-valued properties because doing so enables a wider variety of MAPI components to be used.

Some of the property types have special meaning and use. The property type PT_UNSPECIFIED is used for properties whose type is unknown. You can specify this type in calls to **IMAPIProp::GetProps** to retrieve these properties and find out their types. The correct type will replace the PT_UNSPECIFIED.

The PT_OBJECT type represents a property that is a separate object. The hierarchy table property of a message store, the PR_CONTAINER_HIERARCHY property, is a MAPI table object. Clients can access this table object property either by calling **IMAPIProp::OpenProperty** or **IMAPIContainer::GetHierarchyTable**.

There are three types of string properties: PT_STRING8 for 8 bit ANSI character strings, PT_UNICODE for double byte character strings, and PT_TSTRING for representing either of the first two types. String properties are available in the character set appropriate to their user's environment. For example, if the user is running in a Unicode environment, all PT_TSTRING properties become type PT_UNICODE. If the user is running in an environment with single-character (ANSI) strings, all PT_TSTRING properties become type PT_STRING8. Service providers that can support both Unicode environments and ANSI environments can achieve more widespread distribution than providers who cannot. Double byte character set strings terminate with a single 8-bit byte and are characterized as PT_STRING8.

String properties can be quite large, making it impossible to retrieve their values using the standard **GetProps** method. The body of a message, stored in the PR_BODY property, can be one of the largest string properties. In fact, some service providers this property to 4096 bytes, but other service provider implementations have no such limit. It is necessary to use the **IStream** interface to retrieve message bodies and other large string properties. To access an **IStream** pointer, call the **IMAPIProp::OpenProperty** method. To determine the size of a large string property, call **IStream::Seek** with the flag STREAM_SEEK_END. Be warned that this is not a fast operation.

The following code shows how to access the contents of a file using a stream object and setting

a large string property using OpenProperty.

The **OpenStreamOnFile** function opens and initializes a stream object with the contents of a file.

```
    LPSTREAM pSrcStrm, pDstStrm;
    HRESULT hResult = OpenStreamOnFile (MAPIAllocateBuffer,
                                MAPIFreeBuffer,
                                STGM_CREATE | STGM_READWRITE |
                                STGM_DIRECT | STGM_SHARE_EXCLUSIVE,
                                pszDestFile, NULL, &pDstStrm);
    if (SUCCEEDED(hResult))
    {
        // Open the source stream in the attachment object
        hResult = m_pAttachObj->OpenProperty (PR_ATTACH_DATA_BIN,
                                              (LPIID)&IID_IStream,
                                              0,
                                              0,
                                              (LPUNKNOWN *)&pSrcStrm);
        if (SUCCEEDED(hResult))
        {
            // How big is it?
            STATSTG StatInfo;
            pSrcStrm->Stat (&StatInfo, STATFLAG_NONAME);
            // Copy it all
            hResult = pSrcStrm->CopyTo (pDstStrm, StatInfo.cbSize, NULL,
NULL);
            pSrcStrm->Release();
        }
        pDstStrm->Release();
    }
```

When large string properties are included in a table, the entire value might not be visible. This is because MAPI allows table implementors to truncate these property values at 256 bytes.

The following diagram illustrates a property value structure with all of the potential property types. Notice that not all of the single-valued types have multi-value equivalents.

{ewc msdncd, EWGraphic, group10834 3 /a "SDKEX_11.bmp"}

Figure 3 An SPropValue Property Structure with a List of Valid Property Types

For a comprehensive description of each of these types, refer to the *MAPI Programmer's Reference*.

Property Interfaces

So far, only one MAPI interface for property access has been mentioned - **IMAPIProp**. However, there is another interface, **IPropData**, that enables callers to retrieve and set the access rights for a property and for the object that supports the property and to add to the list of an object's supported properties. **IPropData**, implemented by MAPI, is a utility interface that inherits from **IMAPIProp**. This section discusses how each of these property interfaces are used and provides some tips for implementation that can be used generically by most client applications and service providers.

About IMAPIProp

The **IMAPIProp** interface allows you 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.

Because it is a base interface for so many objects, **IMAPIProp** is one of the most frequently implemented and used interfaces in MAPI. The implementation of its methods varies from object to object. In other words, an implementation of **IMAPIProp** for a profile section object will be different than one for a message. Some objects permit no changes to any of their properties, for example, while other objects allow some of the values of their properties to be modified. Some methods will provide less functionality for one object than for another, returning MAPI_E_NO_SUPPORT or MAPI_E_TOO_COMPLEX in response to unsupported or overly difficult requests. Clients using **IMAPIProp** need to be prepared for all levels of support, and service providers implementing **IMAPIProp** need to know what is expected of them by clients.

About IPropData

The **IPropData** interface provides the ability to retrieve and change the access for a property and its encompassing object. MAPI provides an implementation of **IPropData** that service providers can use by calling the API function **CreateIProp**. The main advantage to using **IPropData** over **IMAPIProp**, besides the additional methods, is that it saves you from having to implement **IMAPIProp**. You can use the implementation provided by MAPI.

The **IPropData** methods are listed in the following table:

Method	Description
HrSetObjAccess	Sets access rights for an object.
HrAddObjProps	Adds object type properties to an object.
HrSetPropAccess	Sets access rights for a property.
HrGetPropAccess	Gets access rights for a property.

HrAddObjProps registers property tags for object type properties (that is, type PT_OBJECT). When clients call this method, they need to check the property problem array to determine if there were problems. 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.

By default, an object has read-write access. That is, if you never call **HrSetObjAccess**, there will be no restrictions on your ability to make changes. Every property in the object will also have read-write access. If you want a read-only object, but you need to add properties to it, start with read-write access. Call **HrAddObjProps** to add the properties and then set the access to read-only by calling **HrSetObjAccess**. If you want an object that has read-write access to some properties and read-only access to others, call **HrSetObjAccess** to set the object's access to read-write and **HrSetPropAccess** to set individual properties to read-only access.

You can also use the **HrSetPropAccess** method 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 flags parameter to this method are positional with respect to the property tag array parameter. Each bit corresponds to a bit in the property tag array.

The **HrGetPropAccess** method for retrieving access rights works in two ways. It will return the access level for individual properties or for all of the object's properties. If a property has been deleted, you won't find it in the property tag array that is returned. If you ask for a deleted property, you will get back zero for the access, meaning that no bit is set.

Working with Properties

Working with MAPI properties is much like working with any piece of data; common operations include reading and writing, saving changes, and copying the data from one object to another. One of the more special operations you as a service provider can perform is to associate a name with a property identifier. Clients and other providers can use the name to refer to the property directly, without regard to its identifier. Another operation, available with other MAPI objects as well, enables the retrieval of detailled information about a prior error.

This section covers in detail these operations from the perspective of both the property implementor and its user.

Retrieving Properties

Retrieving a property involves either making an **IMAPIProp** interface method call or a call to **HrGetOneProp**, an API function provided by MAPI. The **IMAPIProp::GetPropList** method retrieves a list of the properties supported by an object, an array of property tags is returned. The **IMAPIProp::GetProps** method is used most frequently to retrieve a property value structure for one or more small properties.

For large string properties or properties of type PT_OBJECT, **IMAPIProp::OpenProperty** must be used. When you call **OpenProperty**, you ask for access to an interface (other than **IMAPIProp**) that can better retrieve the property value. Callers typically ask for a pointer to an **IStream** interface implementation. **IStream** provides byte level access to data and allows such operations as seek, write, and read.

MAPI provides the **HrGetOneProp** helper function for retrieving one property only at a time. **HrGetOneProp** is useful when the target object exists on the local machine. If the target object is not locally available, avoid using **HrGetOneProp**. The result will be too many remote calls.

When you call **GetProps**, you pass in one or more property tags. A special property tag, PR_NULL, can be specified to indicate that you want to retrieve all of the properties for the object. An optional input parameter is a flag that requests all string properties to be returned in Unicode format. **GetProps** passes back an array of **SPropValue** data structures and a count of the number of structures in the array. A successful call can result in a return value of either S_OK or MAPI_W_ERRORS_RETURNED. The MAPI_W_ERRORS_RETURNED value is used when not all of the properties you requested could be returned. Depending on the implementation of **GetProps**, textual information describing the error is usually placed in the *Value* field of the **SPropValue** data structure instead of the real value for the inaccessible property.

As a property user, you will often want to retrieve a property value. Sometimes all you will have access to is a pointer to a MAPI property object; other times you will also have some property data, such as an identifier. Property values are typically accessed through the **GetProps** method or the **HrGetOneProp** API function. These calls requires as input a property tag. If you only have access to the identifier part of the tag, specify PT_UNSPECIFIED for the type in your call. If you have access to neither the type or the identifier in a property tag, call **IMAPIProp::GetPropList**. If you have a property name, a call to **IMAPIProp::GetIDsFromNames** will return the associated identifier that you can use in your **GetProps** or **HrGetOneProp** call.

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 identifier.

Copying Properties

There are two **IMAPIProp** methods for copying properties between objects. **IMAPIProp**::**CopyTo** copies or moves all of the properties from one object to another; **CopyProps** copies or moves a selected set.

The **CopyTo** method has a facility for specifying individual or object properties that you do not want to include in the copy operation. Individual properties to be excluded are specified in a property tag array; object properties to be excluded are specified in an interface identifier array. One use for this exclusion feature is to avoid copying the properties of an object that are specific to that instance of the object. For example, if you are copying the properties of one message to another message, you will probably want each message to have a unique time stamp for when it was sent. Also, you might want each message to have its own set of file attachments. To avoid copying the date and time of submission, specify PR_MESSAGE_DELIVERY_TIME in the property tag array. To avoid copying the attachments of the first message, specify IID_IAttach in the interface identifier array. If you specify a base interface in the interface identifier array, all of the interfaces that derive from it are also excluded.

Note Do not ever exclude the **IUnknown** interface in your **CopyTo** call. Because **IUnknown** is at the top of the inheritance, excluding it will cause nothing to be copied.

When you are implementing **CopyTo** or **CopyProps**, show a progress dialog box if possible. These operations can take a long time. MAPI provides a progress object that you can use by calling the **IMAPISupport::DoProgressDialog** method. To request a progress dialog box when you are calling one of these methods, pass in the MAPI_DIALOG flag.

The MAPI_DECLINE_OK flag passed to **IMAPIProp::CopyTo** is used mostly when MAPI calls your provider to avoid recursion. This flag gives your provider an opportunity to avoid doing the work and to delegate the work to MAPI. This flag is normally not used by clients.

Sometimes a copy operation is too complicated to handle. Of course, as an implementor, you must try to retrieve as many of your properties as possible from wherever they are stored as quickly as possible. If you cannot complete a request, return MAPI_E_DECLINE_COPY. Callers need to be prepared to receive this return value and take an alternate course of action.

CopyTo passes back a property problem array that sometimes holds error information. If the method succeeds, the array will be empty. Ignore the *ulIndex* member in the **SPPropProblemArray**.

If you're going to be copying properties that are unique to a particular type of object, make sure that your source and destination objects are of the same type. You do not want to inadvertently copy the delivery time property, PR_DELIVER_TIME, to an address book object, for example. MAPI does not prevent clients and service providers from associating properties with objects

The **PropCopyMore** function, which copies a single value at a time, must be used with caution. It is possible to allocate many small blocks of memory, leading to a fragmented memory condition. Consider using **ScCopyProps** which copies values in bulk instead if you can. **ScCopyProps** copies property values that are possibly built from disjointed blocks of memory. After a call to **ScCopyProps**, you will have a new copy of a valid property array. If you need to store this array on disk, you will need to call **ScRelocProps** 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.

Setting Property Data

When you want to set property values, call **IMAPIProp::SetProps** or **HrSetOneProp**. The parameters for **SetProps** are similar to the parameters for **GetProps**, only the caller supplies a property value array containing the new property values.

As an implementor of **SetProps**, you may want to allow your callers to change a property type. You can form a new property tag from an existing identifier and the new property type your callers pass in. For example, address book providers might support changing the home telephone number property from PT_UNICODE to PT_MV_UNICODE when the user adds a second home telephone number.

MAPI provides the **HrSetOneProp** helper function for setting one property at a time. Use **HrSetOneProp** only if you are sure that your target object is local; this function can be expensive when used with remote objects.

Defining New Properties

In spite of the wealth of properties supplied by MAPI for provider use, it may be necessary to define new properties. Service providers can define new properties for their own internal use or for public use by clients and other providers. If you are contemplating defining a new public property for an existing MAPI object or message class, think carefully. One of the primary benefits of using MAPI is that it defines standard identifiers and formats for a large number of message service elements, enabling users to seamlessly mix and match service providers. Service providers that use non-standard properties do not work as well with other service providers. This argument does not apply, of course, to newly created message classes because each message class constitutes a separate name space for message content properties or to properties meant to be used internally.

Remember that when you are defining new properties, you must work with the property types that are already defined. 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 may be stored as ULONG and strings may be stored as PT_STRING8.

For your new property's type, use PT_BINARY to indicate a counted byte array. This property type is handy when you want to extend 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 your application reads data out of such a property, the data is unchanged from how it was stored.

Property identifiers must fall in predefined ranges. Assigning an identifier to your property in the appropriate range helps prevent collisions between properties defined by different vendors or users. If a new property is for a new message class, assign an identifier in the 6800-7BFF range if the property is to be transmitted or in the 7C00-7FFF range if the property is not to be transmitted. Use the named property facility to define new transmittable properties for an existing message class. The identifier range of 8000 and greater is reserved for named properties. Named properties are described in more detail in the next section. If the new provider will be a non-transmittable property to be used internally only, assign an identifier in the 7C00-7FFF range.

Using Named Properties

MAPI provides a facility for assigning names to properties and mapping these names to unique identifiers. As mentioned in the previous section, the named property facility is one way for service providers to define new properties and extend the MAPI property set. After the names of these new properties are published, the properties can be used by clients and other providers without regard to the values of their identifiers. The names can be common names, such as "message address," or unusual names, specific to your service provider. Because each name has a unique identifier associated with it, there is no need to worry about name duplication.

MAPI allows properties to be grouped into property sets, represented by a globally unique identifier, or GUID. There are two property sets that are defined by MAPI: PS_MAPI and PS_PUBLIC_STREAM. Properties defined by MAPI belong to the PS_MAPI property set; properties that are involved in the promotion of document properties belong to the PS_PUBLIC_STREAM property set. Any property belonging to the PS_PUBLIC_STRINGS property set will override a property with the same name in another property set.

The **IMAPIProp** interface methods, **GetNamesFromIDs** and **GetIDsFromNames**, map between property identifiers for named properties and their names. Names are stored in the MAPINAMEID structure either as character strings or as numeric identifiers. The MAPINAMEID structure is used to associate a property name with a globally unique identifier, or GUID, that identifies a property set. The MAPINAMEID structure is defined as follows:

```
typedef struct _MAPINAMEID
{
     LPGUID    lpguid;
     ULONG     ulKind;
     union {
           LONG   lID;
           LPWSTR lpwstrName;
     } Kind;
} MAPINAMEID, FAR *LPMAPINAMEID;
```

When calling **GetIDsFromNames** to retrieve property identifiers for named properties, it is necessary to convert between wide and narrow character sets. The *lppPropTags* parameter for **GetNamesFromIDs** can point to NULL, in which case you get all the names back. You can also pass in NULL for the *lpPropSetGuid* parameter which enables you to retrieve all of the names for all of the property identifiers. That is, the method returns all known mappings.

Objects that provide name-to-identifier mapping of their properties often support a binary property called PR_MAPPING_SIGNATURE. The value of a PR_MAPPING_SIGNATURE property is a MAPIUID structure. Supporting this property provides a shortcut for working with named properties. If two objects have the same mapping signature, they represent the same property and belong to the same property set. Users can compare two named properties simply by checking the value of the PR_MAPPING_SIGNATURE property. They do not have to translate the names of the properties into identifiers and compare them. This is helpful for clients that implement their own copy operation involving named properties and for service providers that implement **IMAPIProp::CopyTo** and **CopyProps**.

When named properties are moved or copied, the name remains constant. Clients can preserve names during a move or copy operation by adjusting property identifiers to match the name-to-identifier mapping of the destination object. An exception to this rule is when the source and destination objects have the same mapping signature. Only IDs in the range of 0x8000 to 0xFFFE use name-to-identifier mapping. MAPI expects a store provider which chooses to support named-properties in contents tables of folders, to use the same name-to-identifier mapping for all objects in a folder. A store provider which chooses to support named properties in contents tables of search folders must use the same name-to-identifier mapping for all objects in the store.

Handling Property Errors

When you are implementing a property object, try to accomplish as much of the requested work as possible, even when errors occur. If a call involves processing multiple properties, such as **IMAPIProp::CopyTo**, and errors occur on only some of the properties, return the value MAPI_W_ERRORS_RETURNED. This warning indicates that the call was successful, but there were a few problems. Some of the methods pass back an array containing data for each of the properties that were processed; other methods pass back an array of property problem data structures, called **SPropProblem** structures.

In implementations of methods that pass back an array, set the property type entry for the failing property to PT_ERROR and the property value entry to the appropriate error code. For example, suppose your implementation of **IMAPIProp::GetProps** is asked to retrieve ten properties and you can only access eight. You should return the warning MAPI_W_ERRORS_RETURNED, set the property type of the two inaccessible properties to PT_ERROR, and set the property value to an error code MAPI_E_NO_ACCESS.

The property problem data structure is defined as follows:

```
typedef struct _SPropProblem {
     ULONG ulIndex;
     ULONG ulPropTag;
     SCODE scode;
} SPropProblem, FAR *LPSPropProblem;
```

The property problem array contains multiple property problem data structures plus a count of the number of structures. For more information about the **IMAPIProp** interface and the **SPropProblem** and **SPropProblemArray** data structures, see the *MAPI Programmer's Reference*.

Considerations for Service Providers

This chapter discusses topics that should apply to most, if not all, types of service providers. Starting with a brief description of the types of files that service providers support, the chapter continues with a discussion of issues pertaining to message service design and configuration. The next section involves session start up and shut down and the tasks that all service providers must perform. Details specific to one or more types of service providers are covered in the chapters that relate to the particular service provider. The last group of sections describe a variety of tasks that are common to all service providers, such as construction of entry identifiers, sending event notifications, and validating parameters.

Guidelines for Service Providers

Each service provider should have three header files: one public header file and two internal files. The public header file should be used for configuration and for documenting properties and their values. One of the internal header files should include all necessary public MAPI headers and should be shared by all of the service provider source files. The other internal file should define resource identifiers.

Executable file names for service providers should not exceed six characters. This is to allow a suffix to be appended to the end of the name to identify the platform.

Implementing a Message Service

Message services, as defined previously, are groupings of one or more related service providers that share installation and configuration code. Service providers can choose to remain as standalone providers, as single-provider message services, or can become members of multiple provider message services. Some of the sample service providers in the MAPI SDK operate as single-provider services. MAPI recommends that all service providers be associated with a message service to make installation and configuration easier for the users of their client applications.

When you implement a message service, you need to create a simple setup program to install the service providers that belong to the service and some software for configuration support. The amount and complexity of the configuration software you will need depends on your message service. The following sections offer suggestions for designing a message service and information about creating the software required to support installation and configuration.

Message Service Design

There are several issues involved with designing message services. First, decide how many service providers will be included in the service. Message services can include either one provider or several. The guideline is that if the message service has multiple providers, these providers should be related in some way. This guideline exists to prevent a provider DLL from being arbitrarily added to an existing message service. The existing service cannot be expected to have all of the information that it needs to configure the new provider and without this information, the provider might not work. To properly integrate the new provider, the service would have to reverse engineer the provider's configuration information and rewrite a great deal of the service configuration code.

An alternative to integrating unrelated providers in a message service is adding them to a profile. For example, suppose your default profile contains a message service with a transport provider and a message store, and you would like to add the capabilities provided by the MAPI SDK sample personal message store. Rather than integrating the sample store into your service, potentially causing problems, add it to your default profile. MAPI will handle the integration.

The other issues involve the organization of the one or more DLL files that are to be used for the message service. Message services can be implemented in one or more DLLs. To simplify coding, installation, and configuration, it is recommended that message services use a single DLL. However, this is not a requirement. Some services use one DLL for message service-specific code and one DLL for each of the providers in the service.

Other services share a DLL. Only related services, or services that have something in common, such as the need to share database code, should share a DLL. If you are planning to implement a message service with another service in the same DLL file, remember that each entry point function in the DLL must have a unique name and that a single DLL cannot include two or more providers of the same type. A single DLL cannot include two or more providers of the same type. That is, only one message store provider, one transport, one address book, and one hook is allowed per DLL. This restriction is because MAPI stores only one entry point per provider type. If your service must include multiple providers of one type, either have them share an entry point or reside in separate DLLs.

The message service DLL can be given any name that is six characters or less. The six character restriction is enforced to allow MAPI to concatenate characters onto the end of the filename to indicate the target platform. If the DLL runs on a 32-bit platform, the characters "32" are concatenated onto the filename; if the DLL runs on a 16-bit platform, nothing is concatenated and the filename remains as originally named. A similar convention applies to provider DLL names.

For example, suppose you build two versions of a message service called TEST, one for a 16 bit platform and the other for a 32 bit platform. The TEST service has one provider that resides in its own DLL, MYAB. The 16 bit versions of the DLLs would be called TEST.DLL and MYAB.DLL while the 32 bit versions would be called TEST32.DLL and MYAB32.DLL.

The message service setup program writes the names of the service and provider DLLs into the MAPISVC.INF file so that they are available for MAPI to use for loading. The message service DLL file is specified using the PR_SERVICE_DLL_NAME property and each provider DLL file is specified using the PR_PROVIDER_DLL_NAME property. Only the base file name and extension is included in MAPISVC.INF.

Message Service Installation

The first step towards implementing a message service involves creating a setup program for installation. The setup program copies the files that make up your service, such as the provider DLLs, to the local drive, creates a default profile, and adds entries to the MAPISVC.INF and WIN.INI files. Depending on the state of your workstation, MAPISVC.INF may already exist. You can use the MERGEINI utility to append your entries, held in a temporary file, to the existing MAPISVC.INF file. Invoke MERGEINI as follows:

```
MERGEINI C:\MAPI\TMP.INI -m -q
```

The easiest way to create a default profile is to invoke one of MAPI's configuration applications, either the Profile Wizard or the Control Panel applet. However, there are no set rules. Setup programs can choose not to involve the user in the process at all by creating the profile programmatically. When you create the profile, make sure that you do not store anything that would be hard or impossible to recreate. Profiles are considered an expendable part of the MAPI architecture. 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. If your profile is password-protected and a user loses the password, the profile must be deleted and a new profile created to take its place.

The entries that you add to MAPISVC.INF are made up of required and optional configuration properties that relate to the message service and its service providers. The required properties include the names of the DLL files in the service and the name of the configuration entry point function for the service.

If your message service is a single provider service, it is wise to store all of entries in the provider section in MAPISVC.INF. Accessing the provider section is faster and more direct. If your message service is a multiple provider service, store the entries in the service section. Since all of the service providers in your service will need access to the same information, it is better to have it in one place. Having one copy eliminates the problem of keeping multiple copies in sync and saves space.

Do not store any passwords or other credentials that require extra protection in MAPISVC.INF. Instead, either do not store them at all or store them as secure properties in the profile. When passwords are not kept in the profile, the user must enter them with every logon. Storing passwords as secure properties in the profile provides the additional protection that passwords require, such as encryption, and makes logging on easier for users.

The entries that are added to the WIN.INI file specify non-default settings for standard MAPI components, such as the DLL files that contain the standard MAPI user interface and profile provider. If your message service intends to use the standard MAPI components, your setup program need not modify the WIN.INI file.

Refer to the *MAPI Programmer's Reference* for details about the specific MAPISVC.INF and WIN.INI entries to include in your message service setup program.

Message Service Configuration

Supporting the configuration of a message service involves writing at least one entry point function, publishing the name of that function in the MAPISVC.INF file, and creating one or more property sheets for showing configuration data. MAPI's Control Panel applet and other configuration clients make a call to this entry point function to display the property sheets. If message services want to also support the Profile Wizard, there are three additional requirements: another entry point function, a standard Windows dialog procedure, and an enhanced message service entry point function.

The following sections describe how to create these entry point functions, the Profile Wizard dialog procedure, and the configuration property sheets.

Creating a Message Service Entry Point Function

The message service entry point function manages service and provider profile data and handles configuration requests from MAPI and client applications. Creating an entry point function is not required, but it is strongly recommended. Without it, MAPI will run, but your service or your service's users will need to perform extra tasks to make up for its absence. For example, a message service that does not store anything in the profile for security reasons would not need an entry point function. However, because MAPI needs access to configuration data to load the service, the user must be required to enter all of the data with every logon.

Message service entry point functions can provide support for interactive configuration, programmatic configuration, or both. As a minimum, these functions are expected to be able to store and retrieve properties from the profile sections that are associated with the message service. Interactive configuration involves the display of one or more property sheets - at least one for the message service and optionally one for each configurable service provider in the service.

Programmatic configuration involves using an array of property values to change configuration options and publishing information about these properties in a public header file. The header file contains property tags and valid settings for each property in the array. For example, the following header file specifies two properties for a fictional message service - one to define a file and the other to uniquely identify the service:

```
/* Property tag definitions   */

#define PR_MY_FILE                 PROP_TAG(PT_TSTRING, 0X6604)
#define PR_MY_UID            PROP_TAG(PT_BINARY, 0X6601)

/*
*  PR_MY_FILE is the full path name of the file for my address list.
*
*  PR_MY_UID is the unique identifier for my service. If you run
*  multiple instances of my service, each instance must have different
*  PR_MY_UIDs.
```

Programmatic configuration is important because it enables message service implementors that are writing special installation programs to configure their service in a totally automated fashion. If your message service will support MAPI's Profile Wizard application, programmatic configuration is required. If your message service does not support the Profile Wizard, programmatic configuration is optional.

All message service entry point functions use the MSGSERVICEENTRY prototype defined by MAPI as follows:

```
HRESULT (STDAPICALLTYPE MSGSERVICEENTRY)
      (HINSTANCE hInstance, LPMALLOC lpMalloc, LPMAPISIP lpMAPISup,
       ULONG ulUIParam, ULONG ulFlags, ULONG ulContext, ULONG cValues,
       LPSPropValue lpProps, LPPROVIDERADMIN lpProviderAdmin,
       LPMAPIERROR FAR * lppMapiError)
```

Message service entry functions receive pointers to a support object, an OLE style allocator, a provider administration object, a set of option flags, and an array of property values. Your entry point function will typically be invoked after your caller, MAPI or a configuration client, has performed or is in the act of performing some operation. The particular operation will be told to you in the *ulContext* parameter.

There are seven different operations, each represented by a constant. Depending on the operation and your service, the entry point function might perform extensive processing or merely return S_OK. Two operations that usually require very little work are the MSG_SERVICE_INSTALL and MSG_SERVICE_UNINSTALL operations. The *ulContext* parameter will be set to MSG_SERVICE_INSTALL after your caller has run the installation program provided by your message service. The entry point function can either return immediately with S_OK or can perform any necessary post-installation processing.

When *ulContext* is set to MSG_SERVICE_UNINSTALL, typically the user 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 or has canceled the removal. If the removal completed successfully, your entry point function can delete any message service-related files or data. If the removal was canceled, query the user to make sure that the choice to cancel was intentional, and, if so, return MAPI_E_USER_CANCEL.

The other operations fall into two categories: operations that apply to a profile that contains or will contain your message service and operations that apply to your message service. The MSG_SERVICE_CREATE, MSG_SERVICE_CONFIGURE, and MSG_SERVICE_DELETE operations apply to a profile. With MSG_SERVICE_CREATE, 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. Your implementation of the create and configure operations will depend on the settings of the flags parameter, *ulFlags*. These flags influence whether a user interface should be involved in the operation, which is sometimes a requirement. The SERVICE_UI_ALWAYS flag indicates that the user always wants a user interface. If this flag is not set during message service creation, your entry point function should fail.

The handling of the MSG_SERVICE_CONFIGURE operation is more complicated. If the *lpProps* parameter contains enough data to successfully configure the service, or if neither the SERVICE_UI_ALWAYS or SERVICE_UI_ALLOWED flags are set, your entry point function should complete the configuration programmatically if it can and not display any user interface. However, if SERVICE_UI_ALLOWED is set and *lpProps* is either empty, contains insufficient data, or your entry point function is not written to handle programmatic configuration, you should display a dialog box to allow the user to handle the configuration.

The *ulContext* parameter is set to MSG_SERVICE_DELETE when your message service is removed from a profile. Deletion occurs on a *per-instance* basis - other instances of the service either in the same profile or a different profile are unaffected and the service can still be added to profiles. Your service entry function should handle this call by returning immediately with a success code.

When a service is copied, MAPI does not make a call to your entry point function. The configuration settings remain as they were in the original service.

The last two values for *ulContext*, MSG_SERVICE_PROVIDER_CREATE and MSG_SERVICE_PROVIDER_DELETE, are specified when your caller is trying to add a service provider to your message service or delete a service provider from your message service. Not all message services support this functionality. If your service does not provide this support, return MAPI_E_NO_SUPPORT. If your service does provide this support, use the **IProviderAdmin** pointer passed into the call in the *lpProviderAdmin* parameter to call either the **CreateProvider** or **DeleteProvider** method.

Note Do not call **MAPIInitialize** or **MAPIUninitialize** in your entry point function. This can lead to a possible deadlock situation. Since both MAPI and OLE may start threads.

Creating Property Sheets

Property sheets can be created using the Windows 95 property sheet API or with a display table. The display table route spares you from having to work with the Windows user interface, reducing your development time.

Creating a Profile Wizard Entry Point Function

The Profile Wizard calls your Profile Wizard entry point function to retrieve information to be displayed in its user interface. This entry point function follows the SERVICEWIZARDENTRY prototype, as is defined below:

```
ULONG (STDAPICALLTYPE SERVICEWIZARDENTRY)
     (HINSTANCE hProviderDLLInstance, LPCSTR *lpcsResourceName,
     FARPROC *lpDlgProc, LPMAPIPROP lpMAPIProp);
```

The Profile Wizard passes in a pointer to an **IMAPIProp** interface implementation. This property object provides access to all of the configuration properties. There are two output parameters: a dialog box resource and a pointer to your dialog procedure. You should keep a reference to your **IMAPIProp** implementation handy while the configuration user interface is visible.

Your Profile Wizard entry point function typically displays a single dialog box resource that may contain multiple dialog boxes within it. This resource includes all of the controls for all of the property pages. When called, the function reveals only the controls for the current page. As the Profile Wizard moves from page to page, your function must hide all of the controls for the old page and expose the controls for the new page.

Creating a Profile Wizard Dialog Procedure

The Profile Wizard dialog procedure is a standard Windows procedure that functions as a callback invoked by the Profile Wizard. This dialog procedure handles the events that occur during the display of your property pages.

The Profile Wizard creates a dialog frame with three buttons (Back, Next or Finish, and Cancel) and a fixed size area into which you display your dialog. Any Windows messages or events that occur within this fixed area will go directly to your Profile Wizard dialog procedure.

Make sure that your property pages are always available because it is possible for the Profile Wizard to proceed in either a forward or backward direction, generating a WM_INITDIALOG message more than once. However, because the Profile Wizard is a single instance application, your dialog procedure does not have to be prepared to display multiple instances of itself. Information can be safely stored in static variables.

Common Entry Point Functions and Methods

Service providers, regardless of their type, implement an entry point function into their DLL, and two interfaces, one for initialization and the other to control the logon session. The interfaces are named according to the type of service provider implementing them. The suggested names for the entry point functions for the types of service providers differ to allow one DLL file to contain any combination of service providers. However, because the entry point function is a prototype, these names may be selected by the service providers. The following table lists the functions as they are defined and the two interfaces:

Type of Service Provider	**DLL Entry Point Function**	**Provider Interface**	**Logon Interface**
Address book	**ABProviderInit**	**IABProvider**	**IABLogon**
Message store	**MSProviderInit**	**IMSProvider**	**IMSLogon**
Transport	**XPProviderInit**	**IXPProvider**	**IXPLogon**

The DLL entry point function performs three tasks. It checks the version of the service provider interface to make sure MAPI is using a version that is compatible with the version that you are using, initializes some global variables, and instantiates a provider object. A pointer to that object is returned for MAPI's use.

The implementations of the provider and logon interfaces across providers is not identical, but similar. All providers include in their provider object a method for logging on and for shutting down. The message store provider adds two other methods - one for spooler logon and one to compare message store identifiers.

The logon object, however, is less similar across providers. The address book and message store logon objects resemble one another in that they share a common set of methods, the address book provider adding a few more methods. The greatest difference lies in the transport provider's implementation. The **IXPLogon** interface includes only one method, **OpenStatusEntry**, that is part of the other two logon interfaces. All of the other transport logon object methods are unique to transport providers.

To give you an idea of the range of difference between the number and type of methods in each of these service provider interfaces, the following table lists the methods for each:

Object	**Address Book Provider**	**Message Store Provider**	**Transport Provider**
Provider (IxxPr	**Shutdown, Logon**	**Shutdown, Logon, SpoolerLogon, CompareStoreIDs**	**Shutdown, TransportLogon**

ovider interface)			
Logon (IxxLogon interface)	**GetLastError, Logoff, OpenEntry, CompareEntryIDs, Advise, Unadvise, OpenStatusEntry, OpenTemplateID, GetOneOffTable, PrepareRecips**	**GetLastError, Logoff, OpenEntry, CompareEntryIDs, Advise, Unadvise, OpenStatusEntry**	**AddressTypes, Idle, RegisterOptions, TransportNotify, TransportLogoff, SubmitMessage, EndMessage, Poll, StartMessage, OpenStatusEntry, ValidateState, FlushQueues**

DLL Entry Point Function

MAPI begins the process of starting your service provider by looking in the profile to locate the name of your DLL file. All service providers are required to register their DLL file name in the profile. After calling the Windows API function **LoadLibrary** to launch your service provider DLL, MAPI calls your DLL entry point function. MAPI can call **LoadLibrary** each time it uses the DLL, regardless of whether it is already loaded, or MAPI can remember internally which DLLs are loaded and only load DLLs when necessary. In either case, MAPI does the appropriate number of **FreeLibrary** calls when it is done with the DLL.

The DLL for a service provider can have additional functions beyond those necessary for MAPI support. For example, a message system might include administration functions in the same DLL. Such external uses should call **LoadLibrary** before using code in the DLL and eventually **FreeLibrary** to remove the DLL from memory after use. Do not count on MAPI keeping the DLL in memory for such functions.

When your DLL entry point function completes, your service provider has been configured and loaded and has identified itself to MAPI. MAPI makes this call at different times depending on the type of service provider and the type of client. When Simple MAPI clients, for example, call **MAPILogon** to begin a session, the default address book and message store providers are automatically started. Address book and message store providers that are used by Extended MAPI clients are started after the logon process, when explicit calls to the session object are made. Clients call **IMAPISession::OpenMsgStore** to load a message store provider and **IMAPISession::OpenAddressBook** to load the integrated address book. Transport providers are not loaded until they are needed to handle the submission or receipt of a message.

A DLL entry point function has two main tasks: to check the version of the service provider interface (SPI) that MAPI is using against your version and to create a provider object. This function has a lengthy parameter list which is the same for every service provider entry point function. For example, the ABPROVIDERINIT prototype for address book providers is defined as follows. Notice that the name of the returned provider object is tailored to the type of service provider; the other entry point functions use *lppMSProvider* or *lppXPProvider*, depending on their type.

```
HRESULT ABProviderInit (STDMAPIINITCALLTYPE ABPROVIDERINIT)
      (HINSTANCE hInstance, LPMALLOC lpMalloc,
       LPALLOCATEBUFFER lpAllocateBuffer, LPALLOCATEMORE lpAllocateMore,
       LPFREEBUFFER lpFreeBuffer, ULONG ulFlags, ULONG ulMAPIVer,
       ULONG FAR *lpulProviderVer, LPABPROVIDER FAR *lppABProvider)
```

As with the message service entry function, the DLL entry point function receives an OLE allocator, the MAPI allocators, and a set of flags. The other parameters are specific to the this entry point function. First, MAPI includes the number of your SPI version and the number of MAPI's version. Service provider interface version checking allows the service provider interface to expand to meet future requirements, while maintaining as much forward and backward compatibility as possible between software made by different companies at different times. The burden of compatibility checking is placed on the software with the higher version number, as it is the one that knows the capabilities of both versions.

The MAPI header files declare CURRENT_SPI_VERSION as the latest version of the SPI. If your version, passed in the *lpulProviderVer* parameter, is the same or lower than the version of the interface that MAPI passes in, the version check should succeed. However, if you are using a less recent version, you should fail the call, returning MAPI_E_VERSION. This way MAPI and service providers jointly define the range of SPI version numbers they can handle, and if nothing matches, then service provider start up fails. In the MAPI 1.0 timeframe this is of little practical use, but in future versions of MAPI it will be more useful.

MAPI's version information is a 32-bit unsigned integer broken into three parts:

- Bits 31-24 are major version byte.
- Bits 23-16 are a minor version byte.
- Bits 15-0 are a minor minor version word.

The version of the provider interface documented in the MAPI header files is always known as CURRENT_SPI_VERSION, whose value changes from time to time. Knowing multiple version numbers is useful for service providers that choose to support more than one version of the MAPI SPI.

Microsoft periodically changes and releases for external users major and minor numbers. Micro version number is used internally to Microsoft, and should be zero in all external versions of the interface.

The last parameter in the DLL entry point function is an output parameter - a pointer to a provider object's IXXProvider interface implementation. The DLL entry point function must create one of these objects and return a pointer to it in this last parameter. MAPI uses the provider object throughout the session for routing requests from the client.

Service Provider Logon

When your DLL entry point function returns, MAPI uses the provider object to call its logon method. The logon method is called **Logon** for address book and message store providers and **TransportLogon** for transport providers. These methods are defined for the three provider types as follows, with the address book provider version appearing first.

```
HRESULT Logon (LPMAPISUP lpMAPISup, ULONG ulUIParam,
     LPTSTR lpszProfileName, ULONG FAR *lpulFlags,
     ULONG FAR *lpulcbSecurity, LPBYTE FAR *lppbSecurity,
     LPMAPIERROR FAR lppMAPIError, LPABLOGON FAR *lppABLogon)

HRESULT Logon (LPMAPISUP lpMAPISup, ULONG ulUIParam,
     LPTSTR lpszProfileName, ULONG cbEntryID, LPENTRYID lpEntryID,
     ULONG FAR *lpulFlags, LPCIID lpInterface,
     ULONG FAR *lpcbSpoolSecurity, LPMAPIERROR FAR lppMAPIError,
     LPMSLOGON FAR *lppMSLogon, LPMDB FAR *lppMDB)

HRESULT TransportLogon (LPMAPISUP lpMAPISup, ULONG ulUIParam,
     LPTSTR lpszProfileName, ULONG FAR *lpulFlags,
     LPMAPIERROR FAR lppMAPIError, LPXPLOGON FAR *lppXPLogon)
```

Notice that some of the parameters are the same across all three prototypes. Every method receives a pointer to a support object, for example. The implementation of this support object varies, depending on which type of provider you are. MAPI provides three different implementations of **IMAPISupport**, each one tailored to the specific type of service provider. Therefore, if you are an address book provider, for example, you will get a support object that includes implementations of the **GetOneOffTable**, **Address**, and **Details** methods but not the **ReadReceipt** and **PrepareSubmit** methods. These latter two methods are included for a message store provider support object.

All logon methods also receive the name of the current profile, a window handle, and a set of flags as input parameters and pass back as output parameters a pointer to a MAPIERROR structure for extended error information and a pointer to a newly created logon object. The other parameters are specific to the type of provider. For information about the portions of the logon method that are provider type-specific, refer to the chapters that relate to programming that type of provider.

One of the basic tasks performed by every logon method is to increment the reference count on the support object. It is critical that you increment the support object's reference count by calling **IMAPISupport::AddRef.** If this step is omitted, MAPI will release the support object and unload your service provider.

Checking the Configuration

Another task involves verifying your provider's configuration. Verifying configuration involves checking the settings for options that are required for your provider to operate successfully. The particular settings that you check depends on your service provider. For example, a message store provider might need to locate its message store, check credentials, and register a row in the status table.

You can retrieve configuration settings for your service provider from the profile by calling **IMAPISupport::OpenProfileSection**. If the profile does not contain all of the settings that your service provider requires, in some cases you can display a user interface to ask. the user to supply the necessary information.. Whether or not you can display a user interface depends on the value of the flags passed in with the call. If a user interface is not permitted or the user cannot supply the information, your service provider cannot be configured and you should return MAPI_E_UNCONFIGURED. A slight variation to this problem occurs when the user cancels the configuration dialog box. In this case, return MAPI_E_USER_CANCEL rather than MAPI_E_UNCONFIGURED.

When you return MAPI_E_UNCONFIGURED, MAPI calls your DLL entry point function again, asking it to locate the required information. If the information can be located, MAPI will call your logon method again. A user interface is never allowed on this second call; all of the necessary data must be passed in. If the information cannot be located, depending on how important your service provider is, the session may terminate.

Registering a Unique Identifier

Every logon method creates a logon object and returns a pointer to that object. MAPI identifies each logon object with a unique identifier, or UID, that is registered as part of the logon method implementation. The UID represents a service provider and is contained within the entry identifier for every object that the service provider owns. MAPI uses the MAPIUID to route calls to the correct provider.To register a UID, providers call **IMAPISupport::SetProviderUID** in the logon method. Most service providers define a static constant.

Some providers register one UID; others register several. Because a single provider can have several logon objects active concurrently, it is recommended that if your service provider can instantiate multiple logon objects, register a unique identifier for each one. This increases the accuracy with which MAPI matches entry identifiers to service providers and saves you some work . When every logon object has its own unique identifier, MAPI can guarantee that any request it routes a logon object can be handled by that object. When logon objects share UIDs, MAPI routes the request to the first instance of the UID it finds. If, in this case, the logon object receiving the request determines that it cannot process the entry identifier, it should pass the request on to the service provider's next logon object before returning an error. If your service provider has only one logon object, or none of the logon objects can service the request, then an error return is in order.

Constructing Entry Identifiers

A service provider uses a piece of binary data known as an entry identifier to refer to one of its object. In the Address Book, for instance, entry identifiers are used to identify mail users, distribution lists, and containers. To clients and other service providers, entry identifiers are just binary data. Only the service provider that has assigned the identifier understands what is inside.

Entry identifiers come in two types: *short-term* and *long-term*. Short-term entry identifiers can be generated quickly, but their uniqueness is guaranteed only over the life of the current session on the current workstation. Short-term entry identifiers are used in tables. There are two types of long term identifiers: temporal and positional. Entry identifiers that last over time are known as temporary identifiers. Entry identifiers that last across computer workstations are known as positional. The only entry identifiers that are both temporary and positional are the identifiers created for custom recipients.

The ENTRYID data structure is used to define an entry identifier. ENTRYID data structures contain an array of flags that describe the identifier and the binary data. The data is made up of a 16 byte MAPIUID field that represents the service provider and a variable amount of other data that is specific to the object. The ENTRYID structure is defined as follows:

```
typedef struct {
     BYTE   abFlags[4];
     BYTE     ab[MAPI_DIM];
} ENTRYID, FAR *LPENTRYID;
```

The first 4-byte flags, of which the last three bytes **must** be zero, describe the type and use of the entry identifier. For example, the value MAPI_THISSESSION indicates that the entry identifier cannot be used to identify objects belonging to other sessions and MAPI_SHORTTERM indicates that this identifier is a short term entry identifier. If you are constructing a long term identifier, these flags must be clear. Clients expect to be able to check the first byte to determine the entry identifier's longevity. A zero indicates a long term identifier and a non-zero indicates a short term identifier.

The next 16 bytes make up a MAPIUID structure that identifies the service provider responsible for creating the entry identifier. When an application calls an **OpenEntry** method, MAPI uses the MAPIUID to determine which service provider DLL to call. The MAPIUID structure contains a type order independent entry identifier. If you are using a profile that must work on two different processors, each with a different byte order, use the MAPIUID structure to describe items.

An updated version of a service provider DLL should use the same MAPIUID as the previous version. Since a message saved in the message store has the entry identifier of the sender stored with it, keeping the same MAPIUID allows a user to reply to a message received months earlier when the user had an earlier version of the provider DLL.

The rest of the *ab* member contains anything that makes sense for your provider. Because a service provider has to be forward and backward compatible with itself, one of the items that makes sense for many providers is version information. Many providers change the format of their entry identifiers from version to version. By storing a version indicator, you can quickly determine the format to use for deciphering any particular entry identifier.

Another helpful piece of information for many providers is an indicator, or hint, of how to locate the object represented by the entry identifier. For example, you could store the disk offset for the last place an entry was stored in a data file. Because hints are only valid for a limited period of time (say, between compresses of a global tree structure that happens only after 128 people are removed from the tree), the entry identifier would also need enough data to determine if the hint is still valid, and to locate the entry even if the hint is out of date. Because hints can change with time, two entry identifiers obtained at different times for the same entry can have different binary values. Call the **CompareEntryIDs** methods found on many objects to determine if two entry identifiers actually refer to the same object.

Whenever a user changes names or moves to a different machine, keep long term identifiers constant.

The more far-reaching an identifier is, the more valid it is. For cross-platform operation, such as use on different byte order machines, you might choose to store all of your entry identifier data in a particular byte order and have the software on one platform always swap bytes. Alternatively, you could choose to indicate the byte order by encoding the format in the entry identifier, and write code on both machines for handling data in either byte order.

If you create entry identifiers that change whenever a move or modify operation is performed, generate notifications to inform clients of these events.

Supporting Notifications

Support for notifications, although common and very useful, is entirely optional. If you are writing a message store or address book provider, mostly likely you will support table notifications for your contents and hierarchy table implementations and notifications on one or more of your other objects. Transport providers typically rely on the MAPI spooler to generate notifications for events that they produce. You can handle the work of keeping track of the registrations and sending the necessary notifications within your advise source objects or you can take advantage of functionality MAPI provides. The **IMAPISupport** interface contains three methods that can help service providers implement notification support: **Subscribe, Unsubscribe**, and **Notify**.

Remember that clients call the **Advise** method of an advise source object to register for notifications and **Unadvise** to deregister. To use the **IMAPISupport** notification methods, create a notification key, or NOTIFKEY data structure, from the entry identifier representing the advise source object. The notification key is binary data that is used by MAPI to identify an advise source object across processes. Pass this key to **IMAPISupport::Subscribe** in your **Advise** implementation. **Subscribe** will call **IMAPIAdviseSink::AddRef**, enabling MAPI to hold onto the advise sink pointer until the **Unadvise** call is made, before returning with a non-zero connection number that you can pass back to the advise sink. You can release the client's advise sink at any time after the **Advise** call. However, this is not true of the connection number. You need to wait until the client makes the **Unadvise** call before you release the connection number.

When it is time to send a notification, pass your notification key and an array of notification structures containing event data in a call to **IMAPISupport::Notify**. Service providers make one call for every advise source for which an event has occurred. This might seem like a lot of calls, but calls to **Notify** are not time consuming. If there are no registered advise sinks for the event, **Notify** ignores the call. When you initialize your notification structures, set all unused members to zero to help clients create smaller, faster, and less error-prone **OnNotify** implementations. For information about how to fill in the individual notification data structures, see the *MAPI Programmer's Reference*.

Service providers can send notifications at any convenient time; there are no rules as to their timing. This can result in clients receiving notifications for events as they are occurring, after they have occurred, or sometimes before they have occurred. It is also possible for clients to receive notifications from an advise source after they have called **Unadvise** to remove their registration.

Notifications on messages are sent either when changes are saved or when the message object is released. Until the notification is sent, no changes are visible in the message store. Note that anyone involved in the session can make changes that cause a notification to be sent: the MAPI subsystem or any client application. Calling the **Unadvise** method on the object that previously received your **Advise** call does not necessarily guarantee that you will no longer get notified by this object. Be careful in your notification callback.

When advise sinks call your **Unadvise** method, forward the call on to MAPI's **Unsubscribe** method in the support object. MAPI will free the advise sink object at this time by calling its **Release** method. Y

Notifications are almost always asynchronous. MAPI does not disallow synchronous notification, however, and service providers who want to implement their notifications synchronously are free to do so. However, all advise sinks, including MAPI, expect asynchronous notification and have no way to request otherwise.

Supporting Status

The status table is a way for service providers to expose status and other dynamic information to clients. Supporting the status table is an option for address book and message store providers; for transport providers it is a requirement. Transport providers must support the following three status table operations:

- creating a row in the table
- updating a row in the table
- providing information to clients about data not stored in the status table row

To add a row to the status table, pass an array of property value structures in a call to **IMAPISupport::ModifyStatusRow.** This call is typically made when your service provided is loaded. The property value structures are used to define the columns to be maintained for your row.

When one of the properties that are included in your row is modified, you must update the status table so that MAPI can notify all interested clients of the change. Updating the status table is accomplished with a call to **IMAPISupport::Notify**.

Service providers are required to include several properties in the property value array that they pass to **ModifyStatusRow**. Beyond this required set, other properties can be included. Including additional properties in a status table row is optional and up to the provider.

The basic set of required properties are:

PR_DISPLAY_NAME	PR_ENTRYID
PR_IDENTITY_DISPLAY	PR_IDENTITY_ENTRYID
PR_INSTANCE_KEY	PR_OBJECT_TYPE
PR_PROVIDER_DISPLAY	PR_PROVIDER_DLL_NAME
PR_RESOURCE_FLAGS	PR_RESOURCE_METHODS
PR_RESOURCE_TYPE	PR_ROWID
PR_STATUS_CODE	PR_STATUS_STRING

There are two ways for clients to access a service provider's status object. One of the ways is through the logon object's **OpenStatusEntry** method. **OpenStatusEntry** exists in every service provider's logon object. If you support a status object, you will return a pointer to it when your **OpenStatusEntry** method is called.

The other way is for clients to call **IMAPISession::OpenEntry** with an entry identifier from the status table that represents your status object. MAPI will route the call to the logon object belonging to the provider that owns the entry identifier by invoking the **OpenStatusEntry** method. If you are passed the MAPI_MODIFY flag, you are being asked to return an object with read-write access. This is only a suggestion; if you do not allow status object modification, you can return a read-only object.

Using the MAPI Idle Functions

The idle engine is a set of API functions that service providers can use for making the best use of idle time. You create a callback function that contains whatever tasks you feel 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, as defined below:

```
BOOL (STDAPICALLTYPE FNIDLE) (LPVOID lpvContext)
```

Use the **FtgRegisterIdleRoutine** to register your callback function with MAPI. 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 the service provider sees fit, and a set of option flags.

You can specify a priority in the *priIdle* parameter that controls how your idle function runs or specify zero if priority is not an issue. Compression or searches should be assigned negative priorities. Tasks that occur once should be assigned positive priorities.

There can be more than one idle function during a session. If you want to rank your idle routines, use positive numbers.

On 32 bit systems, your callback function may be called back on a different thread than the one you used for registering it.

The **DeregisterIdleRoutine** function allows you to deregister a callback function that is active. Realize that the callback function could be invoked at any time during the deregister call, even if **DeregisterIdleRoutine** has returned. This is because **DeregisterIdleRoutine** is an asynchronous operation.

The **ChangeIdleRoutine** function modifies some or all of the characteristics of a callback function. The characteristics that can be modified include the function itself, and/or its priority, time setting, input parameter, and any of the options set with the flags parameter.

Using Structured Storage

Any service provider that accesses large properties needs to use the structured storage interfaces defined initially by OLE. The **IStorage** interface provides high level access to a structured block of storage and the **IStream** interface performs the low level tasks, such as reading and writing bytes from the storage.

Most likely only message store providers will ever need to implement one of these interfaces, typically **IStream**. Because attachments are usually large files or objects themselves, they cannot be managed using the **GetProps** and **SetProps** methods in the standard property object. Attachments require the use of streams.

If your attachment is a file and you want to provide access to it using the methods defined for **IStream**, call **OpenStreamOnFile**. This function allows you to access the file through a newly created stream object.

To create a storage object that implements the **IStorage** interface and works with an implementation of either **IStream** or another OLE storage interface, **ILockBytes**, call the MAPI API function **HrIStorageFromStream.** MAPI will create a storage object that operates with the lower level storage interface.

Because the **IStorage** methods return OLE error values, you will need to call **MapStorageSCode** to translate these values into MAPI error codes.

If your attachment is a message, and you have a readable storage object available, you can call **IMAPIProp::OpenProperty** to open the attachment with a source interface of IID_IMessage. This will result in your getting a message object. Then you can call **OpenIMsgOnIStg** to layer this message object on top of a storage object.

The **OpenIMsgOnIStg** function allows you to specify a callback that is invoked by the new message object when its reference count becomes zero. This can be useful if there is some final processing that needs to be done before the message is completely removed.

This technique is useful when you want to embed a message in an OLE compound document or create a message from arbitrary data in the file system. To accomplish the latter goal, producing a message from data in the file system, first call the OLE helper function **CreateDocfile** to create a storage object. Next pass a pointer to the storage object to **OpenIMsgOnIStg. OpenIMsgOnIStg** creates a message object that accesses its properties using the **IStorage** methods.

The message object that you get back from this scenario is different than the message object that is created by message store providers. You cannot use this message object in calls to **IMsgStore** interface methods, for example, such as **IMsgStore::SubmitMessage**.

Programming with Simple MAPI

This chapter describes how and why your client would use the Simple MAPI functions to add messaging functionality to an application. The topics in this chapter cover the major areas of functionality that a Simple MAPI application 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 Microsoft Windows applications to quickly add a basic level of messaging. Simple MAPI is intended for applications that are concerned with the easy exchange of datafiles and messages.

Because Simple MAPI is only intended for the Microsoft Windows environment and offers limited functionality, it is recommended primarily for backward compatibility with older applications. You are encouraged to use CMC or Extended MAPI for development of new applications whenever possible.You should link Simple MAPI applications 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 allow you to send, address, receive, and reply to messages. Messages sent using Simple MAPI can even include file attachments and OLE objects. Windows applications whose primary purpose is not messaging, such as spreadsheets, word processors, or applications that require only basic messaging can use Simple MAPI to quickly and easily provide messaging functionality.

Simple MAPI has several features that make it a good option for applications 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 gives you freedom to define your own message types as well as the content of messages, and 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 applications 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 application does not need a user interface, you can call the Simple MAPI functions without displaying any dialog boxes.
- Although designed to be called from C programs, you can call the functions with little or no parameter modification from application-specific and standalone scripting packages such as Visual Basic, Actor, Smalltalk, and Object Vision.

If you are writing a new application, you should implement it using Extended MAPI or CMC. However, using Extended MAPI involves learning about objects, interfaces, and the Component Object Model, and it might be that your application only needs the limited messaging functionality available with Simple MAPI. In that case, you should use CMC, since it has the same functionality as Simple MAPI and it works on Microsoft Windows NT, Microsoft Windows for Workgroups 3.1, and Microsoft Windows 95. You should only use Simple MAPI for maintainance of older applications that were written using it.

Initializing a Client

Before starting a Simple MAPI messaging session you need to initialize your client. Initializing your Simple MAPI client involves checking the computer's WIN.INI file to determine whether 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")))
   ```

Similar defined constants exist for the other Simple MAPI functions, so your application simply needs to set the value of the function pointers using **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 messaging services and the providers to manage those services. Before your application can send or receive messages, it needs to log on and establish a session.

There are two types of session, 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 logon*; establishing a persistent session is called *explicit logon*. Applications can use a persistent session for all of the 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 might allow a limited number of sessions, so your application must be able to handle a return value of MAPI_E_TOO_MANY_SESSIONS. This value can be returned when a user is logged on to the system through an email application and a mail-enabled application attempts to log on with a different identity.

Sharing 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. To make this possible, MAPI introduces the idea of a *shared session* that is established once but can be used by other applications. Only one shared session can exist on a given computer at a time. The shared session can be used with any profile. If your application must use a non-shareable session (that is, your application needs to be guaranteed that no other applications will use its session), then it must use the **MAPILogon** function in the Extended 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 application calls **MAPILogon**. If your application logs on with the MAPI_NEW_SESSION flag, then Simple MAPI will create a new shared session for your application to use regardless of whether one already exists.

The session handles returned to applications using a shared session are not the same. Each application, 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

Applications log on explicitly by calling **MAPILogon**. **MAPILogon** automatically opens the default address book for the Simple MAPI caller. When **MAPILogon** returns to the Simple MAPI 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 the application can use in further Simple MAPI calls. This session handle is passed to most Simple MAPI calls and can also be used with Extended MAPI. The support method **ScMAPIXFromSMAPI**, prototyped in the MAPI.H file, converts a Simple MAPI session handle into an Extended MAPI session object pointer.

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

Implicit Logon

Applications log 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, **MAPIDeleteMail** can only delete a message from a pre-existing session; a 0 session handle causes the **MAPIDeleteMail** function to return an error.

When the session handle is 0, Simple MAPI function calls use another common parameter, *flFlags*, 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 mail 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.

Passwords

Whether a password is required for your 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, then your application need not provide a password to **MAPILogon**. If not, then a password is required from the user or from a cache location implemented by the application. 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. Applications running with Microsoft Windows for Workgroups are required to provide a password for **MAPILogon**.

Logon User Interface

Client applications 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 application 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 to **MAPILogon** controls the logon dialog box display.

- To force the dialog box to be displayed

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

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

If you provide 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 to allow the user to enter new credentials.

Specifying a Profile During Logon

The **MAPILogon** call behaves differently depending on whether your application specifies a profile name. Normally a profile name is required. If your application specifies NULL for the *lpszProfileName* parameter in the **MAPILogon** call, 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.

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

Ending a Simple MAPI Session

◆ To end a Simple MAPI session

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

Composing Messages

Composing a message implies either the creation of a new message or the forwarding of or replying to an existing message. The new message may 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 recipt as well.

◆ To create a new message

1. Allocate a **MapiMessage** structure.
2. Fill in its fields with values appropriate for the message the user wants to send.
3. Submit it to the messaging system using the **MAPISendMail** function.

◆ To forward an existing message

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

◆ To reply to a message

1. Retrieve the message using the **MAPIReadMail** function.
2. Modify the message as appropriate for replying:
- Put the original sender's address into the **lpRecips** field.
- Change the subject line to indicate that the message is a reply to an earlier message.
- Allow the user to edit the message body to add their own comments.
3. Submit it to the **MAPISendMail** function.

When your application submits a message with **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

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 Message Headers

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 lists of recipients 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

Use 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 application can pass in an initial array of **MapiRecipDesc** structures which the user can then modify.

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

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

Checking Names

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

Your application should call **MAPIResolveName**. Your application has the option of allowing the display of addressing dialog boxes while resolving the name by including the MAPI_DIALOG flag in the *flFlags* parameter. If the MAPI_DIALOG flag is not present and the recipient's display name resolves to more than one address book entry, **MAPIResolveName** will return the error code MAPI_E_AMBIG_RECIP.

If the MAPI_DIALOG flag is present, Simple MAPI will present the user with a dialog box to resolve the ambiguity. If the MAPI_AB_NOMODIFY flag is also present, then 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 **MAPIAddress**, a successful call to **MAPIResolveName** produces a buffer containing one or more **MapiRecipDesc** structures that must be released with a call to **MAPIFreeBuffer**.

Getting Details About a User

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, the **MAPIDetails** function supports the MAPI_AB_NOMODIFY flag to prevent modification of the properties displayed in the dialog box. The **MAPIDetails** function is most often used with this flag when the user needs more information about an address in order to resolve an ambiguity. The function is most often used without the MAPI_AB_NOMODIFY flag when the user needs to create or edit entries in the Personal Address Book.

Displaying and Editing Addresses

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][Email 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 on 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 application should display friendly names to the user when they are available. When they are not, it is acceptable to display the actual address instead. When forwarding or replying to a message, your application should never allow the user to edit the friendly name or address fields of a **MapiRecipDesc** 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 application 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 application 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) such 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 one of those who the original message was primarily addressed to.
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.

Using the Subject

Using the subject when displaying a message is simple; the subject is simply a null terminated string stored in the **lpszSubject** member of a **MapiMessage** structure.

When replying to or forwarding a message, it is customary for your application to 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 application should not add it again.

Sending Messages

- 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** field of the message structure.
3. Create a text string containing the subject, if any, and place it in the **lpszSubject** field of the message structure.
4. Create a text string containing the message text, if any, and place it in the **lpszNoteText** field of the message structure.
5. Create an array of **MapiFileDesc** structures, if necessary, to contain any attachments and placce it in the **lpFiles** field of the message structure.
6. Submit the message with the **MAPISendMail** function.

Sending messages may involve more or less effort on the part of your application, depending on the way you invoke various Simple MAPI functions to perform the above steps. See the *Programming Examples* topic for more information.

Handling Attachments

Simple MAPI handles attachments by means of an array of **MapiFileDesc** structures pointed to by the **lpFiles** member of a message structure. The **nFileCount** member of the message structure indicates the length of the array, with a value of **0** indicating no attachments.

Simple MAPI recognizes three types of attachments: data files, editable OLE objects, and static OLE objects. The value of the **flFlags** member of a **MapiFileDesc** structure informs Simple MAPI of the type of attachment it describes. 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 in place of the character **lpszNoteText[nPosition]**. The special value **-1** (**0xFFFFFFFF**) indicates that the attachment is not rendered in this way; in this case, the application that receives the message is responsible 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, your application needs to do the following

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 application has received, your application needs to do the following

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

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 to the **MAPISendMail** function.

MapiMessage

The **lpszMessageType** member 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. Applications can use this field 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 non-delivery report (NDR). The messaging system formats the non-delivery for a message type as "REPORT." plus the message type plus ".NDR". For example, if your application uses a message type of "mytype", the NDR for such a message will have "REPORT.mytype.NDR" as its **lpszMessageType** string. For applications that deal with multiple types of message, your application can compare the last four characters of the **lpszMessageType** string against ".NDR" to determine whether the message is a non-delivery report. Naturally, it would be a bad idea for your application to choose a string that ends in ".NDR" as its internal message type. Note that while non-delivery reports generated by MAPI follow this convention, non-delivery 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 application can set the MAPI_RECEIPT_REQUESTED flag to request a receipt. When reading a message, your application can test for the MAPI_UNREAD flag to determine whether the message has not yet been read. Similiarly, your application can test for the MAPI_SENT flag to determine whether the message has been sent. When saving a previously unsaved message, your application should set the MAPI_UNREAD and MAPI_SENT flags as appropriate for the message.

MAPISendMail

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

Flag	Description
MAPI_LOGON_UI	Allow s a logon interface if required. This flag should be set if your application is using an implicit session.
MAPI_NEW_SESSIO N	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 application does not have a **MapiMessage** structure already constructed, or if the user is composing a message interactively.

Programming Examples

The code examples in this chapter are provided to illustrate these concepts in detail. Additional code examples can be found in the SMPCLI application included with the MAPI SDK. SMPCLI is not a typical Simple MAPI application in that 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 an 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 are placed above sections that benefit from explanation.

Sending a Message Simply

This example shows the simplest way of sending a message. An essentially blank message is created and passed to MAPISendMail with parameters that cause Simple MAPI to use dialog boxes to create the content of the message. First, the application defines the variables it needs. Note that a real application would almost certainly not hard-code the attachment pathname 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 application calls **MAPISendMail** and stores the return status so it can detect whether the call succeeded. A real application should use a more sophisticated error reporting mechanism than the **printf** function.

```
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

This example shows how an 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 application starts by defining the variables it needs.

```
// 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, 0, (ULONG)-1, "c:\\budget17.wk3",
"budget17.wk3", NULL};
```

The application then uses **MAPIResolveName** to generate **MapiRecipDesc** structures for the recipients of the message. It could create them directly, as in the previous example, but then no error checking is possible. Since this application 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 application creates the message. Again, a real application would almost certainly not hard-code the actual values of the **MapiMessage**'s members.

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

Again, send the message and record 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 and Displaying Messages

The following example function shows how an application can receive a message with Simple MAPI. To simplify the example, this example assumes that a character interface is being used.

First the 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",// Real app wouldn'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 application 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 application first tests the return code from **MAPIFindNext** against MAPI_E_NO_MESSAGES before checking against SUCCESS_SUCCESS. If the call is successful, the application will have a valid message ID 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 application can now be sure it is safe to retrieve the message. However, it is still a good idea to check the return code from **MAPIReadMail**. If the call fails, the memory pointed to by the application's **lpMessage** pointer will not be accessible by the application. The application 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 application can display the message. As is customary, it begins by displaying the addressing information attached to the message before displaying the subject line and message body. When displaying the addressing information, it is best if the application can display friendly names. However, since friendly names are not always available, the application 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 desire to avoid breaking a receipient's name or address across two lines. This code could be further improved by differentiating between recipients based on their **ulRecipClass** member so that they could 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 application displays the subject line and message body, if they are present. Note that the message text can be printed with a simple call to **printf**. Since the message was read with the MAPI_SUPPRESS_ATTACH flag set, there are guaranteed not to be any 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 application releases the storage that **MAPIReadMail** allocated for the message, closes the session, and returns a successful return code.

```
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.
```

Using Status Objects

This chapter describes what a status object is, how it is implemented, and how it is used. It is divided into two parts following a brief introduction. The first part describes the **IMAPIStatus** interface and the second part describes the properties that are supported by a status object.

About Status Objects

Status objects are MAPI objects that support the **IMAPIStatus** interface. In a typical session, there are several different status objects - some implemented by MAPI, others implemented by service providers. MAPI provides three different status objects: one that represents the overall subsystem, one that represents the MAPI spooler, and one that represents the integrated address book. The MAPI spooler's status object reflects the status of all of the transport providers currently operating. Transport providers are required to supply a status object; it is optional for other service providers.

Status objects are used by clients to learn about the state of session resources. There are two ways a client can access a status object, directly through a service provider's **OpenStatusEntry** method implemented in the logon object or indirectly through the session. Accessing a status object through the session is a two step process. Clients call **IMAPISession::GetStatusTable** to access the status table, a listing of all of the available status objects, and then use one of the columns on the table, PR_ENTRYID, to open the specific status object with **IMAPISession::OpenEntry**.

About IMAPIStatus

The **IMAPIStatus** interface contains four methods, only one of which service providers are required to support. Service providers must support the **ValidateState** method; the **SettingsDialog**, **ChangePassword**, and **FlushQueues** methods are optional. Service providers may return MAPI_E_NO_SUPPORT from the optional methods. The following table briefly describes each of these methods:

Method	**Description**
ChangePassword	Changes a provider-specific password
FlushQueues	Forces a synchronous upload or download of messages
SettingsDialog	Generates a provider-specific configuration dialog
ValidateState	Confirms the status of a provider

Service provider status objects implement **ChangePassword**. **ChangePassword** changes a service provider password programmatically, without user interaction.

The MAPI spooler status object implements **FlushQueues**. **FlushQueues** is an instruction to the MAPI spooler to synchronously give all outgoing messages to the appropriate transport provider or accept all incoming messages from a transport provider. Clients call this method when they want to send or receive messages right away.

Service provider status objects implement **SettingsDialog**. **SettingsDialog** asks a service provider to display a property sheet, which is a dialog box that presents configuration options. It is allowable for service providers to refuse to display a property sheet and return MAPI_E_NO_SUPPORT. However, this is not the recommended course of action. Service providers can implement their own provider sheet or ask MAPI to help by calling **IMAPISupport::DoConfigDialog**. The **DoConfigDialog** method builds a property sheet for display from a display table and configuration data passed in as input parameters.

Service providers can present a read-only property sheet, disallowing any changes of configuration options, or make the property sheet read-write, allowing some or all of the options to be modified. Clients calling the **SettingsDialog** method can request read-only access by setting the UI_READONLY flag, however, this is only a request and it is up to the service provider to ultimately determine what the access will be.

All status objects implement the **ValidateState** method**. ValidateState** prompts a status object to check on its state. The meaning of the phrase "check on its state" varies depending on the particular status object implementation. The MAPI subsystem's **ValidateState** method validates all of the resources owned by service providers and the subsystem itself. The MAPI spooler's **ValidateState** method performs a logon of all of the transports, regardless of whether they were already logged on. The MAPI address book's **ValidateState** method checks the entries in its profile section.

Transport status objects might be passed several option flags in their **ValidateState** call. For example, clients can dynamically reconfigure a transport by making the appropriate changes and then passing the CONFIG_CHANGED flag to **ValidateState**. Some service providers display a progress dialog box if the validation is expected to take a long time - an optional, but helpful feature.

One of the properties of the status object, PR_RESOURCE_METHODS, indicates which of these **IMAPIStatus** methods are supported. PR_RESOURCE_METHODS is a bitmask; there is one bit that corresponds to each of the **IMAPIStatus** methods. When the bit for a particular method is set, that method is operational. Clients check the value of the PR_RESOURCE_METHODS property to determine which of the **IMAPIStatus** methods are fully implemented for a particular status object before making calls.

About Status Object Properties

The following table lists all of the required properties associated with a status object. All of these properties appear in the status table.

PR_DISPLAY_NAME	PR_ENTRYID
PR_IDENTITY_DISPLAY	PR_PROVIDER_DLL_NAME
PR_IDENTITY_ENTRYID	PR_RESOURCE_FLAGS
PR_IDENTITY_SEARCH_KEY	PR_RESOURCE_TYPE
PR_INSTANCE_KEY	PR_ROWID
PR_OBJECT_TYPE	PR_STATUS_CODE
PR_PROVIDER_DISPLAY	PR_STATUS_STRING

Status object implementors can also, as an option, provide the PR_STORE_RECORD_KEY property.

The PR_RESOURCE_TYPE property indicates the type of MAPI component, or resource, that has implemented the status object. The values used to set PR_RESOURCE_TYPE are the following constants:

MAPI_AB	MAPI_STORE_PROVIDER
MAPI_AB_PROVIDER	MAPI_SUBSYSTEM
MAPI_PROFILE_PROVIDER	MAPI_TRANSPORT_PROVIDER
MAPI_SPOOLER	

The PR_RESOURCE_METHODS property is used to indicate which of the **IMAPIStatus** methods have full support for a particular status object. The property consists of a bitmask - one bit for each method. If the bit for a method is set, an implementation for the method exists and can be called. If the bit for the method is not set, the implementation, if called, will return MAPI_E_NO_SUPPORT.

The PR_RESOURCE_FLAGS property is a common property that describes many objects, among them the status object. There are several flags that apply only to status objects. For example, the STATUS_DEFAULT_STORE flag is meaningful for message store providers. If set, it indicates that this message store operates as the default message store. Conversely, STATUS_NO_DEFAULT_STORE indicates that a message store cannot be a default store.

The STATUS_PRIMARY_IDENTITY and STATUS_NO_PRIMARY_IDENTITY flags can be used to indicate whether or not a service provider supplies identity information for the session. If STATUS_PRIMARY_IDENTITY is set, the entry identifier stored in the PR_IDENTITY_ENTRYID property for this status object is the primary identity for the session.

Other flags include STATUS_DEFAULT_OUTBOUND, STATUS_OWN_STORE, STATUS_SIMPLE_STORE, STATUS_TEMP_SECTION and STATUS_XP_PREFER_LAST.

All status objects can set the PR_STATUS_CODE property to STATUS_AVAILABLE, STATUS_OFFLINE, or STATUS_FAILURE. STATUS_AVAILABLE indicates that the service provider or MAPI resource is operational. STATUS_OFFLINE indicates that only local data or services are available. STATUS_FAILURE indicates problems. When STATUS_FAILURE is set, the service provider may soon be shut down, ending the current session.

Transport providers can also set PR_STATUS_CODE to:

STATUS_INBOUND_ACTIVE	STATUS_OUTBOUND_ACTIVE
STATUS_INBOUND_ENABLED	STATUS_OUTBOUND_ENABLED
STATUS_INBOUND_FLUSH	STATUS_OUTBOUND_FLUSH

The inbound flags relate to the transport's ability to receive messages and the actions it can take on those messages. The outbound flags relate to the sending of messages. The ENABLED setting indicates that a transport provider can deliver messages to or accept messages from the MAPI spooler. The ACTIVE setting indicates that a transport provider is currently receiving or sending a message. The FLUSH setting relates to "foreground" message reception and transmission.

Using Tables

This chapter describes how tables are implemented and used. Beginning with a definition and an overview of common features, the chapter continues with a description of the interfaces that are used to access a table's underlying data and the data structures that are passed to the interface methods. The next part covers common table operations, such as sorting, defining a column set, and building restrictions. The last section provides information about each type of table, including the properties that appear in the columns.

As with the other chapters that discuss MAPI objects, this chapter presents tables from both the user's and the implementor's viewpoint. Tables are implemented by MAPI and by service providers to be used by clients and service providers. Therefore, service providers have a dual role as user and implementor. Which role a service provider assumes at any one time depends on the situation and the type of table. The next section delves deeper into describing the different types of MAPI tables and their implementation.

About Tables

MAPI tables are used for looking at the properties of other MAPI objects in a structured way A table can be thought of as a two-dimensional array, the two dimensions being rows and columns. Each row represents a MAPI object; each column represents a property of the object. A MAPI table resembles the traditional database table in that it has many of the same characteristics - the data is presented in row and column format, a cursor keeps track of the current position, and the client or provider using the table has the ability to establish search criteria and a sort order to create a customized view.

A MAPI table is different than most other MAPI objects in that it does not support a set of its own properties. Its purpose is to display the properties of other MAPI objects. The interface that it implements, **IMAPITable**, inherits directly from the **IUnknown** interface and not from the **IMAPIProp** interface. **IMAPITable** provides read-only access to the property data, enabling multiple simultaneous views of the data. Users cannot change the actual property data, they can only change the way it is presented.

Figure 1 following shows one of the frequent uses of a table in MAPI - to display the the contents of a folder. Two messages are shown; each message has several properties. A message is divided into two parts to separate the properties that describe the message and the actual message data. The descriptive properties include the name of the recipient and the sender, represented by the display name property, or PR_DISPLAY_NAME, and the sender name property, or PR_SENDER_NAME. The date and subject of the message are the other descriptive properties. The date value comes from the PR_ORIGINAL_DELIVERY_TIME property and the subject from the PR_SUBJECT property. Because the message uses a plain text format, it is stored in the PR_BODY property. Message text using the rich text format is stored in another property. The PR_ENTRYID property stores the internal identifier of the message. The column set of the contents table, or set of properties that the user wants to see, has been limited to four of these properties, sorted in order by entry identifier.

{ewc msdncd, EWGraphic, group10838 0 /a "SDKEX.bmp"}

Figure 1   Contents table for a folder.

There are many different types of tables. Tables are differentiated by the information that they present. Some tables present information about a single object; other tables present information collected from multiple objects. An example of the first type of table is the recipients table. Message store providers supply clients with a recipients table for every message that contains information about all of the recipients for that particular message. An example of the second type is the providers table. The providers table is implemented by MAPI to supply service providers with information about all of the service providers in the current session. Whereas a typical session involves many messages and many recipients tables, there is only one providers table.

You can determine the implementor of most tables by the table type. As mentioned above, recipients tables are always implemented by the implementor of messages, a message store provider. Attachment tables, which also contain message information are also implemented by message store providers. MAPI implements several different tables, some for use by clients, some by service providers, and some by both. The message stores table is used by clients to retrieve information about the various message store providers installed in the current profile. Display tables are used by service providers to help with user interface implementation. The message services and profiles tables can be used by either clients or service providers to support message service configuration.

As stated above, the type of table usually determines its implementor. However, MAPI defines two types of tables that are implemented by both address book and message store providers. Contents tables hold the contents of address book containers when implemented by address book providers and folders when implemented by message store providers. Hierarchy tables show the hierarchy of the integrated address book when implemented by address book providers and the message store hierarchy when implemented by message store providers.

Some tables are static, meaning that once they are instantiated they will not change. Although the underlying property data might change, the table's view of it does not. Other tables are more dynamic; the view is updated when the underlying data changes. The message stores table is an example of a dynamic table. It always reflects what is in the profile. If a client's user edits the profile, changing the default message store, for example, the values of the PR_DEFAULT_STORE properties for the affected message stores are updated in the table view.

MAPI defines several data structures that are useful for working with tables. There is one data structure to represent a single row and another to represent a set of rows. This data structure is used frequently in table operations. Other data structures are used in specific operations, such as sorting and specifying restrictions. The next section describes these data structures in detail.

Table Data Structures

There are three structures that are used to describe table data: one for the actual data in a row and two for data related to how that data is shown to the user. The row set, or **SRowSet**, is a hierarchical data structure that contains the data for each row. The sort order set, or **SSortOrder**, is a hierarchical data structure that describes the order in which the rows of a table are displayed. The **SRestriction** data structure is used to establish search criteria.

About Row Sets

Table data is organized using a hierarchy of data structures. At the top of the hierarchy is the row set, or **SRowSet** data structure. Each **SRowSet** is made up of an array of row, or **SRow** structures and a count of the number of row structures in the array. Each **SRow** contains an array of property values, or **SPropValue** structures and, again, a count of the number of structures in the array. The **SPropValue** array contains the property tags and values for the properties in the columns of the table.

The following diagram illustrates the bottom two layers of the hierarchy, starting with the **SRow**:

{ewc msdncd, EWGraphic, group10838 1 /a "SDKEX_17.bmp"}

Figure 2   An SRow data structure

About Sort Order Sets

The sort order set data structure, **SSetOrderSet**, contains information about the ordering of data in a table. It is usedfor both regular sorting and categorized sorting and contains an array of sort order (**SSortOrder)** data structures. Other members of the structure specify the number of sort order structures included, and the number of categories that are included and should be expanded when the table is displayed. Sort order sets are defined as follows:

```
typedef struct _SSortOrderSet
{
  ULONG       cSorts;
  ULONG       cCategories;
  ULONG       cExpanded;
  SSortOrder  aSort[];
} SSetOrderSet, FAR * LPSSortOrderSet;
```

The **SSortOrder** data structure is made up of a property tag and a constant that indicates whether the table should be in ascending or descending order or, for categorized tables, whether the property should become a separate category. **SSortOrder** structures are defined as follows:

```
typedef struct _SSortOrder
{
  ULONG  ulPropTag;
  ULONG  ulOrder;
} SSetOrder, FAR * LPSSortOrder;
```

About Restrictions

Building restrictions allows you to specify exactly how much and what type of data you wish to include in a table view. Restrictions are built using the **SRestriction** data structure which containsa union of more specialized restriction structures and an indicator of the type of structure included in the union.

The **SRestriction** data structure is defined as follows:

```
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;
```

To illustrate how the structures involved in building a restriction fit together, consider the **SAndRestriction** data structure. The **SAndRestriction** data structure allows you to compare two or more restriction structures using the logical AND operator. To build a logical AND restriction, you specify RES_AND in the *rt*, or restriction type, field and include the appropriate number of pointers to the **SRestriction** data structures to be compared.

{ewc msdncd, EWGraphic, group10838 2 /a "SDKEXa.bmp"}

Figure 3 An SRestriction data structure of type RES_AND

Some common restrictions include the bit mask restriction, stored in the **SBitMaskRestriction** structure, for testing if all of the bits in a flag are on or off, and the comment restriction, stored in the **SCommentRestriction** structure, for associating a name with a set of named properties. Clients might want to use the comment restriction for viewing hidden text in a word processing document. The **SSubRestriction** structure is used to place a subobject restriction on a table. Subobject restrictions are useful operations that allow you, for example, to search for all messages from a particular recipient or for all attachments in a search-results folder. The property restriction, stored in the **SPropertyRestriction** structure, allows you to compare the value of a property with another value and exclude those rows that have values which do not match. A property restriction can be used, for example, to search for a particular type of attachment, such as an executable file.

About IMAPITable

MAPI tables implement the **IMAPITable** interface. **IMAPITable** is a read-only interface, meaning that its methods do not allow any modifications to the table data. Because service providers as well as MAPI implement many different types of tables to be used by other providers and clients, **IMAPITable** is one of the most commonly implemented and used interfaces in MAPI.

As stated earlier, there are many types of tables and each type has its own **IMAPITable** implementation. The differences in implementation are apparent in the work that each method does and the amount of support each method provides. For example, consider the difference between a hierarchy table, which is a multi-level list of rows, and the providers table, which is a flat list. Multi-level tables can expand and contract their view; single level tables do neither. Therefore, hierarchy tables will fully implement the **GetCollapseState** and **SetCollapseState** methods in **IMAPITable** and the providers table will return MAPI_E_NO_SUPPORT.

For more information on using and implementing **IMAPITable**, see the *MAPI Programmer's Reference* The "Working with Tables" section describes how to implement frequent tasks.

About ITableData

There is another interface, **ITableData**, that MAPI implements to help service providers access and maintain the underlying data upon which the table view is based. **ITableData** is a read-write interface, allowing users to add, delete, and change the data displayed in the rows. Like **IMAPITable**, **ITableData** does not support any of its own properties and inherits directly from **IUnknown**. MAPI provides two different interfaces for table access to allow users to simultaneously view and modify table data.

Service providers call the API function **CreateTable** to instantiate a table data object and retrieve a pointer to an **ITableData** interface implementation. One of the parameters to **CreateTable** is an array of property tags specifying the properties that should be included in the table. Each property tag includes a property type and identifier. The property type cannot be PT_UNSPECIFIED, PT_ERROR, or PT_NULL.

The **ITableData** interface operates with an index column, a property that is used to locate rows for modification. Specifying a multi-valued property for the index column is not possible.

Figure 4 shows the relationship between **IMAPITable** and **ITableData**. The service provider object on the left controls the underlying data using **ITableData**. Each time a client requests a view of the data, the service provider creates a unique MAPI table object based on the request and returns a pointer to the **IMAPITable** implementation.

{ewc msdncd, EWGraphic, group10838 3 /a "SDKEX.bmp"}

Figure 4. Relationship between IMAPITable and ITableData.

The **ITableData** methods are summarized in the following table..

Method	**Description**
HrGetView	Returns a MAPI table object
HrModifyRow	Adds or modifies a row in a table
HrDeleteRow	Deletes a row from a table
HrQueryRow	Returns the specified property values for a row in a table
HrEnumRow	Returns all of the property values for a row in a table
HrModifyRows	Adds or modifies multiple rows in a table
HrNotify	Requests a table notification be sent
HrInsertRow	Inserts a row into a table
HrDeleteRows	Deletes multiple rows in a table

When clients make calls to retrieve tables from service providers, service providers can call the **ITableData::HrGetView** method. **HrGetView** returns a table object that can be sorted or restricted.

ITableData defines separate methods for operating on a single row or operating on multiple rows. For example, the **HrDeleteRows** method will delete as many rows as are specified on input. If a particular row is not found, the method skips it. The number of successfully deleted rows is stored in the *cRowsDeleted* output parameter passed back. A single notification is generated, regardless of the number of rows deleted.

Both **HrModifyRow** and **HrModifyRows** will make changes to existing rows. If the target row does not exist, a row is added.

The **HrEnumRow** method uses the internal sequential number assigned to each row when it is added to the table to get rows out of a table in the order that they were put in.

The **HrInsertRow** method is used to insert a row at a specified position.

The **HrNotify** method can be used to generate a notification to all MAPI table objects that share a view of the data. A table modified notification is sent for all rows that match the property values passed into **HrNotify** as an input parameter. A notification is sent regardless of whether the data actually changed. When display tables, or tables that show information about user interface controls, receive this type of notification, they reload the data associated with the control.

Working with Tables

Working with a MAPI table is like working with a relational database table. Many of the table operations, such as setting search criteria, retrieving rows, rearranging the view, are typical database table operations. Sometimes these operations take a long time. As a table user, consider allowing the implementor to process asynchronously and take advantage of deferred errors. As an implementor, bear in mind that there is no way to show progress on these operations. Therefore, try to make your implementations as efficient as possible.

This section covers in detail these common operations from the perspective of the table user and the table implementor.

Setting Columns

All tables have a group of properties that make up their default column set. For most tables, you have the choice of accepting this default set of columns or modifying it, removing certain columns or adding new ones. Modifying the column set of a table is known as setting columns, a task that is performed through the **IMAPITable::SetColumns** method.

To retrieve rows from a table, you call **IMAPITable::QueryRows**. If you make the call to **QueryRows** before you call **IMAPITable::SetColumns** to modify the column set, the columns that you will be returned will be those in the default column set. Some tables, like the session table, do allow column set modification. Whenever you call **QueryRows** on the session table, you will always get back the same set of columns.

In addition to the default column set, some tables have a required set. This is the group of properties that must be present in every row of the table. As a table implementor, you must provide these columns, and as a table user, you are guaranteed that these columns will be present. Sometimes there is overlap between column sets when the default and required sets are the same. Tables that do not allow you to modify their column set, like many MAPI tables, operate this way.

As a table implementor, the set of columns that you permit your caller to pass as an input parameter to **SetColumns** will typically be the set of columns that you return in your **IMAPITable::QueryColumns** implementation. However, you can allow other columns to be specified if you are able to add them dynamically to your default column set. If extra columns are specified that you cannot handle, you can return PT_ERROR in the appropriate member of the property problem array when your caller calls **IMAPITable::GetLastError**.

SetColumns can be prompted to work asynchronously or synchronously; however, table implementors can reject the request and return MAPI_E_NO_SUPPORT. Table users set the TBL_BATCH flag in their **SetColumns** call to request that it work asynchronously.If the table implementor supports asynchronous column modification, **SetColumns** will return before the operation completes. Upon completion, a table notification is sent and, upon receipt of this notification, the user can assume that the column set is now as requested.

The **HrAddColumnsEx** API function allows you to modify the current column set of a table, much like **SetColumns.** However, **HrAddColumnsEx** differs from **SetColumns** in that it never removes columns. This call only adds to the beginning of the current set. Whatever was in the column set before the call will still be there after the call, positioned after the newly added columns. **HrAddColumnsEx** is used most frequently to add to the column set of recipient tables.

Displaying Multi-Valued Columns

A multi-valued column is a column that contains the value for a multi-valued property. Multi-valued properties are displayed in a table in one of two ways:

- As a single row, with all of the values appearing in the column. This is the default.
- As a series of single valued properties appearing in multiple rows. The values of the other columns in the row are duplicated. 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. Each multi-valued property should have as many rows as it has values.

As a table user, you request a multiple row display by setting the MVI_FLAG flag in the property type and applying the MVI_PROP macro to the property tag

As an implementor, you do not have to support the alternative of using multiple rows to display multi-valued properties. You may return MAPI_E_TOO_COMPLEX when a caller requests it. However, the rule of thumb is if the underlying data for your table is based on multi-valued properties, you should support this alternative.

Working with String Properties

String property values can be represented in tables using standard 8 byte characters, property type PT_STRING8, or 16 byte Unicode characters, property type PT_UNICODE. Table implementors are free to choose exactly how much Unicode support they offer. Because Unicode adds value to both clients and service providers by extending the feature set, supporting Unicode wherever possible is recommended. All of the MAPI table object implementations support Unicode.

Many table methods accept a flag that dictates whether or 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 MAPI_E_BAD_CHAR_WIDTH.

The values for string properties can be very long. When a lengthy string property is returned to you as a table user, its size might or might not be truncated, depending on the implementor. Most table implementors truncate a column containing a string property to 255 bytes. However, when the same lengthy string property is returned to you as a user of the object to which the property belongs, its entire size is available. Truncation of string property values affects only its display in a table.

Limiting string properties to 255 bytes may or may not impact any restrictions that are built. Some implementors allow a restriction to be built that matches against the entire property value, and other implementors allow the match to occur only on the first 255 bytes. Depending on the implementor, you may be able to build a restriction that matches against the entire property rather than the first 255 bytes.

Table Positioning

MAPI tables have two resources for determining and establishing a position. One of the resources is the cursor, a built-in indicator of the current position. Cursors cannot be changed by a table's user; they are set by the table's implementor to always reflect the user's present position.

The other resource is the bookmark, an indicator of any position in the table. Bookmarks are used to mark a particular position in a table, enabling you to return to that position at a later time. Using bookmarks can make your table operations faster.

A call to **IMAPITable::CreateBookmark** establishes a bookmark at the current position. Table implementors are required to at least provide bookmarks for thecurrent position, the beginning of the table, and the end of the table. Other bookmarks are optional. To position the cursor using a required bookmark, pass one of the following constants to either **IMAPITable::SeekRow** or **IMAPITable::FindRow**: BOOKMARK_CURRENT, BOOKMARK_BEGINNING, or BOOKMARK_END. Because bookmarks are limited resources, it is important to free them by calling **IMAPITable::FreeBookmark**. At times **CreateBookmark** might be unable to allocate sufficient memory, in which case the error MAPI_E_UNABLE_TO_COMPLETE is returned.

The **IMAPITable::SeekRow** or **IMAPITable::SeekRowApprox** methods change a table's position. **SeekRow** moves the cursor to a position determined by an established bookmark and a specified number of rows; **SeekRowApprox** uses a fraction to determine where to move the cursor. **SeekRow** is useful when you want to position your table ten rows from the current position, for example, or when you want to start over at the beginning. **SeekRowApprox** is useful for implementing a scroll bar; it lets you move to the row that is one third of the rows from the beginning of the table, for example.

The **IMAPITable::QueryPosition** method returns the row at the position indicated by a specified fraction. **QueryPosition** is used to get the row that is one-third of the way into the table, for example. It can also be used for scroll bar implementation. Implementations of **QueryPosition** and **SeekRowApprox** can be quite expensive on categorized tables if the implementations choose to be accurate.

Retrieving Rows

Users of tables typically interact with the table data by retrieving one or more rows in a predefined order. There are three available methods to use to retrieve rows from a table: **IMAPITable::QueryRows**, **ITableData::HrQueryRow**, and **HrQueryAllRows**.

Clients call **IMAPITable::QueryRows** to retrieve one or more rows at a time, beginning with the current cursor position. The properties that you get back are either the ones that you have specified in a **SetColumns** call or the ones included in the table's default column set, if **SetColumns** has not been called.

Service providers call **ITableData::HrQueryRow** to retrieve a specific row with all of the properties associated with the row.

All table users can call **HrQueryAllRows**, a helper function that combines the functionality of the **SetColumns**, **SortTable**, **Restrict**, and **QueryRows**. **HrQueryAllRows** always calls **SeekRow** to get to the beginning of the table before starting its processing. If you want to retrieve all of the rows of a table, this function offers a valuable shortcut. However, it is possible to run out of memory when you are retrieving rows from a table with this function. To process an entire table while limiting the amount of memory consumed, set the *crowsMax* parameter to a reasonable number and call **HrQueryAllRows** in a loop. This parameter limits the number of rows that are returned with each call. The position of the cursor upon return from **HrQueryAllRows** is undefined. However, usually it will be at the bottom of the table.

After you have retrieved a row, be careful to check for unexpected or invalid column values, such as PT_ERROR or MAPI_E_NOT_FOUND. Sometimes properties are included in the column set of a table in spite of the fact that they are not supported. The table implementor places a value in the column to indicate that there is no valid value. This is a different error than a column missing from a table altogether.

Expanding the View

All tables support to some degree ways to create customized views. Customized views are created by building restrictions, specifying a sort order, or defining a column set. Restrictions, sometimes called search criteria, narrow the view of the data. For example, if the table you are using contains the contents of a large folder, there may be many hundreds of messages in it. Maybe you are interested in looking only at the most recent messages or the ones from a particular recipient. Building a restriction on the table enables you to do just that. The data for these rows remains unchanged; it is just not part of your view. MAPI provides many different ways to build restrictions, such as specifying a bitmask or a property type as a basis for comparison. Some tables support all restriction types; other tables support only a few.

Rows can be displayed as flat or as multi-level lists. Most tables are flat lists of rows; only two tables, the hierarchy table and the categorized contents table, are multi-level. Multi-level tables provide two views of their data: an expanded view and a collapsed view. When a top level row is expanded, it is marked with a minus sign and all of its lower level rows are visible. When a top row is collapsed, it is marked with a plus sign and none of its lower level rows are visible. Many hierarchy tables that display message store contents use this feature to allow users to view the messages within many sets of folders without having to scroll.

The **IMAPITable** methods, **SetCollapseSet** and **GetCollapseState,** help you to work with data associated with the state of the display. **GetCollapseState** allows you to save data that enables you to recreate a multi-level display; **SetCollapseState** recreates the display based on this saved data. The data that you save includes:

- the row where you should begin rebuilding the collapsed or expanded state
- all of the plus and minus indicators at all table depths
- the top row for the current view.

The PR_INSTANCE_KEY property for the row where you begin rebuilding the display is stored in the *lpbInstanceKey* parameter passed to **GetCollapseState**; this property is used to set the bookmark that identifies this row for the **SetCollapseState** method.

As an implementor of **SetCollapseState**, operate as if you had been asked to reload your table. This method reestablishes the state of the display to the state that exists with the last **GetCollapseState** call. You are responsible for keeping the sort order and restrictions exactly the same as they were before the call. Also, you must return a bookmark in the *lpbkLocation* parameter that corresponds to the row specified by the instance key parameter passed to **GetCollapseState**. If this row no longer exists, pass back BOOKMARK_BEGINNING in *lpbkLocation*. The caller should free the bookmark by calling **IMAPITable::FreeBookmark.**

The **ExpandRow** and **CollapseRow** methods allow the state of a row to be modified. As implementors of these methods, you do not have to send a notification on a top level row indicating that the value of its PR_ROW_TYPE property has changed. If clients maintain the data, they can update their copy of the lower level rows when making the **ExpandRow** or **CollapseRow** call.

When **ExpandRow** is called, the top level row to undergo the expansion is represented by the instance key specified in the *pbInstanceKey* parameter. The returned row set is inserted in the table after this row. No notification is generated by these rows appearing in the table. If the expanded section is larger than the value of the *ulRowCount* parameter, then only *ulRowCount* rows are returned. The expand state is not saved across restrictions.

If **ExpandRow** cannot return all of the requested rows because of insufficient memory, the total number of rows that were added is returned in the *lpulMoreRows* parameter. This output parameter returns the number of additional rows that would have been returned if resource limits had not been exceeded. If the contents of *lpulMoreRows* is non-zero, then the position in the table is adjusted to the first of the rows that could not be returned.

When **CollapseRow** is called, the number of rows removed, including top level and lower level rows, is returned in the contents of the *pulRowCount* parameter. No notification is generated when these rows disappear from the table.

A bookmark pointing to a row which is collapsed out of the view is still valid. If used, it is positioned at the next visible row. The call on which such a bookmark is used returns MAPI_W_POSITION_CHANGED. Your implementation of **CollapseRow** can move the bookmark at the time of collapse or the time of use. If the position is changed at the time of collapse, you must set and return a bit with the bookmark to indicate whether its position has changed since its last use.

Sorting Tables

Specifying a sort order involves supplying one or more columns whose values dictate the order in which the rows of your table are displayed. Maybe youwant to see all of your messages in alphabetic order by subject so that all of the threads of a conversation are together. The sort feature on MAPI tables enables you to customize your view in this way. A special type of sorting, called categorization, is available with contents tables. Categorization sorts data by groups, or categories.

The rows that you retrieve can be sorted using **IMAPITable::SetColumns** or **IMAPITable::SortTable**. **SetColumns** establishes an initial sort order using the columns that it includes in the table. When a call is made to retrieve the rows, they will appear in this order. Tables that do not allow their column set to be altered often have a default sort order. The providers and message service tables, for example, are sorted by PR_DISPLAY_NAME.

SortTable establishes a subsequent sort order, sometimes using columns that appear in the table view, sometimes using columns that do not appear in the view, and sometimes using predefined categories. **SortTable** sorts rows in ascending or descending order, depending on your choice as a caller. All NULL entries are placed last.

The instance key, or PR_INSTANCE_KEY property, uniquely identifies each row in a table. PR_INSTANCE_KEY is a required column in most tables. Instance keys are unique to each table instance; two views of the same table do not share instance keys. The PR_INSTANCE_KEY property is used as an index column, for notification, and, if the table supports it, for categorization.

All implementations of **SortTable** allow users to define a sort order based on columns that are currently part of the view. Some implementations will add value by supporting a sort on inactive columns or categories. Implementations that cannot support these additional features can return MAPI_E_TOO_COMPLEX if asked to perform an unsupported sort.

Whenever **SortTable** fails, the sort order that was in effect before the failure will still be in effect.

If you are planning to limit your column set, build a restriction, and request a sort, make sure you do those tasks in that exact order. That is, call **SetColumns** before **Restrict** and **Restrict** before **SortTable**. This limits the number of rows and columns that will be sorted, thereby improving performance.

Using Categorized Tables

A categorized table is a table that organizes its data into named groups. Categorization is a feature that some contents tables support.

To request a categorized table, call **SortOrder** specifying the properties that you want to use as categories in the array of **SSortOrder** data structures. Categories may be in either ascending or descending order. The *cCategories* field in the **SSortOrderSet** data structure should be set to the number of categories that you have specified and the *cExpanded* field should be set to the number of categories that are initially displayed in their expanded state.

After **SortOrder** completes, you can call **SetColumns** to define the columns that you would like to include in your view. You can specify any of the columns from the **SSortOrder** array or columns that were not included in the array. The ordering of the columns in the **SSortOrder** structure and in the **SetColumns** call do not have to match.

When you call **QueryRows** to retrieve your data, each row will have a PR_ROW_TYPE property associated with it. PR_ROW_TYPE is a 32-bit value that indicates whether a row is a lower-level data row (TBL_LEAF_ROW) or a top level category row which should be shown as expanded (TBL_EXPANDED_CATEGORY) or collapsed (TBL_COLLAPSED_CATEGORY).

There will be a category heading row containing one or more of the categorization properties. All other properties, except for PR_INSTANCE_KEY and PR_ROW_TYPE, are not returned. The top-level category rows contain only the category properties in the following pattern: the most top level row contains the first category, the second level row contains two categories, and the third level row contains three categories. This pattern continues until the category list is exhausted.

To implement **SortTable** for categorization, begin by expanding the first category, and keep expanding the categories in the order they appear in the **SSortOrder** array until the number expanded equals the value of *cExpanded*. The rest of the remaining categories are collapsed. If the *cExpanded* field is equal to zero, only the rows containing the top-level category names are visible. If the *cExpanded* field is equal to one less than the number of categories, then all of the category rows are visible and non of the lower-level non-category rows are visible. If the *cExpanded* field is equal to the number of categories, then the table is fully expanded.

The PR_INSTANCE_KEY property will change on a row in a table that is categorized by a multi-value column after a call to **SortTable**, **SetColumns**, or **Restrict.** This is because the data for a multi-value column might be included in multiple rows, with one row for every value. When a table is categorized ona column that contains a single value property, the value of its PR_INSTANCE_KEY property remains constant.

The PR_DEPTH property indicates a level of a row. PR_DEPTH is set to 0 for top-level categorization rows and 1 for second-level categorization rows. For each level, the PR_DEPTH value increases by one. Lower-level data rows have a value of n where n is the number of categories for PR_DEPTH.

The PR_UNREAD_COUNT property, a computed property that contains the number of data rows in a category, is included as a column on a category row. This is the only property that appears on a category row that cannot be used for a category. If an attempt is made to define PR_UNREAD_COUNT as a category in the **SortTable**; the table implementor should return MAPI_E_INVALID_PARAMETER from the **SortTable** call.

Building Restrictions

One of the most common ways users work with tables is to search for data that matches specific search criteria. Users specify search criteria by building restrictions. As described earlier in the "About Restrictions" section, there are many types of restrictions, each represented with its own unique data structure. When you set a restriction, you select the data structure that applies to the type of restriction you are building, fill an instance of the structure with the appropriate data, and pass it in your call to **IMAPITable::Restrict** or **IMAPITable::FindRow**.

The **IMAPITable::Restrict** method applies the restriction to the table without retrieving any rows. To see the results of a **Restrict** call, you must call **IMAPITable::QueryRows**. **IMAPITable::FindRow**, however, applies the restriction and retrieves a row, returning the next matching row in the table. **Restrict** is used when you want to work with the rows that result from applying a restriction for an extended length of time whereas **FindRow** applies a temporary restriction, in existence only for the duration of the call.

Content restrictions use the concept of fuzzy level to describe the degree of exactness or looseness that is applied when searching text. The *ulFuzzyLevel* parameter in the **SContentRestriction** data structure can have six different values as follows:

Fuzzy Level	**Description**
FL_FULLSTRING	The exact string is searched for as a strict match on the entire property.
FL_SUBSTRING	A match occurs if the string occurs anywhere within the property.
FL_PREFIX	A match occurs only if the string occurs at the beginning of the property. This flag if present, takes precedence over FL_FULLSTRING.
FL_IGNORECASE	The comparison ignores differences in the case of letters.
FL_IGNORENONSPACE	The comparison ignores differences in non-spacing characters or in diacritics.
FL_LOOSE	Loose matching.

These constants can be OR'ed together. For example, FL_SUBSTRING **|** FL_IGNORECASE results in a loose match substring.

Providers can choose to ignore the FL_IGNORECASE and the FL_IGNORENONSPACE values. In fact, the degree of support for fuzzy levels is implementation-specific. Therefore, when you call **Restrict** or **FindRow** and pass in an unsupported fuzzy level, you will most likely get back the error MAPI_E_TOO_COMPLEX.

MAPI supports a limited form of regular expression notation for use in building restrictions. A regular expression specifies a set of character strings. A member of this set of strings is said to match the regular expression. An ordinary character is a one-character regular expression that matches itself.

You form regular expressions using special characters like the period ".", asterisk "*", and opening bracket"[". If you want to use a special character literal (that is, without its regular meaning) in a regular expression, you must quote it by preceding it with a backslash, "\". 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 relational operator is RELOP_RE, so do not be tempted to quote the special characters everywhere.

Characters that are considered special, except when they appear within brackets, include:

- period (.)
- asterisk (*)

- opening bracket ([)
- backslash (\)

Other characters are special under certain circumstances. These characters are as follows:

- The caret (^), which is special at the beginning of an entire regular expression or when it immediately follows an opening bracket([]).
- The dollar sign ($), which is special at the end of an entire regular expression.

The period is a one character regular expression that matches any character except the new line character. A non-empty character string enclosed in 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 the new line character and the remaining characters in the string.

The asterisk (*) 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 or last in the string. The closing bracket (]) does not terminate such a string when it is the first character within it for example, []a-f], matches either a closing bracket (]) or one of the letters "a" through "f". The period, asterisk, opening bracket, and backslash lose their special meaning within such a string of characters.

Use the following rules to construct regular expressions from one-character regular expressions:

1. A one-character regular expression matches itself.
2. A one-character regular expression followed by an asterisk (*) 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.
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.
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.
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.
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):

1. A caret (^) at the beginning of an entire regular expression constrains that regular expression to match an initial segment of a line.
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.

Table Notifications

Users can register to be notified when a change to a table occurs by creating an advise sink object and calling **IMAPITable::Advise**. Typical changes that cause notifications to be sent include the addition or deletion of a row or column and any critical error.

Table implementors can manage the registration and generation of notifications by hand or rely on the helper methods in **IMAPISupport**. Either way, when you send a table notification, set the *ulEventType* field in the NOTIFICATION structure to TABLE_NOTIFICATION and include a TABLE_NOTIFICATION data structure in the *info* field. Set the *propIndex* field to the value of the PR_INSTANCE_KEY property for the affected row and the *propPrior* field to the value of the PR_INSTANCE_KEY property for the row preceding the affected row. The **SRow** structure contains all of the property data for the affected row. The properties that are included in the **SRow** structure are ordered using the column set that was established from the previous **IMAPITable::SetColumns** call. Both the property values and the property tags are in this order.

The TABLE_NOTIFICATION data structure is defined as follows:

```
typedef struct _TABLE_NOTIFICATION
{
         ULONG       ulTableEvent;
         HRESULT     hResult;
         SPropValue  propIndex;
         SProvValue  propPrior;
         SRow        row;

}   TABLE_NOTIFICATION;
```

There are several different event types that you can specify in the *ulTableEvent* field. As an advise sink implementing **IMAPIAdviseSink::OnNotify**, you should be prepared to handle all of these. A TABLE_CHANGED event is a somewhat nebulous event indicating that only something about the table has changed, but no details are available. The table's state is as it was before the event. All instance keys (PR_INSTANCE_KEY properties), bookmarks, and current positioning are still valid. A TABLE_CHANGED event is sent when, for example, the underlying data is stored in a database and the database is replaced.

The TABLE_ERROR event is sent when a critical error occurs, usually during an asynchronous operation. 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 not returned as the result of a method call.

The TABLE_RELOAD event indicates a need to re-read the data and start over. When clients receive this event, they should assume that nothing about the table is still valid. All bookmarks, instance keys, status and positioning information are obsolete.

The TABLE_RESTRICT_DONE, TABLE_SETCOL_DONE, and TABLE_SORT_DONE events signal the end of an asynchronous operation - either to set a restriction, set columns, or perform a sort on the table.

The TABLE_ROW_ADDED notification is sent when a new row has been added to the table. The *propPrior* instance key is for the row above where the row was added. As a client receiving this event, bear in mind that although the data in the *propPrior* and *SRow* fields was correct when the notification was generated, it might no longer be correct when you receive the notification. Between the time the notification was generated and the time that it was sent, other changes might have occurred.

The TABLE_ROW_DELETED event identifies a row that is no longer part of the table. The TABLE_NOTIFICATION structure does not contain a value for the *propPrior* field for this event.

The TABLE_ROW_MODIFIED event identifies a changed row. The **SRow** structure contains new data for the row. If the affected row is now the first row in the table, the value of the *propPrior* field is PR_NULL. Clients receiving this event must duplicate the entire SRow structure if they want to keep a copy of the properties included in the structure. Multiple TABLE_ROW_MODIFIED events should be in chronological order with respect to the view that is seen by the user.

Now, for a couple of warnings. First, because of the asynchronous nature of event notification, the data passed in the TABLE_NOTIFICATION structure may not be the most current information about the state of the table. Clients may be aware of changed or added rows before they are officially notified.

Second, although the order of the properties that are returned in the TABLE_NOTIFICATION structure should be the order that you specified in your last **SetColumns** call, do not assume that this is the case. Senders of table notifications are not required to follow any rules when ordering these properties.

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.

Types of Tables

MAPI defines many different tables to enable you to readily access and manipulate the important properties of many types of objects. These next sections focus on these different types and their own unique characteristics. Each section includes the following information, if appropriate:

- methods for instantiation and how to retrieve the **IMAPITable** interface pointer
- typical implementor
- typical user
- properties for the required columns
- properties for the default columns
- properties for the default sort order
- characteristics specific to the table

Attachment Table

The 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 clients, who call **IMessage::GetAttachmentTable** to retrieve a pointer to the table object.

Attachment tables have three required columns: PR_ATTACH_NUM, PR_RECORD_KEY, and PR_RENDERING_POSITION. Message store providers are free to add other columns.

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.

A common property for many objects, PR_RECORD_KEY is used in attachment tables to alsoidentify 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. Not all attachments use PR_RENDERING_POSITION. A value of 0xFFFFFFFF indicates that an alternate method for rendering the attachment should be used.

The attachment table is dynamic, so it can change while it is open. That is, if the 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.

Attachment tables, when initially opened, are not necessarily sorted in any particular order.

Contents Table

The contents table lists information about child objects, such as messaging users, distribution lists, and messages, that are contained in a parent object, such as an address book container, or folder. The information contained in a contents table is used to display address book recipients and messages to users of client applications.

Address book and message store providers implement contents tables as part of the **IMAPIContainer** interface that address book container objects and folder objects support. To create a contents table, clients call **IMAPIContainer::GetContentsTable**. This method 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. This allows the call to succeed, returning some or all of the data. For information on dealing with deferred errors, see the "Handling Errors" section in Chapter 8.

The MAPI_ASSOCIATED flag, applicable only to message store contents tables, indicates that only associated entries should be included in the new table. Message store contents tables have two types of entries: standard and associated. When a client creates a message, a flag can be set on the **IMAPIFolder::CreateMessage** call to indicate that the new message is somehow special. Perhaps it contains hidden data or is an alternate representation of a standard message. Associated and standard messages are stored separately, and clients ask for one or the other when creating contents tables.

In addition to the use of the MAPI_ASSOCIATED flag, there are a few other differences between the address book contents tables and message store contents tables. With message store contents tables, a given column is empty if the message store does not support the associated message property.

All contents tables use the PR_INSTANCE_KEY property as the index.

Contents tables have a lengthy list of required columns. Some of the columns are required by both types of contents tables; other columns are specific to address books or message stores. The following table lists all of the required columns and the table or tables in which they belong.

Column	**Address Book**	**Message Store**
PR_ADDRTYPE	X	
PR_CLIENT_SUBMIT_TIME		X
PR_DISPLAY_NAME	X	
PR_DISPLAY_CC		X
PR_DISPLAY_TO		X
PR_DISPLAY_TYPE	X	
PR_ENTRYID	X	X
PR_HASATTACH		X
PR_INSTANCE_KEY	X	X
PR_LAST_MODIFICATION_TIME		X
PR_MAPPING_SIGNATURE		X
PR_MESSAGE_CLASS		X
PR_MESSAGE_DELIVERY_TIME		X
PR_MESSAGE_FLAGS		X
PR_MESSAGE_SIZE		X
PR_MSG_STATUS		X
PR_NORMALIZED_SUBJECT		X
PR_OBJECT-TYPE	X	X

PR_PARENT_ENTRYID	X
PR_PRIORITY	X
PR_RECORD_KEY	X
PR_SENDER_NAME	X
PR_SENSITIVITY	X
PR_STORE_ENTRYID	X
PR_STORE_RECORD_KEY	X
PR_SUBJECT	X

There are a few more properties that are not part of the required column set, but are often included in address book contents tables. These optional columns are:

PR_EMAIL_ADDRESS	PR_SEND_RICH_KEY
PR_SEARCH_KEY	PR_TRANSMITABLE_DISPLAY_NAME

Message store providers must include PR_PARENT_DISPLAY for search-results folders contents tables.

This list of properties does not include all the properties that can be included in a contents table. Clients may request other properties; it is up to the service provider as to whether the table can include them.

Display Table

The display table describes how to show a dialog box. Each row represents a control on the dialog box; each column represents a property of the control. One of the common ways that display tables are used in MAPI is for the display of configuration property sheets. With a display table, you can copy property data from your local workstation to a server workstation and have the server control a display of the data on your workstation. Building a display table is similar to creating a program with a scripting language.

Display tables are implemented by MAPI and used mostly by service providers, with occasional use by clients. To build a display table, you fill in one or more display table page, or DTPAGE. data structures and pass them along with several other parameters to the API function, **BuildDisplayTable**.

BuildDisplayTable returns to callers a pointer to a MAPI table object and, if desired, a pointer to a read-write table data object as well. Clients who require access to the table can be given the **IMAPITable** pointer to the read-only view. The table data object is returned only if a valid **ITableData** pointer is passed into the call. Service providers that do not require access to the underlying data can pass NULL.

The DTPAGE data structure, used to describe the dialog box, includes a count of the number of controls that appear on the dialog box, the name of the dialog resource, and a pointer to an array of display table control, or DTCTL, data structures.

The DTCTL data structure contains a constant that indicates the control type, a set of flags, an item identifier, a character filter for edit or combo box controls, notification data, and a union of different control type data structures. Each control type structure holds information specific to the type of control. The control type corresponds to the PR_CONTROL_TYPE property; the set of flags corresponds to the PR_CONTROL_FLAGS property.

Each of the controls in a display table has notification data associated with it.

The DTCTL data structure is defined as follows:

```
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;
      LPDTBLDLLBX        lpdllbx;
      LPDTBLCHECKBOX     lpcheckbox;
      LPDTBLGROUPBOX     lpgroupbox;
      LPDTBLBUTTON       lpbutton;
      LPDTBLRADIOBUTTON  lpradiobutton;
      LPDTBLMVLISTBOX    lpmvlistbox;
      LPDTBLMVDDLBX      lpmvddlbx;
      LPDTBLPAGE         lppage;
  }  ctl;
} DTCTL, FAR *LPDTCTL;
```

Because there are so many different types of control objects that represent the rows in a display table, there is no generic way to handle a table notification when it is received. The processing that occurs when you receive a notification depends on the type of object that has been affected.

The following table describes how all of the possible control types handle a table notification. The display table control types are listed by their structure names.

Display Table Control Type	**Recommended Action**
DTBLBUTTON	Retrieve the button's value.
DTBLCHECKBOX	Reread the value for *ulPRPropertyName.*
DTBLCOMBOBOX	Open the table, reread all of the rows including the value for *ulPRPropertyName.*
DTBLDDLBX	Reopen the table, read all of the rows, and call **GetProps** to retrieve the values for *ulPRDisplayProperty* and the *ulPRSetProperty* properties
DTBLEDIT	Reread the property tag and redisplay.
DTBLGROUPBOX	Ignor the notification.
DTBLLABEL	Ignor the notification.
DTBLLBX and *ulPRSetProperty* is PR_NULL (multi-select list box)	If one of the columns is an entry identifier, this is considered to be the same as if you're looking at the contents table of a distribution list of an address book container, for example, a details display on a distribution list. There is no concept of selection. Simply refresh the list box. Do not close or reload the object.
DTBLLBX and *ulPRSetProperty* is not PR_NULL (single selection list box)	Read the set property and try to identify it.
DTBLMVDDLBOX (list box with multi-value string properties)	Reread the property and re-populate the list box.
DTBLMVLISTBOX	Reread the property and re-populate the list box.
DTBLPAGE	If *ulbLpszComponent* is non-zero, there is an encapsulated string that points to a help file. No notifications on this control; everything is static.
DTBLRADIOBUTTON	The *ulcButtons* field contains the number of buttons in the group. The *ulPropTag* field is the name of the property associated with the buttons. When a notification is received, reread the *ulPropTag* and make the appropriate selection on the controls.

There are eight properties that are part of the required column set for display tables.

PR_XPOS	PR_YPOS
PR_DELTAX	PR_DELTAY

PR_CONTROL_TYPE	PR_CONTROL_FLAGS
PR_CONTROL_STRUCTURE	PR_CONTROL_ID

Hierarchy Table

The hierarchy table displays information about child objects, such as messages, folders, messaging users, and distribution lists, that are contained in the current container. Each row of a hierarchy table contains a set of columns holding information about one child 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. Clients retrieve a hierarchy table by calling the **IMAPIContainer::GetHierarchyTable** method.

The hierarchy table can include all columns visible in the current table view, but for many tables the initial set of columns are those that most implementations can produce quickly. Typically, this is the default column set.

All hierarchy table implementations must include at least the following properties in their default column set:

PR_COMMENT	PR_DEPTH
PR_DISPLAY_NAME	PR_ENTRYID
PR_STATUS	

In addition to these columns, address book hierarchy tables must include the following properties:

PR_CONTAINER_FLAGS	PR_DISPLAY_TYPE
PR_INSTANCE_KEY	PR_OBJECT_TYPE

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

Message Services Table

The message services table lists information about the message services in the current profile. There is one message services table for every MAPI session. The message services 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 services table is implemented by MAPI and used by special purpose clients that provide configuration support. Clients retrieve a message services table object by calling **IMsgServiceAdmin::GetMsgServicesTable.**

There are eight properties that are part of the required column set for the message services table.

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

The PR_INSTANCE_KEY property is the index for the table; the default sort column is PR_DISPLAY_NAME.

The PR_SERVICE_NAME property 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.

The PR_RESOURCE_FLAGS property describes the message service's capabilities.

Message Stores Table

The message stores table lists information about message store providers in the current profile. There is one message stores table for each session that is implemented by MAPI. Clients access this table by calling **IMAPISession::GetMsgStoresTable**.

This table is indexed by PR_INSTANCE_KEY and there is no default sort order.

There are ten properties that are part of the required column set for the message stores table.

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

One-Off Table

The one-off table lists the templates that an address book provider supports for creating custom recipient addresses. Address book providers implement one-off tables in their **IABLogon::GetOneOffTable** method which MAPI calls when the user of a client application elects to create a custom recipient.

Custom recipients are individual recipients or distribution lists that do not as yet belong to any of the address book providers in the current profile. These custom recipients are created using dialog box templates supplied by an address book provider.

Once open, MAPI keeps the one-off table open. When the address book provider makes changes to the table, notifications are sent to the client.

The following properties are part of 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	

Outgoing Queue Table

The outgoing queue table lists all of the outgoing messages for a message store. Message store providers implement the outgoing queue tables for the MAPI spooler to use. Clients do not use 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. The MAPI spooler is designed to accept messages from the message store in ascending order of submission time. Because of this requirement, there may 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.

It is the responsibility of the message store provider to set up notifications for the outgoing message queue and ensure that the notification callback function is called when the contents of the queue change.

The following properties are required columns 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	

Profile Table

The 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 during the session, the table will not change.

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 for the profile table:

PR_DEFAULT_PROFILE	PR-DISPLAY_NAME

Providers Table

The providers table lists information about service providers for client use. There are two different providers 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 are required in the column sets of providers 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

To display the current transport order, build a restriction to retrieve only those rows with the PR_RESOURCE_TYPE property set to MAPI_TRANSPORT_PROVIDER. Then sort the table on the PR_PROVIDER_ORDINAL property and retrieve all of the rows, using the **IMAPITable::QueryRows** method or the API function **HrQueryAllRows**.

To change the transport order, apply the same restriction and retrieve the rows. 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.

Providers that have been deleted, or are in use but have been marked for deletion, are not included in the providers table. Once a providers table has been returned to a client, it will not reflect changes to the environment, such as the addition or deletion of a provider.

Receive Folder Table

The 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 clients use them by making a call to the **IMsgStore::GetReceiveFolderTable** method.

Receive folder tables have three properties in their required column set:

PR_ENTRYID	PR_MESSAGE_CLASS
PR_RECORD_KEY	

Recipients Table

The recipients table lists information about all of the recipients for a message. Both messages under composition and received messages are included in the table. Message store providers implement recipients tables and clients use them. Clients access a recipients table by making a call to the **IMessage::GetRecipientsTable** method. Changes to a recipients table can be made by calling the **IMessage::ModifyRecipients** method. A warning when calling **ModifyRecipients** - you must specify every column that you want to appear in the table and not just the changed ones. **ModifyRecipients** will delete all columns that you do not specify.

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 the recipients table implementation. The AND, OR, EXISTS, and Property restrictions are among the restrictions that should be available to recipients 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

Categorization and sorting are optional features.

Recipients tables have a different column set depending on whether the message has been submitted. All messages should include the following properties in their recipients tables:

PR_DISPLAY_NAME	PR_ENTRYID
PR_RECIPIENT_TYPE	PR_ROWID

The optional properties are as follows:

PR_DISPLAY_TYPE	PR_OBJECT_TYPE
PR_SPOOLER_STATUS	

Submitted messages should include these additional properties in their required column set:

PR_ADDRTYPE	PR_CLIENT_SUBMIT_TIME
PR_SENDER_ENTRYID	PR_RESPONSIBILITY
PR_SENDER_NAME	PR_TRANSPORT_STATUS

Depending on a provider's implementation, additional columns might be in the table.

Status Table

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 service providers. MAPI provides data for three rows: one for the MAPI subsystem, one for the MAPI spooler, and one 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 provide status table support.

To use the status table, clients call the **IMAPISession::GetStatusTable** method. Status table information can be used in a variety of ways. For example, it is possible to open the 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, so use its methods to change a service provider password, flush the message queue, display a configuration property sheet, or confirm status with the 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.

The status table has a large number of required properties. Besides these required properties, service providers can choose whether or not to store other properties. PR_IDENTITY_SEARCH_KEY is a property that some service providers may elect to include. The basic set of required properties is:

PR_DISPLAY_NAME	PR_ENTRYID
PR_IDENTITY_DISPLAY	PR_IDENTITY_ENTRYID
PR_INSTANCE_KEY	PR_OBJECT_TYPE
PR_PROVIDER_DISPLAY	PR_PROVIDER_DLL_NAME
PR_RESOURCE_FLAGS	PR_RESOURCE_METHODS
PR_RESOURCE_TYPE	PR_ROWID
PR_STATUS_CODE	PR_STATUS_STRING

Developing a Transport Provider

This chapter describes the transport provider's role in the MAPI message system and describes the implementation details necessary to develop features common to all transport providers. Implementation details necessary to add TNEF, remote, X.400, and EDI capabilities to transport providers are covered in separate chapters.

‹Tell the reader what they can expect out of this chapter: overview, how to implement must and optional features, provide a pointer to the other chapters they need, such as the specific follow-on XP chapters. Also let them know what they should have already boned up on before they got to this point.

Transport Provider's Role in the MAPI Subsystem

Message transport provider dynamic link libraries (DLLs) provide the interface between the MAPI spooler and that part of the underlying messaging system(s) 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. Multiple transports can be installed on the same system, but MAPI supplies the one spooler required.

Client applications do not communicate directly with the transport; rather, the MAPI spooler sends outgoing messages to the appropriate transport 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 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 message system:

- Register the address types the transport provider handles.
- 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.
- Inform the MAPI spooler when an incoming message needs to be handled.
- Pass incoming message data to the MAPI spooler by using message objects.
- Fill in all required MAPI message properties on incoming messages.
- Deletes the message from the underlying messaging system after delivery.
- Provide status information for the MAPI spooler and client applications.

Transport Provider-MAPI Spooler Operational Model

Transport provider initialization, startup, processing, shutdown and deinitialization 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 the **XPProviderInit** function, passes a support object, gets the provider object, and performs the version handshake.
2. The MAPI spooler calls the **IXPProvider::TransportLogon** method of the XPProvider object. A session is established between the 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** function. The transport returns a list of the UIDs and email address types it cares about.
4. The MAPI spooler calls the **IXPLogon::RegisterOptions** function for recipient options. The transport returns a list of the available per-recipient options for any of the email address types for which it indicated interest in the **AddressTypes** call.
5. The MAPI spooler calls the **RegisterOptions** function for message options. The transport returns a list of the available per-message options for any of the email address types for which it indicated interest in the **AddressTypes** call.
6. The MAPI spooler calls the **IXPLogon::TransportNotify** function to enable message transmission and/or reception.
7. If requested by the transport provider in its return for the **TransportLogon** call, the MAPI spooler periodically calls the **IXPLogon::Idle** method.
8. 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.
9. The MAPI spooler calls the **TransportNotify** method to disable message transmission and/or reception.
10. The MAPI spooler releases the logon and provider objects. See **IXPProvider::Shutdown** for more information.

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 spooler calls **IXPLogon::SubmitMessage**, with passing in an **IMessage** instance, to begin the process.
2. The transport then places a reference value (a transport-defined value 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 has accepted 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 transports 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 mail 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 may use **IXPLogon::TransportNotify** to abort message processing during the **SubmitMessage** or **EndMessage** calls.

Message Reception Model

The transport provider can tell the MAPI spooler whether it must be polled for incoming mail or whether it performs a callback to the spooler − by calling **IMAPISupport::SpoolerNotify**− when incoming mail is available by setting the SP_LOGON_POLL flag with its return for **IXPProvider::TransportLogon**. After learning that incoming mail is available, the spooler (at a convenient time) 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**.
4. The transport provider stores the message data by using 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 import.

Required Features Transport Providers Must Implement

All MAPI transport providers must implement certain required 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 elsewhere in this programming guide.
- Your transport provider 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.

For more detailed information on implementing the preceding function and interfaces, see the *MAPI Programmer's Reference*.

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.
- 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 for memory freed by client applications.
- Optionally support event notification for interested client applications.
- Maintain a profile, 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) to define common constants, structures, interfaces, and return values.

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 is the transport provider initialization function **XPProviderInit**. For more information on how to implement your provider's initialization function, see **MAPI Programmer's Reference**.

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 based on what type of service provider is being called, **MSProviderInit** for message stores, **ABProviderInit** for address books, or **XPProviderInit** for transports. The names are different so that one DLL can contain any combination of providers.

The MAPISPI.H header file has a type definition for the function prototype of each type of initialization function, and a predefined procedure name for each initialization routine. By naming the initialization routines in your C files with the same names used by **GetProcAddress** and using a straight forward export declaration in your DLL.DEF file, you automatically get type checking of the parameters on your initialization routine. See the sample provider source code for examples.

Note ***XX*** in the prototype is **AB**, **MS**, or **XP** depending on the provider type.

Also, if a provider's initialization call succeeds but returns an 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 simply fails with a MAPI_E_VERSION return value.

The last step in getting access to service provider resources is to log onto the service provider. You access the logon methods from the **MSProvider**, **ABProvider**, or **XPProvider** object returned from the initialization routine. This is the call where credentials (if used) are checked, dialog boxes may be allowed, and so on. The logon methods have different parameters and slightly different functions, depending on the type of provider.

If a second store, address book, or transport session is opened on the same DLL, process, and MAPI session, a second provider object is *not* created. Instead the first provider object is used to log on to the second store, address book, or transport. Thus, multiple transport sessions, stores or address books can be opened from a single provider object. MAPI creates a second provider object if two MAPI sessions are used on the same process.

Releasing the Transport Provider

Here are the processing steps when MAPI or the MAPI spooler finish using a transport logon object.

1. MAPI or the MAPI spooler calls the transport provider's **IXPLogon::TransportLogoff** method.
2. The transport provider invalidates all objects opened within the object and not yet released, including status objects, by calling the **IMAPISupport::MakeInvalid** method.
3. The MAPI support object **Release** method removes the provider's row from the status table and removes from internal tables any UIDs set that were set with the **IMAPISupport::SetProviderUID** method. It decrements the count of known logon objects active on this provider object, and if the count reaches zero MAPI calls **IXPProvider::Shutdown** and **Release** on the provider object. If this was the last known provider object using this DLL on this process, MAPI remembers internally to call **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 a batch of messages to and from a transport provider. Typically, the MAPI spooler will flush the queues for all transports that are logged onto the session, starting with the first transport as set in the transport order section of the user's profile. Transport providers must handle the **FlushQueues** call as described in the following sequence of steps to allow proper message processing and to allow 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 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	The 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. 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	The **SubmitMessage**-**EndMessage** loop

	transport provider	continues until all messages in the queue have been downloaded. After the last **EndMessage** call, the transport provider should free any external resources so they can be used by other transport providers to flush their queues.
8.	MAPI spooler	Notifies the transport provider that it is has finished downloading messages by calling the transport's **IXPLogon::TransportNotify** method with the NOTIFY_END_OUTBOUND_FLUSH flag set.
9.	Transport provider	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**.
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. The transport provider must then clear the STATUS_INBOUND_FLUSH bit in the PR_STATUS_CODE property of its status row using **ModifyStatusRow**. When the transport provider is done uploading messages it should ensure that all external resources have been released so they are available for use by other transport providers.
14.	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. The MAPI spooler then 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.

A transport provider can stop the **FlushQueues** operation at any time by clearing the STATUS_OUTBOUND_FLUSH and STATUS_INBOUND_FLUSH bits in its status row. If the MAPI spooler is shutting down and wants 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.

For more information on specific implementation details, see the reference entries for the methods in the *MAPI Programmer's Reference*.

Interacting with the Spooler

The MAPI spooler does its work and makes its calls to transport providers when foreground applications are idle. After 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. To reduce disturbance to foreground processing, transport providers should repeatedly call back to the MAPI spooler to release the CPU whenever they are performing long operations. Transport providers for 16-bit Windows platforms should take particular care to break up any operation that takes more than 0.2 seconds.

Transport providers 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 update using the **IMAPISupport::ModifyStatusRow** method.

Since network activity often takes more than 0.2 seconds, transport providers should, whenever possible, use asynchronous network requests. This allows them to initiate a request, release the CPU by calling back to the MAPI spooler, and when the MAPI spooler again gives them control, they 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 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 behave appropriately as a background process by yielding on its own when appropriate. A transport requiring the disabling of this behavior or 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 until the transport ends critical section processing with another call to **SpoolerNotify**.

Setting Properties on Inbound Messages

The client applications within the MAPI message system 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 message.
PR_BODY	Text body of the message.
PR_MESSAGE_DELIVERY_TIME	Date/Time message was delivered
PR_SENDER_NAME	Display name of originator of message.
PR_SENDER_ENTRYID	Address Book EntryID of originator of message.
PR_SENDER_SEARCH_KEY	Address Book search key of originator of message.
PR_CLIENT_SUBMIT_TIME	The time that the message was submitted to its messaging system by the sender's mail client.
PR_SENT_REPRESENTING_NAME	Name of representative delegate for sending.
PR_SENT_REPRESENTING_ENTRYID	Address Book EntryID of sending delegate.
PR_SENT_REPRESENTING_SEARCH_KEY	Address Book search key of sending delegate.
PR_RCVD_REPRESENTING_NAME	Name of representative delegate for receiving.
PR_RCVD_REPRESENTING_ENTRYID	Address Book EntryID of receiving delegate.
PR_RCVD_REPRESENTING_SEARCH_KEY	Address Book search key of receiving delegate.
PR_REPLY_RECIPIENT_NAMES	List of delegated recipient display names, separated by semicolon/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 PR_SENT_REPRESENTING_??? to the same values as PR_SENDER_???, PR_RCVD_REPRESENTING_??? to the same values as PR_RECEIVED_BY_???, and build the PR_REPLY_RECIPIENT_??? based on PR_SENDER_???.

ENTRYID or ENTRYLIST properties can be generated if necessary using the **IMAPISupport::CreateOneOff** method.The XXX_SEARCH_KEY properties may be generated by concatenating the PR_ADDRTYPE property assocated with a user, a colon ':', and the PR_EMAIL_ADDRESS property assocated with the user, then folding the result to uppercase. The Windows API **CharUpperBuff** is a convenient function to use for this purpose.

Using the Support Object

MAPI provides an implementation of the **IMAPISupport** interface with the support object service providers receive during logon. Transport Providers can call the following **IMAPISupport** methods:

IMAPISupport::Address
IMAPISupport::CreateOneOff
IMAPISupport::DoConfigPropSheet
IMAPISupport::GetLastError
IMAPISupport::GetMemAllocRoutines
IMAPISupport::GetSvcConfigSupportObj
IMAPISupport::IStorageFromStream
IMAPISupport::MakeInvalid
IMAPISupport::ModifyStatusRow
IMAPISupport::NewUID
IMAPISupport::Notify
IMAPISupport::OpenAddressBook
IMAPISupport::OpenEntry
IMAPISupport::OpenProfileSection
IMAPISupport::RegisterPreprocessor
IMAPISupport::SpoolerNotify
IMAPISupport::SpoolerYield
IMAPISupport::StatusRecips
IMAPISupport::Subscribe
IMAPISupport::Unsubscribe
IMAPISupport::WrapStoreEntryID

Providing Status

Service providers expose status and other dynamic information to applications using the MAPI status table. When an application establishes a session with a provider, the provider creates a row in this table for each resource owned by it. A message store provider, for instance, should create a row for each message store it opens when it is being logged on. MAPI manages the table interface and notifies client applications of status changes. Service providers must update MAPI's table when a change occurs, and must support application calls to obtain information not exposed in the table.

Each provider must support three operations on status table entries:

- Create a row in the status table.
- Update a row in the status table.
- Service an application request for further status information.

The information that a provider gives MAPI appears as a single row in a **IMAPITable** object. When status changes , the provider should call **IMAPISupport::ModifyStatusRow** to inform the messaging subsystem of the change. The subsystem broadcasts the notification to interested MAPI 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_DEFAULT_OUTBOUND	Indicates any message going through this transport provider uses this identity, unless overridden with the PR_SENT_REPRESENTING_NAME property (for additional information on this property, see Chapter 11, "Messages"). Always on in exactly one status row for each transport provider.
STATUS_DEFAULT_STORE	Indicates this message store is the default for use by Simple MAPI.
STATUS_PRIMARY_IDENTITY	Indicates the entry ID in this row's PR_IDENTITY_ENTRYID column is the primary identity for this session.

Optional Features Transport Providers Can Implement

Optional features transport providers can implement include the following:

- Register message and recipient options specific to the transport provider.
- Perform any verification of credentials required by the messaging system.
- Support event notification for interested client applications.
- Maintain a profile, if necessary, to store credentials to the underlying messaging system.
- Display configuration property sheets to allow users to configure the transport provider's settings.
- Provide message delivery reports to client applications.

Implementing Security

Each transport provider is responsible for implementing an appropriate level of security for access to its underlying messaging system. Each incoming or outgoing message sent through a transport by the MAPI spooler is handled in the context of a provider logon session. The transport provider may display a logon dialog box to the user that prompts for a user's credentials before establishing such a connection. Alternatively, the transport provider could 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, remember 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 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 email-address. A user can have a separate e-mail-address for each installed transport provider, or different addresses for each session on a single provider.

For more information on storing credentials in profile sections, see the reference entry for the **IProfSect** interface.

Displaying Configuration Property Sheets

Transport providers use the **IMAPISupport::DoConfigPropSheet** method to implement configuration property sheets; for more information, see *MAPI Programmer's Reference*.

Providing Message Delivery Reports

Sometimes messages are sent to transport provider that it doesn't understand. How the transport provider determines whether message delivery or nondelivery reports are sent to client applications is an implementation detail specific to individual transport providers. Once it has been determined that a report is to be sent to client applications, transport providers use the **IMAPISupport::StatusRecips** method to notify MAPI.

Developing a Remote Transport Provider

This chapter will describe the remote transport architecture and provide how-to information for developing a remote transport provider.

Remote Transport Architecture

Remote transport writers need a way to be able to execute special slow link functionality. The following section outlines MAPIs remote architecture and discusses the functionality that MAPI transport providers can implement. Specifically, MAPI provides support for remote transport providers to:

- download headers
- mark messages for download and deletion
- provide download status
- allow direct connect and disconnect control

MAPI remote architecture consists of the parts and subsystems shown here:

{ewc msdncd, EWGraphic, group10840 0 /a "SDKEX_24.bmp"}

Remote Viewer Application

This is a client application that is designed to work with remote transports. Remote viewer applications can check the MAPI status table for transports that have registered themselves as remote transport providers, and adjust their user interface appropriately. Remote client applications 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.

Remote Transport Provider

Remote transport provider have the STATUS_REMOTE_ACCESS bit set in the PR_STATUS_CODE property of their provider's row in the status table. A remote transport provider may also wish to set some of the other status bits to control the flow of mail both incoming and outgoing.

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 client 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.

The folder is expected to be a flat list with no subfolders. While the existence of subfolders should not cause client application problems, these folders are not displayed. Opening the folder does not cause the folder's contents to be downloaded. In this way message headers can be downloaded and looked at off-line or at a later time. Remote transport providers might want to provide some sort of local cache or storage facility for the downloaded message headers. Remote transport providers are encouraged to fully support the methods of the **IMAPITable** interface for the folder's contents table.

Remote transport providers must support the methods of the **IMAPIStatus** and **IMAPIFolder** interfaces.

Developing Transports that Support TNEF

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.

MAPI supplies an implementation of the **ITnef** interface for use by MAPI transport providers when working with TNEF objects created with the **OpenTnefStream** or **OpenTnefStreamEx** functions.

The **ITnef** interface provides the following methods: **ITnef::AddProps**, **ITnef::EncodeRecips**, **ITnef::ExtractProps**, **ITnef::Finish**, **ITnef::FinishComponent**, **ITnef::OpenTaggedBody**, **ITnef::SetProps**

For more information on using the **ITnef** interface, see *MAPI Programmer's Reference, Volume 1*.

TNEF Processing Overview

The process that a transport provider uses to send a message that includes a TNEF stream is as follows. The message properties that are supported by the underlying messaging system are set. The **ITnef** interface is obtained and used to encapsulate into a TNEF stream the message properties not supported by the messaging system. The **ITnef** interface is used to insert tags describing the rendering information of any message attachments into the message body. then the message is sent.

Transport providers receiving messages containing TNEF streams use the following process to retrieve the encapuslated properties. First the properties supported by the underlying message system are written into a new message. Then the **ITnef** interface is used to decode the TNEF stream and write the ecapuslated properties into the new message. Then the message body is parsed to reunite the attachments with their rendering information in the message body.

Two examples of TNEF usage follow. The first example, peer to peer, describes how a transport represents an entire message in a single file. The second example, custom processing, describes how TNEF uses custom encoding of attachments to allow the underlying messaging system to use its own method of transporting attachments.

Peer to Peer

When a message is submitted, the transport provider can choose to create a file that is used to contain the message storage during transmission. Next a stream interface is wrapped around the file. The transport provider then writes out the envelope properties to the stream in a tagged format that allows the properties to be easily decoded by receiving transport providers..

A TNEF object is obtained by passing the stream interface into the **OpenTnefStream** function. The transport provider gets a list of all defined properties for the message by calling the **IMAPIProp::GetPropList** method. The transport provider then excludes all non-transmittable properties and all properties that it has encoded in the message envelope. **ITnef::AddProps** is called to encode the remaining properties, including all attachments. After all the requested properties are added, **ITnef::Finish** is called to encode the message into the stream. Finally, the tagged message body is obtained by calling the **ITnef::OpenTaggedBody** method. This tagged body is written out to the outbound stream. The transport provider calls **IUnknown::Release** to release the TNEF object and transmits the message.

On receipt of an inbound message, the transport provider decodes the message envelope properties into the message. Once the message properties are decoded, the TNEF data is wrapped into a stream object and passed to **OpenTnefStream**. The transport provider then decodes the information in the TNEF data by calling the **ITnef::ExtractProps** method. It is important to note that anything decoded by the TNEF object will overwrite properties decoded from the message envelope, that is, TNEF will overwrite the existing properties in a message. After the properties have been extracted into the message, the decorated message body is processed with a call to **ITnef::OpenTaggedBody.** The body is opened, written, and saved. The TNEF objects are released by calling **IUnknown::Release** and the message is saved with a call to **IMAPIProp::SaveChanges**.

Custom Processing

Custom processing can be used when a transport wants to transmit attachments separately or via the underlying messaging systems attachment model. TNEF uses a mechanism that allows the transport writer to send the attachments apart from the message and reconnect them on the receiving side.

Custom processing is similar to the peer to peer method up to the point when the **ITnef::AddProps** is called. Instead of a single call, the transport provider makes multiple calls to **ITnef::AddProps**. The first call is made to add only the properties on the message via the TNEF_PROP_MESSAGE_ONLY flag.

After the message properties are added, the transport provider customizes the attachment processing using one of two options.

The first, and recommended, option is to make one call to **ITnef::AddProps** and one call to **ITnef::SetProps** for each attachment. The **ITnef::AddProps** call passes in TNEF_PROP_EXCLUDE flag with a property tag array that contains the PR_ATTACH_DATA_xxx property and an attachment identifier that specifies the attachment to be processed. **ITnef::SetProps** is called for each attachment that has a property tag of PR_ATTACH_TRANSPORT_NAME and a unique value that identifies the attachment as transported by the underlying message system. The remaining outbound TNEF processing is the same as in the peer to peer example. On the receiving side, the message is created as described above, the TNEF is decoded, and the tagged body is processed. Then the transport provider checks each attachment looking for the PR_ATTACH_TRANSPORT_NAME property. When an attachment is found with that property, the transport provider uses that property's value to reattach the attachment data to the message. The message is then saved and delivered.

The second option for customizing attachment processing uses a single call to **ITnef::AddProps**, passing in the TNEF_PROP_CONTAINED flag and with the *lpvData* parameter pointing to a string that is used as the value for the PR_ATTACH_TRANSPORT_NAME property. The purpose of this call is to exclude the PR_ATTACH_DATA_BIN property and add the PR_ATTACH_DATA_xxx property.

With this alternate model, the inbound TNEF processing is the same as in the peer to peer example.

Encoding Recipient Tables

The encoding of a recipient table into the TNEF stream is rarely necessary. In general, the recipient properties are transmitted in the containing envelope. When inclusion of the recipient table is necessary, TNEF can be prompted to encode the recipient table as a part of its normal processing. This is done during the initial phase of TNEF processing. If a transport wants to include the recipient table in the TNEF stream, **ITnef::EncodeRecips** is called. If no table was passed in the *lpRecipTable* parameter, TNEF gets the recipient table from the message, queries the column set, and process every row of the table into the TNEF stream. If a table is supplied at the time of the call, TNEF uses that table in its current state.

A secondary method is available for those transports that do not need special processing of the recipient table. This alternate method is implemented by calling **ITnef::AddProps** with the PR_MESSAGE_RECIPIENTS property specified in the inclusion list. This is equivalent to calling **ITnef::EncodeRecips** and passing in NULL for the *lpRecipTable* parameter.

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>*or *<File Name>* are variables contain values that are taken from the attachment itself. In most cases, these values come from the PR_ATTACHMENT_TITLE property. In cases where those values are not available, the title is defaulted by TNEF based on the attachment type.

The *<KeyNum>* is a variable containing the textual representation of the attachment key assigned to the attachment by TNEF. The base value of the key is passed in to the **OpenTnefStream** call. This value must not be non-zero.

The last field is either the stream name passed into the **OpenTnefStream** call or the value of the PR_ATTACH_TRANSPORT_NAME property.

The tagging of the message body is performed when a transport provider asks for a tagged message body by calling **ITnef::OpenTaggedBody**. When reading from the tagged body stream, TNEF replaces the single character that was in the message body at the index provided in the PR_RENDERING_POSITION property with the appropriate tag. When writing to the tagged body 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 body by underlying messaging systems.

Legal Notice

About This Book

MAPI Programmer's Reference, Volume 1, one of three books accompanying the Messaging Application Programming Interface (MAPI) Software Development Kit (SDK), provides a complete reference to the interfaces used with Extended MAPI objects. This book should be used together with *MAPI Programmer's Reference, Volume 2,* and *MAPI Programmer's Guide* to develop messaging-based applications and services.

Intended Audience

MAPI Programmer's Reference, Volume 1, is written for C and C++ developers with a range of needs and experience with messaging. For those developers who want to use MAPI to augment their applications with messaging features, no specific prerequisite knowledge is required. However, for workgroup developers who plan to use MAPI to create full-scale messaging applications, to create extensions or customized solutions to existing products, or to create messaging services, a background in OLE programming is recommended.

How This Book Is Organized

This book's contents are organized as follows:

- Chapter 1, "Understanding Extended MAPI Interfaces and Methods," introduces the basic design and use of MAPI interfaces, including interface inheritance and object accessibility, common features of all interfaces, input and output parameter usage, flag usage, method similarity, parameter commonality, and return value approach.
- Chapter 2, "Extended MAPI Interfaces," consists of an alphabetic listing of all interfaces that developers can implement using MAPI.

How Interface Reference Entries Are Structured

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 applications that must provide an implementation of this interface for other applications or MAPI to call.
Called by:	A list of the applications that typically call the methods of this interface.

After the "At a Glance" table, a table follows 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 derived 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 followed by this information:

Heading	**Content**
Syntax	The syntax for the method.
Parameters	A description of each parameter of the method.
Return Values	A description of each value that the method can return.
Comments	A description of why and how the method is used.
See Also	Cross-references to other topics in *MAPI Programmer's Reference, Volume 1, MAPI Programmer's Reference, Volume 2,* and *MAPI Programmer's Guide* that can help you use this method.

Document Conventions

The following typographical conventions are used throughout this book:

Typographical Convention	Meaning
Bold	Indicates an interface, method, structure, or other keyword in MAPI, Microsoft® Windows®, 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. In addition, italics are used to highlight the first use of terms and to emphasize meaning.
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.
`monospace`	Indicates source code, structure syntax, examples, user input, and program output. For example: `ptbl->SortTable(pSort, TBL_BATCH);`
SMALL CAPITALS	Indicate the names of keys on the keyboard. When you see a plus sign (+) between two key names, you should hold down the first key while pressing the second. The carriage return key, sometimes marked as a bent arrow on the keyboard, is called ENTER.

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* by Charles Petzold.

For More Information

For information on MAPI not covered in this book, see *MAPI Programmer's Guide* and *MAPI Programmer's Reference, Volume 2*, both contained within the MAPI SDK. *MAPI Programmer's Guide* provides a conceptual overview of the MAPI architecture and describes a wide range of programming tasks illustrated with extensible and reusable sample code. *MAPI Programmer's Reference, Volume 2* provides a complete reference to Extended MAPI functions, macros, structures, data types, properties, and return values and to the Simple MAPI and CMC programming models. It also includes a glossary of MAPI terms.

For more information about OLE programming, see *Inside OLE, Second Edition,* by Kraig Brockschmidt (Redmond, WA: Microsoft Press®, 1995) and *OLE Programmer's Reference, Volume One* and *Volume Two,* in the Microsoft Win32® SDK, published on CD-ROM by the Microsoft Developer Network (MSDN) and by Microsoft Visual C++™.

For more information about the Microsoft Windows programming environment, see Microsoft Windows SDK for Microsoft Windows, Version 3.1. Another good source is *Programming Windows* by Charles Petzold (Redmond, WA: Microsoft Press, 1990).

For more information about Windows 95 programming, see the Microsoft Windows 95 SDK.

For more information about Windows NT™ programming, see the Microsoft Windows NT SDK.

Understanding Extended MAPI Interfaces and Methods

This chapter covers the following concepts that are integral to working with MAPI interfaces and methods:

- How MAPI supports the OLE component object model (COM).
- The MAPI interface hierarchy.
- How MAPI interfaces are implemented and used.
- Conventions for working with MAPI methods, such as, C calling convention, method similarities, parameter commonalities, input, output, and input-output parameters, flag usage, and method return values.

This chapter does not provide a full explanation of MAPI architecture and concepts; for a more complete conceptual discussion of MAPI, see *MAPI Programmer's Guide*.

MAPI Objects

A MAPI object is somewhat different from a C++ object, although you can effectively use C++ objects to implement MAPI objects. A MAPI *object* is a C data structure or C++ class that supports a set of properties and subscribes to the component object model (COM) introduced with OLE. A binary standard for objects, the COM dictates that objects support one or more *interfaces*, or groups of related functions. These interfaces are derived from a base interface, **IUnknown**, that defines methods for interface query and reference counting. Interface query is the OLE mechanism for determining an object's capabilities − that is, what interfaces it supports. When a client application or service provider queries for an interface that an object has implemented, a pointer to that interface is returned, and the application can call interface methods through the pointer.

When an object supports an interface, it provides implementations for all of the methods specified in the interface. The object can provide a unique implementation of a method or delegate to another, previously defined implementation from which the object is derived. To use an object, you obtain a pointer to it; the pointer enables you to call methods belonging to one of the interfaces the object supports.

MAPI objects are simpler than most COM objects because they typically support only one interface deriving from **IUnknown** rather than several. However, because an interface can derive from **IUnknown** indirectly through another interface, a MAPI object can support methods from multiple interfaces. For example, the address-book container object (in other words, the address book container), supports the **IABContainer** interface. **IABContainer** derives from the **IMAPIContainer** interface, which derives from the **IMAPIProp** interface, which ultimately derives from **IUnknown**. Therefore, the address book container supports, in addition to the methods it uniquely implements, all those methods implemented by those interfaces from which its main interface is derived.

MAPI Interfaces

As with other interfaces that support the OLE component object model, a MAPI interface is a group of related functions. MAPI defines each interface's methods, specifying their parameter type and order, without always providing implementations for the methods. What supplies the implementation − MAPI, a client application, or a service provider − depends on the purpose of the interface, the interaction between the MAPI client application and MAPI service provider, and the service provider's particular feature set.

Besides the lack of a guaranteed implementation, an interface such as a MAPI interface that adheres to the component object model also participates in interface query, as previously described. All component object model interfaces inherit from **IUnknown**, either directly or indirectly. When you implement an object with an interface that inherits from another interface, you must implement all of the methods specified for both the inherited interface and the derived interface. When you call an object, it is important to know the inheritance hierarchy so that when you obtain a pointer to an object, you know what methods are available.

Note For more information about the **IUnknown** interface and about programming with the component object model, see *Inside OLE, Second Edition,* by Kraig Brockschmidt (Redmond, WA: Microsoft Press, 1995) and *OLE Programmer's Reference, Volume One* and *Volume Two,* in the Microsoft Win32 SDK, published on CD-ROM by the Microsoft Developer Network (MSDN) and by Microsoft Visual C++.

MAPI Inheritance Hierarchy

The MAPI inheritance hierarchy is very simple and straightforward. Most MAPI objects inherit directly from **IUnknown**. Some inherit from the properties interface, **IMAPIProp**, and a few inherit from the container interface, **IMAPIContainer**, as shown in the following figure.

{ewc msdncd, EWGraphic, group10843 0 /a "SDKEX_06.bmp"}

Implementing and Using MAPI Interfaces

MAPI interfaces can be understood from three different perspectives: what implements them, what uses (that is, calls) them, and what they are used for.

What Implements the MAPI Interfaces

Some interfaces are implemented by both MAPI and service providers. For example, the **IMAPIProp** and **IMAPITable** interfaces are implemented by MAPI and all service providers. This functionality enables service provider developers to write their own implementations for interfaces that MAPI must also implement. Service providers can wrap method calls and pass them to MAPI if they choose to use MAPI's implementation instead. When multiple implementations of an interface are available, access is controlled by the function pointers returned for the object that exposes the interface. The following table describes what implements the MAPI interfaces.

Implemented by	Interfaces
MAPI	**IAddrBook**, **IMAPIControl**, **IMAPIProgress**, **IMAPIProp**, **IMAPISession**, **IMAPISupport**, **IMAPITable**, **IMsgServiceAdmin**, **IProfAdmin**, **IProfSect**, **IPropData**, **IProviderAdmin**, **ITableData**, **ITnef**
Client applications	**IMAPIAdviseSink**, **IMAPIViewContext**
Address book providers	**IABContainer**, **IABLogon**, **IABProvider**, **IDistList**, **IMailUser**, **IMAPIContainer**, **IMAPIProp**, **IMAPITable**
Message store providers	**IAttach**, **IMAPIContainer**, **IMAPIFolder**, **IMAPIProp**, **IMAPIStatus**, **IMAPITable**, **IMessage**, **IMsgStore**, **IMSLogon**, **IMSProvider**
Transport providers	**IMAPIProp**, **IMAPIStatus**, **IMAPITable**, **IXPLogon**, **IXPProvider**, **ITnef**
Message hook providers	**ISpoolerHook**
Form registry providers	**IMAPIFormContainer**, **IMAPIFormInfo**, **IMAPIFormMgr**
Forms	**IMAPIForm**, **IPersistMessage,** **IMAPIFormAdviseSink**, **IMAPIMessageSite**, **IMAPIViewAdviseSink**, **IMAPIViewContext,** **IMAPIFormFactory**

What Uses the MAPI Interfaces

The following table describes what implementations can call which interfaces and identifies the MAPI header file where each interface is defined.

Used by	Interfaces	MAPI header file
Client applications only	**IMAPISession**, **IAddrBook**, **IProfAdmin**, **IMsgServiceAdmin**	MAPIX.H
Service providers only	**IMAPISupport**, **IABProvider**, **IMSProvider**, **IXPProvider**, **IABLogon**, **IMSLogon**, **IXPLogon**	MAPISPI.H
All implementations – MAPI, clients, and providers	**IAttach**, **IABContainer**, **IDistList**, **IMAPIAdviseSink**, **IMAPIControl**, **IMAPIContainer**, **IMAPIFolder**, **IMAPIProgress**, **IMAPIProp**, **IMAPIStatus**, **IMAPITable**, **IMailUser**, **IMessage**, **IMsgStore**, **IProfSect**, **IPropData**, **IProviderAdmin**, **ISpoolerHook**, **ITableData**, **ITnef**	MAPIDEFS.H, MAPIHOOK.H, MAPIUTIL.H, and TNEF.H
Form objects	**IMAPIForm**, **IMAPIFormAdviseSink**, **IMAPIFormContainer**, **IMAPIFormFactory**, **IMAPIFormInfo**, **IMAPIFormMgr**, **IMAPIMessageSite**, **IPersistMessage**, **IMAPIViewAdviseSink**, **IMAPIViewContext**	MAPIFORM.H

What the MAPI Interfaces Are Used For

The following table describes MAPI interfaces based on the actions they are used for.

Used for	Interfaces
Working with address books	**IABContainer, IABLogon, IABProvider, IAddrBook, IAttach, IDistList, IMailUser, IMAPIContainer**, **IMAPIProp, IMAPITable**
Working with message stores	**IAttach, IMAPIContainer, IMAPIFolder, IMAPIProp, IMAPIStatus, IMAPITable, IMessage, IMsgStore, IMSLogon, IMSProvider**
Working with transports	**IXPLogon, IXPProvider, ITnef**
Working with forms	**IMAPIForm, IMAPIFormAdviseSink, IMAPIFormContainer, IMAPIFormFactory, IMAPIFormInfo, IMAPIFormMgr, IMAPIMessageSite, IPersistMessage, IMAPIViewAdviseSink, IMAPIViewContext**
Working with profiles	**IMsgServiceAdmin, IProfAdmin, IProfSect, IProviderAdmin**
Managing logon	**IIABLogon, IMAPISession, IMSLogon, IXPLogon**
Working with tables	**IMAPITable, ITableData**
Working with properties	**IMAPIProp, IPropData**
Determining object status	**IMAPIStatus**
Working with notifications	**IMAPIAdviseSink, IMAPIFormAdviseSink**
Supporting service providers	**IMAPISupport**
Working with message hooks	**ISpoolerHook**

Some Conventions for Working with MAPI Methods

The following topics provide information on some conventions common to all MAPI methods. Learning about these similarities helps in learning and applying the MAPI programming model consistently throughout your program.

Calling MAPI Methods Using C

All MAPI interfaces are documented using C++ and OLE conventions. When you call an interface method from a C language program, you must add a parameter to the list of parameters that are documented in the MAPI reference. The additional parameter, which must be the first parameter in the argument list, supplies information to the C language program that is provided by the C++ program's *this* pointer.

For example, consider the **IABLogon** interface method **GetOneOffTable**. In C++, the call syntax is as follows:

```
lpABLogon->GetOneOffTable(lppTable);
```

When you call this method from a C program, you must add a pointer as the first parameter to the ABLogon object, and the call syntax is through the vtable as follows:

```
hResult = lpABLogon->lpVtbl->GetOneOffTable(lpABLogon, lppTable)
```

Method Similarities

Some MAPI interfaces have methods with the same names as methods defined for other interfaces; **GetLastError** and **OpenEntry**, methods which both appear in multiple interfaces, are two such examples. In most cases, such methods take the same arguments and have the same functionality for all interfaces that define them, the only difference being that they work on different objects. For example, the **IABContainer::CopyEntries** and **IDistList::CopyEntries** methods work in the same way, but one works on a container object and one works on a distribution list object. In such cases, you should always call the method using a pointer to the most specific object. For example, if you want to copy a folder it is more efficient to call **CopyFolder** using a pointer to the actual folder object than it is to call **CopyFolder** using a pointer to the message store's support object.

In the case of **IMAPIStatus::ValidateState** and **IXPLogon::ValidateState**, although the two identically named methods take the same arguments, they have completely different functionality. They are implemented and called differently because one method is used by client applications and the other by the MAPI spooler.

In other cases, although the method names are the same, they take different arguments and have different functionality. This is true for most of the interfaces that deal with forms. Because the object type through which the method is being called must match the *this* pointer (the first parameter of all MAPI methods), calls are passed through to the correct method in such cases.

The following three tables provide an overview of the similarities and differences for like-named methods. For specific implementation details, see the reference entry for a particular method.

The following methods have the same parameters and the same functionality in each interface in which they are found, but each version of the method works on a different object.

Methods	**Interfaces**
CopyEntries	**IABContainer**, **IDistList**
CopyFolder	**IMAPIFolder**, **IMAPISupport**
CopyMessages	**IMAPIFolder**, **IMAPISupport**
CreateEntry	**IABContainer**, **IDistList**
CreateOneOff	**IAddrBook**, **IMAPISupport**
DeleteEntries	**IABContainer**, **IDistList**
Details	**IAddrBook**, **IMAPISupport**
FlushQueues	**IMAPIStatus**, **IXPLogon**
GetLastError	**IABLogon**, **IMAPIControl**, **IMAPIFormContainer**, **IMAPIProp**, **IMAPISession**, **IMAPISupport**, **IMAPITable**, **IMsgServiceAdmin**, **IMSLogon**, **IProfAdmin**, **IProviderAdmin**
GetOneOffTable	**IABLogon**, **IMAPISupport**
GetProviderTable	**IMsgServiceAdmin**, **IProviderAdmin**
NewEntry	**IAddrBook**, **IMAPISupport**
OpenAddressBook	**IMAPISession**, **IMAPISupport**
OpenStatusEntry	**IABLogon**, **IMSLogon**, **IXPLogon**
OpenTemplateID	**IABLogon**, **IMAPISupport**
PrepareRecips	**IABLogon**, **IAddrBook**
ResolveNames	**IABContainer**, **IDistList**
Shutdown	**IABProvider**, **IMSProvider**, **IXPProvider**
Unadvise	**IABLogon**, **IAddrBook**, **IMAPIForm**, **IMAPISession**, **IMAPITable**, **IMsgStore**, **IMSLogon**

The following method has the same parameters in each interface but different functionality.

Method	Interface
ValidateState	**IMAPIStatus**, **IXPLogon**

The following methods have different parameters in each interface and different functionality.

Method	Interface
AdminServices	**IMAPISession**, **IProfAdmin**
CalcFormPropSet	**IMAPIFormContainer**, **IMAPIFormInfo**, **IMAPIFormManager**
Logoff	**IABLogon**, **IMAPISession**, **IMSLogon**
Logon	**IABProvider**, **IMSProvider**, **IXPLogon**
OpenFormContainer	**IMAPIFormInfo**, **IMAPIFormMgr**
ResolveMessageClass	**IMAPIFormContainer**, **IMAPIFormMgr**
ResolveMultiple MessageClasses	**IMAPIFormContainer**, **IMAPIFormMgr**
SetProps	**IMAPIProp**, **ITnef**

The following table lists methods found in multiple interfaces that don't fall into the earlier categories.

Methods	Interfaces	Comments
Advise	**IABLogon**, **IAddrBook**, **IMAPIForm**, **IMAPISession**, **IMAPITable**, **IMsgStore**, **IMSLogon**	The **IMAPIForm** and **IMAPITable** versions of **Advise** take different parameters and have different functionality from the other **Advise** methods and from each other. The other methods all take the same parameters but work on different objects.
CompareEntryIDs	**IABLogon**, **IAddrBook**, **IMAPISession**, **IMAPISupport**, **IMsgStore**, **IMSLogon**, **IMSProvider**	All versions take the same parameters, but each works on a different type of object. Method functionality is slightly different when comparing methods for different objects.
OpenEntry	**IABLogon**, **IAddrBook**, **IMAPIContainer**, **IMAPISession**, **IMAPISupport**, **IMsgStore**, **IMSLogon**	Same parameters, but each version of the method works on a different object. Method functionality is slightly different when comparing methods for different objects.
OpenProfileSection	**IMAPISession**, **IMAPISupport**, **IMsgServiceAdmin**,	The **IMAPISupport:: OpenProfileSection** method takes different

	IProviderAdmin	parameters and has slightly different functionality from the other versions of the method. The other versions of the method take the same parameters but work on different objects, and among the other methods functionality is slightly different when comparing methods for different objects.
SubmitMessage	**IMAPIMessageSite**, **IMessage**, **IXPLogon**	**IXPLogon** has different parameters and different functionality from the others. **IMAPIMessageSite** and **IMessage** take the same parameters but work on different objects and have completely different functionality.

Parameter Commonalities

Many MAPI methods take similar parameters as arguments. Learning about these similarities helps in learning and applying the MAPI programming model. Shown here are the more commonly used parameters and the way in which each is used throughout the MAPI methods.

Parameter	**Usage**
lpInterface	Pointer to the interface to be used.
ulUIParam	Handle of the window where the action is taking place.
ulFlags	Flags passed in.
lpulFlags	Flags passed in and passed back on return.
lpMAPIError	Pointer to a **MAPIERROR** structure.
lpMessage	Pointer to a message object.
lpProblems	Pointer to an array of problems returned on operations involving properties.
lpProgress	Pointer to a progress object.
lpTable	Pointer to a table object.
lpUnk	Pointer to an object returned from a call to a **QueryInterface** method.
lpPropTagArray	Pointer to an array of property tags.
lpEntryID	Pointer to an entry identifier for an object.
lpulConnection	Pointer to the connection number used to register a notification.

Input, Output, and Input-Output Parameters

In the parameter descriptions, this documentation has adopted the convention of indicating whether a parameter is an input, output, or input-output parameter. The distinction is important for knowing how to efficiently manage memory.

As previously stated, the memory management model employed in MAPI is an enhancement of the rules specified in OLE's component object model. As with OLE, these rules apply only to calls made to public interface methods, and within the public method, only to memory allocated for parameters that are not pointers to interfaces, such as strings and pointers to structures. When internally allocating and freeing memory not related to MAPI processes within your client application or provider, you can use whatever mechanism makes sense.

Parameters fall into one of three groups. They can be *input* parameters, set by the calling application with information to be used by the called method, *output* parameters, set by the called method and returned to the calling application, or *input-output* parameters, a combination of the two groups. The MAPI memory management model dictates how memory for these groups of parameters is allocated and freed as follows:

Parameter group	**Allocating memory**	**Freeing memory**
Input	Calling application is responsible for allocating memory and can use any mechanism.	Calling application is responsible for freeing memory and can use any mechanism.
Output	Called application is responsible for allocating memory and must use the **MAPIAllocateBuffer** function.	Called application is responsible for freeing memory and must use the **MAPIFreeBuffer** function.
Input-output	Calling application is responsible for the initial allocation; called application can reallocate memory if necessary using the **MAPIAllocateMore** function.	Called application is responsible for initial memory freeing if reallocation is necessary. Calling application must free the final return value.

When a process fails, providers need to pay attention to output and input-output parameters, because the calling application generally has no way to release the memory for the failed process. If a function returns a value indicating failure, then each output or input-output parameter must either be left at the value initialized by the calling application or set to a value for which memory will be released without any action on the part of the calling application. For example, a provider must leave an output pointer parameter of void ** ppv as it was on input; otherwise, it must set the returned pointer to NULL (*ppv = NULL).

Flag Usage

MAPI methods (as well as MAPI functions and structures) make extensive use of bitmasks to pass flags so that bits can be set to control how an operation is performed. The most common usage of bitmasks is in *ulFlags* parameters, but you will find them used in other parameters as well.

MAPI has predefined all of the allowable flags as constants. These constants take the form of all uppercase letters with each word separated by an underscore, for example MAPI_UNICODE. Your application should only set the bits in the bitmask by passing in the constants allowed for that parameter; it should not set the bits itself or define its own flags. The only valid flags for a particular parameter of a given method are the ones that are listed under the parameter description. Passing any other flags, or setting any other bits by hand, will result in the error code MAPI_E_UNKNOWN_FLAGS being returned.

For example, **IMessage::SetReadFlag** takes a single parameter, *ulFlags*. The allowable flags are CLEAR_READ_FLAG, GENERATE_RECEIPT_ONLY, MAPI_DEFERRED_ERRORS, and SUPPRESS_RECEIPT. You would set the bit for CLEAR_READ_FLAG like this:

```
pMessage->SetReadFlag(CLEAR_READ_FLAG);
```

Some methods take a *ulFlags* parameter in which all of the bits are reserved for MAPI's use. In such cases, the parameter description will read "Reserved; must be zero." Passing any value other than zero in such parameters results in the MAPI_E_UNKNOWN_FLAGS error being returned. If more than one flag is defined for a parameter, any combination of flags can be passed using the logical-OR operator, unless the documentation for that parameter specifically mentions that some flags are mutually exclusive. For example, you would set all of the flags for **SetReadFlag** like this:

```
pMessage->SetReadFlag(CLEAR_READ_FLAG |
GENERATE_RECEIPT_ONLY | MAPI_DEFERRED_ERRORS |
SUPPRESS_RECEIPT);
```

The called method then examines each relevant bit of the *ulFlags* parameter to see if you set that bit.

Return Values

All MAPI methods return HRESULTs. Your implementation should return S_OK (zero) on success, and one of the predefined MAPI return values (nonzero) for failure or warning conditions. The return values for each method are documented as part of the method's reference entry. Additionally, Extended MAPI Return Values groups the errors that are common to most MAPI methods.

Extended MAPI Interfaces

This chapter consists of an alphabetic listing of all interfaces that developers can implement using Extended MAPI.

IABContainer : IMAPIContainer

[New - Windows 95]

The **IABContainer** interface provides further access to address book containers. **IABContainer** can make such changes as creating, copying, and deleting entries, in addition to performing bulk searches. Address book providers must implement all methods of this interface. Providers might return MAPI_E_NO_SUPPORT, however, if their address book does not support certain operations. For example, an address book that did not support the copying of entries would return MAPI_E_NO_SUPPORT when its **IABContainer::CopyEntries** method was called.

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
Called by:	Extended MAPI client applications

Vtable Order

CreateEntry	Creates a new entry in a modifiable address book container.
CopyEntries	Copies one or more entries into a modifiable address book container.
DeleteEntries	Removes one or more entries from a modifiable address book container.
ResolveNames	Resolves entries from an address book container.

IABContainer::CopyEntries

[New - Windows 95]

Copies one or more entries, typically messaging users or distribution lists, into an address book container that supports modification.

Syntax

HRESULT CopyEntries(**LPENTRYLIST** *lpEntries*, **ULONG** *ulUIParam*, **LPMAPIPROGRESS** *lpProgress*, **ULONG** *ulFlags*)

Parameters

lpEntries
Input parameter pointing to an array of **ENTRYLIST** structures containing entry identifiers to copy into the container.

ulUIParam
Input parameter containing the handle of the window the dialog box is modal to. The *ulUIParam* parameter must be zero if the AB_NO_DIALOG flag is set in the *ulFlags* parameter.

lpProgress
Input parameter pointing to the address of a progress object that contains client-supplied progress information to be displayed in a progress user interface during a lengthy copy operation. If NULL is passed in the *lpProgress* parameter, MAPI provides progress information. The *lpProgress* parameter is ignored if the AB_NO_DIALOG flag is set in the *ulFlags* parameter.

ulFlags
Input parameter containing a bitmask of flags that control how the copy operation is performed. If no flags are set, the entry is created. 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 duplicate matches than using the flag CREATE_CHECK_DUP_STRICT. For example, a provider can define a loose match as the same display name, while defining a strict match as the same display name and the same e-mail address.

CREATE_CHECK_DUP_STRICT
Indicates a strict level should be used for duplicate entry checking, which returns fewer duplicate matches than using the flag CREATE_CHECK_DUP_LOOSE. For example, a provider can define a loose match as the same display name, while defining a strict match as the same display name and the same e-mail address.

CREATE_REPLACE
Duplicate entries replace entries in the container of which they are duplicates.

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 copied. Use the HR_FAILED macro to test for this warning, but the call should be handled as a successful return.

Comments

Use the **IABContainer::CopyEntries** method to copy entries from one container to another. Copying entries with **IABContainer::CopyEntries** is the preferred way to copy entries between containers, even if there is only one entry in the container, because the copy operation is done within the provider rather than under client application control. Using the **IABContainer::CopyEntries** copy operation is functionally equivalent to creating new entries in the target container using calls to the following sequence of methods for each entry in the entry list: **IABContainer::CreateEntry**, **IMAPIProp::SaveChanges**, and **IUnknown::Release**.

Address book providers must support all the flags that can be set in the *ulFlags* parameter; however, your provider is free to determine what the semantics of CREATE_CHECK_DUP_LOOSE and CREATE_CHECK_DUP_STRICT mean within the context of its implementation. Providers that cannot determine whether the entry is a duplicate should allow the entry to be copied.

If the CREATE_CHECK_DUP_LOOSE and CREATE_CHECK_DUP_STRICT flags are set, and the implementation does not copy the entry because it is a duplicate, the MAPI_W_PARTIAL_COMPLETION warning is not returned. The MAPI_W_PARTIAL_COMPLETION warning is only returned when a nonduplicate entry cannot be copied.

If the CREATE_CHECK_DUP_LOOSE, CREATE_CHECK_DUP_STRICT, and CREATE_REPLACE flags are not set, then the entry is copied, even if it is a duplicate.

See Also

ENTRYLIST structure, **IABContainer::CreateEntry** method, **IMAPIProgress : IUnknown** interface, **IMAPIProp::SaveChanges** method

IABContainer::CreateEntry

[New - Windows 95]

Creates a new entry in an address book container supporting modification. A new container entry can be a messaging user, a distribution list, or another container.

Syntax

HRESULT CreateEntry(**ULONG** *cbEntryID*, **LPENTRYID** *lpEntryID*, **ULONG** *ulCreateFlags*, **LPMAPIPROP FAR** * *lppMAPIPropEntry*)

Parameters

cbEntryID
Input parameter containing the number of bytes in the *lpEntryID* parameter.

lpEntryID
Input parameter pointing to the entry identifier of an existing entry that is copied to create the new entry.

ulCreateFlags
Input parameter containing a bitmask of flags that control 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 duplicate matches than using the flag CREATE_CHECK_DUP_STRICT. For example, a provider can define a loose match as the same display name, while defining a strict match as the same display name and the same e-mail address.

CREATE_CHECK_DUP_STRICT
Indicates a strict level should be used for duplicate entry checking, which returns fewer duplicate matches than using the flag CREATE_CHECK_DUP_LOOSE. For example, a provider can define a loose match as the same display name, while defining a strict match as the same display name and the same e-mail address.

CREATE_REPLACE
Duplicate entries replace entries in the container of which they are duplicates.

lppMAPIPropEntry
Output parameter pointing to the variable that receives a pointer to the newly created object.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

Comments

Use the **IABContainer::CreateEntries** method to create a new entry in the specified container and return a pointer to the newly created object. The types of entries that can be created within a container are defined by the container's PR_CREATE_TEMPLATES property. The PR_CREATE_TEMPLATES property can be retrieved by passing it in the *ulPropTag* parameter when calling the **IMAPProp::OpenProperty** method.

Client applications can't assume that they can access the entry identifier returned in the *lppMAPIPropEntry* parameter until after the **IMAPIProp::SaveChanges** method has been called.

Address book providers must support all the flags that can be set in the *ulFlags* parameter; however, your provider is free to determine what the semantics of CREATE_CHECK_DUP_LOOSE and CREATE_CHECK_DUP_STRICT mean within the context of its implementation. Providers that cannot determine whether the entry is a duplicate should allow the entry to be created.

If the CREATE_CHECK_DUP_LOOSE and CREATE_CHECK_DUP_STRICT flags are set, and the implementation does not create the entry because it is a duplicate, the MAPI_W_PARTIAL_COMPLETION warning is not returned. The MAPI_W_PARTIAL_COMPLETION warning is only returned when a nonduplicate entry cannot be created.

If the CREATE_CHECK_DUP_LOOSE, CREATE_CHECK_DUP_STRICT, and CREATE_REPLACE flags are not set, then the entry is created, even if it is a duplicate.

The duplicate checking operation, although the flags are passed in with CreateEntry, does not occur until **SaveChanges** is called.

Errors such as MAPI_E_COLLISION, that can occur when the entry is created, are returned on the **SaveChanges** call, not from **CreateEntry**.

See Also

IABContainer::CopyEntries method, **IMAPIProp::OpenProperty** method, **IMAPIProp::SaveChanges** method, PR_CREATE_TEMPLATES property

IABContainer::DeleteEntries

[New - Windows 95]

Removes recipients from an address book or other container that enables the user to delete entries. The removed recipients are typically messaging users, distribution lists, or containers.

Syntax

HRESULT DeleteEntries(**LPENTRYLIST** *lpEntries*, **ULONG** *ulFlags*)

Parameters

lpEntries
Input parameter pointing to an array of **ENTRYLIST** structures containing entry identifiers for the recipients 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. Use the HR_FAILED macro to test for this warning, but the call should be handled as a successful return.

IABContainer::ResolveNames

[New - Windows 95]

Resolves entries from an address book container.

Syntax

HRESULT ResolveNames(**LPSPropTagArray** *lpPropTagArray*, **ULONG**, *ulFlags*, **LPADRLIST** *lpAdrList*, **LPFlagList** *lpFlagList*)

Parameters

lpPropTagArray
Input parameter pointing to a **SPropTagArray** structure containing the properties required for each recipient name being resolved. To request that the default set of property columns for the container's contents table be returned in the address list, send a value of NULL in the *lpPropTagArray* parameter.

ulFlags
Input parameter containing a bitmask of flags controlling the type of the text in the returned strings. The following flag can be set:

MAPI_UNICODE
Indicates the returned strings of the default column set are to be in Unicode™ format. If the MAPI_UNICODE flag is not set, the strings are in 8-bit format.

lpAdrList
Input-output parameter that on input points to an **ADRLIST** structure containing the list of recipients whose names need to be resolved. On output, the *lpAdrList* parameter returns the list of resolved names.

lpFlagList
Input-output parameter containing a list of values; each value corresponds to an entry in the address list in the *lpAdrList* parameter and provides the name-resolution status for that particular entry. The values in *lpFlagList* are in the same order as the entries in the *lpAdrList* parameter. The following flags can be set:

MAPI_AMBIGUOUS
Indicates that the corresponding entry in the recipient list is resolved, but that it did not resolve to a unique entry identifier. If this flag is returned, other containers should ignore this entry.

MAPI_RESOLVED
Indicates that the corresponding entry in the recipient list has been resolved to a unique entry identifier. If this flag is returned, other containers should ignore this entry.

MAPI_UNRESOLVED
Indicates that the corresponding entry in the recipient list 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.

Comments

Address book providers use the **IABContainer::ResolveNames** method to resolve unresolved names in an address book container. When a client application calls **IAddrBook::ResolveNames**, MAPI's implementation calls **IABContainer::ResolveNames** for each address book provider. Address book providers that want to perform bulk name resolutions should implement this method. Address book providers that don't implement **IABContainer::ResolveNames** resolve names by restricting the container's contents table using the PR_ANR property in the **SPropertyRestriction** structure. For more information on using restrictions to resolve names, see *MAPI Programmer's Guide*.

Address book providers that do implement **IABContainer::ResolveNames** take the address list passed in the *lpAdrList* parameter and try to resolve all the recipients that have their corresponding MAPI_UNRESOLVED flag set in the *lpFlagList* parameter. Entries in the *lpAdrList* parameter that are lacking the PR_ENTRYID property are considered unresolved.

The array of flags **ResolveNames** returns in the *lpFlagList* parameter is used to track the results of the name resolution operation; the array shows for each recipient name whether it is resolved, unresolved, or matches more than one recipient. If, after looking at an unresolved entry, your container can match that entry uniquely, your provider should also set the flag for that entry in *lpFlagList* to MAPI_RESOLVED and add the PR_ENTRYID property for that entry to that entry's **ADRENTRY** structure in the address list in the *lpAdrList* parameter. The entry identifier added in such a case does not have to be permanent. If your container contains more than one entry that matches a given unresolved entry, then your provider should set the flag for that entry in *lpFlagList* to MAPI_AMBIGUOUS and leave that entry's **ADRENTRY** alone.

If a provider can't return all the properties requested for the unresolved recipient, and these properties don't exist in the **ADRENTRY** structure passed in, it should set the property type of each property not returned to PT_ERROR. Any property columns that are already included with a recipient entry should be retained if that entry is resolved.

If your provider is replacing an **ADRENTRY**, it should first free the **ADRENTRY** using the **MAPIFreeBuffer** function, then allocate the replacement **ADRENTRY** using the **MAPIAllocateBuffer** function.

MAPI needs certain properties to submit a message and will call providers to get those properties as part of its **IAddrBook::PrepareRecips** and **IMAPISupport::ExpandRecips** implementations. Providers implementing **ResolveNames** can eliminate the MAPI callbacks on PrepareRecips and ExpandRecips, and thus improve performance for message submission, by returning the following property columns when they resolve recipient names:

PR_ADDRTYPE
The address type of the entry.

PR_DISPLAY_NAME
The display name of the recipient.

PR_EMAIL_ADDRESS
The e-mail address of the recipient.

PR_ENTRYID
The entry identifier for the object.

PR_OBJECT_TYPE
Indicates the type of the MAPI object.

PR_SEARCH_KEY
The search key used to identify the message within tables.

PR_TRANSMITTABLE_DISPLAY_NAME
The transmittable display name for the recipient.

Clients applications can also use the returned property columns in their **IMessage::ModifyRecipients** calls.

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

[New - Windows 95]

The **IABLogon** interface is used to access resources in an address book provider object.

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 an application off the 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 in the address book.
Advise	Registers an object for notifications about changes within the address book.
Unadvise	Removes a registration for notifications previously established with a call to the **IABLogon::Advise** method.
OpenStatusEntry	Opens a status object.
OpenTemplateID	Enables this provider to generate for another address book provider an interface for an object within the current address book provider.
GetOneOffTable	Returns the table of custom recipient address templates that can be used for creating recipients for a message.
PrepareRecips	Converts short-term entry identifiers to long-term entry identifiers, updates those recipients that belong to this address book provider, and, if necessary, retrieves those recipients' permanent entry identifiers along with any additional properties requested .

IABLogon::Advise

[New - Windows 95]

Registers an object for notifications about changes within the address book.

Syntax

HRESULT Advise(**ULONG** *cbEntryID*, **LPENTRYID** *lpEntryID*, **ULONG** *ulEventMask*,
LPMAPIADVISESINK *lpAdviseSink*, **ULONG FAR** * *lpulConnection*)

Parameters

cbEntryID
Input parameter containing the number of bytes in 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.

ulEventMask
Input parameter containing an event mask of the types of notification events occurring for the object for which MAPI will generate notifications to filter 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 data structures:

Notification event type	Corresponding data structure
fnevCriticalError	**ERROR_NOTIFICATION**
fnevObjectCreated	**OBJECT_NOTIFICATION**
fnevObjectDeleted	**OBJECT_NOTIFICATION**
fnevObjectModified	**OBJECT_NOTIFICATION**
fnevObjectCopied	**OBJECT_NOTIFICATION**
fnevTableModified	**TABLE_NOTIFICATION**
fnevStatusObjectModified	**STATUS_OBJECT_NOTIFICATION**

lpAdviseSink
Input parameter pointing to the advise sink object to be called when an event for the object occurs about which notification has been requested.

lpulConnection
Output parameter pointing to a variable that upon a successful return holds the connection number for this 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 the *lpEntryID* parameter.

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 could not be found to handle the entry identifier.

Comments

Use the **IABLogon::Advise** method to register an address book object for notification callbacks. MAPI will forward this call to the service provider active on the session that is responsible for the object indicated by the entry identifier in the *lpEntryID* parameter. Whenever a change occurs to the indicated object, the provider checks to see what event mask bit has been set in the *ulEventMask* parameter and thus what type of change has 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, your client should use the **HrThisThreadAdviseSink** function.

To provide notifications, the service provider implementing **Advise** needs to keep a copy of the pointer to the advise sink object; to do so, it calls the **IUnknown::AddRef** method for the advise sink object to maintain the 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, client applications must be prepared for the advise sink object to be released. Clients should therefore release their advise sink object after **Advise** returns, unless they have a specific long-term use for it.

See Also

HrThisThreadAdviseSink function, **IABLogon::Unadvise** method, **IMAPIAdviseSink::OnNotify** method, **NOTIFICATION** structure

IABLogon::CompareEntryIDs

[New - Windows 95]

Compares two entry identifiers to determine if they refer to the same service provider 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.

Syntax

HRESULT CompareEntryIDs(**ULONG** *cbEntryID1*, **LPENTRYID** *lpEntryID1*, **ULONG** *cbEntryID2*, **LPENTRYID** *lpEntryID2*, **ULONG** *ulFlags*, **ULONG FAR** * *lpulResult*)

Parameters

cbEntryID1
Input parameter containing the number of bytes in the *lpEntryID1* parameter.

lpEntryID1
Input parameter pointing to the first entry identifier to be compared.

cbEntryID2
Input parameter containing the number of bytes in 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 that receives the result of the comparison; 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.

Comments

Use the **IABLogon::CompareEntryIDs** method to compare two entry identifiers for a given service provider object and 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 could occur, for example, trying to compare a short-term entry identifier with a long-term entry identifier.

IABLogon::GetLastError

[New - Windows 95]

Returns a **MAPIERROR** structure containing information about the last error that occurred for the address book object.

Syntax

HRESULT GetLastError(**HRESULT** *hResult*, **ULONG** *ulFlags*, **LPMAPIERROR FAR** * *lppMAPIError*)

Parameters

hResult
Input parameter containing the result returned for the last call on the session object that returned an error.

ulFlags
Input parameter containing a bitmask of flags controlling the type of the returned strings. The following flag can be set:

MAPI_UNICODE
Indicates the strings returned 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 8-bit 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.

Comments

Use the **IABLogon::GetLastError** method to retrieve information to display as a message to the user regarding the last error returned from a method call on the session object.

To release all the memory allocated by MAPI, client applications need only call the **MAPIFreeBuffer** function for the **MAPIERROR** structure.

The return value from **GetLastError** must be S_OK for the application to make use of the **MAPIERROR** structure. Even if the return value is S_OK, it is still possible that a **MAPIERROR** structure won't be returned. If the implementation cannot determine what the last error was, or if a MAPIERROR structure is not available for that error, a pointer to NULL is returned in the *lppMAPIError* parameter instead.

See Also

MAPIERROR structure, **MAPIFreeBuffer** function

IABLogon::GetOneOffTable

[New - Windows 95]

Returns the table of custom recipient address templates that can be used for creating recipients for a message.

Syntax

HRESULT GetOneOffTable(**ULONG** *ulFlags*, **LPMAPITABLE FAR** * *lppTable*)

Parameters

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 8-bit format.

lppTable
Output parameter pointing to a pointer to the returned table object that contains a list of custom recipient addresses supported by this provider for general uses, such as filling in the recipient fields of a message.

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 tables.

Comments

MAPI calls the **IABLogon::GetOneOffTable** method to make templates for creating new custom recipient addresses (also called one-off addresses) available to client applications. These custom recipient addresses, created by the user based on dialog box templates, can be used to send individual messages to recipients.

The following properties must be available as columns:

PR_ADDRTYPE
The address type of the entry. If this address type is only determined by user selection, then it should contain a zero-length string.

PR_DEPTH
How far to indent the display name in the list. A value of zero means the display name is indented all the way to the left.

PR_DISPLAY_NAME
The display name that shows up in the listbox of items that can be created.

PR_DISPLAY_TYPE
The display type of the entry, usually DT_MAILUSER.

PR_ENTRYID
The entry identifier that gets passed into **IABContainer::CreateEntry**, **IABContainer::OpenEntry**, and **IAddrBook::NewEntry**.

PR_INSTANCE_KEY

The index column for the row.

PR_SELECTABLE

Indicates whether or not this item can be selected from the list box to create a custom recipient address. MAPI allows a provider to group items visually by indenting and allowing certain rows to function as headings by being unselectable.

For example, if you had several templates to build X.400 addresses, you could display them this way:

FAX templates (depth 0, not selectable)

Local (depth 1, selectable)

Long-distance (depth 1, selectable)

MAPI keeps this table open and changes to this table should be handled through table notifications. If your provider changes the table, it should call the **IMAPIAdviseSink::OnNotify** method with the appropriate event masks for any client applications 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 custom recipient table by the **IMAPITable::QueryColumns** method. The initial active columns for a custom recipient table are those columns the **QueryColumns** method returns before the application that contains the custom recipient 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 application that contains the custom recipient table calls the **IMAPITable::SetColumns** method.
- Controls the property types of the sort order specification returned by the **IMAPITable::QuerySortOrder** method before the application that contains the custom recipient table calls the **IMAPITable::SortTable** method.

See Also

BuildDisplayTable function, **IABContainer::CreateEntry** method, **IAddrBook::NewEntry** method, **IMAPISupport::GetOneOffTable** method

IABLogon::Logoff

[New - Windows 95]

Logs the client application off the address book provider.

Syntax

HRESULT Logoff(**ULONG** *ulFlags*)

Parameters

ulFlags
Reserved; must be zero.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

Comments

Address book providers use the **IABLogon::Logoff** method to forcibly shut down the address book. **IABLogon::Logoff** is called in the following situations:

- While MAPI is logging off a client application after a call to the **IMAPISession::Logoff** method.
- While MAPI is logging off an address book provider. In this case, **IABLogon::Logoff** is called as part of MAPI's processing the **IUnknown::Release** method of the support object that the address book provider creates while it is processing an **IUnknown::Release** method call on an address book object.

See Also

IABProvider::Logon method

IABLogon::OpenEntry

[New - Windows 95]

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.

Syntax

HRESULT OpenEntry(**ULONG** *cbEntryID*, **LPENTRYID** *lpEntryID*, **LPCIID** *lpInterface*, **ULONG** *ulFlags*, **ULONG FAR *** *lpulObjType*, **LPUNKNOWN FAR *** *lppUnk*)

Parameters

cbEntryID
Input parameter containing the number of bytes in the *lpEntryID* parameter.

lpEntryID
Input parameter pointing to the entry identifier for the container or recipient object to be opened.

lpInterface
Input parameter pointing to the interface identifier (IID) for the object. Passing NULL for the *lpInterface* parameter indicates the MAPI interface for the object will be returned. The *lpInterface* parameter can also be set to an identifier for another appropriate interface for the object.

ulFlags
Input parameter containing a bitmask of flags used to control how the object is opened. The following flags can be set:

MAPI_BEST_ACCESS
Indicates the object should be opened with the maximum privileges allowed to the user. For example, if the client application has write privilege, open the object with write privilege; if the client application has read-only privilege, open the object with read-only privilege. The client application can learn the privilege 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 application. 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 client applications should not assume that write access was granted.

lpulObjType
Output parameter pointing to a variable where the object type for 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 to modify a read-only object or an attempt to access an object for which the user has insufficient permissions.

MAPI_E_UNKNOWN_ENTRYID
The object indicated in the *lpEntryID* parameter is not recognized. This return value is typically returned if the address book provider that the object is contained within is not open.

MAPI_E_NOT_FOUND
The object indicated in the *lpEntryID* parameter does not exist.

Comments

MAPI uses the **IABLogon::OpenEntry** method to open a container or recipient Your provider only receives an **OpenEntry** call for entry identifiers which MAPI has determined belong to your provider. Your provider returns a pointer that provides further access to the object to be opened. 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 your 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 on the opened object. Passing NULL in *lpInterface* indicates the standard MAPI interface for that type of object should be used. The address book provider can implement other interfaces for the object if needed by passing the interface identifier for the interface that should be returned.

If MAPI passes NULL for the *lpEntryID* parameter, it indicates that your provider should open root container in its container hierarchy.

IABLogon::OpenStatusEntry

[New - Windows 95]

Opens a status object.

Syntax

HRESULT OpenStatusEntry(**LPCIID** *lpInterface*, **ULONG** *ulFlags*, **ULONG FAR *** *lpulObjType*, **LPMAPISTATUS FAR *** *lppMAPIStatus*)

Parameters

lpInterface
Input parameter pointing to the interface identifier (IID) to be used for the returned object. Passing NULL for the *lpInterface* parameter indicates the MAPI interface for the object will be returned, in this case the **IMAPIStatus** interface. The *lpInterface* parameter can also be set to an identifier for another appropriate interface for the object.

ulFlags
Input parameter containing a bitmask of flags used to control how the status entry is opened. The following flag can be set:

MAPI_MODIFY
Requests write access. By default, objects are created with read-only access, and client applications should not assume that write access was granted.

lpulObjType
Output parameter pointing to a variable that holds the type of the opened object.

lppEntry
Output parameter pointing to a variable where the pointer to the returned object is stored.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

Comments

Message store providers use the **IABLogon::OpenStatusEntry** method to open a status object. This status object is then used to enable client applications 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

IABLogon::OpenTemplateID

[New - Windows 95]

Allows your provider to bind code to data in a foreign address book provider.

Syntax

HRESULT OpenTemplateID(**ULONG** *cbTemplateID*, **LPENTRYID** *lpTemplateID*, **ULONG** *ulTemplateFlags*, **LPMAPIPROP** *lpMAPIPropData*, **LPCIID** *lpInterface*, **LPMAPIPROP FAR *** *lppMAPIPropNew*, **LPMAPIPROP** *lpMAPIPropSibling*)

Parameters

cbTemplateID
: Input parameter containing the number of bytes in the *lpTemplateID* parameter. This value is obtained from the address book entry's PR_TEMPLATEID property.

lpTemplateID
: Input parameter pointing to the template identifier for the template to be used. This value is obtained from the address book entry's PR_TEMPLATEID property.

ulTemplateFlags
: Input parameter containing a bitmask of flags controlling how the address book entry is opened. The following flag can be set:

  FILL_ENTRY
  : Indicates this is the first time this entry is being created in the foreign provider's address book and that the address book provider should copy all relevant properties, including name to identifier mapping, to the object pointed to by the *lpMAPIPropData* parameter. Your provider must perform this operation.

lpMAPIPropData
: Input parameter pointing to the property object that is linked to the code in address book provider. This is the object that receives the template information from the address book provider. This object must support the interface being requested in the *lpInterface* parameter and it must be set to read-write access.

lpInterface
: Input parameter pointing to the interface identifier that is being used for both the property objects in *lppMAPIPropData* and *lppMAPIPropNew* parameters. Passing NULL in the *lpInterface* parameter indicates the return value is cast to the standard interface for the address-book entry, which can be assumed to be IID_IMailUser.

lppMAPIPropNew
: Output parameter pointing to a variable where the pointer to the interface indicated by the *lpInterface* parameter is stored.

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 in is unrecognized by your provider.

Comments

Address book providers use the **IABLogon**::**OpenTemplateID** method to bind code that exists in their address book provider to data that is contained in the foreign address book provider. This functionality allows your provider to update the properties in the entry passed in the *lpTemplateID* parameter.

Any address book provider that can contain entries with template identifiers copied from other containers can check for the PR_TEMPLATEID property after calls to the **IABContainer::OpenEntry** or **IABLogon::CreateEntry** methods. If the address book entry contains the PR_TEMPLATEID property, the provider calls **IMAPISupport::OpenTemplateID** so that the address book provider responsible for the entry has the opportunity to update the entry.

If the FILL_ENTRY flag is passed in the *ulFlags* parameter, the address book provider must set all properties on the entry.

If your provider doesn't recognize the entry identifier passed in the *lpTemplateID* parameter, it should return MAPI_E_UNKNOWN_ENTRYID.

See Also

PR_TEMPLATEID property, **IMAPISupport::OpenTemplateID** method, **IPropData : IMAPIProp** interface

IABLogon::PrepareRecips

[New - Windows 95]

Prepares a recipient list for later use by the messaging system. Converts recipients' short-term entry identifiers to long-term entry identifiers, updates those recipients that belong to this address book provider, and, if necessary, retrieves those recipients' permanent entry identifiers along with any additional properties requested.

Syntax

HRESULT PrepareRecips(**ULONG** *ulFlags*, **LPSPropTagArray** *lpPropTagArray*, **LPADRLIST** *lpRecipList*)

Parameters

ulFlags
Reserved; must be zero.

lpPropTagArray
Input parameter pointing to a counted array of property tags for the properties that require updating by the calling application. 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.

Comments

The **IABLogon::PrepareRecips** method is called to ensure that all recipients your address book provider recognizes have permanent entry identifiers and that they have all the properties requested in the *lpPropTagArray* parameter. 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 recognized by your provider, it 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 property is not available to the calling application.

Like the **ADRLIST** structure, each **SPropValue** property value structure passed in the *lpPropTagArray* parameter must be separately allocated using **MAPIAllocateBuffer** and **MAPIAllocateMore** 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

[New - Windows 95]

Removes an object's registration for notification of address book changes previously established with a call to the **IABLogon::Advise** method.

Syntax

HRESULT Unadvise(**ULONG** *ulConnection*)

Parameters

ulConnection
Input parameter containing the number of the registration connection returned by **IABLogon::Advise**.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

Comments

The **IABLogon::Unadvise** method releases the pointer to the advise sink object passed in the *lpAdviseSink* parameter in the previous call to **IABLogon::Advise** and thereby cancels a notification registration. As part of discarding the pointer to 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 **IMAPIAdviseSink::OnNotify** on the advise sink object, the **Release** call is delayed until the **OnNotify** method returns.

See Also

IABLogon::Advise method, **IMAPIAdviseSink::OnNotify** method

IABProvider : IUnknown

[New - Windows 95]

The **IABProvider** interface provides a method to log onto 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
Corresponding pointer type:	LPABPROVIDER
Implemented by:	Address book providers
Called by:	MAPI

Vtable Order

Shutdown	Closes your provider's address book object in an orderly fashion.
Logon	Logs MAPI onto one instance of your address book provider.

IABProvider::Logon

[New - Windows 95]

Logs MAPI onto one instance of your address book provider.

Syntax

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 MAPI support object, which is an object that provides access to functions required by providers.

ulUIParam
Input parameter containing the handle of the window the dialog box is modal to. MAPI passes in the value from the **MAPILogon** call

lpszProfileName
Input parameter pointing to a string containing the profile name of the user.

ulFlags
Input parameter containing a bitmask of flags used to control the logon. The following flags can be set:

AB_NO_DIALOG
Indicates that 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 information that is not already contained with the profile specified in the *lpszProfileName* parameter.

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_UNICODE
Indicates the passed-in strings are in Unicode format. If the MAPI_UNICODE flag is not set, the strings are in 8-bit format.

lpulcbSecurity
Input-output parameter pointing to a variable containing the size, in bytes, of the security credentials structure for which a pointer is returned in 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 variable where the pointer to the buffer containing the security credentials is stored. 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. A value of NULL can be returned if there is no **MAPIERROR** structure to return.

lppABLogon
Output parameter pointing to the address of a variable in which the address book provider passes back its logon object.

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. MAPI calls the provider's message service entry function.

MAPI_E_USER_CANCEL
: The user canceled the operation, typically by clicking the cancel button in a dialog box.

Comments

When a client application calls the **IMAPISession::OpenAddressBook** method, MAPI loads your address book provider into memory and then calls the **IABProvider::Logon** method each time your provider is added to the client application's profile. The address book provider should, during the processing of each **IABProvider::Logon** call, use the **IMAPISupport::SetProviderUID** method to register the set of supported unique identifiers (UIDs). MAPI uses these UIDs for comparison to the UIDs found in entry identifiers so as to route calls from various objects that use entry identifiers. For example, routing **OpenEntry** calls to the correct address-book logon object. If your provider is logged in more than once to the same address-book provider object, and your provider's UID is constant, then your provider can provide a means to route an entry identifier used on any provider logon object to the correct logon context within your provider. For more information on using multiple logon objects, see *MAPI Programmer's Guide*.

The MAPI support object exists to provide access to functions needed by the address book and other providers. The support object can be different for each logon session; your address book provider should use the current object, indicated by the *lpMAPISup* parameter, for its session. Your provider must call the **IUnknown::AddRef** method on the support object if the **IABProvider::Logon** call is successful, otherwise your provider is unloaded.

The local name of the user's profile in the *lpszProfileName* parameter is provided as a convenience for address book providers. It can be used in error dialog boxes, logon screens, or in other user interfaces where you want to show the user the name of the profile. If you want your provider to keep and use this profile name, the provider must copy it to storage that your provider has allocated. The profile name is displayed in the character set of the user's application as indicated by the presence or absence of the MAPI_UNICODE flag in the *ulFlags* parameter.

The address book provider should create an address-book 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 **IABLogon::OpenEntry** and **IABLogon::Logoff** methods for your provider.

An address book provider that needs to log on to an underlying messaging system or directory service can use the MAPI support method **IMAPISupport::OpenProfileSection** for saving and retrieving security credential sets about this particular logon session. If your provider finds that all the required information is not in the profile it should return MAPI_E_UNCONFIGURED so that the provider's message service entry function gets called by MAPI.

If your provider requires a password during logon, a logon dialog box is displayed, unless the AB_NO_DIALOG flag was set in the *ulFlags* parameter. If the user cancels the logon process, typically by clicking the cancel button in the dialog box, your provider should return the value MAPI_E_USER_CANCEL.

When MAPI logs on to your provider, the MAPI spooler also logs on to your provider. The values in the *lpulcbSecurity* and *lppbSecurity* parameters are copied in 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 **MAPIAllocateBuffer**.

See Also

IABLogon::Logoff method, **IABLogon::OpenEntry** method, **IMAPISupport::OpenProfileSection** method, **IMAPISupport::SetProviderUID** method, **MSGSERVICEENTRY** function prototype

IABProvider::Shutdown

[New - Windows 95]

Closes your provider's address book object in an orderly fashion.

Syntax

HRESULT Shutdown (**ULONG** *lpulFlags*)

Parameters

lpulFlags
Reserved; must be a pointer to zero.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

Comments

When MAPI calls your provider's **IABLogon::Shutdown** method, your provider is free to do whatever it needs to do when it shuts down. Shutdown is only called after your final logon object is released.

IAttach : IMAPIProp

[New - Windows 95]

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 *MAPI Programmer's Guide*.

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
Called by:	Client applications

Vtable Order

No unique methods

IDistList : IMAPIContainer

[New - Windows 95]

The **IDistList** interface is used to provide further access to distribution lists within address book containers that are modifiable. **IDistList** can make such changes as creating, copying and deleting distribution lists, in addition to performing bulk searches. 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, see the reference entries for **IABContainer**. For more information on working with distribution list objects, see *MAPI Programmer's Guide*.

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:	Extended MAPI 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 distribution lists from the address book container.
ResolveNames	Resolves entries from an address book container.

IMailUser : IMAPIProp

[New - Windows 95]

The **IMailUser** interface has no unique methods of its own; methods inherited from the **IMAPIProp** interface can be called through **IMailUser** for use with mail user objects. For more information about using mail user objects, see *MAPI Programmer's Guide*.

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

IMAPIAdviseSink : IUnknown

[New - Windows 95]

The **IMAPIAdviseSink** interface is used to implement an advise sink for receiving notification events. Typically, the client application creates a different advise sink object each time it registers an object for notification about changes to another object (that is, for each call to an **Advise** method). The application chooses what data is retained in the advise sink object's internal data structures, and it controls what the **IMAPIAdviseSink::OnNotify** method does when a notification event occurs. For more information on **IMAPIAdviseSink::OnNotify**, see its reference entry.

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	Informs a client application that one or more notification events of the type indicated by the event mask passed in a previous **Advise** call has occurred.

IMAPIAdviseSink::OnNotify

[New - Windows 95]

Informs a client application that one or more notification events of the type indicated by the event mask passed in a previous **Advise** call has occurred.

Syntax

HRESULT 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 Values

S_OK
The call succeeded and has returned the expected value or values.

Comments

Service providers call the **IMAPIAdviseSink::OnNotify** method of a client application's advise sink object when changes occur to an object or a table for which notifications have been requested.

Each notification describes a separate event. In the *lpNotifications* parameter, the client application passes a **NOTIFICATION** structure; the type of each specific event is indicated within a substructure of that **NOTIFICATION**. The contents of the **NOTIFICATION** thus vary depending on the event type; not all members of the **NOTIFICATION** are used for each event type. The following table lists the possible event types along with their corresponding substructures within **NOTIFICATION**:

Notification event type	**Corresponding data 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 application should not modify or free the **NOTIFICATION** structure passed to **OnNotify**. The data in the structure is valid only until the **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 of multiple MAPI calls can be also combined and delivered in one call to **OnNotify**, depending upon the provider implementation.

A typical advise sink object is one 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 data structures describing the contents of the dialog box. The **OnNotify** method of such an 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 application retains the notification connection number returned in the *ulConnection* parameter by **Advise**. When the application no longer requires notifications, 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 your 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, your provider should use 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**.

See Also

HrAllocAdviseSink function, **HrThisThreadAdviseSink** function, **IMAPISupport::Notify** method, **NOTIFICATION** structure

IMAPIContainer : IMAPIProp

[New - Windows 95]

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
Called by:	Client applications

Vtable Order

GetContentsTable	Returns a pointer to the contents table of a container.
GetHierarchyTable	Returns a pointer to a hierarchy table object, which holds hierarchically grouped information about the child containers in the current container.
OpenEntry	Opens a message or child container object contained in the currently open container and returns a pointer to the object to provide further access.
SetSearchCriteria	Sets the search criteria for a particular search-results container.
GetSearchCriteria	Obtains the search criteria for a particular search results container.

IMAPIContainer::GetContentsTable

[New - Windows 95]

Returns a pointer to the contents table of a container.

Syntax

HRESULT GetContentsTable(**ULONG** *ulFlags*, **LPMAPITABLE FAR** * *lppTable*)

Parameters

ulFlags
Input parameter containing a bitmask of flags used to control how the contents table is returned. The following flags can be set:

MAPI_ASSOCIATED
Used with folders, but not address book containers or distribution lists. Indicates the provider should return the table of associated entries rather than the standard table. The "associated entries" contents table is completely separate from the standard contents table, and client applications can (for example) use this 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 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 8-bit format.

lppTable
Output parameter pointing to a variable where the pointer to the contents table 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_NO_SUPPORT
The table has no content.

Comments

Use the **IMAPIContainer::GetContentsTable** method to get a pointer to the contents table for a container. The returned table contains the default column set for that table. To examine the information the contents table holds, your application can use the methods of the **IMAPITable** interface.

Address book contents tables typically contain the following property columns:

PR_ADDRTYPE
The address type for the entry.

PR_DISPLAY_NAME
The object's display name.

PR_DISPLAY_TYPE
Integer mapping the entry to its associated icon or other display element.

PR_ENTRYID
The object's entry identifier.

PR_INSTANCE_KEY
    The search key for the row.
PR_OBJECT_TYPE
    Indicate the type of the MAPI object.

Folder contents tables typically contain the following property columns:

PR_CLIENT_SUBMIT_TIME
    Contains the time the message sender marked the message for submission.
PR_DISPLAY_TO
    The names of the message's primary recipients.
PR_ENTRYID
    The object's entry identifier.
PR_HASATTACH
    Indicates whether the message contains any attachments.
PR_INSTANCE_KEY
    The search key for the row.
PR_LAST_MODIFICATION_TIME
    The data and time when the object was last modified.
PR_MAPPING_SIGNATURE
    Contains the mapping signature for named properties of the object.
PR_MESSAGE_CLASS
    Contains a text string identifying the sender-defined message class for the message.
PR_MESSAGE_DELIVERY_TIME
    Contains the date and time when the message was delivered by a transport provider.
PR_MESSAGE_FLAGS
    Contains a bitmask of flags indicating the current state of the message.
PR_MESSAGE_SIZE
    The approximate size, in bytes, of the data transferred if the message is moved from one message store to another.
PR_MSG_STATUS
    Contains a bitmask of client application or service provider-defined flags describing the status of the message.
PR_NORMALIZED_SUBJECT
    The subject of a message with any reply of forwarding prefixes removed.
PR_OBJECT_TYPE
    The type of an object, such as folder, message, distribution list, and so on.
PR_PARENT_ENTRYID
    The entry identifier containing the message object.
PR_PRIORITY
    The relative priority of a message.
PR_RECORD_KEY
    A unique binary-comparable identifier for an object.
PR_SENDER_NAME
    The message sender's display name as set by the transport provider handling the message.
PR_SENSITIVITY
    The message sender's opinion about the sensitivity of the message.
PR_STORE_ENTRYID
    The unique entry identifier for the message store where the object resides.
PR_STORE_RECORD_KEY
    A unique binary-comparable identifier for the message store in which the object resides.

PR_SUBJECT
    The subject of a message.

Providers that implement **GetContentsTable** must also support client application calls to the **IMAPIProp::OpenProperty** method attempting to open the PR_CONTAINER_CONTENTS property. Providers must also return PR_CONTAINER_CONTENTS with any **IMAPIProp::GetProps** or **IMAPIProp::GetPropList** calls.

Strings in contents table columns returned by **GetContentsTable** might be truncated at 255 characters by service providers. To determine whether a string within a column is truncated, a client application should check all strings exactly 255 characters in length by opening the column entry from which the string came and checking the corresponding property. Depending on your application'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 the **QueryColumns** method returns before the application that contains the content stable 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 application that contains the contents table calls the **IMAPITable::SetColumns** method.
- Controls the property types of the sort order specification returned by the **IMAPITable::QuerySortOrder** method before the application 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

[New - Windows 95]

Returns a pointer to a hierarchy table object, which holds hierarchically grouped information about the child containers in the current container.

Syntax

HRESULT GetHierarchyTable(**ULONG** *ulFlags*, **LPMAPITABLE FAR** * *lppTable*)

Parameters

ulFlags
Input parameter containing a bitmask of flags used to control 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 row indicates the depth, relative to the open container, of the given child container. The open container's immediate child containers are at depth zero. If the flag CONVENIENT_DEPTH is not given, the table contains the immediate child containers only.

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_UNICODE
Indicates the returned strings are in Unicode format. If the MAPI_UNICODE flag is not set, the strings are in 8-bit format.

lppTable
Output parameter pointing to a variable where the pointer to the hierarchy table 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_NO_SUPPORT
The table has contents but no folders.

Comments

Use the **IMAPIContainer::GetHierarchyTable** method to get a pointer to a hierarchy table object, which holds information about the child containers in the current container. To examine the information the hierarchy table holds, your application can use the methods of the **IMAPITable** interface.

Each row of the hierarchy table contains a set of columns holding information about one child container. The hierarchy table can include all columns visible in the current table view, but for many tables the initial active columns are those that most implementations can produce quickly. The initial active columns for a hierarchy table are those columns the **IMAPITable::QueryColumns** method returns before the application that contains the hierarchy table calls the **IMAPITable::SetColumns** method. The initial active columns for a newly opened address book hierarchy table must include at least the following property columns:

PR_CONTAINER_FLAGS
Flags describing the address book container.

PR_DEPTH
: The relative depth of indentation of the child container's name in the hierarchy table. The depth is zero-based, with zero being leftmost.

PR_DISPLAY_NAME
: The child container's display name.

PR_DISPLAY_TYPE
: Integer mapping the entry to its associated icon or other display element.

PR_ENTRYID
: The child container's entry identifier.

PR_INSTANCE_KEY
: The search key for the row.

PR_OBJECT_TYPE
: Indicate the type of the MAPI object.

PR_AB_PROVIDER_ID
: Identifier indicating the address book provider responsible for the data in the address book.

Hierarchy tables for folders typically include the following property columns:

PR_DEPTH
: The relative depth of indentation of the child container's name in the hierarchy table. The depth is zero-based, with zero being leftmost.

PR_DISPLAY_NAME
: The child container's display name.

PR_COMMENT
: A messaging user- defined comment about the purpose or content of the child container.

PR_ENTRYID
: The child container's entry identifier.

PR_INSTANCE_KEY
: The search key for the row.

PR_STATUS
: A bitmask of flags defining the child container's status. PR_STATUS only applies to folders.

PR_SUBFOLDERS
: A property that contains TRUE if the child container contains subfolders, and FALSE otherwise. PR_SUBFOLDERS only applies to folders.

PR_FOLDER_TYPE
: A property that indicates whether the child container is a root folder, a normal folder, or a search-results folder. PR_FOLDER_TYPE only applies to folders.

PR_SUBJECT
: The subject of the entry.

Providers that implement **GetHierarchyTable** must also support client application calls to the **IMAPIProp::OpenProperty** method attempting to open the PR_CONTAINER_HIERARCHY property. Providers must also return PR_CONTAINER_HIERARCHY with any **IMAPIProp::GetProps** or **IMAPIProp::GetPropList** call.

Strings in hierarchy table columns returned by **GetHierarchyTable** might be truncated at 255 characters by service providers. To determine whether a string within a column is truncated, a client application should check all strings exactly 255 characters in length by opening the column entry from which the string came and checking the corresponding property. 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 the **IMAPITable::QueryColumns** method. The initial active columns for a hierarchy table are those columns the **QueryColumns** method returns before the application that contains the hierarchy table calls the **IMAPITable::SetColumns** method.
- 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 application that contains the hierarchy table calls the **IMAPITable::SetColumns** method.
- Controls the property types of the sort order specification returned by the **IMAPITable::QuerySortOrder** method before the application that contains the hierarchy table calls the **IMAPITable::SortTable** method.

See Also

IMAPIProp::GetPropList method, **IMAPIProp::GetProps** method, **IMAPITable : IUnknown** interface, PR_CONTAINER_HIERARCHY property

IMAPIContainer::GetSearchCriteria

[New - Windows 95]

Obtains the search criteria for a particular search results container.

Syntax

HRESULT GetSearchCriteria(**ULONG** *ulFlags*, **LPSRestriction FAR** * *lppRestriction*, **LPENTRYLIST FAR** * *lppContainerList*, **ULONG FAR** * *lpulSearchState*)

Parameters

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 8-bit format.

lppRestriction
Output parameter pointing to a pointer to a structure holding the restriction criteria that defines the conditions of the search. If your application passes NULL in the *lppRestriction* parameter, the **IMAPIContainer::GetSearchCriteria** method returns no restriction set.

lppContainerList
Output parameter pointing to a pointer to a structure that lists the entry identifiers included in the search. If your application passes NULL in the *lppContainerList* parameter, **GetSearchCriteria** doesn't return an entry identifier list.

lpulSearchState
Output parameter containing a bitmask of flags used to indicate the current state of the search. If your application sends NULL in the *lpulSearchState* parameter, **GetSearchCriteria** doesn't return any flags. The following flags can be set for *lpulSearchState:*

SEARCH_FOREGROUND
Indicates the search will run at high priority relative to other searches. If this flag is not set it means that the search will run 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 it means that the CPU-intensive part of the search's operation is over. This flag only has meaning if the search is active (SEARCH_RUNNING is set).

SEARCH_RECURSIVE
Indicates the search will look in indicated folders and all their subfolders for matching messages. If this flag is not set it means that only the folders explicitly included in the last call to **SetSearchCriteria** on this search folder will be looked in for matching messages.

SEARCH_RUNNING
Indicates the search is active (live) and its contents table is being updated to reflect database changes in the store. If this flag is not set it means the search is inactive and its 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 application did not set its search criteria before **IMAPIFolder::GetSearchCriteria** was called.

Comments

Use the **IMAPIContainer::GetSearchCriteria** method to obtain the search criteria for a particular search-results container. A search-results container is a folder that contains the findings of a query that searched for information based on specific criteria. The search criteria is created using the **IMAPIContainer::SetSearchCriteria** method. **GetSearchCriteria** is primarily used with folders; 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 advanced search dialog boxes with address books, see *MAPI Programmer's Guide*.

A call to **GetSearchCriteria** can only be used on folders of type FOLDER_SEARCH. If **GetSearchCriteria** is called before your application sets search criteria for the search-results folder, **GetSearchCriteria** returns the value MAPI_E_NOT_INITIALIZED.

When your application is done with the data structures returned by **GetSearchCriteria** in the *lppRestriction* and *lppContainerList* parameters, 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

[New - Windows 95]

Opens a message or child container object contained in the currently open container and returns a pointer to the object to provide further access.

Syntax

HRESULT OpenEntry(**ULONG** *cbEntryID*, **LPENTRYID** *lpEntryID*, **LPCIID** *lpInterface*, **ULONG** *ulFlags*, **ULONG FAR** * *lpulObjType*, **LPUNKNOWN FAR** * *lppUnk*)

Parameters

cbEntryID
Input parameter containing the number of bytes in the entry identifier pointed to by the *lpEntryID* parameter.

lpEntryID
Input parameter pointing to the entry identifier of the object to be opened.

lpInterface
Input parameter pointing to the interface identifier for the indicated object. Passing NULL in the *lpInterface* parameter indicates that the return value is cast to the standard interface for the indicated object. The *lpInterface* parameter can also be set to an appropriate interface identifier for the object being opened.

ulFlags
Input parameter containing a bitmask of flags used to control how the object is opened. The following flags can be used:

MAPI_BEST_ACCESS
Indicates the object should be opened with the maximum privileges allowed to the user. For example, if the client application has write privilege, open the object with write privilege; if the client application has read-only privilege, open the object with read-only privilege. The client application can learn the privilege 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 application. 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 client applications should not assume that write access was granted.

lpulObjType
Output parameter pointing to a variable where the object type for 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 to modify a read-only object or an attempt to access an object for which the user has insufficient permissions.

MAPI_E_UNKNOWN_ENTRYID
The object indicated in the *lpEntryID* parameter is not recognized. This return value is typically returned if the message store or address book provider that the object is contained within is not

open.

MAPI_E_NOT_FOUND
The object indicated in the *lpEntryID* parameter does not exist.

Comments

Use the **IMAPIContainer::OpenEntry** method to open a message or child container contained in the currently open container. The entry identifier in the *lpEntryID* parameter indicates which object to open. **IMAPIContainer::OpenEntry** returns a pointer to the opened object. Client applications are guaranteed to be able to open entries that are within the container using the **IMAPIContainer::OpenEntry** call. 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; in that case your provider should return a modifiable object. If your 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 on the opened object. Passing NULL in *lpInterface* indicates the standard MAPI interface for that type of object should be used. The address book provider can implement other interfaces for the object if needed by passing the interface identifier for the interface that should be returned.

If your application passes NULL for the *lpEntryID* parameter, it indicates the service provider should open the top-level container in its container hierarchy (that is, the root folder). Your application should check the value in the *lpulObjType* parameter to verify the returned object has the expected object type. After performing this check, your application should cast the address returned in the *lppUnk* parameter to the address of a message or folder object pointer.

IMAPIContainer::SetSearchCriteria

[New - Windows 95]

Sets the search criteria for a particular search-results container.

Syntax

HRESULT SetSearchCriteria(**LPSRestriction** *lpRestriction*, **LPENTRYLIST** *lpContainerList*, **ULONG** *ulSearchFlags*)

Parameters

lpRestriction
Input parameter pointing to a structure holding the restriction criteria that define the conditions of the search. If your application passes NULL in the *lpRestriction* parameter, then the most recently used search criteria for this search-results folder are used again. Your application should not pass NULL in *lpRestriction* for the first search within a container.

lpContainerList
Input parameter pointing to a structure that lists the entry identifiers to be included in the search. If your application passes NULL in the *lpContainerList* parameter, then the most recently used entry identifiers for this search-results folder are included in the new search. Your application should not pass NULL in *lpContainerList* for the first search within a container.

ulSearchFlags
Input parameter containing a bitmask of flags used to control how the search is performed. The following flags can be set:

BACKGROUND_SEARCH
Indicates that the search will run at normal priority relative to other searches. This flag cannot be set at the same time as FOREGROUND_SEARCH.

FOREGROUND_SEARCH
Indicates the search has foreground priority. This flag cannot be set at the same time as BACKGROUND_SEARCH.

RECURSIVE_SEARCH
Searches indicated folders and all their subfolders. Cannot be set at the same time as SHALLOW_SEARCH.

RESTART_SEARCH
Indicates that a search which is inactive should be restarted. In addition to restarting an inactive search, this flag must be passed in on the first call to **SetSearchCriteria** in order to initiate a search. This flag cannot be set at the same time as STOP_SEARCH.

SHALLOW_SEARCH
Indicates the search will only look in the indicated folders for matching messages. This flag cannot be set at the same time as RECURSIVE_SEARCH.

STOP_SEARCH
Stops an ongoing search (if any). Cannot be set at the same time as RESTART_SEARCH.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

Comments

Use the **IMAPIContainer::SetSearchCriteria** method to set the search criteria for a particular search-results folder within a container. A search-results folder is a folder that contains the findings of a query that searches for information based on specific criteria. A call to **SetSearchCriteria** can only be performed for folders of type FOLDER_SEARCH. Address books providers do not set up their search criteria using **SetSearchCriteria**; they set restrictions on the contents table for the address book container. For more information on setting search criteria on address book containers, see *MAPI Programmer's Guide*.

An application generally accomplishes a search through a sequence of MAPI calls. First, the application must create a search-results folder by using **IMAPIFolder::CreateFolder** to create a folder and to specify for the folder a type of FOLDER_SEARCH. Next, the application calls the **SetSearchCriteria** method to define the search criteria and to initiate the search. Finally, the application browses through the search results by using the **IMAPIContainer::GetContentsTable** method on the search-results folder to obtain a table that describes the messages in the folder.

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 restriction has been applied. A search operation only works against this merged contents table, it does not search through other search-results folders. The results of the search only returns the messages that match the search criteria; the folder hierarchy is not returned.

The search criteria define the conditions of the search, including the search keys, the folders to be searched, any conditional operations using the logical-OR, logical-AND, and logical-NOT operators, and so on.

The restrictions that set search criteria for **SetSearchCriteria** can contain subobject restrictions for a message's recipient table and attachment tables. A subobject restriction is a search restriction on an object held within the object being searched (for example, an attachment within a message); such a restriction is held in an **SSubRestriction** structure. The PR_MESSAGE_RECIPIENTS and PR_MESSAGE_ATTACHMENTS properties are used when requesting subobject restrictions. For instance, suppose a client application needs to create a search-results folder whose contents are messages containing attached files with extension ".MSS." In this case, the application sets a search criterion by using an **SSubRestriction** structure for the PR_MESSAGE_ATTACHMENTS property that points to an **SPropertyRestriction** structure holding a property restriction of PR_ATTACH_EXTENSION = MSS. Service providers that do not support subobject restrictions should return the value MAPI_E_TOO_COMPLEX for attempted subobject restrictions.

If your application sets the FOREGROUND_SEARCH flag in the *ulSearchFlags* parameter, the provider performs the indicated search at high priority, possibly degrading the performance of the client application. If your application does not set FOREGROUND_SEARCH, the provider performs the search in the background. In either case, MAPI immediately returns control to the calling application after finishing the search.

An application 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 begin again, but other changes to the search criteria can cause a search to begin again.

When an application is through using a search-results folder, it can delete the folder or leave it for later use. If your application deletes 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 folders. For more information on supporting these operations, see the reference entries for **IMAPIContainer::OpenEntry** and the **IMAPIFolder** interface. These operations apply to items within the search folder, not the search folder itself. Items cannot be created within, or copied into, search folders.

Providers can use the following idle functions for idle time processing: **FtgRegisterIdleRoutine**, **DeregisterIdleRoutine**, **EnableIdleRoutine**, and **ChangeIdleRoutine**.

See Also

ChangeIdleRoutine function, **DeregisterIdleRoutine** function, **EnableIdleRoutine** function, **FtgRegisterIdleRoutine** function, **IMAPIContainer::GetContentsTable** method, **IMAPIContainer::OpenEntry** method, **IMAPIFolder::CreateFolder** method, **IMAPIFolder : IMAPIContainer** interface, **SPropertyRestriction** structure, **SRestriction** structure, **SSubRestriction** structure

IMAPIControl : IUnknown

[New - Windows 95]

The **IMAPIControl** interface is used to activate and change the state of controls defined by display tables and configuration property sheets, as well as message and recipient option dialog boxes. The most common usage is for enabling button controls. To get a control object supporting the **IMAPIControl** interface, call **IMAPIProp::OpenProperty** on the display table containing the control. For more information on working with display tables and control objects, see *MAPI Programmer's Guide*.

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:	Client applications, MAPI

Vtable Order

GetLastError	Returns a **MAPIERROR** structure containing information about the last error that occurred for a control object.
Activate	Activates a control object.
GetState	Retrieves the state of the control (that is, whether it is enabled or disabled).

IMAPIControl::Activate

[New - Windows 95]

Activates a control object.

Syntax

HRESULT Activate(**ULONG** *ulFlags*, **ULONG** *ulUIParam*)

Parameters

ulFlags
  Reserved; must be zero.

ulUIParam
  Input parameter containing a handle to the parent window that the dialog box is modal to.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

Comments

Client applications call the **IMAPIControl::Activate** method to activate a control object. Usually, the control is a button in a dialog box or property sheet. In the *ulUIParam* parameter, your client should pass the handle for the window the dialog box is modal to. Your application should not call **Activate** if the MAPI_DISABLED value was returned in the *lpulState* parameter when **IMAPIControl::GetState** was called.

See Also

IMAPIControl::GetState method

IMAPIControl::GetLastError

[New - Windows 95]

Returns a **MAPIERROR** structure containing information about the last error that occurred for the session object.

Syntax

HRESULT GetLastError(**HRESULT** *hResult*, **ULONG** *ulFlags*, **LPMAPIERROR FAR** * *lppMAPIError*)

Parameters

hResult
Input parameter containing the result returned for the last call on the session object that returned an error.

ulFlags
Input parameter containing a bitmask of flags. The following flag can be set:

MAPI_UNICODE
Indicates the strings returned 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 8-bit 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.

Comments

Use the **IMAPIControl::GetLastError** method to retrieve information to display as a message to the user regarding the last error returned from a method call on the control object.

To release all the memory allocated by MAPI, client applications need only call the **MAPIFreeBuffer** function for the **MAPIERROR** structure.

The return value from **GetLastError** must be S_OK for the application to make use of the **MAPIERROR** structure. Even if the return value is S_OK, it is still possible that a **MAPIERROR** structure won't be returned. If the implementation cannot determine what the last error was, or if a MAPIERROR structure is not available for that error, a pointer to NULL is returned in the *lppMAPIError* parameter instead.

See Also

MAPIERROR structure, **MAPIFreeBuffer** function

IMAPIControl::GetState

[New - Windows 95]

Retrieves the state of the control (that is, whether it is enabled or disabled).

Syntax

HRESULT GetState(**ULONG** *ulFlags*, **ULONG FAR** * *lpulState*)

Parameters

ulFlags
Reserved; must be zero.

lpulState
Output parameter containing the value used to indicate the state of the control. One of the following values can be set:

MAPI_DISABLED
Indicates that the control should be disabled and unselectable.

MAPI_ENABLED
Indicates that the control should be enabled.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

Comments

Client applications call the **IMAPIControl::GetState** method to retrieve the state of a control object – that is, whether the control is enabled or disabled. Usually, the control is a button in a dialog box. In the case of a button, if it is enabled it can respond to a mouse click or key press; if it is disabled, it is displayed as grayed and cannot be selected by the user.

See Also

IMAPIControl::Activate method

IMAPIFolder : IMAPIContainer

[New - Windows 95]

The **IMAPIFolder** interface is used to perform operations on folders such as creating, copying, and deleting 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
Called by:	Client applications

Vtable Order

CreateMessage	Creates a new message within a folder.
CopyMessages	Copies or moves one or more messages between two folders.
DeleteMessages	Deletes one or more messages from a folder.
CreateFolder	Creates a new folder as a subfolder within the message store.
CopyFolder	Copies or moves a folder from its current parent folder to another.
DeleteFolder	Deletes a subfolder from the given folder.
SetReadFlags	Sets or clears the read flags for the messages in a folder, and manages the sending of read receipts.
GetMessageStatus	Obtains the status associated with a message in a folder – for example, whether that message is marked for deletion.
SetMessageStatus	Sets the status associated with a message in a particular folder – for example, whether that message is marked for deletion.
SaveContentsSort	Sets the default sort order for a folder's contents table.
EmptyFolder	Deletes all items from a folder without deleting the folder itself.

IMAPIFolder::CopyFolder

[New - Windows 95]

Copies or moves one or more messages in the current folder to another.

Syntax

HRESULT CopyFolder(**ULONG** *cbEntryID*, **LPENTRYID** *lpEntryID*, **LPCIID** *lpInterface*, **LPVOID** *lpDestFolder*, **LPTSTR** *lpszNewFolderName*, **ULONG** *ulUIParam*, **LPMAPIPROGRESS** *lpProgress*, **ULONG** *ulFlags*)

Parameters

cbEntryID
Input parameter containing the number of bytes in the entry identifier pointed to by the *lpEntryID* parameter.

lpEntryID
Input parameter pointing to the entry identifier of the folder 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 for the *lpInterface* parameter indicates the MAPI interface for the destination folder object will be returned. Client applications must pass NULL. Message store providers can also set the *lpInterface* parameter to an identifier for another appropriate interface for the destination folder object. For example, a message can be copied with 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 copied or moved.

lpszNewFolderName
Input parameter pointing to a string containing the name to be given to the newly created or moved folder. If the client application 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 to the window the dialog box is modal to. The *ulUIParam* parameter is ignored unless the client application 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 the *lpProgress* parameter, the progress information is provided by MAPI. The *lpProgress* parameter is ignored unless the FOLDER_DIALOG flag is set in the *ulFlags* parameter.

ulFlags
Input parameter containing a bitmask of flags used to control how the copy or move operation is accomplished. The following flags can be set:

COPY_SUBFOLDERS
Indicates that all subfolders are included in the copy operation. This is optional for copy operations and is implied for move operations.

FOLDER_DIALOG
Displays a progress-information user interface while the operation proceeds.

FOLDER_MOVE
Indicates that the folder is to be moved. If this flag is not set, the folder will be copied.

MAPI_DECLINE_OK
Informs the provider that if it chooses not to implement **IMAPIFolder::CopyFolder**, that it can 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 8-bit 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 is the same as the name of a sub-folder in the destination folder. Folder names must be unique.

MAPI_E_DECLINE_COPY
Indicates that the provider has chosen not to implement this 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 object might be partially modified.

MAPI_W_PARTIAL_COMPLETION
The call succeeded, but not all entries were successfully copied. Use the HR_FAILED macro to test for this warning, but the call should be handled as a successful return.

Comments

Message store providers use 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 the subfolders of the affected folder. To copy or move all subfolders nested within the copied or moved folder, the client application passes the COPY_SUBFOLDERS flag in the *ulFlags* parameter.

To allow copy or move operations involving more than one folder to continue even if one or more folders specified for the operation do not exist or have already been moved elsewhere, and thus cannot be copied or moved, a message store provider should attempt to 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 application, it returns the value S_OK. If one or more folders cannot be copied or moved, **CopyFolder** returns the value MAPI_W_PARTIAL_COMPLETION. If **CopyFolder** returns a different value, such as MAPI_E_NOT_ENOUGH_MEMORY, that indicates the call did not complete, though it might already have copied or moved one or more folders without being able to continue. The calling application 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 the *lpEntryID* parameter, **CopyFolder** returns MAPI_W_PARTIAL_COMPLETION or MAPI_E_NOT_FOUND, depending on a message store's implementation.

IMAPIFolder::CopyMessages

[New - Windows 95]

Copies or moves one or more messages between two folders.

Syntax

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 be copied or moved.

lpInterface
Input parameter pointing to the interface identifier (IID) for the destination folder object indicated in the *lpDestFolder* parameter. Passing NULL for the *lpInterface* parameter indicates the MAPI interface for the destination folder object will be returned. Client applications must pass NULL. Message store providers can also set the *lpInterface* parameter to an identifier for another appropriate interface for the destination folder object. For example, a message can be copied with IID_IUnknown, IID_IMAPIProp, IID_IMAPIContainer, or IID_IMAPIFolder.

lpDestFolder
Input parameter pointing to the open destination folder where the messages identified in the *lpMsgList* parameter are to be copied or moved.

ulUIParam
Input parameter containing the handle of the window the dialog box is modal to. The *ulUIParam* parameter is ignored unless the client application 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 the *lpProgress* parameter, the progress information is provided by MAPI. The *lpProgress* parameter is ignored unless the MESSAGE_DIALOG flag is set in the *ulFlags* parameter.

ulFlags
Input parameter containing a bitmask of flags used to control how the copy or move operation is accomplished. The following flags can be set:

MAPI_DECLINE_OK
Informs the provider that if it chooses not to implement **IMAPIFolder::CopyMessage**, that it can immediately return MAPI_E_DECLINE_COPY.

MESSAGE_DIALOG
Displays a progress-information user interface 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_DECLINE_COPY
Indicates that the service provider has chosen not to implement this operation.

MAPI_W_PARTIAL_COMPLETION
The call succeeded, but not all entries were successfully copied. Use the HR_FAILED macro to test for this warning; however, the call should be handled as a successful return

Comments

Message store providers use the **IMAPIFolder::CopyMessages** method to copy or move messages from one folder to another. Your 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 the messages by parent folder.

Messages that do not exist, have already been moved elsewhere, are open with read-write access, or are currently submitted, cannot be copied or moved. However, to allow copy or move operations involving more than one message to continue even if one or more messages specified for the operation cannot be copied or moved, a message store provider should attempt to complete the operation as best it can for each message specified. The provider should stop the operation without completing it only in case of failures it cannot control, such as running out of memory or disk space, message store corruption, and so on.

If **CopyMessages** successfully completes the copy or move operation for every message requested by the client application, S_OK is returned. If one or more messages cannot be copied or moved, **CopyMessages** returns the value MAPI_W_PARTIAL_COMPLETION. If **CopyMessages** returns a different value, such as MAPI_E_NOT_ENOUGH_MEMORY, that indicates the call did not complete, though it might already have copied or moved one or more messages without being able to continue. The calling application cannot proceed on the assumption that an error return implies no work was done.

IMAPIFolder::CreateFolder

[New - Windows 95]

Creates a new folder as a subfolder within the message store.

Syntax

HRESULT CreateFolder(**ULONG** *ulFolderType*, **LPTSTR** *lpszFolderName*, **LPTSTR** *lpszFolderComment*, **LPCIID** *lpInterface*, **ULONG** *ulFlags*, **LPMAPIFOLDER FAR *** *lppFolder*)

Parameters

ulFolderType
Input parameter indicating the type of folder to create. Your message store provider should pass in the *ulFolderType* parameter to one of the following values:

FOLDER_GENERIC
Indicates a generic folder should be created.

FOLDER_SEARCH
Indicates a search folder should be created.

lpszFolderName
Input parameter pointing to a string containing the name to be given to the new folder.

lpszFolderComment
Input parameter pointing to a string containing the comment to be associated with the new folder. This string becomes the value of the folder's PR_COMMENT property. If NULL is passed in the *lpszFolderComment* parameter, the folder has no initial comment.

lpInterface
Input parameter indicating the interfaces identifier that should be used for the folder being returned in the *lppFolder* parameter. Passing NULL for the *lpInterface* parameter indicates IID_IMAPIFolder is to be used. Client applications must pass NULL. Message store providers can also set the *lpInterface* parameter to any of the interfaces IMAPIFolder is derived from, that is: IID_IUnknown, IID_IMAPIProp, or IID_IMAPIContainer.

ulFlags
Input parameter containing a bitmask of flags used to control the creation of the folder. 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 application. 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 8-bit format.

OPEN_IF_EXISTS
Allows the method to succeed, even if the folder already exists, by returning a new folder object for the existing folder. Note that message stores that allow sibling folders to have the same name might fail to open an existing folder if more than one exists with the name supplied in the *lpszFolderName* parameter.

lppFolder
Output parameter pointing to a variable where the 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 the *lpszFolderName* parameter already exists. Folder names must be unique.

Comments

Message store providers use the **IMAPIFolder::CreateFolder** method to create new normal 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. Client applications should be able to handle the value MAPI_E_COLLISION returned if this rule is not followed.

To determine the entry identifier of the newly created folder, your implementation calls the **IMAPIProps::GetProps** method to read the new folder's PR_ENTRYID property while the folder is still open.

See Also

IMAPIProp::GetProps method

IMAPIFolder::CreateMessage

[New - Windows 95]

Creates a new message within a folder.

Syntax

HRESULT CreateMessage(**LPCIID** *lpInterface*, **ULONG** *ulFlags*, **LPMESSAGE FAR** * *lppMessage*)

Parameters

lpInterface
Input parameter indicating the interfaces identifier that should used for the message being returned in the *lppMessage* parameter. Passing NULL for the *lpInterface* parameter indicates IID_IMessage is to be used. Client applications must pass NULL. Message store providers can also set the *lpInterface* parameter to any of the interfaces IMessage is derived from, that is: IID_IUnknown, or IID_IMAPIProp.

ulFlags
Input parameter containing a bitmask of flags used to control how the message is created. The following flags can be set:

MAPI_ASSOCIATED
Indicates that an associated item should be created.

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.

lppMessage
Output parameter pointing to a variable where the pointer to the newly created message object is stored.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

Comments

Message store providers use the **IMAPIFolder::CreateMessage** method to create a new message in a folder.

The MAPI_ASSOCIATED flag should be set in the *ulFlags* parameter to create objects associated with the folder and listed in the associated contents table. Associated entries allow client applications to associate invisible data with a folder.

Although a new message has a unique PR_RECORD_KEY property, its PR_ENTRYID property might not be available until a client application 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 application calls the **SaveChanges** method to save changes.

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

[New - Windows 95]

Deletes a subfolder from the given folder.

Syntax

HRESULT DeleteFolder(**ULONG** *cbEntryID*, **LPENTRYID** *lpEntryID*, **ULONG** *ulUIParam*, **LPMAPIPROGRESS** *lpProgress*, **ULONG** *ulFlags*)

Parameters

cbEntryID
Input parameter containing the number of bytes in the entry identifier pointed to by the *lpEntryID* parameter.

lpEntryID
Input parameter pointing to the entry identifier of the folder to delete.

ulUIParam
Input parameter containing the handle of the window the dialog box is modal to. 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, the progress information is provided by MAPI. The *lpProgress* parameter is ignored unless the FOLDER_DIALOG flag is set in the *ulFlags* parameter.

ulFlags
Input parameter containing a bitmask of flags used to control the deletion of the folders. The following flags can be set:

DEL_FOLDERS
Deletes all subfolders of the subfolder indicated in the *lpEntryID* parameter.

DEL_MESSAGES
Deletes all messages in the folder indicated in the *lpEntryID* parameter.

FOLDER_DIALOG
Displays a progress-information user interface while the operation proceeds.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

MAPI_E_HAS_FOLDERS
The folder being deleted contains folders and the DEL_FOLDERS flag was not set. The folder was not deleted.

MAPI_E_HAS_MESSAGES
The folder being deleted contains messages and the DEL_MESSAGES flag was not set. The folder containing the messages was not deleted.

MAPI_W_PARTIAL_COMPLETION
The call succeeded, but not all of the entries were successfully deleted. Use the HR_FAILED macro to test for this warning, though the call should be handled as a successful return.

Comments

Message store providers use 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 application sets the

DEL_MESSAGES flag in the *ulFlags* parameter; to delete all subfolders in a subfolder, the client application sets the DEL_FOLDERS flag in *ulFlags*. However, no flag need be set to delete any associated items, such as views or form definitions, from a folder.

To allow deletions of more than one subfolder to continue even if one or more subfolders marked for deletion by the calling application do not exist or have been moved elsewhere, and thus cannot be deleted, a message store provider should attempt to complete the deletion as best it can for each subfolder specified. 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 folder marked for deletion, it returns the value S_OK. If one or more folders cannot be deleted, **DeleteFolder** returns the value 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 folders or messages without being able to continue. The calling application cannot proceed on the assumption that an error return implies no work was done.

During a **DeleteFolder** call, messages that are being processed by the MAPI spooler are not deleted, nor does the message store provider attempt to call the **IMsgStore::AbortSubmit** method on 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 folders from being deleted because they still have contents.

IMAPIFolder::DeleteMessages

[New - Windows 95]

Deletes one or more messages from a folder.

Syntax

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 to the window the dialog box is modal to. 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, the progress information is provided by MAPI. The *lpProgress* parameter is ignored unless the MESSAGE_DIALOG flag is set in the *ulFlags* parameter.

ulFlags
: Input parameter containing a bitmask of flags used to control how the messages are deleted. The following flag can be set:

  MESSAGE_DIALOG
  : Displays a progress-information user interface 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. Use the HR_FAILED macro to test for this warning, though the call should be handled as a successful return.

Comments

Message store providers use the **IMAPIFolder::DeleteMessages** method to delete messages from a folder.

Messages that do not exist, have been moved elsewhere, are open with read-write access, or are currently submitted, cannot be deleted. However, to allow deletions of more than one message to continue even if one or more messages marked for deletion by the calling application cannot be deleted, a message store provider should attempt to complete the deletion as best it can for each message specified. 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 the value S_OK. If one or more messages cannot be deleted, **DeleteMessages** returns the value 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 application cannot proceed on the assumption that an error return implies no work was done.

During a **DeleteMessages** call, submitted messages are not deleted, nor does the message store provider attempt to call the **IMsgStore::AbortSubmit** method on such messages. The message is left in the folder in which it resides.

IMAPIFolder::EmptyFolder

[New - Windows 95]

Deletes all items from a folder without deleting the folder itself.

Syntax

HRESULT EmptyFolder(**ULONG** *ulUIParam*, **LPMAPIPROGRESS** *lpProgress*, **ULONG** *ulFlags*)

Parameters

ulUIParam
Input parameter containing the handle of the window the dialog box is modal to. 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, the progress information is provided by MAPI. The *lpProgress* parameter is ignored unless the FOLDER_DIALOG flag is set in the *ulFlags* parameter.

ulFlags
Input parameter containing a bitmask of flags used to control how deletion is performed for the messages and subfolders of a folder. The following flags can be set:

DEL_ASSOCIATED
Subfolders are deleted in their entirety, including all subfolders and associated items. The DEL_ASSOCIATED flag only has meaning for the top-level folder acted on by the call.

FOLDER_DIALOG
Displays a progress-information user interface 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 successfully deleted. Use the HR_FAILED macro to test for this warning, though the call should be handled as a successful return.

Comments

Message store providers use the **IMAIPIFolder::EmptyFolder** method to delete all of a folder's contents without deleting the folder itself.

When the calling application sets the DEL_ASSOCIATED flag in the *ulFlags* parameter, **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 application does not set DEL_ASSOCIATED, **EmptyFolder** does not delete associated information. The DEL_ASSOCIATED flag only has meaning for the top-level folder acted on by the call.

If **EmptyFolder** successfully deletes all subfolders and messages, it returns the value S_OK. If one or more items cannot be deleted, **EmptyFolder** returns the value 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 application cannot proceed on the assumption that an error return implies that no work was done.

During an **EmptyFolder** call, submitted messages are not deleted, nor does the message store provider attempt to call the **IMsgStore::AbortSubmit** method on such messages. The message is left in the folder in which it resides.

IMAPIFolder::GetMessageStatus

[New - Windows 95]

Obtains the status associated with a message in a particular folder – for example, whether that message is marked for deletion.

Syntax

HRESULT GetMessageStatus(**ULONG** *cbEntryID*, **LPENTRYID** *lpEntryID*, **ULONG** *ulFlags*, **ULONG FAR** * *lpulMessageStatus*)

Parameters

cbEntryID
Input parameter containing the number of bytes in 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 receiving a bitmask of flags indicating the message's status. Bits 5 through 15 are reserved and must be zero; bits 16 through 31 are available for application-specific use.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

Comments

Message store providers use the **IMAPIFolder::GetMessageStatus** method to find out the status of a message. **GetMessageStatus** returns status information in the *lpulMessageStatus* parameter.

How your application sets, clears, and uses the message status bits in *lpulMessageStatus* depends entirely on your application's 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 application. Users can choose which IPM client they will use. Applications that store their messages in other subtrees can use bits 16 through 31 for their own purposes.

See Also

IMAPIFolder::SetMessageStatus method

IMAPIFolder::SaveContentsSort

[New - Windows 95]

Sets the default sort order for a folder's contents table.

Syntax

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 used to control how the default sort order is set. The following flags can be set:

RECURSIVE_SORT
Indicates the default sort order should be for the indicated folder and all of 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.

Comments

Message store providers use 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**; such providers return MAPI_E_NO_SUPPORT.

See Also

IMAPIContainer::GetContentsTable method, **SSortOrderSet** structure

IMAPIFolder::SetMessageStatus

[New - Windows 95]

Sets the status associated with a message in a particular folder – for example, whether that message is marked for deletion.

Syntax

HRESULT SetMessageStatus(**ULONG** *cbEntryID*, **LPENTRYID** *lpEntryID*, **ULONG** *ulNewStatus*, **ULONG** *ulNewStatusMask*, **ULONG FAR *** *lpulOldStatus*)

Parameters

cbEntryID
: Input parameter containing the number of bytes in 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 a message within a folder in the message store. 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 the *ulNewStatusMask* parameter mask, the corresponding bit in *ulNewStatus* is set or cleared for the message within the message store.

lpulOldStatus
: Output parameter pointing to a variable that receives the previous value of the message status flags.

Return Values

S_OK
: The call succeeded and has returned the expected value or values.

Comments

Message store providers use the **IMAPIFolder::SetMessageStatus** method to set the status associated with a message in a folder.

Some message store providers use **SetMessageStatus** to enable applications to negotiate a message lockout operation among themselves. To do so, client applications choose a bit for the lockout bit. Your provider sets the lockout bit using **SetMessageStatus** and examines the previous value in the *lpulOldStatus* parameter to determine if the application caused the chosen 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

[New - Windows 95]

Sets or clears the read flags for the messages in a folder, and manages the sending of read receipts.

Syntax

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 the read flags. If the client application passes NULL in the *lpMsgList* parameter, all messages are processed.

ulUIParam
: Input parameter containing the handle of the window the dialog box is modal to. The *ulUIParam* parameter is ignored unless the client application 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 the *lpProgress* parameter, the progress information is provided by MAPI. The *lpProgress* parameter is ignored unless the FOLDER_DIALOG flag is set in the *ulFlags* parameter.

ulFlags
: Input parameter containing a bitmask of flags used to control the setting of each message's read flag − that is, each message's MSGFLAG_READ flag in its PR_MESSAGE_FLAGS property − and the processing of read receipts. The following flags can be set in the *ulFlags* parameter:

CLEAR_READ_FLAG
: Resets the MSGFLAG_READ flag. No read receipt is sent.

MESSAGE_DIALOG
: Displays a progress-information user interface while the operation proceeds.

GENERATE_RECEIPT_ONLY
: Generates a read receipt but does not change the state of the MSGFLAG_READ flag in PR_MESSAGE_FLAGS.

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.

SUPPRESS_RECEIPT
: Indicates the messaging system should not send a read receipt for each message, but that each message should be marked as read. If the MSGFLAG_READ bit is already set, do nothing.

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 receipts.

MAPI_W_PARTIAL_COMPLETION
: The call succeeded, but not all the entries were successfully processed. Use the HR_FAILED macro to test for this warning, though the call should be handled as a successful return.

Comments

Message store providers use the **IMAPIFolder::SetReadFlags** method to set or clear the read flags for the messages in a folder, and to manage the sending of read receipts, when messages are being moved or copied from one message store to another. **SetReadFlags** sets or clears the MSGFLAG_READ bit in the PR_MESSAGE_FLAGS property.

Messages that do not exist, have been moved elsewhere, are open with read-write access, or are currently submitted, cannot have their read flags set. However, to allow read-flag setting operations involving more than one message to continue even if one or more messages specified cannot have their read flags set, a message store provider should attempt to 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.

If none of the flags are set in the *ulFlags* parameter, the following rules apply:

- If the MSGFLAG_READ bit is already set, do nothing.
- If the PR_READ_RECEIPT_REQUESTED bit is set, send the read receipt and set the MSGFLAG_READ bit.

MAPI_E_INVALID_PARAMETER is returned 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 SUPPRESS_RECEIPT and GENERATE_RECEIPT_ONLY are set in *ulFlags*, the PR_READ_RECEIPT_REQUESTED bit, if it is set, is turned off, and a read receipt should not be sent.

Note Providers can optimize report behavior so that a client application's setting a message attribute to get a read or delivery report is only a request and the provider can support not sending read or delivery reports. However, some message stores do not support the suppression of read reports for some messages. If the client application calls **SetReadFlag** on such a message with the SUPPRESS_RECEIPT flag set in the *ulFlags* parameter, **SetReadFlag** returns the value MAPI_E_NO_SUPPRESS; in this case, MAPI does not mark the message as read and does not generate a report.

If **SetReadFlags** successfully completes processing for every message specified, it returns the value S_OK. If one or more messages cannot be processed, **SetReadFlags** returns the value 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 application cannot proceed on the assumption that an error return implies that no work was done.

See Also

IMessage::SetReadFlag method, PR_MESSAGE_FLAGS property

IMAPIForm : IUnknown

[New - Windows 95]

The **IMAPIForm** interface is implemented by form servers for the benefit of form viewers.

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.
GetViewContext	Returns the current view context for a form.
ShutdownForm	Closes a form.
DoVerb	Requests that a form object perform one of its verbs.
Advise	Registers a form viewer for notification on changes to a form.
Unadvise	Removes a form viewer's registration for notification of form changes.

IMAPIForm::Advise

[New - Windows 95]

Registers a form viewer for notification on changes to a form object.

Syntax

HRESULT Advise(**LPMAPIVIEWADVISESINK** *pAdvise*, **ULONG FAR** * *pulConnection*)

Parameters

pAdvise
: Input parameter pointing to the view advise-sink object to be called when an event for the form object occurs about which notification has been requested.

pulConnection
: Output parameter pointing to a variable where the connection number for the notification registration is stored. The connection number must be nonzero.

Return Values

S_OK
: The call succeeded and has returned the expected value or values.

Comments

Form viewers call the **IMAPIForm::Advise** method to register for notification callbacks when changes occur on the form. The 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 that pointer with its notification callbacks. The form server implementation should call the **IUnknown::AddRef** method on the view advise-sink object to retain the object until notification registration is canceled. The **Advise** implementation assigns a nonzero connection number to the notification registration, which is returned in the *pulConnection* parameter.

To cancel a notification registration, your form viewer calls the **IMAPIForm::Unadvise** method. During the call to **Unadvise**, or shortly thereafter, the saved pointer to the view advise-sink object in *pAdvise* is released.

See Also

IMAPIForm::Unadvise method, **IMAPIViewAdviseSink : IUnknown** interface

IMAPIForm::DoVerb

[New - Windows 95]

Requests a form object perform one of its verbs.

Syntax

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. This parameter can be NULL.

hwndParent
Input parameter containing the handle of the parent window the dialog box is modal to. This parameter should be NULL if the dialog box is not modal.

lprcPosRect
Input parameter pointing to a **RECT** structure used to display the form's window.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

OLEOBJ_S_CANNOT_DOVERB_NOW
Indicates that the verb is valid, but that the object cannot perform the operation now.

Comments

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 the *lpViewContext* parameter, 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, are expected to be modal with respect to the **DoVerb** call (that is, the printing must be finished before the **DoVerb** call returns). Non-modal verbs can be made to act like modal verbs by passing in a view context object in the *lpViewContext* parameter whose **IMAPIViewContext::GetViewStatus** call returns the VCSTATUS_MODAL flag.

The handle in the *hwndParent* parameter must be assumed to have a life span of the **DoVerb** call, but because it can be destroyed immediately upon the call's return, forms should not save the handle.

The **RECT** structure used by the form's window can be obtained using the Windows **GetWindowRect** function.

See Also

IMAPIForm::SetViewContext method, **IMAPIViewContext::GetViewStatus** method

IMAPIForm::GetViewContext

[New - Windows 95]

Returns the current view context for the form object.

Syntax

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

[New - Windows 95]

Sets a form view context as the current view context for the form object.

Syntax

HRESULT SetViewContext(**LPMAPIVIEWCONTEXT** *pViewContext*)

Parameters

pViewContext
Input parameter pointing to the view context object to be set as current.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

Comments

Form viewers call the **IMAPIForm::SetViewContext** method to set a particular form view context to be 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 to determine which status flags have been set for the view context to be set as current.

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 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.

See Also

IMAPIViewContext::GetViewStatus method, **IMAPIViewContext::SetAdviseSink** method

IMAPIForm::ShutdownForm

[New - Windows 95]

Closes a form object.

Syntax

HRESULT Close(**ULONG** *ulSaveOptions*)

Parameters

ulSaveOptions
Input parameter containing a flag used to control how or whether the form's contents are saved when the form is closed. The following mutually exclusive flags 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 choose to implement this as either SAVEOPTS_NOSAVE or SAVEOPTS_SAVEIFDIRTY if no user interface is currently being displayed.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

Comments

Form viewers call the **IMAPIForm::ShutdownForm** method to close a form object. A typical form server's implementation of **ShutdownForm** goes through the following process:

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::OnClose** method to update notifications.
6. Calls **IMAPIViewContext::SetAdviseSink**, 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 viewer and message site interfaces.
10. Returns S_OK.

Form viewers can choose to ignore errors returned by **ShutdownForm** and release the form interface after making the call.

See Also

IMAPIViewAdviseSink::OnShutdown method

IMAPIForm::Unadvise

[New - Windows 95]

Removes a form viewer's registration for notification of form object changes previously established with a call to the **IMAPIForm::Advise** method.

Syntax

HRESULT Unadvise(**ULONG** *ulConnection*)

Parameters

ulConnection
Input parameter containing the number of the registration connection previously returned by a call to **IMAPIForm::Advise**.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

Comments

Form viewers call the **IMAPIForm::Unadvise** method to cancel notifications for a form object. To do so, **Unadvise** releases the pointer to the view advise-sink object passed in the *pAdvise* parameter in the previous call to **IMAPIForm::Advise**. 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 **Unadvise**, 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

[New - Windows 95]

The **IMAPIFormAdviseSink** interface is implemented by form objects, which receive notifications from form viewers. Although part of the form object, **IMAPIFormAdviseSink** is not an interface on a form object. Hence, client applications should not expect to use the **QueryInterface** method from 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 object
Called by:	Form viewers

Vtable Order

OnChange	Notifies a form about a new status for handling commands that activate the next or previous message handled by the form.
OnActivateNext	Identifies whether the message class of the next message can be handled by the current form.

IMAPIFormAdviseSink::OnActivateNext

[New - Windows 95]

Identifies whether the message class of the next message to be displayed can be handled by the current form object.

Syntax

HRESULT OnActivateNext(LPCSTR *lpszMessageClass*, **ULONG** *ulMessageStatus*, **ULONG** *ulMessageFlags,* **LPPERSISTMESSAGE FAR *** *ppPersistMessage*)

Parameters

lpszMessageClass
: Input parameter pointing to the string naming the message class of the form.

ulMessageStatus
: Input parameter containing a bitmask of client application- or service provider-defined flags, copied from the PR_MSG_STATUS property of the message, 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 message, that indicates the current state of the message.

ppPersistMessage
: Output parameter pointing to a variable where the pointer to the IPersistMessage object used for the new form is stored. 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.

Comments

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 NULL in the *ppPersistMessage* parameter and an HRESULT of S_OK if the message class is handled by the current form object. If the form object cannot handle the message's class, 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 that 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

[New - Windows 95]

Notifies a form object about a change in the viewer's status.

Syntax

HRESULT OnChange(**ULONG** *ulDir*)

Parameters

ulDir

Input parameter containing a bitmask of flags for handling changes to the form's viewer status. The value can contain the following flags:

VCSTATUS_READONLY
: The delete, submit, and move operations should be disabled.

VCSTATUS_NEXT
: Indicates that there is a next message in the viewer.

VCSTATUS_PREV
: Indicates that there is a previous message in the viewer.

VCSTATUS_INTERACTIVE
: If this flag is not set, indicates the form should suppress displaying user interface even in response to a verb which might otherwise cause user interface to be displayed.

VCSTATUS_MODAL
: Indicates the form is to be modal to the viewer.

Return Values

S_OK
: The call succeeded and has returned the expected value or values.

Comments

Form viewers call the **IMAPIFormAdviseSink::OnChange** method to notify the form object about a change in the viewer status. Normally the only change is setting or clearing the VCSTATUS_NEXT or VCSTATUS_PREVIOUS flags 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.

The values for VCSTATUS_MODAL and VCSTATUS_INTERACTIVE cannot change values in a view context once it has been created.

See Also

IMAPIViewContext::ActivateNext method

IMAPIFormContainer : IUnknown

[New - Windows 95]

The **IMAPIFormContainer** interface creates and manages form registries within folders. This interface is used to create application-specific form registries.

At a Glance

Specified in header file:	MAPIFORM.H
Object that supplies this interface:	Form container object
Corresponding pointer type:	LPMAPIFORMCONTAINER
Implemented by:	Form registry providers
Called by:	Client applications

Vtable Order

InstallForm	Installs a form into a form registry.
RemoveForm	Removes a particular form from the form registry.
ResolveMessageClass	Resolves a message class to a form within a form container and returns a form information object for that form.
ResolveMultipleMessageClasses	Resolves group 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

[New - Windows 95]

Returns an array of the properties used by all forms installed within a form container.

Syntax

HRESULT CalcFormPropSet(ULONG *ulFlags*, **LPMAPIFORMPROPARRAY FAR *** *ppResults*)

Parameters

ulFlags
Input parameter containing a bitmask of flags used to control how the property array in the *ppResults* parameter is returned. 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 8-bit format.

FORM_PROPSET_INTERSECTION
Indicates the returned array contains the intersection of the forms' properties.

FORM_PROPSET_UNION
Indicates the returned array contains the union of the forms' properties.

ppResults
Output parameter pointing to a variable where the pointer to the returned **MAPIFORMPROPARRAY** array is stored. This array 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.

Comments

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 returns a **MAPIFORMPROPARRAY** structure containing information on 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 registry 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

[New - Windows 95]

Returns the display name of a form container.

Syntax

HRESULT GetDisplay(**ULONG** *ulFlags*, **LPTSTR FAR *** *pszDisplayName*)

Parameters

ulFlags
Input parameter containing a bitmask of flags. The following flag can be set:

MAPI_UNICODE
Indicates returned strings are in Unicode format. If the MAPI_UNICODE flag is not set, the strings are in 8-bit 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

[New - Windows 95]

Installs a form into a form container.

Syntax

HRESULT InstallForm(**ULONG** *ulUIParam*, **ULONG** *ulFlags*, **LPCTSTR** *szCfgPathName*)

Parameters

ulUIParam
Input parameter containing the handle of the parent window that the dialog box is modal to if the *ulFlags* parameter contains the MAPI_DIALOG flag. This parameter can be NULL if the MAPI_DIALOG flag is not also passed.

ulFlags
Input parameter containing a bitmask of flags used to control 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 8-bit format.

MAPIFORM_OVERWRITEONCONFLICT
Indicates a previous definition of the form is to be replaced with the definition of the form being installed.

szCfgPathName
Input parameter containing the path to the forms 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; call **IMAPIFormContainer::GetLastError** to get the **MAPIERROR** structure associated with the error.

MAPI_E_USER_CANCEL
The user canceled the installation of the form, typically by clicking the cancel button in the dialog box.

Comments

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_OVERWRITEONCONFLICT flag is set. Otherwise the form installation is merged with the current form description, if one exists.

- That the path to the form configuration file is a Unicode string, if the MAPI_UNICODE flag is set.

Form registry 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.

Client applications should call **IMAPIFormContainer::GetLastError** if **InstallForm** returns MAPI_E_EXTENDED_ERROR and check the returned **MAPIERROR** structure to determine the condition causing the error.

IMAPIFormContainer::RemoveForm

[New - Windows 95]

Removes a particular form from the form container.

Syntax

HRESULT RemoveForm(**LPCSTR** *szMessageClass*)

Parameters

szMessageClass
: Input parameter containing a string naming the message class of the form to be removed from the registry. 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 the match the message class for any forms in the form registry.

IMAPIFormContainer::ResolveMessageClass

[New - Windows 95]

Resolves a message class to a form within a form container and returns a form information object for that form.

Syntax

HRESULT ResolveMessageClass(**LPCSTR** *szMessageClass*, **ULONG** *ulFlags*, **LPMAPIFORMINFO FAR** * *ppforminfo*)

Parameters

szMessageClass
Input parameter containing a string naming the message class of the form being resolved.

ulFlags
Input parameter containing a bitmask of flags used to control how the message class is resolved. The following flag can be set:

MAPIFORM_EXACTMATCH
Indicates that 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 forms in the form container.

Comments

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 form's properties.

To resolve a message class to a form, your application 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. Client applications should not assume that the class identifier exists in the OLE registry 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

[New - Windows 95]

Resolves group message classes to their forms within a form container and returns an array of form information objects for those forms.

Syntax

HRESULT ResolveMultipleMessageClasses (**LPSMESSAGECLASSARRAY** *pMsgClassArray*,
ULONG *ulFlags*, **LPSMAPIFORMINFOARRAY FAR *** *ppfrminfoarray*)

Parameters

pMsgClassArray
Input parameter pointing to an array containing the message classes to be resolved.

ulFlags
Input parameter containing a bitmask of flags used to control how the message classes are resolved. The following flag can be set:

MAPIFORM_EXACTMATCH
Indicates that 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 the client application passes NULL in the *pMsgClassArray* parameter, the *ppfrminfoarray* contains form information objects for all forms in the registry.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

Comments

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 the *ppforminfoarray* parameter provides further access to each of the forms' properties. If a message class cannot be resolved to a form, NULL is returned for that message class. So although the method returns S_OK, applications should not assume that all message classes have been successfully resolved; applications should check the values in the returned array.

Your application 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.

See Also

IMAPIFormContainer::ResolveMessageClass method

IMAPIFormFactory : IUnknown

[New - Windows 95]

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 *OLE Programmer's Reference, Volume One,* and *Inside OLE, Second Edition.*

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

[New - Windows 95]

Returns a class factory object for a form.

Syntax

HRESULT CreateClassFactory(**REFCLSID** *clsidForm*, **ULONG** *ulFlags*, **LPCLASSFACTORY FAR *** *lppClassFactory*)

Parameters

clsidForm
: Input parameter indicating 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.

Comments

The **IMAPIFormFactory::CreateClassFactory** method is used to obtain a class factory for a specific form from a form runtime package. The same object can be returned by **CreateClassFactory** on multiple calls for the same message class identifier; creating a new instance is not required. The class factory returned will be used to generate a new instance of the form..

IMAPIFormFactory::LockServer

[New - Windows 95]

Keeps an open form server in memory.

Syntax

HRESULT LockServer(**ULONG** *ulFlags*, **ULONG** *fLockServer*)

Parameters

ulFlags
    Reserved; must be zero.

fLockServer
    Input parameter indicating TRUE if the lock count is to be incremented, FALSE otherwise.

Return Values

S_OK
    The call succeeded and has returned the expected value or values.

Comments

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

[New - Windows 95]

The **IMAPIFormInfo** interface gives client applications access to useful properties particular to form definition. By keeping form information in a separate object, the form registry provider can describe a form to a client without activating the form itself.

Unlike most interfaces defined in the MAPIFORM.H 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 registry providers
Called by:	Client applications

Vtable Order

CalcFormPropSet	Returns a pointer to the complete set of properties published 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 the current form in a configuration file with the given filename.
OpenFormContainer	Returns a pointer to the form container object in which the current form is registered.

IMAPIFormInfo::CalcFormPropSet

[New - Windows 95]

Returns a pointer to the complete set of properties published by a form.

Syntax

HRESULT CalcFormPropSet (**ULONG** *ulFlags*, **LPMAPIFORMPROPARRAY FAR *** *ppFormPropArray*)

Parameters

ulFlags
Input parameter containing a bitmask of flags. The following flag can be set:

MAPI_UNICODE
Indicates the returned strings are to be in Unicode format. If the MAPI_UNICODE flag is not set, the strings are in 8-bit format.

ppFormPropArray
Output parameter pointing to a variable where the pointer to the array of returned properties is stored.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

IMAPIFormInfo::CalcVerbSet

[New - Windows 95]

Returns a pointer to the complete set of verbs published by a form.

Syntax

HRESULT CalcVerbSet(**ULONG** *ulFlags*, **LPMAPIVERBARRAY FAR *** *ppMAPIVerbArray*)

Parameters

ulFlags
Input parameter containing a bitmask of flags. The following flag can be set:

MAPI_UNICODE
Indicates the strings are returned in Unicode format. If the MAPI_UNICODE flag is not set, the strings are in 8-bit 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.

Comments

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

[New - Windows 95]

Builds an icon from an icon property of a form.

Syntax

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.

Comments

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 and builds an icon that can be displayed in table views that include property columns for icons.

IMAPIFormInfo::OpenFormContainer

[New - Windows 95]

Returns a pointer to the form container object in which the current form is registered.

Syntax

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

[New - Windows 95]

Saves a description of the current form in a configuration file with the given filename.

Syntax

HRESULT SaveForm(**LPCTSTR** *szFileName*)

Parameters

szFileName
Input parameter containing a string to name 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.

Comments

Client applications call the **IMAPIFormInfo::SaveForm** method to save a description of the current form in a file with the given filename. **SaveForm** creates a configuration file, and the filename passed in the *szFileName* parameter must have the .FDM extension.

Forms can be reinstalled later by being selected from a list of form descriptor messages in a dialog box displayed by form registry providers. The recommended extension for form descriptor messages is "FDM."

IMAPIFormMgr : IUnknown

[New - Windows 95]

The **IMAPIFormMgr** interface is used by form-viewer client applications 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:	Forms registry providers
Called by:	Form viewers

Vtable Order

LoadForm	Launches a form so as to open an existing message.
ResolveMessageClass	Resolves a message class to a form within a form registry and returns a form information object for that form.
ResolveMultipleMessageClasses	Resolves group message classes to their forms within a form registry and returns an array of form information objects for those forms.
CalcFormPropSet	Returns an array of the properties published by a group of forms.
CreateForm	Launches a form so as to create a new message based on that form.
SelectForm	Presents a dialog box that enables the user to select a form, then returns a form information object describing that form.
SelectMultipleForms	Presents a dialog box that enables the user to select multiple forms, then returns an array of form information objects describing the forms.
SelectFormContainer	Presents a dialog box that enables the user to select the form registry into which to install a form, then returns an interface for the registry object the user selected.
OpenFormContainer	Opens an **IMAPIFormContainer** interface on a specific form container.
PrepareForm	Downloads a form for launching.
IsInConflict	Determines whether the form can handle its own conflicts.

IMAPIFormMgr::CalcFormPropSet

[New - Windows 95]

Returns an array of the properties used by a group of forms.

Syntax

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 published properties.

ulFlags
Input parameter containing a bitmask of flags used to control how the property array in the *ppResults* parameter is returned. 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 8-bit format.

FORM_PROPSET_INTERSECTION
Indicates that the returned array contains the intersection of the form's properties.

FORM_PROPSET_UNION
Indicates that the returned array contains the union of the form's properties.

ppResults
Output parameter pointing to a variable where the pointer to the returned **SMAPIFormPropArray** structure is stored. This array contains the properties published 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.

Comments

Client applications 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 returns an **SMAPIFormPropArray** structure containing information on 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 registry providers that do not support Unicode strings should return MAPI_E_BAD_CHARWIDTH if MAPI_UNICODE is passed.

See Also

SMAPIFormPropArray structure

IMAPIFormMgr::CreateForm

[New - Windows 95]

Launches a form so as to create a new message based on that form.

Syntax

HRESULT CreateForm(**ULONG** *ulUIParam*, **ULONG** *ulFlags*, **IMAPIFormInfo** *pfrminfoToActivate*,
  REFIID *refiidToAsk*, **LPVOID FAR** * *ppvObj*)

Parameters

ulUIParam
  Input parameter containing a handle to the main window of the calling application, cast to a ULONG. The *ulUIParam* parameter is used in this case to display a progress user interface while the form is being loaded. The *ulUIParam* parameter is ignored unless the MAPI_DIALOG flag is set in the *ulFlags* parameter.

ulFlags
  Input parameter containing a bitmask of flags used to control the loading of the form. The following flag 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.

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 on the form object created. The *refiidToAsk* parameter must not be NULL.

ppvObj
  Output parameter pointing to a variable where the pointer to the interface returned is stored.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

MAPI_E_NO_INTERFACE
  The requested interfaced is not supported by the form object.

Comments

Client applications call the **IMAPIFormMgr::CreateForm** method to launch a form so as to create a new message based on the form. **CreateForm** launches the form by creating an instance of its form server for that form as described in the given form information object. If required, **CreateForm** will call **IMAPIFormMgr::PrepareForm** to download the form server code to the user's disk.

The *pfrminfoToActivate* parameter must point to a form information object for the form that has been correctly resolved.

After the form has been launched, the calling application must set up a message using the **IPersistMessage** interface and can optionally set up an **IMAPIViewContext** for the form. For more information on forms launching, see *MAPI Programmer's Guide*.

See Also

IMAPIViewContext : IUnknown interface, **IPersistMessage : IUnknown** interface

IMAPIFormMgr::IsInConflict

[New - Windows 95]

Determines whether the form can handle its own conflicts.

HRESULT IsInConflict(**ULONG** *ulMessageFlags*, **ULONG** *ulMessageStatus*, **LPCSTR** *szMessageClass*)

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 application- or service 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.

Return Values

S_OK
The form does not handle its own conflicts.

S_FALSE
The form handles its own conflicts, or the message is not in conflict.

Comments

Client applications call the **IMAPIFormMgr::IsInConflict** method to discover if a particular form does not handle its own 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. S_FALSE is returned if the form handles its own conflicts.

A form that does not handle its own conflicts must be launched using **IMAPIForm::LoadForm** and cannot reuse an existing form object.

See Also

IMAPIFormAdviseSink::OnActivateNext method, PR_MESSAGE_FLAGS property, PR_MSG_STATUS property

IMAPIFormMgr::LoadForm

[New - Windows 95]

Launches a form so as to open an existing message.

Syntax

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 a handle to the main window of the calling application, cast to a ULONG. The *ulUIParam* parameter is used in this case to display a progress user interface while the form is being loaded. The *ulUIParam* parameter is ignored unless the MAPI_DIALOG flag is set in the *ulFlags* parameter.

ulFlags
Input parameter containing a bitmask of flags used to control the loading of the form. The following flag 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.

MAPIFORM_EXACTMATCH
Indicates that only message class strings that are an exact match should be resolved.

lpszMessageClass
Input parameter pointing to the message class for the message to be loaded. If NULL is passed in this parameter, the message class is determined from the message pointed to by the *pmsg* parameter.

ulMessageStatus
Input parameter containing a bitmask of client application- or service provider-defined flags, copied from the PR_MSG_STATUS property of the message, that provides information on the state of the message. Must be set if *lpszMessageClass* is nonnull, otherwise this parameter is ignored.

ulMessageFlags
Input parameter containing a bitmask of flags, copied from the PR_MESSAGE_FLAGS property of the message, that indicates the current state of the message. Must be set if *lpszMessageClass* is nonnull; otherwise this parameter is ignored.

pFolderFocus
Input parameter pointing to the folder containing the message. *pFolderFocus* can be NULL if a folder doesn't exit.

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 requested interface 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 the *lpszMessageClass* parameter does not match the message class for any forms in the form registry.

Comments

Client applications call the **IMAPIFormMgr::LoadForm** method to launch a form so as to create a form object for an existing message. **LoadForm** launches the form object, loads the message into the form object, sets up the 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 the *lpszMessageClass* parameter, then the implementation will obtain the message class and the PR_MSG_STATUS and PR_MESSAGE_FLAGS properties. If a message class string is provided in *lpszMessageClass*, then the implementation should set the flags in *ulMessageStatus* and *ulMessageFlags*.

See Also

PR_MESSAGE_FLAGS property, PR_MSG_STATUS property

IMAPIFormMgr::OpenFormContainer

[New - Windows 95]

Opens an **IMAPIFormContainer** interface on a specific form container.

Syntax

HRESULT OpenFormContainer(**HFRMREG** *hfrmreg,* **LPUNKNOWN** *lpunk,*
LPMAPIFORMCONTAINER FAR * *lppfcnt*)

Parameters

hfrmreg
Input parameter containing an **HFRMREG** enumeration indicating the form registry to open. A **HFRMREG** enumeration is an enumeration specific to a form registry provider. Possible **HFRMREG** values include:

HFRMREG_DEFAULT
Indicates a convenient registry container.

HFRMREG_ENTERPRISE
Indicates the public folder registry.

HFRMREG_FOLDER
Indicates a folder registry container.

HFRMREG_PERSONAL
Indicates the container for the default message store.

HFRMREG_LOCAL
Indicates the local registry 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 the *lpUnk* parameter does not support the required interface.

Comments

Client applications call the **IMAPIFormMgr::OpenFormContainer** method to open an **IMAPIFormContainer** interface for a specific form container. This interface can then be used for installing and removing forms.

If the value in the *hfrmreg* parameter is HFRMREG_FOLDER case, which indicates a folder container, the interface identifier used in the *lpunk* parameter must be non-null and must allow **QueryInterface** calls to a **IMAPIFolder** interface.

A call to **OpenFormContainer** or the **MAPIOpenLocalFormContainer** function must be used to open the local registry container; **IMAPIFormMgr::SelectFormContainer** cannot be used to allow the user to select the local form container.

See Also

IMAPIFormContainer::InstallForm method, **IMAPIFormMgr::SelectFormContainer** method, **MAPIOpenLocalFormContainer** function

IMAPIFormMgr::PrepareForm

[New - Windows 95]

Downloads a form for launching.

Syntax

HRESULT PrepareForm(**ULONG** *ulUIParam*, **ULONG** *ulFlags,* **LPMAPIFORMINFO** *pfrmiInfo*)

Parameters

ulUIParam
: Input parameter containing a handle to the main window of the calling application, cast to a ULONG. The *ulUIParam* parameter is used in this case to display a progress user interface 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 used to control how the form is downloaded. The following flag 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.

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.

Comments

Client applications call the **IMAPIFormMgr::PrepareForm** method to download a form from the form registry for launching. Most clients do not need to call **PrepareForm** as both the **IMAPIFormMgr::CreateForm** and **IMAPIFormMgr::LoadForm** methods call **PrepareForm** if necessary.

PrepareForm can be used by clients to obtain the dynamic-link libraries (DLLs) and other files associated with a form in order to modify them. If the modified form is to be loaded back into its form registry, the form must be reinstalled.

See Also

IMAPIFormMgr::CreateForm method, **IMAPIFormMgr::LoadForm** method

IMAPIFormMgr::ResolveMessageClass

[New - Windows 95]

Resolves a message class to a form within the form registry and returns a form information object for that form.

Syntax

HRESULT ResolveMessageClass(**LPCSTR** *szMsgClass*, **ULONG** *ulFlags*, **LPMAPIFOLDER** *pFolderFocus*, **LPMAPIFORMINFO FAR** * *ppResult*)

Parameters

szMsgClass
Input parameter containing a string naming the message class to be resolved.

ulFlags
Input parameter containing a bitmask of flags used to control how the message class is resolved. The following flag can be set:

MAPIFORM_EXACTMATCH
Indicates that 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 *szMessageClass* parameter does not the match the message class for any forms in the form registry.

Comments

Client applications call the **IMAPIFormMgr::ResolveMessageClass** method to resolve a message class to a form within the form registry. The form information object returned in the *ppResult* parameter provides further access to the form's properties. Your application passes in the name of the message class to be resolved, for example, IPM.HelpDesk.Software. The MAPIFORM_EXACTMATCH flag can be passed in the *ulFlags* parameter to force the resolution to be exact, that is, prohibiting resolution to super classes of the message class. The *pFolderFocus* parameter should contain the folder containing the message being resolved; it can, however, be NULL, in which case the message-class resolution process does not search a folder registry container.

The order of the containers searched is dependent on the implementation of the form registry provider. The default registry searches the local container, the folder container for the passed-in folder, the personal container registry, and finally, the enterprise registry 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. Client applications should not assume that the class identifier exists in the OLE registry 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

IMAPIFormMgr::ResolveMultipleMessageClasses

[New - Windows 95]

Resolves group message classes to their forms within the form registry and returns an array of form information objects for those message classes.

Syntax

HRESULT ResolveMultipleMessageClasses(**LPSMESSAGECLASSARRAY***pMsgClasses*, **ULONG** *ulFlags*, **LPMAPIFOLDER** *pFolderFocus*, **LPSMAPIFORMINFOARRAY FAR *** *ppfrminfoarray*)

Parameters

pMsgClasses
Input parameter pointing to an array containing the message classes to be resolved.

ulFlags
Input parameter containing a bitmask of flags used to control how the message classes are resolved. The following flag can be set:

MAPIFORM_EXACTMATCH
Indicates that 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 to be resolved. The *pFolderFocus* parameter can be NULL.

ppfrminfoarray
Output parameter pointing to a variable where the pointer to an array of form information objects describing the forms and their locations is stored. If the client application passes NULL in the *pMsgClasses* parameter, the *ppfrminfoarray* contains information for all forms in the registry.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

Comments

Client applications call the **IMAPIFormMgr::ResolveMultipleMessageClasses** method to resolve a group of message classes to forms within the form registry. The array of form information objects returned in the *ppforminfoarray* parameter provides further access to each of the forms' properties. If a message class cannot be resolved to a form, NULL is returned for that message class. So although the method returns S_OK, applications should not assume that all message classes have been successfully resolved; applications should check the values in the returned array.

Your application passes in an array of message class names to be resolved. The MAPIFORM_EXACTMATCH flag can be passed in the *ulFlags* parameter to force the resolution to be exact, that is, preventing resolution to super classes of the message class.

Message class names are always ANSI strings, never Unicode.

See Also

IMAPIFormMgr::ResolveMessageClass method

IMAPIFormMgr::SelectForm

[New - Windows 95]

Presents a dialog box that enables the user to select a form, then returns a form information object describing that form.

Syntax

HRESULT SelectForm(ULONG *ulUIParam*, **ULONG** *ulFlags*, **LPCTSTR** *pszTitle*, **LPMAPIFOLDER** *pfld*, **LPMAPIFORMINFO FAR *** *ppfrminfoReturned*)

Parameters

ulUIParam
Input parameter containing a handle to the main window of the calling application, cast to a ULONG. The *ulUIParam* parameter is used in this case to display the dialog box.

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 8-bit format.

pszTitle
Input parameter pointing to a string containing the caption of the dialog box. If the *pszTitle* parameter is NULL, the form registry 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 enterprise 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 dialog box.

Comments

Client applications 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 client passes the MAPI_UNICODE flag in the *ulFlags* parameter, all strings are Unicode. Form registry providers that do not support Unicode strings should return MAPI_E_BAD_CHARWIDTH if MAPI_UNICODE is passed.

IMAPIFormMgr::SelectMultipleForms

[New - Windows 95]

Presents a dialog box that enables the user to select multiple forms, then returns an array of form information objects describing the forms.

Syntax

HRESULT SelectMultipleForms(**ULONG** *ulUIParam*, **ULONG** *ulFlags*, **LPCTSTR** *pszTitle*, **LPMAPIFOLDER** *pfld*, **LPMAPIFORMINFOARRAY** *pfrminfoarray*, **LPMAPIFORMINFOARRAY FAR *** *ppfrminfoarray*)

Parameters

ulUIParam
Input parameter containing a handle to the main window of the calling application, cast to a ULONG. The *ulUIParam* parameter is used in this case to display the dialog box.

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 8-bit format.

pszTitle
Input parameter pointing to a string containing the caption of the dialog box. If the *pszTitle* parameter is NULL, the form registry provider 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 enterprise form container.

pfrminfoarray
Input parameter pointing to an array of form information objects which are preselected for the user.

ppfrminfoarray
Output parameter pointing to a variable where the pointer to the returned array of form information objects describing the selected forms 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 dialog box.

Comments

Client applications 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 independent of their hiddenness. If a client passes the MAPI_UNICODE flag in the *ulFlags* parameter, all strings are Unicode. Form registry providers that do not support Unicode strings should return MAPI_E_BAD_CHARWIDTH if MAPI_UNICODE is passed.

IMAPIFormMgr::SelectFormContainer

[New - Windows 95]

Presents a dialog box that enables the user to select a form registry container and return an interface for the container object the user selected.

Syntax

HRESULT SelectFormContainer(**ULONG** *ulUIParam*, **ULONG** *ulFlags*, **LPMAPIFORMCONTAINER FAR *** *lppfcnt*)

Parameters

ulUIParam
: Input parameter containing a handle to the main window of the calling application, cast to a ULONG. The *ulUIParam* parameter is used in this case to display the dialog box.

ulFlags
: Input parameter containing a bitmask of flags used to control how the registry is selected. The following flags can be set:

  MAPIFORM_SELECT_ALL_REGISTRIES
  : Indicates selection can be made from all for registry containers. This is the default action.

  MAPIFORM_SELECT_FOLDER_REGISTRIES_ONLY
  : Indicates selection can be made only from folder registry containers.

  MAPIFORM_SELECT_NON_FOLDER_REGISTRIES_ONLY
  : Indicates selection can be made only from form registry containers not associated with folders.

lppfcnt
: Output parameter pointing to a variable where the pointer to the returned interface for the registry object the user selected is stored.

Return Values

S_OK
: The call succeeded and has returned the expected value or values.

Comments

The **IMAPIFormMgr::SelectFormContainer** method is commonly used to select a form container into which a form is installed. **SelectFormContainer** cannot be used to select the local application form registry, which is HFRMREG_LOCAL.

IMAPIMessageSite : IUnknown

[New - Windows 95]

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 the form can use to launch another form.
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 the message site object about the capabilities of the message site on the message.

IMAPIMessageSite::CopyMessage

[New - Windows 95]

Copies the current message to a folder.

Syntax

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.

Comments

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 on the new message is returned to the form.

A typical implementation of **CopyMessage** does the following:

1. Creates the new message to be copied to.
2. Calls **IPersistMessage::Save**(pMsgNew, FALSE).
3. Calls **IPersistMessage::SaveCompleted**(NULL).
4. Calls **SaveChanges** on the new message.

See Also

IMAPIProp::SaveChanges method, **IPersistMessage::Save** method, **IPersistMessage::SaveCompleted** method

IMAPIMessageSite::DeleteMessage

[New - Windows 95]

Deletes the current message.

Syntax

HRESULT DeleteMessage(**LPMAPIVIEWCONTEXT** *pViewContext*, **LPCRECT** *prcPosRect*)

Parameters

pViewContext
  Input parameter pointing to a view context object.

prcPosRect
  Input parameter pointing to the **RECT** structure containing the current form's window size and position. This window rectangle is used by the next form displayed.

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.

Comments

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, the implementation is required to call **IMAPIViewContext::ActivateNext** passing the VCDIR_DELETE flag prior to performing the actual deletion. If a form viewer's implementation of **DeleteMessage** moves the message, it must save changes to the message if it has been modified.

Following the return of **DeleteMessage**, form objects must check for a new message and then dismiss themselves if none exists. Form servers wanting to determine whether the message is deleted or whether it is being moved to a deleted items folder can check for the DELETE_IS_MOVE flag on the return from a call to the **IMAPIMessageSite::GetSiteStatus** method.

A typical implementation of **DeleteMessage** does the following:

1. If moving the message, call **IPersistMessage::Save**(NULL,TRUE).
2. Call **IMAPIViewContext::ActivateNext**(VCDIR_DELETE).
3. If the **ActivateNext** call failed, return. If **ActivateNext** returned S_FALSE, call **IPersistMessage::HandsOffMessage**.
4. Delete or move the message.

See Also

IMAPIMessageSite::GetSiteStatus method, **IMAPIViewContext::ActivateNext** method, **IPersistMessage::HandsOffMessage** method, **IPersistMessage::Save** method

IMAPIMessageSite::GetFolder

[New - Windows 95]

Returns the folder in which the current message was created or opened, if one exists.

Syntax

HRESULT GetFolder(**LPMAIFOLDER 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
There was no folder for the message.

IMAPIMessageSite::GetFormManager

[New - Windows 95]

Returns a form manager interface, which the form can use to launch another form.

Syntax

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

[New - Windows 95]

Returns the current message.

Syntax

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 form.

Comments

Form objects call the **IMAPIMessageSite::GetMessage** method to obtain a message interface for the current message. This is the same message as was passed in with either the **IPersistMessage::InitNew**, **IPersistMessage::Load,** or the **IPersistMessage::SaveCompleted** methods.

GetMessage returns S_FALSE if no message currently exists. This can occur after calls to **IPersistMessage::HandsOffMessage** or before the next call to **IPersistMessage::Load** or **IPersistMessage::SaveCompleted** has been made.

See Also

IPersistMessage : IUnknown interface

IMAPIMessageSite::GetSession

[New - Windows 95]

Returns the MAPI session in which the current message was created or opened.

Syntax

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

[New - Windows 95]

Returns information from the message site object about the capabilities of the message site on the message.

Syntax

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 delete operation on the message is really a move to a deleted items folder.

VCSTATUS_MOVE
Indicates the message can be moved.

VCSTATUS_NEWMESSAGE
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.

Comments

Form objects call the **IMAPIMessageSite::GetSiteStatus** method to obtain the message site's capabilities on the message. The flags returned in the *lpulStatus* parameter provide information on the message site; typically they are used to enable or disable menu commands based on the capabilities of the message site's 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 rechecked. Some message sites, particularly read-only sites, cannot permit messages to be saved or deleted.

See Also

IPersistMessage::Load method, **IPersistMessage::SaveCompleted** method

IMAPIMessageSite::GetStore

[New - Windows 95]

Returns the message store containing the current message, if such a store exists.

Syntax

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

[New - Windows 95]

Moves the current message to a folder.

Syntax

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 the **RECT** structure used by the form's window. 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.

Comments

Form objects call the **IMAPIMessageSite::MoveMessage** method to move the current message to a new folder. A form viewer's implementation of **MoveMessage** is required to call **IMAPIViewContext::ActivateNext**, passing the VCDIR_MOVE flag prior to actually moving the form to a new folder. Following the return of **MoveMessage**, form objects must check for a current message and then dismiss themselves if none exists.

The **RECT** structure used by the form's window can be obtained using the Windows **GetWindowRect** function.

See Also

IMAPIViewContext::ActivateNext method

IMAPIMessageSite::NewMessage

[New - Windows 95]

Creates a new message.

Syntax

HRESULT NewMessage(ULONG *fComposeInFolder*, **LPMAPIFOLDER** *pFolderFocus*, **LPPERSISTMESSAGE** *pPersistMessage*, **LPMESSAGE FAR *** *ppMessage*, **LPMAPIMESSAGESITE FAR *** *ppMessageSite*, **LPMAPIVIEWCONTEXT FAR *** *ppViewContext*)

Parameters

fComposeInFolder
Input parameter indicating in which folder the message should be composed. If FALSE, *pFolderFocus* is ignored and the form viewer implementation can compose the message in any folder. If TRUE, and NULL is passed in the *pFolderFocus* parameter, the message is composed in the current folder. If TRUE and a non-null value is passed in *pFolderFocus*, the message is composed in the folder pointed to by the *pFolderFocus* parameter.

pFolderFocus
Input parameter pointing to the folder where the new message is created.

pPersistMessage
Input parameter pointing to the **IPersistMessage** object for the new form.

ppMessage
Output parameter pointing to a pointer to the new message.

ppMessageSite
Output parameter pointing to a pointer to the message site object.

ppViewContext
Output parameter pointing to a pointer to a view context appropriate for passing to a new form with the new message. NULL can be passed in the *ppViewContext* parameter if the form implements its own view context.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

Comments

Form objects call the **IMAPIMessageSite::NewMessage** method to enable a form 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 pass the message site directly to the new form object.

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 expected, NULL should be passed in the *ppViewContext* parameter.

See Also

IMAPIViewContext : IUnknown interface

IMAPIMessageSite::SaveMessage

[New - Windows 95]

Requests that the current message be saved.

Syntax

HRESULT SaveMessage()

Parameters

None.

Return Values

S_OK
    The call succeeded and has returned the expected value or values.

Comments

Form objects call the **IMAPIMessageSite::SaveMessage** method to request that a message be saved. After being saved, the message generates the normal **IPersistMessage** object behavior.

IMAPIMessageSite::SubmitMessage

[New - Windows 95]

Requests that the current message be submitted to the MAPI spooler.

Syntax

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 right away.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

Comments

Form objects 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**, form objects must check for a current message and then dismiss themselves if none exists.

See Also

IPersistMessage::HandsOffMessage method

IMAPIProgress : IUnknown

[New - Windows 95]

The **IMAPIProgress** interface is used by providers to provide client applications with information to update progress-information user interfaces, which indicate the progress towards 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:	Providers

Vtable Order

Progress	Updates the progress-information user interface with status on the progress made towards completion of the operation in question.
GetFlags	Returns the flag settings from the progress object for the level of operation on which progress information is calculated.
GetMax	Returns the maximum setting for the whole operation from the progress object.
GetMin	Returns the minimum setting for the whole operation from the progress object.
SetLimits	Sets the minimum value of items operated on, the maximum value of items operated on, and the flags that control how progress information is calculated for the operation in question.

IMAPIProgress::GetFlags

[New - Windows 95]

Returns the flag settings from the progress object for the level of operation on which progress information is calculated.

Syntax

HRESULT GetFlags(**ULONG FAR *** *lpulFlags*)

Parameters

lpulFlags
: Output parameter containing a bitmask of flags used to control 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* (the item being operated on) and *ulTotal* (total items to operate on) parameters to increment progress being made on the operation. For example, if you were copying a folder that contained 15 subfolders and the MAPI_TOP_LEVEL flag is set in the *lpulFlags* parameter, progress will increment for each subfolder copied: 1 of 15, 2 of 15, 3 of 15, and so on.

Return Values

S_OK
: The call succeeded and has returned the expected value or values.

Comments

Use the **IMAPIProgress::GetFlags** method to get the flag settings from the progress object for the level of operation on which progress information is calculated. This value, retrieved from **GetFlag**'s *lpulFlags* parameter, is set using the **IMAPIProgress::SetLimits** method. Your application uses this value to control the level of operation for which progress information is given. For instance, if you copy a folder containing 15 subfolders and the MAPI_TOP_LEVEL flag is set in *lpulFlags*, progress is calculated in terms of subfolders copied: 1 of 15, 2 of 15, 3 of 15, and so on. If the MAPI_TOP_LEVEL flag is not set in *lpulFlags*, the copying of the 15 subfolders is considered a single operation and progress is calculated as a percentage of the total operation: 20 percent, 40 percent, 60 percent, and so on.

See Also

IMAPIProgress::SetLimits method

IMAPIProgress::GetMax

[New - Windows 95]

Returns the maximum setting for the whole operation from the progress object.

Syntax

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.

Comments

Use the **IMAPIProgress::GetMax** method to obtain the maximum setting from the progress object. The value in **IMAPIProgress::GetMax**'s *lpulMax* parameter is set using the **IMAPIProgress::SetLimits** method. The values in *lpulMax* and the *lpulMin* parameter obtained through the **IMAPIProgress::GetMin** method are used by the **IMAPIProgress::Progress** method to calculate the number that is used as the value in **Progress**'s *ulValue* parameter. This calculation is done by comparing *lpulMin* and *lpulMax* and expressing the comparison as a percentage.

See Also

IMAPIProgress::GetMin method, **IMAPIProgress::Progress** method, **IMAPIProgress::SetLimits** method

IMAPIProgress::GetMin

[New - Windows 95]

Returns the minimum setting for the whole operation from the progress object.

Syntax

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.

Comments

Use the **IMAPIProgress::GetMin** method to obtain the minimum setting from the progress object. The value in **IMAPIProgress::GetMin**'s *lpulMin* parameter is set using the **IMAPIProgress::SetLimits** method. The values in *lpulMin* and the *lpulMax* parameter obtained through the **IMAPIProgress::GetMax** method are used by the **IMAPIProgress::Progress** method to calculate the number that is used as the value in **Progress**'s *ulValue* parameter. This calculation is done by comparing *lpulMin* and *lpulMax* and expressing the comparison as a percentage.

See Also

IMAPIProgress::GetMin method, **IMAPIProgress::Progress** method, **IMAPIProgress::SetLimits** method

IMAPIProgress::Progress

[New - Windows 95]

Updates the progress-information user interface with status on the progress made towards completion of the operation.

Syntax

HRESULT Progress(**ULONG** *ulValue*, **ULONG** *ulCount*, **ULONG** *ulTotal*)

Parameters

ulValue
Input parameter containing a number indicating the percentage, relative to *ulCount* and *ulTotal* if the MAPI_TOP_LEVEL flag is set in **IMAPIProgress::SetLimits**, or that expresses the comparison of the values in the *lpulMin* and *lpulMax* parameters of the **SetLimits** method if the MAPI_TOP_LEVEL flag is not set on **SetLimits**.

ulCount
Input parameter containing the number of the item currently being operated on.

ulTotal
Input parameter containing the total number of items to be operated on.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

Comments

Service providers use the **IMAPIProgress::Progress** method to update the progress-information user interface for the progress object passed in by a client application. **Progress**'s *ulCount* parameter is the number of the current item being operated on; its *ulTotal* parameter is the total number of items to be operated on. For example, if 10 folders are being copied and 4 of them have already been copied, then 5 should be sent in the *ulCount* parameter and 10 should be sent in the *ulTotal* parameter.

If the MAPI_TOP_LEVEL flag has been set in the *lpulFlags* parameter of **IMAPIProgress::SetLimits**, then *ulValue* is expressed as a percentage based on the values in the *ulCount* and the *ulTotal* parameters. For example, if *ulCount* is 20 and *ulTotal* is 50, then *ulValue* is 40 and indicates 40 percent completion. If MAPI_TOP_LEVEL is not set, then *ulValue* is expressed as a percentage based on the *lpulMax* and the *lpulMin* parameters of **SetLimits**. For example, if *lpulMax* is 100 and *lpulMin* is 0, *ulValue* is 100 and indicates 100 percent.

If your application is copying all messages within a single folder, then the value in the *ulTotal* parameter should indicate the total number of messages being copied. If your application is copying a folder, then the value in *ulTotal* should be the number of subfolders within the folder, not the number of subfolders plus the number of messages. 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

[New - Windows 95]

Sets the minimum value of items operated on, the maximum value of items operated on, and the flags that control how progress information is calculated for the operation in question.

Syntax

HRESULT SetLimits(**LPULONG** *lpulMin*, **LPULONG** *lpulMax*, **LPULONG** *lpulFlags*)

Parameters

lpulMin
Input parameter pointing to a variable containing the minimum number of items in the operation.

lpulMax
Input parameter pointing to a variable containing the maximum number of items in the operation.

lpulFlags
Input parameter containing a bitmask of flags used to control 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* (the item being operated on) and *ulTotal* (total items to operate on) parameters to increment progress being made on the operation. For example, if you were copying a folder that contained 15 subfolders and the MAPI_TOP_LEVEL flag is set in the *lpulFlags* parameter, progress will increment for each subfolder copied: 1 of 15, 2 of 15, 3 of 15, and so on.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

Comments

Client applications use the **IMAPIProgress::SetLimits** method to establish the values that control how operation progress is calculated for the progress-information user interface. Service providers should not call **SetLimits** to change the values if they are passed a progress object by a client application; they should use the values passed to them.

The *lpulFlags* parameter is used to control the level of operation for which progress information is given. For instance, if you copy a folder containing 15 subfolders and the MAPI_TOP_LEVEL flag is set in *lpulFlags*, progress is calculated in terms of subfolders copied: 1 of 15, 2 of 15, 3 of 15, and so on. If the MAPI_TOP_LEVEL flag is not set in *lpulFlags*, the copying of the 15 subfolders is considered a single operation and progress is calculated as a percentage of the total operation: 20 percent, 40 percent, 60 percent, and so on.

The values in the *lpulMin* and *lpulMax* parameters are used by the **IMAPIProgress::Progress** method to calculate the number that is used as the value in **Progress**'s *ulValue* parameter if the MAPI_TOP_LEVEL flag is not passed in the lpulFlags parameter.. This calculation is done by comparing *lpulMin* and *lpulMax* and expressing the comparison as a percentage.

See Also

IMAPIProgress::Progress method

IMAPIProp : IUnknown

[New - Windows 95]

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:	All service providers
Called by:	All applications

Vtable Order

GetLastError	Returns 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 the properties of an object.
OpenProperty	Opens an interface on a property of an object.
SetProps	Sets the property value of one or more properties of an object.
DeleteProps	Deletes the given list of properties.
CopyTo	Copies or moves all properties of a source object to destination object.
CopyProps	Copies 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

[New - Windows 95]

Copies or moves a selected set of properties from the source object to a given destination object. (The source object is the object on which the call to the **IMAPIProp::CopyProps** method is made.)

Syntax

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 be copied or moved from this object to the destination object. The *lpIncludeProps* parameter value cannot be NULL.

ulUIParam
Input parameter containing the handle of the window the dialog box is modal to. In this case, it is used to indicate the dialog box where progress information is displayed.

lpProgress
Input parameter pointing to a progress object that contains client- or provider-supplied progress information. If NULL is sent in the *lpProgress* parameter, 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 interface identifier for the interface of the destination object to which properties are being copied as indicated in the *lpDestObj* parameter. *lpInterface* must not be NULL.

lpDestObj
Input parameter pointing to the open destination object.

ulFlags
Input parameter containing a bitmask of flags used to control how the copy or move operation is performed. The following flags can be set:

MAPI_DECLINE_OK
Used by MAPI to limit recursion within MAPI's **CopyProps** implementation. Informs the provider that if it chooses not to implement **IMAPIProp::CopyProps**, that it should immediately return MAPI_E_DECLINE_COPY.

MAPI_DIALOG
Displays a dialog box 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 a value of 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_DECLINE_COPY
: Indicates that the provider has chosen not to implement the copy operation.

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 object might be partially modified.

MAPI_E_INTERFACE_NOT_SUPPORTED
: An appropriate interface cannot be obtained.

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.

The following values can be returned in the **SPropProblemArray** structure, but not as a return value 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 is not a severe error; allow the process to continue.

MAPI_E_INVALID_TYPE
: The property type is invalid.

MAPI_E_UNEXPECTED_TYPE
: The type of the property value is not the type expected by the calling application.

Comments

Use the **IMAPIProp::CopyProps** method to copy or move to the destination object those properties designated in the **SPropTagArray** structure passed in the *lpIncludeProps* parameter that are present in the source object. When copying properties between like objects, for example between two message objects, the interface identifiers and the object types must be the same for both the source and the destination object. 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 by copied or moved data 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, respectively. For folder objects, the PR_CONTAINER_HIERARCHY, PR_CONTAINER_CONTENTS, and PR_FOLDER_ASSOCIATED_CONTENTS properties can be included in the *lpIncludeProps* **SPropTagArray** structure to permit copying or moving of subfolders, messages, or associated objects from a source folder to a destination object. If subfolders are copied or moved, they are copied or moved in their entirety, regardless of the use of properties indicated by the **SPropTagArray** structure in the source object itself.

If the source object directly or indirectly contains the destination object, the overall call fails and returns the value MAPI_E_FOLDER_CYCLE. Some implementations, however, perform significant work before discovering this error and leave the source and destination objects partially modified, so applications should try to avoid such calls. If the same pointer is used for both the source and destination objects, MAPI_E_NO_ACCESS is returned.

The interface of the destination object, indicated in the *lpInterface* parameter, is usually exactly the same interface as that for the source object. If *lpInterface* is set to NULL, then the value MAPI_E_INVALID_PARAMETER is returned for **CopyProps**. If an acceptable interface is given in the *lpInterface* parameter but an invalid pointer is passed in the *lpDestObj* parameter, the results are unpredictable, with the most likely result being that the calling application stops.

If the MAPI_DIALOG flag is not set in the *ulFlags* parameter, then **CopyProps** ignores the *ulUIParam* and *lpProgress* parameters and no progress-information user interface is provided. If a client application sets the MAPI_DIALOG flag in the *ulFlags* parameter and passes a value of NULL in the *lpProgress* parameter, then the provider is responsible for generating a progress-information user interface. If a client application sets the MAPI_DIALOG flag in the *ulFlags* parameter and passes a progress object in the *lpProgress* parameter, 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 of the selected properties, the value S_OK is returned and an **SPropProblemArray** structure is returned 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, exactly 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, the PR_MESSAGE_RECIPIENTS or PR_MESSAGE_ATTACHMENTS property is returned in the **SPropProblemArray** structure.

The value returned in the *lppProblems* parameter 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 **IMAPIProp::GetLastError** to get the **MAPIERROR** structure for the error.

The calling application must free the returned **SPropProblemArray** structure by calling the **MAPIFreeBuffer** function, but this should only be done if **CopyProps** returns with S_OK.

See Also

IMAPIFolder::CopyMessages method, **IMAPIProp::CopyTo** method, **IMAPIProp::GetLastError** method, **SPropProblemArray** structure, **SPropTagArray** structure

IMAPIProp::CopyTo

[New - Windows 95]

Copies or moves all properties of the source object to a destination object, except for a given set of properties that are excluded from the copy or move operation. (The source object is the object on which the call to the **IMAPIProp::CopyTo** method is made.)

Syntax

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 be excluded when copying or moving properties.

rgiidExclude
: Input parameter containing an array of interface identifiers indicating interfaces that should not be used when copying or moving supplemental information to the destination object.

lpExcludeProps
: Input parameter pointing to the **SPropTagArray** structure containing the property identifiers of the properties that should not be copied or moved to the destination object. Passing a value of NULL in the *lpExcludeProps* parameter indicates all properties are to be copied or moved. Passing zero in the **cValues** member of the *lpExcludeProps* **SPropTagArray** structure results in the value MAPI_E_INVALID_PARAMETER being returned.

ulUIParam
: Input parameter containing the handle of the window the dialog box is modal to. In this case it is used to indicate the dialog box where progress information is displayed.

lpProgress
: Input parameter pointing to a progress object that contains client- or provider-supplied progress information. If NULL is sent in the *lpProgress* parameter, 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 interface identifier for the interface of the destination object to which properties are being copied.

lpDestObj
: Input parameter pointing to the open destination object.

ulFlags
: Input parameter containing a bitmask of flags used to control how the copy or move operation is to be performed. The following flags can be set:

  MAPI_DECLINE_OK
  : Used by MAPI to limit recursion within MAPI's **CopyTo** implementation. Informs the provider that if it chooses not to implement **IMAPIProp::CopyTo**, that it should immediately return MAPI_E_DECLINE_COPY.

  MAPI_DIALOG
  : Displays a dialog box 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 a value of 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_DECLINE_COPY
Indicates that the provider has chosen not to implement the copy operation.

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 object might be partially modified.

MAPI_E_INTERFACE_NOT_SUPPORTED
An appropriate interface cannot be obtained.

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.

The following values can be returned in the **SPropProblemArray** structure, but not as a return value 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 is not a severe error; allow the process to continue.

MAPI_E_INVALID_TYPE
The property type is invalid.

MAPI_E_UNEXPECTED_TYPE
The type of the property value is not the type expected by the calling application.

Comments

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 by copied or moved data 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 being copied or moved to the destination object. For example, passing in a value of 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, respectively. For folder objects, the PR_CONTAINER_HIERARCHY, PR_CONTAINER_CONTENTS, and PR_FOLDER_ASSOCIATED_CONTENTS properties can be included in the *lpExcludeProps* **SPropTagArray** structure to prevent copying or moving of subfolders, messages, or associated objects from the source folder to the destination object. If subfolders are copied or moved, they are copied or moved in their entirety, regardless of the use of properties indicated by the **SPropTagArray** structure in the source object itself.

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.

If the source object directly or indirectly contains the destination object, the overall call fails and returns the value MAPI_E_FOLDER_CYCLE. Some implementations, however, perform significant work before discovering this error and leave the source and destination objects partially modified, so applications should try to avoid such calls. If the same pointer is used for both the source and destination objects, MAPI_E_NO_ACCESS is returned.

The interface of the destination object, indicated in the *lpInterface* parameter, is usually exactly the same interface as that for the source object. If *lpInterface* is set to NULL, then the value MAPI_E_INVALID_PARAMETER is returned for **CopyTo**. If an acceptable interface is given in the *lpInterface* parameter but an invalid pointer is passed in the *lpDestObj* parameter, the results are unpredictable, with the most likely result being that the calling application stops.

Some objects contain supplemental information beyond that, which can be accessed with the interface pointer in the *lpInterface* parameter. To copy or move such information, a provider's **IMAPIProp::CopyTo** implementation first uses **IUnknown::QueryInterface** on the destination object to see if it can accept the extra data. Conversely, if the calling application is aware of supplemental information and requires that **CopyTo** not copy or move it, the application can specify in the array passed in the *rgiidExclude* parameter interface identifiers for the properties that **CopyTo** should not copy or move. For example, if the application wants to copy messages, but not embedded objects within the messages, it can pass IID_IMessage in the exclude array. **CopyTo** ignores any interfaces listed in *rgiidExclude* that 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 **IMAPIProp** also excludes **IMAPIFolder**, **IMessage**, **IAttach**, 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 the *ulFlags* parameter, then **CopyTo** ignores the *ulUIParam* and *lpProgress* parameters and no progress-information user interface is provided. If a client application sets the MAPI_DIALOG flag in the *ulFlags* parameter and passes a value of NULL in the *lpProgress* parameter, then the provider is responsible for generating a progress-information user interface. If a client application sets the MAPI_DIALOG flag in the *ulFlags* parameter and passes a progress object in the *lpProgress* parameter, 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 of the selected properties, the value S_OK is returned and an **SPropProblemArray** structure is returned 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, exactly 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, the PR_MESSAGE_RECIPIENTS or PR_MESSAGE_ATTACHMENTS property is returned in the **SPropProblemArray** structure.

The value returned in the *lppProblems* parameter 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 **IMAPIProp::GetLastError** to get the **MAPIERROR** structure for the error.

If an error occurs on the **CopyTo** call, do not use of free the SPropProblemArray structure. Applications should ignore the ulIndex member in property problem sets returned by **CopyTo**.

The calling application must free the returned **SPropProblemArray** structure by calling the **MAPIFreeBuffer** function, but this should only be done if **CopyTo** returns with S_OK.

See Also

IMAPIFolder::CopyMessages method, **IMAPIProp::GetLastError** method, **SPropProblemArray** structure, **SPropTagArray** structure

IMAPIProp::DeleteProps

[New - Windows 95]

Deletes the given list of properties.

Syntax

HRESULT DeleteProps(**LPSPropTagArray** *lpPropTagArray*, **LPSPropProblemArray FAR *** *lppProblems*)

Parameters

lpPropTagArray
Input parameter pointing to an **SPropTagArray** structure containing the properties to be deleted. 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 the value MAPI_E_INVALID_PARAMETER being returned.

lppProblems
Output parameter pointing to a variable where the pointer to an **SPropProblemArray** structure is stored. If a value of 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 to modify a read-only object or an attempt to access an object for which the user has insufficient permissions.

Comments

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 of the selected properties, the value S_OK is returned and an **SPropProblemArray** structure is returned in the *lppProblems* parameter. 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, exactly which properties were not successfully deleted can be determined from the **SPropProblemArray** structure. 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 the *lppProblems* parameter 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 **IMAPIProp::GetLastError** to get the **MAPIERROR** structure for the error.

The calling application must free the returned **SPropProblemArray** structure by calling the **MAPIFreeBuffer** function, but this should only be done if **DeleteProps** returns with S_OK.

See Also

IMAPIProp::GetProps method, **IMAPIProp::SaveChanges** method, **MAPIFreeBuffer** function, **SPropTagArray** structure

IMAPIProp::GetIDsFromNames

[New - Windows 95]

Provides property identifiers, given a list of property names.

Syntax

HRESULT GetIDsFromNames(ULONG *cPropNames*, **LPMAPINAMEID FAR** * *lppPropNames*, **ULONG** *ulFlags*, **LPSPropTagArray FAR** * *lppPropTags*)

Parameters

cPropNames
Input parameter containing the number of pointers to **MAPNAMEID** structures returned in the *lppPropNames* parameter. If the *lppPropNames* parameter has a value of NULL, the value in the *cPropNames* parameter must be zero.

lppPropNames
Input parameter pointing to an array of pointers to **MAPINAMEID** structures containing names of properties. A value of NULL requests property identifiers for all property names about which the object has information. If MAPI_CREATE is set in *ulFlags*, *lppPropNames* 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* that is 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 the pointer to the array of existing or newly assigned property identifiers is stored. The property types 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
Indicates that the operation cannot be performed because too many properties tags are being 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. Use the HR_FAILED macro to test for this warning, but the call should be handled as a successful return.

Comments

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 the *lppPropNames* parameter. **GetIDsFromNames** returns the type portion of each tag as PT_UNSPECIFIED; call **IMAPIProp::SetProps** to set them.

Message store providers wanting to extend property sets typically implement the **GetIDsFromNames** and **IMAPIProp::GetNamesFromIDs** methods for message and folder objects. Other interfaces derived from **IMAPIProp** typically return the value MAPI_E_NO_SUPPORT for calls to these methods.

Only identifiers in the range of 0x8000 to 0xFFFE use name-to-identifier mapping. MAPI expects 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 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.

Applications 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 the PR_MAPPING_SIGNATURE property, in which case the application can skip this step.

If a name set with **GetIDsFromNames** does not have an identifier, **GetIDsFromNames** returns the value 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 sent with **GetIDsFromNames** does not have an identifier and the MAPI_CREATE flag was set in the *ulFlags* parameter, the **GetIDsFromNames** call allocates an identifier for the name and internally registers the identifier in the name-to-identifier mapping table. When the value in the *lppPropNames* parameter is NULL, MAPI returns all identifiers defined for the entire PR_MAPPING_SIGNATURE (the value in the *cPropNames* parameter must also be zero if NULL is passed in the *lppPropNames* parameter). If the number of defined identifiers in the store exceeds the implementation limit, MAPI_E_TOO_BIG is returned and client applications should query by identifier.

See Also

IMAPIProp::GetNamesFromIDs method, **MAPINAMEID** structure, **MAPIUID** structure

IMAPIProp::GetLastError

[New - Windows 95]

Returns a **MAPIERROR** structure containing information about the last error that occurred for the object.

Syntax

HRESULT GetLastError(**HRESULT** *hResult*, **ULONG** *ulFlags*, **LPMAPIERROR FAR** * *lppMAPIError*)

Parameters

hResult
: Input parameter containing the result returned for the last call on the session object that returned an error.

ulFlags
: Input parameter containing a bitmask of flags controlling the type of the returned strings. The following flag can be set:

  MAPI_UNICODE
  : Indicates the strings returned 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 8-bit 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.

Comments

Use the **IMAPIProp::GetLastError** method to retrieve information to display as a message to the user regarding the last error returned from a method call on the session object.

To release all the memory allocated by MAPI, client applications need only call the **MAPIFreeBuffer** function for the **MAPIERROR** structure.

The return value from **GetLastError** must be S_OK for the application to make use of the **MAPIERROR** structure. Even if the return value is S_OK, it is still possible that a **MAPIERROR** structure won't be returned. If the implementation cannot determine what the last error was, or if a MAPIERROR structure is not available for that error, a pointer to NULL is returned in the *lppMAPIError* parameter instead.

See Also

MAPIERROR structure, **MAPIFreeBuffer** function

IMAPIProp::GetNamesFromIDs

[New - Windows 95]

Provides property identifiers, given a list of property names.

Syntax

HRESULT GetNamesFromIDs(**LPSPropTagArray FAR** * *lppPropTags*, **LPGUID** *lpPropSetGuid*, **ULONG** *ulFlags*, **ULONG FAR** * *lpcPropNames*, **LPMAPINAMEID FAR * FAR** * *lpppPropNames*)

Parameters

lppPropTags
Output parameter pointing to a variable where the pointer to an **SPropTagArray** structure of property tags, including the identifiers for which names are needed, is stored. Passing zero in the **cValues** member of the **SPropTagArray** structure results in the value MAPI_E_INVALID_PARAMETER being returned. *lppPropTags* can be a pointer to NULL, in which case all of the names are returned.

lpPropSetGuid
Input parameter pointing to a globally unique identifier (GUID) for the property set. To get all of the names, pass in the value PS_PUBLIC_STRINGS and NULL for the *lppPropNames* parameter.

ulFlags
Input parameter containing a bitmask of flags used to control how names are mapped. The following flags can be used:

MAPI_NO_IDS
Indicates that identifiers are not returned.

MAPI_NO_STRINGS
Indicates that strings are not returned.

lpcPropNames
Output parameter pointing to a variable containing the number of strings in the *lpppPropNames* parameter.

lpppPropNames
Input or output parameter pointing to a variable pointer to an array of pointers to **MAPINAMEID** structures containing names of properties. It is an input parameter if it points to a value. If it points to NULL, all of the GUIDs for the property set are returned.

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. Use the HR_FAILED macro to test for this warning, but the call should be handled as a successful return.

Comments

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, there are some applications that also need to associate names in Unicode string format with some properties in their objects.

GetNamesFromIDs ignores the property type in each property tag sent in the *lppPropTags* parameter

– only the property identifier is significant. If a specified identifier does not have a Unicode name, **GetNamesFromIDs** returns a NULL in that identifier's place in the structure returned in the *lpppPropNames* parameter and also returns the value MAPI_W_ERRORS_RETURNED. If the value in the *lppPropTags* parameter 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.

There are two distinguished GUIDs which identify interesting property sets: PS_MAPI and PS_PUBLIC_STRINGS.

If you want to use a MAPI folder as a container for string properties, use the PS_PUBLIC_STRINGS as the property set for storing the ad-hoc document property strings. Applications that want to get the public names within the container call **GetNamesFromIDs** with a pointer to a NULL property tag array and a GUID of PS_PUBLIC_STRINGS and set the MAPI_NO_IDS flag in the *ulFlags* parameter.

Applications that want to get 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.

See Also

IMAPIProp::GetIDsFromNames method, **MAPIFreeBuffer** function, **MAPINAMEID** structure, **SPropTagArray** structure

IMAPIProp::GetPropList

[New - Windows 95]

Returns a list of all the properties of an object.

Syntax

HRESULT GetPropList(**ULONG** *ulFlags*, **LPSPropTagArray FAR** * *lppPropTagArray*)

Parameters

ulFlags
: Input parameter containing a bitmask of flags controlling the format of 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 8-bit 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.

Comments

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 from the list of returned properties for which the calling application does not have access.

The calling application must free the property tag structure pointed to by the *lppPropTagArray* parameter by calling the **MAPIFreeBuffer** function.

If the object 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

[New - Windows 95]

Retrieves the property values of one or more properties of an object.

Syntax

HRESULT GetProps(**LPSPropTagArray** *lpPropTagArray*, **ULONG** *ulFlags*, **ULONG FAR** * *lpcValues*, **LPSPropValue FAR** * *lppPropArray*)

Parameters

lpPropTagArray
Input parameter pointing to an **SPropTagArray** structure containing the property tags of the properties whose values are to be retrieved. If a value of 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, the value MAPI_E_INVALID_PARAMETER is returned.

ulFlags
Input parameter containing a bitmask of flags controlling the type of the text for the returned prop value if you pass in PT_UNSPECIFIED 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 8-bit format.

lpcValues
Output parameter pointing to a variable in which is placed the number of properties for which tags are returned in the *lppPropArray* parameter. *lpcValues* 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.

The following values can be returned in the **SPropProblemArray**, but not as a return value for **SetProps**:

MAPI_E_INVALID_ENTRYID
The named properties are not supported.

MAPI_E_NOT_ENOUGH_MEMORY
Insufficient memory was available to complete the operation. You should open the property to get the data.

MAPI_E_NOT_FOUND
The requested object does not exist.

MAPI_E_OBJECT_DELETED
The folder has been deleted since it was opened.

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. Use the HR_FAILED macro to test for this warning, but the call should be handled as a successful return.

Comments

Use the **IMAPIProp::GetProps** method to get the property values of one or more properties from an object. Your application 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 the *lppPropArray* parameter exactly matches the order in which the **SPropTagArray** structure in the *lpPropTagArray* parameter requested the properties. If property types are specified in the **SPropTagArray** structure in *lpPropTagArray*, then the returned property values in the **SPropValue** structure returned in *lppPropArray* have types that exactly match the requested types, unless an error is returned instead.

If the calling application 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 will be indicated in the returned **SPropValue** structure.

When your application 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 **IMAPIProp::GetProps**, but the data in the returned **SPropValue** structure contains no useful information. To request values for secure properties, your application must explicitly request the properties by identifier; secure properties' values are not returned when you pass NULL in the *lppPropTagArray* parameter.

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 process must free the returned **SPropValue** structure by calling **MAPIFreeBuffer** but should only do so if **GetProps** returns S_OK or the warning MAPI_W_ERRORS_RETURNED.

When access to one or more properties fails, for example when the specified properties do not exist, **GetProps** returns the warning MAPI_W_ERRORS_RETURNED. The calling process should check the property tag of each of the returned properties to determine for which access failed. Those for which access failed have their property type set to PT_ERROR, and their values indicate which error occurred.

If a property for which a property value is requested does not exist, **GetProps** returns an individual **SPropValue** structure containing the value MAPI_E_NOT_FOUND. If none of the properties for which values are requested exist, as in the case of an empty profile section, your service provider should allocate an LPSPropTagArray pointer with every property tag set to the appropriate error value and return MAPI_W_ERRORS_RETURNED.

If no properties exist, and the calling application has requested values for all properties by passing NULL in the *lpPropTagArray* parameter, **GetProps** returns S_OK, sets the count value in the **cValues** member of the **SPropTagArray** structure to zero, and returns a zero length in the *lpcValues* parameter and NULL in the *lppPropArray* parameter. **GetProps** must not return multivalued properties with **cValues** set to zero.

If **GetProps** encounters an individual string or binary property that is too large (typically 4K or 8K) to conveniently be returned, **GetProps** marks that property with property type PT_ERROR, sets its value to MAPI_E_NOT_ENOUGH_MEMORY, and returns MAPI_W_ERRORS_RETURNED. Call **IMAPIProp::OpenProperty** to get the data within the property.

If the MAPI_UNICODE flag is set in the *ulFlags* parameter, then any string properties not specified to be returned in 8-bit format are returned in Unicode format. If MAPI_UNICODE is not set, string properties with unspecified types are returned in 8-bit format. String properties with unspecified types occur when **GetProps** is called with a NULL value for the *lpPropTagArray* parameter and when the property type 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.

See Also

IMAPIProp::OpenProperty method, **MAPIAllocateBuffer** function, **MAPIAllocateMore** function, **MAPIFreeBuffer** function, **SPropTagArray** structure, **SPropValue** structure

IMAPIProp::OpenProperty

[New - Windows 95]

Opens an interface on a property of an object.

Syntax

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 to be used. Must not be NULL.

ulInterfaceOptions
Input parameter indicating interface-specific behavior.

ulFlags
Input parameter containing a bitmask of flags used to control 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 you set MAPI_CREATE, you should also set MAPI_MODIFY.

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 write access. The default interface is read-only. MAPI_MODIFY must be set when MAPI_CREATE is also 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 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 the MAPI_CREATE flag was not set in the *ulFlags* parameter.

Comments

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, and this method can also be used for other properties (typically, large string and binary properties) depending on the implementation.

When opening an interface for an object with **OpenProperty**, your application uses an interface identifier (IID) to identify the interface.

The interface identifier IID_IStream is used for string and binary properties, and the interface identifier IID_IMessage is used when opening a PR_ATTACH_DATA_OBJ property for a message; a PR_ATTACH_DATA_OBJ property contains an embedded OLE object or embedded message data. The IID_IStreamDocFile interface identifier can be used for those properties that are known to be a document file encoding of an IStorage object.

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 the value MAPI_E_INVALID_PARAMETER.

Note Applications using an **IStream** interface to access double byte character set (DBCS) text should not seek or call **IStream::SetSize** on the stream other than with a zero position or size variable. Also, the application should not rely upon the value of the *plibNewPosition* output parameter. Applications should not call the **IMAPIProp::GetProps**, **IMAPIProp::SetProps**, or **IMAPIProp::DeleteProps** method or perform other calls that affect an open property while a property is open 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 are undefined and provider-dependent.

When a client application requires the ability to write to a stream it is opening, the client application should set the MAPI_MODIFY flag in the *ulFlags* parameter, whether or not the MAPI_CREATE flag is set in *ulFlags*. Some providers return the value MAPI_E_INVALID_PARAMETER when an application attempts to create a stream without setting the MAPI_MODIFY flag. If the MAPI_CREATE flag is set, MAPI_MODIFY must also be set.

The calling application is responsible for recasting the interface pointer returned in the *lppUnk* parameter to the one appropriate for the interface specified in the *lpiid* parameter. The calling application 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 on those properties. If a requested interface is not available on the given property, **OpenProperty** returns the error MAPI_E_INTERFACE_NOT_SUPPORTED.

See Also

HrIStorageFromStream function, **IMAPIProp::DeleteProps** method, **IMAPIProp::GetProps** method, **IMAPIProp::SetProps** method, **IMAPISupport::IStorageFromStream** method

IMAPIProp::SaveChanges

[New - Windows 95]

Makes permanent any changes made to an object since the last save operation.

Syntax

HRESULT SaveChanges(**ULONG** *ulFlags*)

Parameters

ulFlags

Input parameter containing a bitmask of flags used to control what happens to the object when the **IMAPIProp::SaveChanges** method is called. If no flags are set, the client application wants to commit changes to the object but will make no further calls to the object except for calling **Release**. If a provider returns an error in this case, it is because the provider couldn't save the changes. If neither KEEP_OPEN_READWRITE or KEEP_OPEN_READONLY are set, the results are implementation-specific; some providers treat it as equivalent to passing KEEP_OPEN_READWRITE, while others can choose to interpret it like KEEP_OPEN_READONLY, while others actually go to the effort of shutting down the object. The following flags can be set:

FORCE_SAVE

Writes changes to the object and closes it. Write permission must have previously been granted for the operation to succeed. This flag only forces the save if MAPI_E_OBJECT_CHANGED was returned from a preceding SaveChanges call. It overrides the previous changes made to the object.

KEEP_OPEN_READONLY

Indicates the client application wants to commit changes and keep the object open for reading. This flag informs the provider that the client application does not want to modify the object and does not intend to 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 choose to leave the object open for READ/WRITE access. However, the provider cannot close off all access to the object when this flag is given.

This flag is meant to suggest to the provider that the client does not intend to make further modifications to the object. A client that passes KEEP_OPEN_READONLY and then calls **IMAPIProp::SetProps** followed by a call to **SaveChanges** can expect the application to crash in some provider implementations.

KEEP_OPEN_READWRITE

Indicates that the client application wants to commit changes and keep the object open for read-write access. This usually occurs when the object was initially opened for read-write access. After calling **SaveChanges**, the client can choose to make further changes to the object. 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.

After receiving such an error, the client continues to have read-write access and might pass KEEP_OPEN_READONLY or no KEEP_OPEN_* flags. It is up to the provider to decide if it can fully support this flag. In no case can a provider leave the object in a read-only state when the KEEP_OPEN_READWRITE flag is given.

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.

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_OBJECT_CHANGED
The object has changed since it was opened.

MAPI_E_OBJECT_DELETED
The object has been deleted since it was opened.

Comments

Use the **IMAPIProp::SaveChanges** method to make changes permanent for some objects, specifically messages, attachments, address book containers, and messaging users. 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 the client application saves the message changes using **SaveChanges** and releases the message objects using **IUnknown::Release**. 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 the KEEP_OPEN_READONLY flag set in the *ulFlags* parameter.

Furthermore, changes to properties, such as the message subject, that are cached in a folder's summary table cannot be processed until your application has called **SaveChanges** and in some situations **Release** as well.

When making bulk changes, such as saving attachments to multiple messages, your application should defer error processing by setting the MAPI_DEFERRED_ERRORS flag in the *ulFlags* parameter. 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 constant should be omitted from the **SaveChanges** call on 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 constant altogether.) If the KEEP_OPEN_READWRITE and KEEP_OPEN_READONLY flags are set on the same call as MAPI_DEFERRED_ERRORS, providers probably will ignore MAPI_DEFERRED_ERRORS. If MAPI_DEFERRED_ERRORS is not set in the *ulFlags* parameter, one of the previously deferred errors are returned for the **SaveChanges** call.

Standard client application 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 and examination of any returned error code. If the original object has been modified, the application 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 application warns the user, who can save the message somewhere else.

If the user deletes an open object, the client application is unable to force a save, even when the FORCE_SAVE flag has been set in the *ulFlags* parameter.

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

[New - Windows 95]

Sets the property value of one or more properties of an object.

Syntax

HRESULT SetProps(ULONG *cValues*, **LPSPropValue** *lpPropArray*, **LPSPropProblemArray FAR *** *lppProblems*)

Parameters

cValues
Input parameter containing the number of values in the *lpPropArray* parameter. The *cValues* parameter must not be zero.

lpPropArray
Input parameter pointing to an array of **SPropValue** structures holding property values.

lppProblems
Output parameter pointing to a variable where the pointer to an **SPropProblemArray** structure is stored. If a value of 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 a return value 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 can't be written because it is computed by the destination object's 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_UNEXPECTED_TYPE
The type of the property value is not the type expected by the calling application.

Comments

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, your application 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 the value MAPI_E_INVALID_PARAMETER being returned. Furthermore, your application cannot set multivalued properties with zero values in the **cValues** member of the **SPropValue** structure. Calls to set zero-valued multivalued properties should return the value MAPI_E_INVALID_PARAMETER.

If the call succeeds overall but there are problems with setting some of the selected properties, the value S_OK is returned and an **SPropProblemArray** structure is returned 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, exactly which properties were not successfully set can be determined from the **SPropProblemArray** structure. For example, if a property type is invalid or not supported, the **SetProps** call returns the value MAPI_E_INVALID_TYPE for that property in the **SPropProblemArray** structure.

The value returned in the *lppProblems* parameter 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 application should not use or free the **SPropProblemArray** structure; call **IMAPIProp::GetLastError** to get the **MAPIERROR** structure for the error.

The calling application must free the returned **SPropProblemArray** structure by calling the **MAPIFreeBuffer** function, but this should only be done if **SetProps** returns with 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** methods. When such a call is received, the client attempts to transmit the cached information to the provider and the client can receive error codes related to the earlier calls. For information about how a specific service provider implements **SetProps**, see the provider's documentation.

See Also

IMAPIProp::GetProps method, **IMAPIProp::SaveChanges** method, **MAPIFreeBuffer** function, **SPropProblemArray** structure, **SPropValue** structure

IMAPISession : IUnknown

[New - Windows 95]

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

Vtable Order

GetLastError	Returns information about the last error that occurred for the session object.
GetMsgStoresTable	Returns a table that provides information about each of the message stores configured in the 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.
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 client-application session.
OpenEntry	Opens an object by using its entry identifier.
CompareEntryIDs	Compares two entry identifiers to determine whether they refer to the same object.
Advise	Registers your application for notifications on changes to the session object.
Unadvise	Removes a registration for notification of changes previously established with a call to the **IMAPISession::Advise** method.
MessageOptions	Displays a modal dialog box showing the message options for a particular message.
QueryDefaultMessageOpt	Returns the available message options and their default settings for a particular e-mail address type.
EnumAdrTypes	Returns the e-mail address types for which a transport provider has registered support.
QueryIdentity	Returns an entry identifier representing the primary identity for a MAPI session.
Logoff	Ends a MAPI session.
SetDefaultStore	Sets the default message store provider.

AdminServices	Administers configuration changes to the profile.
ShowForm	Displays a message form for editing and sending.
PrepareForm	Creates a message instance for use by **ShowForm**.

IMAPISession::AdminServices

[New - Windows 95]

Administers configuration changes to the profile.

Syntax

HRESULT AdminServices(**ULONG** *ulFlags*, **LPSERVICEADMIN FAR *** *lppServiceAdmin*)

Parameters

ulFlags
: Reserved; must be zero.

lppServiceAdmin
: Output parameter pointing to a variable where the pointer to the returned service-administration object is stored.

Return Values

S_OK
: The call succeeded and has returned the expected value or values.

Comments

Use the **IMAPISession::AdminServices** method to acquire a pointer to an **IMsgServicesAdmin** interface. Once your application has the pointer to the interface, it can call **IMsgServicesAdmin** methods to change the message service configuration within a profile. Changes made to the **IMsgServicesAdmin** methods will not affect the current running session. If profile configuration was the primary purpose for creating the session, log on using the MAPI_NO_MAIL flag.

See Also

IMsgServiceAdmin : IUnknown interface, **IProfAdmin::AdminServices** method

IMAPISession::Advise

[New - Windows 95]

Registers your application for notifications on changes to the session object.

Syntax

HRESULT Advise(**ULONG** *cbEntryID*, **LPENTRYID** *lpEntryID*, **ULONG** *ulEventMask*,
LPMAPIADVISESINK *lpAdviseSink*, **ULONG FAR *** *lpulConnection*)

Parameters

cbEntryID
Input parameter containing the number of bytes in 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. An entry identifier of a status object cannot be used. A value of NULL can be used to detect logoffs from shared sessions.

ulEventMask
Input parameter containing an event mask of the types of notification events occurring for the object for which MAPI will generate notifications to filter specific cases. The following table lists the possible event types along with their corresponding data structures:

Notification event type	**Corresponding data structure**
fnevCriticalError	**ERROR_NOTIFICATION**

lpAdviseSink
Input parameter pointing to an advise sink object to be called when an event for the session object occurs about which notification has been requested. This advise sink object must have already been allocated.

lpulConnection
Output parameter pointing to a variable that on 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 the *lpEntryID* parameter.

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 could not be found to handle the entry identifier.

Comments

Use the **IMAPISession::Advise** method to register a session object for notification callbacks. MAPI will forward this call to the service provider active on the session that is responsible for the object indicated by the entry identifier in the *lpEntryID* parameter. Whenever a change occurs to the indicated object, the provider checks to see what event mask bit has been set in the *ulEventMask* parameter and thus what type of change has 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, your client should use the **HrThisThreadAdviseSink** function.

To provide notifications, the service provider implementing **Advise** needs to keep a copy of the pointer to the advise sink object; to do so, it calls the **IUnknown::AddRef** method for the advise sink object to maintain the 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, client applications must be prepared for the advise sink object to be released. Clients should therefore release their advise sink object after **Advise** returns unless they have a specific long term use for it.

See Also

ERROR_NOTIFICATION structure, **HrThisThreadAdviseSink** function, **IMAPIAdviseSink::OnNotify** method, **IMAPISession::Unadvise** method

IMAPISession::CompareEntryIDs

[New - Windows 95]

Compares two entry identifiers to determine if they refer to the same service provider 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.

Syntax

HRESULT CompareEntryIDs(**ULONG** *cbEntryID1*, **LPENTRYID** *lpEntryID1*, **ULONG** *cbEntryID2*, **LPENTRYID** *lpEntryID2*, **ULONG** *ulFlags*, **ULONG FAR *** *lpulResult*)

Parameters

cbEntryID1
Input parameter containing the number of bytes in 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 number of bytes in 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 that receives the result of the comparison; 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.

Comments

Use the **IMAPISession::CompareEntryIDs** method to compare two entry identifiers for a given service provider object and 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 could occur, for example, after a new version of a service provider is installed.

If **CompareEntryIDs** returns an error, the calling application should make no assumptions about the comparison and should take the most conservative approach based on what your application is trying to do. **CompareEntryIDs** might fail if, for example, no provider has registered for one of the entry identifiers. If your application compares message-store entry identifiers when one or both of the stores has not yet opened, **CompareEntryIDs** returns the value MAPI_E_UNKNOWN_ENTRYID.

IMAPISession::EnumAdrTypes

[New - Windows 95]

Returns the e-mail address types for which a transport provider has registered support.

Syntax

HRESULT EnumAdrTypes(**ULONG** *ulFlags*, **ULONG FAR** * *lpcAdrTypes*, **LPTSTR FAR * FAR** * *lpppszAdrTypes*)

Parameters

ulFlags
Input parameter containing a bitmask of flags controlling 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 8-bit format.

lpcAdrTypes
Output parameter pointing to the number of strings indicating e-mail address types returned in the *lpppszAdrTypes* parameter.

lpppszAdrTypes
Output parameter pointing to a variable where the array of pointers to strings containing e-mail address types (such as FAX, SMTP, X500, and so on) is stored.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

Comments

Use the **IMAPISession::EnumAdrTypes** method to determine which address types are supported by the transport providers loaded on a given session. The list of address types returned by **EnumAddrTypes** depends on what providers are loaded at the time. For example, if a transport provider is not open, the address types that the unopened provider supports do not appear in the list.

Your application should release the string array pointed to by the *lpppszAdrTypes* parameter when done with them by calling the **MAPIFreeBuffer** function.

See Also

MAPIFreeBuffer function

IMAPISession::GetLastError

[New - Windows 95]

Returns a **MAPIERROR** structure containing information about the last error that occurred for the session object.

Syntax

HRESULT GetLastError(**HRESULT** *hResult*, **ULONG** *ulFlags*, **LPMAPIERROR FAR *** *lppMAPIError*)

Parameters

hResult
Input parameter containing the result returned for the last call on the session object that returned an error.

ulFlags
Input parameter containing a bitmask of flags controlling the type of the returned strings. The following flag can be set:

MAPI_UNICODE
Indicates the strings returned 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 8-bit 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.

Comments

Use the **IMAPISession::GetLastError** method to retrieve information to display as a message to the user regarding the last error returned from a method call on the session object.

To release all the memory allocated by MAPI, client applications need only call the **MAPIFreeBuffer** function for the **MAPIERROR** structure.

The return value from **GetLastError** must be S_OK for the application to make use of the **MAPIERROR** structure. Even if the return value is S_OK, it is still possible that a **MAPIERROR** structure won't be returned. If the implementation cannot determine what the last error was, or if a MAPIERROR structure is not available for that error, a pointer to NULL is returned in the *lppMAPIError* parameter instead.

See Also

MAPIERROR structure, **MAPIFreeBuffer** function

IMAPISession::GetMsgStoresTable

[New - Windows 95]

Returns a table that provides information about each of the message stores configured in the session's profile.

Syntax

HRESULT GetMsgStoresTable(**ULONG** *ulFlags*, **LPMAPITABLE FAR** * *lppTable*)

Parameters

ulFlags
Input parameter containing a bitmask of flags controlling 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 8-bit 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.

Comments

Use the **IMAPISession::GetMsgStoresTable** method to retrieve a table holding information about each open message store configured in the session's profile. The message-store information table contains the following required property columns:

PR_DEFAULT_STORE
If this column has an entry, and its value is true, it indicates that this message store is where the MAPI spooler delivers incoming messages. One and only one message store per profile has this value set to true.

PR_DISPLAY_NAME
The display name of the message store resource listed in the current row.

PR_ENTRYID
The entry identifier for the message store resource listed in the current row. This entry identifier is used to open the resource with the **IMAPISession::OpenMsgStore** method. This is a long-term entry identifier, except in the case of a message store not fully configured in the profile.

PR_INSTANCE_KEY
The index column for the row.

PR_PROVIDER_DISPLAY
The display name of the service provider.

PR_RECORD_KEY
This is the unique identifier for the message store object.

PR_RESOURCE_TYPE
For messages stores, the resource type value is MAPI_STORE_PROVIDER.

The message-store information table can also contain the following optional columns:

PR_MDB_PROVIDER
    Indicates the provider type that furnished the underlying data for the message store.
PR_RESOURCE_FLAGS
    Contains message service-specific flags for the message store.

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. The changes include addition of new, permanent message stores to the table, removal of existing stores, and changes in which message store is the default. Client applications 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 the **QueryColumns** method returns before the application that contains the message-store information 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 application that contains the message-store information table calls the **IMAPITable::SetColumns** method.
- Controls the property types of the sort order specification returned by the **IMAPITable::QuerySortOrder** method before the application 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

[New - Windows 95]

Gets the status table for a given client-application session.

Syntax

HRESULT GetStatusTable(**ULONG** *ulFlags*, **LPMAPITABLE FAR** * *lppTable*)

Parameters

ulFlags
Input parameter containing a bitmask of flags controlling 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 8-bit format.

lppTable
Output parameter pointing to a pointer to the returned status table object.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

Comments

Use the **IMAPISession::GetStatusTable** method to open the status table object which contains information about everything that is 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 **IMAPISession::OpenEntry** on an entry identifier within the status table returned from **GetStatusTable**. The status table contains at least the following property columns:

PR_DISPLAY_NAME
Name of the component.

PR_ENTRYID
Unique identifier for the object.

PR_IDENTITY_DISPLAY
The display name for the service provider as defined by the service provider.

PR_INSTANCE_KEY
The search key for the row.

PR_OBJECT_TYPE
Indicates the type of the MAPI object.

PR_PROVIDER_DLL_NAME
Filename of the provider DLL.

PR_PROVIDER_DISPLAY
The display name of the service provider.

PR_RESOURCE_METHODS
Flags indicating which methods are supported by the associated status object.

PR_RESOURCE_FLAGS
Flags describing the resource.

PR_RESOURCE_TYPE
Indicates what type of service provider this provider is.

PR_ROWID

Contains a unique identifier for the row.

PR_STATUS_CODE
: Status of the component

The following properties are optional:

PR_IDENTITY_ENTRYID
: Entry identifier of the user on this resource.

PR_IDENTITY_SEARCH_KEY
: The search key used to locate the service provider.

PR_STATUS_STRING
: String to describe what the state of the component is. If absent, the default value is used based on PR_STATUS_CODE.

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 the **QueryColumns** method returns before the application that contains the status 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 application that contains the status table calls the **IMAPITable::SetColumns** method.
- Controls the property types of the sort order specification returned by the **IMAPITable::QuerySortOrder** method before the application that contains the message store information 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

[New - Windows 95]

Ends a MAPI session.

Syntax

HRESULT Logoff(**ULONG** *ulUIParam*, **ULONG** *ulFlags*, **ULONG** *ulReserved*)

Parameters

ulUIParam
Input parameter whose usage depends on the platform. On Windows platforms, the *ulUIParam* parameter is the handle to the main window of the calling application, cast to a ULONG.

ulFlags
Input parameter containing a bitmask of flags used to control the logoff process. The following flags can be set:

MAPI_LOGOFF_SHARED
Indicates that all applications logged in using the shared session are notified of the shared logoff and can log off if they want to. Any application that uses the shared session can set this flag. If the specified session is not part of the shared session, this flag is ignored.

MAPI_LOGOFF_UI
Indicates that a logoff dialog box (some implementations of MAPI might request confirmation of logoff if operations are pending) should be displayed. 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.

Comments

Use 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 it returns an HRESULT rather than a ULONG.

If the MAPI_LOGOFF_SHARED flag is set in the *ulFlags* parameter, notifications are sent to all client applications using the shared session indicating that they should also log off.

After **Logoff** has been called, the calling application 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

[New - Windows 95]

Displays a modal dialog box showing the message options for a particular message. You can edit the message options in the dialog box.

Syntax

HRESULT MessageOptions(**ULONG** *ulUIParam*, **ULONG** *ulFlags*, **LPTSTR** *lpszAdrType*, **LPMESSAGE** *lpMessage*)

Parameters

ulUIParam
Input parameter containing the handle to the window that the dialog box is modal to.

ulFlags
Input parameter containing a bitmask of flags controlling the type of the 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 8-bit format.

lpszAdrType
Input parameter pointing to a string containing the messaging address type for which the options dialog box should be displayed; examples of e-mail address types are FAX, SMTP, X500, and so on. If all registered address types should be shown, your application should pass NULL in the *lpszAdrType* parameter. The *lpszAdrType* parameter's value must not be zero.

lpMessage
Input parameter pointing to the message for which a dialog box is to be 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.

Comments

Use 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 become persistent after the application calls **IMAPIProp::SaveChanges** on the message. Changes made to the message options for the message do not affect the default options for that address type. These options are usually messaging service elements specific to a transport provider.

IMAPISession::OpenAddressBook

[New - Windows 95]

Opens the MAPI address book and returns a pointer that provides further access.

Syntax

HRESULT OpenAddressBook(**ULONG** *ulUIParam*, **LPCIID** *lpInterface*, **ULONG** *ulFlags*, **LPADRBOOK FAR *** *lppAdrBook*)

Parameters

ulUIParam
Input parameter containing the handle to the window that the address book window is modal to.

lpInterface
Input parameter pointing to the interface identifier for the address book object. Passing NULL in the *lpInterface* parameter indicates that the return value is cast to the standard interface for the address book object.

ulFlags
Input parameter containing a bitmask of flags used to control the return of the columns. The following flag can be set:

AB_NO_DIALOG
Suppresses display of dialog boxes while underlying address book providers are initialized during logon. If AB_NO_DIALOG is not set in the *ulFlags* parameter, 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_W_ERRORS_RETURNED
The call succeeded, but one or more of the address book providers could not be loaded. Use the HR_FAILED macro to test for this warning, but the call should be handled as a successful return.

Comments

Use 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 addressing dialog boxes.

This method can return MAPI_W_ERRORS_RETURNED if it cannot load an address book provider This is a warning, not an error, and should be handled as a successful return. 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, your application can still use the **IAddrBook** interface, and must release it when done.

Your application can call the **IMAPISession::GetLastError** method on the session object to obtain a **MAPIERROR** structure containing information about the address book providers that failed to 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.

See Also

IMAPISession::GetLastError method, **MAPIERROR** structure

IMAPISession::OpenEntry

[New - Windows 95]

Opens an object given its entry identifier.

Syntax

HRESULT OpenEntry(**ULONG** *cbEntryID*, **LPENTRYID** *lpEntryID*, **LPCIID** *lpInterface*, **ULONG** *ulFlags*, **ULONG FAR *** *lpulObjType*, **LPUNKNOWN FAR *** *lppUnk*)

Parameters

cbEntryID
Input parameter containing the number of bytes in the entry identifier pointed to by the *lpEntryID* parameter.

lpEntryID
Input parameter pointing to the entry identifier of the object to be opened.

lpInterface
Input parameter pointing to the interface identifier for the indicated object. Passing NULL in the *lpInterface* parameter indicates that the return value is cast to the standard interface for the indicated object. The *lpInterface* parameter can also be set to an appropriate interface identifier for the object being opened.

ulFlags
Input parameter containing a bitmask of flags used to control how the object is opened. The following flags can be used:

MAPI_BEST_ACCESS
Indicates the object should be opened with the maximum privileges allowed to the user. For example, if the client application has write privilege, open the object with write privilege; if the client application has read-only privilege, open the object with read-only privilege. The client application can learn the privilege 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 application. 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 client applications should not assume that write access was granted.

lpulObjType
Output parameter pointing to a variable where the object type for 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 to modify a read-only object or an attempt to access an object for which the user has insufficient permissions.

MAPI_E_UNKNOWN_ENTRYID
The object indicated in the *lpEntryID* parameter is not recognized. This return value is typically returned if the message store or address book provider that the object is contained within is not open.

MAPI_E_NOT_FOUND
The object indicated in the *lpEntryID* parameter does not exist.

Comments

Use 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 application does not have information on where to locate the object. The **IMAPISession::OpenEntry** call returns a pointer that provides further access to the object to be opened. 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.

In the MAPI implementation running with Windows, 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 that status object.

Calling **OpenEntry** and passing in the entry identifier for a message store in the *lpEntryID* parameter, as your application 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, which, without displaying a logon dialog box, opens that message store and returns a pointer to it. 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 application should check the value returned in the *lpulObjType* parameter to determine that the object type returned is what was expected. Commonly, after the application 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

[New - Windows 95]

Opens a message store and returns a pointer that provides further access to the open store.

Syntax

HRESULT OpenMsgStore(**ULONG** *ulUIParam*, **ULONG** *cbEntryID*, **LPENTRYID** *lpEntryID*, **LPCIID** *lpInterface*, **ULONG** *ulFlags*, **LPMDB FAR** * *lppMDB*)

Parameters

ulUIParam
Input parameter whose usage depends on the platform. On Windows platforms, the *ulUIParam* parameter is the handle to the main window of the calling application, cast to a ULONG.

cbEntryID
Input parameter containing the number of bytes in 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 indicated message-store object. Passing NULL in the *lpInterface* parameter indicates that the return value is cast to the standard interface for a message store object.

ulFlags
Input parameter containing a bitmask of flags used to control how the message store is accessed. The following flags can be used:

MAPI_BEST_ACCESS
Indicates the object should be opened with the maximum privileges allowed to the user. For example, if the client application has write privilege, open the object with write privilege; if the client application has read-only privilege, open the object with read-only privilege. The client application can learn the privilege 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 application. 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, the error code 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 the 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 to not notify the MAPI spooler that this message store is being opened.

MDB_TEMPORARY
Used to log the store on so that information can be retrieved programmatically from the profile section. Instructs MAPI that the store is not to be added to the message stores information table and that the store cannot be made permanent.

MDB_WRITE
Requests write access.

lppMDB
Output parameter pointing to a variable where the 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 in the *lpEntryID* parameter does not exist.

MAPI_W_ERRORS_RETURNED
: The call succeeded, but the message store provider has information available. Use the HR_FAILED macro to test for this warning, but the call should be handled as a successful return. Call **IMAPISession::GetLastError** to get the information from the provider.

Comments

Use the **IMAPISession::OpenMsgStore** method during the logon process to get access to a particular message store using the entry identifier from the message store table. The default behavior is to open the object as read-only, unless an application 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 bits 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.
- Calling the **IMAPISession::OpenEntry** method or the **IMAPIProp::OpenProperty** method with the flag MAPI_MODIFY, which requests write access, fails.
- Calls to the following methods fail: **IMAPIFolder::CreateMessage**, **IMAPIFolder::DeleteMessages**, **IMAPIFolder::CreateFolder**, **IMAPIFolder::DeleteFolder**, **IMAPIFolder::SetMessageStatus**, **IMAPIProp::SetProps**, **IMAPIProps::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**, **IMAPIFolder::CopyTo**.

IMAPISession::OpenProfileSection

[New - Windows 95]

Opens a section of the current profile and returns a pointer that provides further access to the profile object.

Syntax

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) that identifies the profile section.

lpInterface
Input parameter pointing to the interface identifier used to open the profile section. Passing NULL in the *lpInterface* parameter indicates that the return value is cast to the standard interface for a profile section. The *lpInterface* parameter can also be set to an appropriate interface identifier. Valid interface identifiers are IID_IMAPIProp and IID_IProfSect.

ulFlags
Input parameter containing a bitmask of flags used to control 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 application. 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 client applications should not assume that write access was granted.

lppProfSect
Output parameter pointing to a variable that receives the pointer to the open profile object.

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 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.

Comments

Use 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. The default behavior is to open the profile section as read-only, unless an application sets the MAPI_MODIFY flag in the *ulFlags* parameter. Profile sections belonging to service providers cannot be opened by calls to the **IMAPISession::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 application has the profile section open, a read-write open operation fails and returns the value 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 the MAPI_MODIFY flag in the *ulFlags* parameter, the call creates the section. If an **OpenProfileSection** call attempts to open a nonexistent section with read-only access, the value MAPI_E_NOT_FOUND is returned.

All open operations should be as brief as possible, but an application that is writing to a profile section can keep it open while displaying a modification dialog box.

See Also

IMAPIProp : IUnknown interface, **IProfSect : IMAPIProp** interface, **MAPIUID** structures

IMAPISession::PrepareForm

[New - Windows 95]

Creates a message instance for use by **ShowForm**.

Syntax

HRESULT PrepareForm(**LPCIID** *lpInterface*, **LPMESSAGE** *lpMessage*, **ULONG FAR *** *lpulMessageToken*)

Parameters

lpInterface
: Input parameter pointing to the interface identifier (IID) for a message object. Passing NULL for the *lpInterface* parameter indicates the MAPI interface for the message object will be returned. A client application can also pass in *lpInterface* 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 the returned message token, which must be passed in the following **IMAPISession::ShowForm** call.

Return Values

S_OK
: The call succeeded and has returned the expected value or values.

Comments

Use the **IMAPISession::PrepareForm** method to create a message instance for use by the **IMAPISession::ShowForm** method. Client applications should only have a single reference to the message passed in **PrepareForm**'s *lpMessage* parameter. If the call to **PrepareForm** succeeds, the client application 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

[New - Windows 95]

Returns the available message options and their default setting for a particular e-mail address type.

Syntax

HRESULT QueryDefaultMessageOpt(**LPTSTR** *lpszAdrType*, **ULONG** *ulFlags*, **ULONG FAR** * *lpcValues*, **LPSPropValue FAR** * *lppOptions*)

Parameters

lpszAdrType
Input parameter pointing to a string containing the e-mail address type in question; examples of e-mail address types are FAX, SMTP, X500, and so on.

ulFlags
Input parameter containing a bitmask of flags controlling the type of the strings. The following flag can be set:

MAPI_UNICODE
Indicates the passed-in and the returned strings are in Unicode format. If the MAPI_UNICODE flag is not set, the strings are in 8-bit format.

lpcValues
Output parameter pointing to the number of property values returned in the *lppOptions* parameter.

lppOptions
Output parameter pointing to a variable where the pointer to the 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.

Comments

Use the **IMAPISession::QueryDefaultMessageOpt** method to find out what message options are available for a particular e-mail 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

[New - Windows 95]

Returns an entry identifier representing the primary identity for a user for a MAPI session.

Syntax

HRESULT QueryIdentity(**ULONG FAR** * *lpcbEntryID*, **LPENTRYID FAR** * *lppEntryID*)

Parameters

lpcbEntryID
: Output parameter pointing to a variable in which the **IMAPISession::QueryIdentity** method returns the number of bytes in 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. Use the HR_FAILED macro to test for this warning, although the call should be handled as a successful return.

Comments

Use the **IMAPSession::QueryIdentity** method to retrieve a user's primary identity for the current session. The *primary identity* is a string that represents the user of a MAPI session.

Each messaging service provider that MAPI has information about establishes an identity for each of its users. This identity can be established when a client application 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 bit in the PR_RESOURCE_FLAGS property. Client applications can call the **IMsgServiceAdmin::SetPrimaryIdentity** method to designate one of the many identities established for a user by messaging service providers as the primary identity for that user.

The **QueryIdentity** method provides applications a convenient way to retrieve this primary identity. Once your application 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 e-mail address. Sophisticated applications 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 applications, which assume 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 the *lppEntryID* parameter.
- If some rows make available the PR_IDENTITY_ENTRYID property 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 the client application 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.

See Also

IMAPISession::OpenEntry method, **IMsgServiceAdmin::SetPrimaryIdentity** method, **MAPIFreeBuffer** function

IMAPISession::SetDefaultStore

[New - Windows 95]

Sets the default message store provider.

Syntax

HRESULT SetDefaultStore(**ULONG** *ulFlags*, **ULONG** *cbEntryID*, **LPENTRYID** *lpEntryID*)

Parameters

ulFlags

Input parameter containing a bitmask of flags used to control 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 of the PR_RESOURCE_FLAGS column of the status table.

MAPI_PRIMARY_STORE

Sets the STATUS_PRIMARY_STORE flag of the PR_RESOURCE_FLAGS column of the status table. Indicates the store that client applications should try to use when they log on. If the primary store is not the default store, client applications should set it to be so.

MAPI_SECONDARY_STORE

Sets the STATUS_SECONDARY_STORE flag of the PR_RESOURCE_FLAGS column of the status table. Indicates the store that client applications should try to use if the primary store is not available. If the client application cannot open the primary store it should open the secondary store and set it to be the default store.

MAPI_SIMPLE_STORE_PERMANENT

Causes the STATUS_SIMPLE_STORE flag of the PR_RESOURCE_FLAGS column of the status table to be set.

MAPI_SIMPLE_STORE_TEMPORARY

Causes the STATUS_SIMPLE_STORE flag of the PR_RESOURCE_FLAGS column of the status table to be set.

cbEntryID

Input parameter containing the number of bytes in 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 your application sends a value of NULL in the *lpEntryID* parameter, no message store is selected for the default.

Return Values

S_OK

The call succeeded and has returned the expected value or values.

Comments

Use the **IMAPISession::SetDefaultStore** method to make a single Extended MAPI 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 bits in the PR_STORE_SUPPORT_MASK property.

See Also

PR_RESOURCE_FLAGS property, **PR_STORE_SUPPORT_MASK** property, **TABLE_NOTIFICATION** structure

IMAPISession::ShowForm

[New - Windows 95]

Displays a message form for editing and sending.

Syntax

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 for the window in which the form is displayed.

lpMsgStore
Input parameter pointing to the message store where the folder pointed to by the *lpParentFolder* parameter resides.

lpParentFolder
Input parameter pointing to the folder in which the message was created.

lpInterface
Reserved; must be NULL.

ulMessageToken
Input parameter containing the message token returned in 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 will not be released.

lpMessageSent
Reserved; must be NULL.

ulFlags
Input parameter containing a bitmask of flags used to control how the message is saved after being sent. The following flags can be set:

MAPI_NEW_MESSAGE
Indicates that the message has not previously been saved (i.e. **IMAPIProp::SaveChanges** has not been called on the message).

MAPI_POST_MESSAGE
Indicates that the message should be saved to the parent folder for the message. 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 application- or service 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 the *ulMessageToken* parameter. The flags indicate the current state of the message.

ulAccess
Input parameter containing flags copied from the PR_ACCESS_LEVEL property of the message referenced to the message in the *ulMessageToken* parameter. The flags indicate the level of read-write access permitted on the message.

lpszMessageClass

Input parameter pointing to a string naming the message class of the message referenced to the message in the *ulMessageToken* parameter. This string is copied from the PR_MESSAGE_CLASS property of that 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.

Comments

Use the **IMAPISession::ShowForm** method together with **IMAPISession::PrepareForm** to display a message form for editing or sending. Client applications should only have a single reference to the message passed in **PrepareForm**'s *lpMessage* parameter. If the call to **PrepareForm** succeeds, the client application 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::PrepareForm method

IMAPISession::Unadvise

[New - Windows 95]

Removes an object's registration for notification of changes previously established with a call to the **IMAPISession::Advise** method.

Syntax

HRESULT Unadvise(**ULONG** *ulConnection*)

Parameters

ulConnection
Input parameter containing the number of the registration connection previously returned by a call to the **IMAPISession::Advise** method.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

Comments

Client applications use 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** and thereby cancel 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 **IMAPIAdviseSink::OnNotify** on the advise sink object, the **Release** call is delayed until the **OnNotify** method returns.

See Also

IMAPIAdviseSink::OnNotify method, **IMAPISession::Advise** method

IMAPIStatus : IMAPIProp

[New - Windows 95]

The **IMAPIStatus** interface is used by 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
Called by:	Client applications

Vtable Order

ValidateState	Confirms the external status information available for a service provider by checking with the service provider itself.
SettingsDialog	Displays a dialog box enabling the user to change the configuration of the active service provider.
ChangePassword	Changes a provider-specific password.
FlushQueues	Forces all messages waiting to be sent or received by a particular transport provider to be uploaded or downloaded synchronously.

IMAPIStatus::ChangePassword

[New - Windows 95]

Changes a provider-specific password for the provider. A user interface is never displayed.

Syntax

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 controlling the type of the password 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 8-bit 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 bit is not set in the PR_RESOURCE_METHODS property.

See Also

PR_RESOURCE_METHODS property

IMAPIStatus::FlushQueues

[New - Windows 95]

Forces all messages waiting to be sent or received by a particular transport provider to be uploaded or downloaded synchronously.

Syntax

HRESULT FlushQueues(**ULONG** *ulUIParam*, **ULONG** *cbTargetTransport*, **LPENTRYID** *lpTargetTransport*, **ULONG** *ulFlags*)

Parameters

ulUIParam
Input parameter containing the handle of the window the dialog box is modal to.

cbTargetTransport
Input parameter containing the number of bytes in the *lpTargetTransport* parameter. Only valid when called by the MAPI spooler's status object. To flush all queues pass a value of zero.

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). Only valid when called from the MAPI spooler's status object. To flush all queues pass a value of NULL.

ulFlags
Input parameter containing a bitmask of flags used to control message queue flushing. 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 operation is complete.

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. Asynchronous transports cannot respond to unforced **IXPLogon::FlushQueues** and **IMAPIStatus::FlushQueues** method calls.

FLUSH_NO_UI
Indicates the transport provider should not display user interface components. This flag is used only by the MAPI spooler; transport 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 another operation is attempted.

MAPI_E_NO_SUPPORT
The STATUS_FLUSH_QUEUES bit is not set in the PR_RESOURCE_METHODS property.

Comments

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 user interface. **FlushQueues** processing can take a long time; MAPI_E_BUSY should be returned for asynchronous requests so that client applications can continue. 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

[New - Windows 95]

Displays a dialog box enabling the user to change the configuration of the active service provider.

Syntax

HRESULT SettingsDialog(**ULONG** *ulUIParam*, **ULONG** *ulFlags*)

Parameters

ulUIParam
Input parameter containing the handle to the window the dialog box is modal to.

ulFlags
Input parameter containing a bitmask of flags used to control how the configuration dialog box is displayed. The following flags can be set:

UI_READONLY
Suggests the provider not allow 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 bit is not set in the PR_RESOURCE_METHOD property.

Comments

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

[New - Windows 95]

Confirms the external status information available for a service provider by checking with the service provider itself.

Syntax

HRESULT ValidateState(**ULONG** *ulUIParam*, **ULONG** *ulFlags*)

Parameters

ulUIParam
Input parameter containing the handle of the window the dialog box is modal to.

ulFlags
Input parameter containing a bitmask of flags used to control the status check for the service provider. The following flags can be set:

ABORT_XP_HEADER_OPERATION
Indicates that the user has requested that the operation be canceled (typically by pressing the cancel button). The transport has the option to continue working on the operation, or the transport can abort the operation and return MAPI_E_USER_CANCELED.

CONFIG_CHANGED
When the MAPI spooler is called with the flag set, it validates the state of currently loaded transports by calling the transport's **IXPLogon::AddressTypes** and **IMAPISession::MessageOptions** methods at its next convenient time. Also provides the MAPI spooler an opportunity to correct critical transport failures without forcing the client applications to log off and then log on again. When a specific transport is called with this flag set, the transport calls **IMAPISupport::SpoolerNotify** to notify the MAPI spooler if its profile section has been updated.

FORCE_XP_CONNECT
Indicates that the user selected a connect operation. When this flag is used with the REFRESH_XP_HEADER_CACHE or PROCESS_XP_HEADER_CACHE flags, the connect action occurs without caching.

FORCE_XP_DISCONNECT
Indicates that the user selected a disconnect operation. When this flag is used with the REFRESH_XP_HEADER_CACHE or PROCESS_XP_HEADER_CACHE flags, the disconnect action occurs without caching.

PROCESS_XP_HEADER_CACHE
Indicates that entries in the header cache table should be processed and that all messages marked as MSGSTATUS_REMOTE_DOWNLOAD should be downloaded and that all messages marked as MSGSTATUS_REMOTE_DELETE 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 mailbag headers should be downloaded and that all message status marks should be cleared.

SUPPRESS_UI
Prevents the provider from displaying user interface components.

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 another operation is attempted.

MAPI_E_NO_SUPPORT
: The STATUS_VALIDATE_STATE bit 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 the dialog box. This flag is only returned by remote transport providers.

Comments

Use the **IMAPIStatus::ValidateState** method to have a service 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

[New - Windows 95]

MAPI provides a support object for all service providers as a part of the registration process during service provider logon. Some methods of the MAPI support object provide functionality used by all provider types; for example, the **GetLastError** method is used by all providers. Other methods of the support object are specific to one provider type; for example, the **CopyMessages** method is only used by message store providers. Methods that are not implemented for a particular provider return MAPI_E_NO_SUPPORT when called.

MAPI creates a separate support object for each provider. When a provider makes calls to methods of the **IMAPISupport** interface, it should use the same support object passed in the *lpMAPISup* parameter when MAPI logged the provider onto that session. Providers working with more than one support object should maintain consistency between support objects and the session they are linked with.

The following table indicates which methods are implemented for particular providers.

Method	Address book provider	Message store provider	Transport provider
Address	•		
CompareEntryIDs	•	•	•
CompleteMsg		•	
CopyFolder		•	
CopyMessages		•	
CreateOneOff	•	•	•
Details	•		
DoConfigPropsheet	•	•	•
DoCopyProps		•	
DoCopyTo		•	
DoProgressDialog	•	•	
DoSentMail		•	
ExpandRecips		•	
GetLastError	•	•	•
GetMemAllocRoutines	•	•	•
GetOneOffTable	•		
GetSvcConfigSupportObj	•	•	•
IStorageFromStream	•	•	•
MakeInvalid	•	•	•
ModifyProfile		•	
ModifyStatusRow	•	•	•
NewEntry	•		
NewUID	•	•	•
Notify	•	•	•
OpenAddressBook		•	•
OpenEntry	•	•	•
OpenProfileSection	•	•	•

OpenTemplateID	•		
PrepareSubmit		•	
ReadReceipt		•	
RegisterPreprocessor			•
SetProviderUID	•	•	
SpoolerNotify		•	•
SpoolerYield			•
StatusRecips			•
StoreLogoffTransports		•	
Subscribe	•	•	•
Unsubscribe	•	•	•
WrapStoreEntryID	•	•	•

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:	Providers

Vtable Order

GetLastError	Returns information about the last error that occurred for the support object.
GetMemAllocRoutines	Retrieves the addresses of the three MAPI memory allocation functions, **MAPIAllocateBuffer**, **MAPIAllocateMore**, and **MAPIFreeBuffer**.
Subscribe	Sets up a subscription for notification of changes.
Unsubscribe	Remove an object's subscription for notification of changes.
Notify	Notifies interested applications about changes to an object that the provider owns.
ModifyStatusRow	Creates the status table column values for the service providers.
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, unique MAPI identifier (a 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 that the transport provider needs servicing.

CreateOneOff	Creates an entry identifier for a custom recipient.
SetProviderUID	Informs MAPI of the MAPI unique identifier (MAPIUID) assigned to the service provider using this support object.
CompareEntryIDs	Compares two entry identifiers to determine if they refer to the same support object.
OpenTemplateID	Allows runtime binding of one address book provider's code to the data in another address book provider.
OpenEntry	Opens an object given its entry identifier.
GetOneOffTable	Returns the table of custom recipient address templates that can be created onto a message.
Address	Displays MAPI's default addressing dialog box.
Details	Displays details about an address book entry using MAPI's default user interface.
NewEntry	Creates a new address book entry using MAPI's default user interface.
DoConfigPropsheet	Enables service providers to display property sheets using MAPI's default user interface. Primarily used for configuration.
CopyMessages	Copies or moves messages from the current folder to another folder.
CopyFolder	Copies or moves a folder to a destination folder.
DoCopyTo	Copies or moves all the properties of the source object to a destination object, except for given properties that are excluded from the operation.
DoCopyProps	Copies or moves a set of properties from the source object to a destination object.
DoProgressDialog	Provides a default implementation of the **IMAPIProgress** interface 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 a sentmail message for an open message.
OpenAddressBook	Opens the address book and returns a pointer that provides further access.
CompleteMsg	Calls the MAPI spooler to complete message delivery processing.
StoreLogoffTransports	Specifies the orderly release of the 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 a message store provider's profile section permanent.
IStorageFromStream	Implements an **IStorage** object on an **IStream** object.
GetSvcConfigSupportObj	Creates a new support object for use by calls to a messaging service provider's configuration entry point function.

IMAPISupport::Address

[New - Windows 95]

Displays MAPI's default addressing dialog box.

Syntax

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. A window handle must always be passed in on input. If the DIALOG_SDI flag is set in the **ADRPARM** structure, then on output the window handle of the modeless dialog is returned.

lpAdrParms
Input-output parameter pointing to an **ADRPARM** structure that controls the presentation and behavior of the addressing dialog box.

lppAdrList
Input-output parameter pointing to a variable where the pointer to an **ADRLIST** structure holding the recipient list updated by **IAddrBook::Address** is stored. *lppAdrList* can be NULL on input.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

Comments

Use the **IMAPISupport::Address** method to pass the current list of recipients in a message and return a list updated using the MAPI addressing dialog box. A client application passes the current list of recipients for a message, possibly empty, in the *lppAdrList* parameter when calling **Address**, 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 both the passed and returned **ADRLIST** structures consist of **ADRENTRY** structures, with each holding an individual recipient that are organized 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 (that is, a recipient that receives a copy of a message), and MAPI_BCC (that is, a recipient that receives a copy of a message without other recipients being made aware).

ADRLIST structures can hold both resolved and unresolved recipient entries. An *unresolved entry* does not have a PR_ENTRYID property. Applications that enable users to type recipient names directly into a message, in addition to choosing names from a predetermined list, create unresolved entries by creating an ADRENTRY with just the PR_DISPLAY_NAME and the PR_RECIPIENT_TYPE properties. A *resolved entry* will at least contain the following properties: PR_ENTRYID, PR_RECIPIENT_TYPE, PR_DISPLAY_NAME, PR_ADDRTYPE, and PR_DISPLAY_TYPE.

Your 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 is passed in by the calling application, or if NULL is passed in the *lppAdrList* parameter on input, then **Address** allocates a larger buffer for the **ADRLIST** structure it returns using **MAPIAllocateBuffer** and returns this buffer's address in the *lppAdrList* parameter. **Address** frees the old buffer by calling the **MAPIFreeBuffer** function.

Each **SPropValue** property value structure is also 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 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

[New - Windows 95]

Compares two entry identifiers to determine if they refer to the same service provider object.

Syntax

HRESULT CompareEntryIDs(ULONG *cbEntryID1*, **LPENTRYID** *lpEntryID1*, **ULONG** *cbEntryID2*, **LPENTRYID** *lpEntryID2*, **ULONG** *ulFlags*, **ULONG FAR *** *lpulResult*)

Parameters

cbEntryID1
: Input parameter containing the number of bytes in 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 number of bytes in 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 Boolean variable that receives the result of the comparison; 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.

Comments

Use the **IMAPISupport::CompareEntryIDs** method to compare two entry identifiers for a given service provider object and 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 provider should make no assumptions about the comparison and should take the most conservative approach based on what your provider is trying to do. **CompareEntryIDs** can fail if, for example, no provider has registered for one of the entry identifiers. If your provider compares message-store entry identifiers when one or both of the stores has not yet opened, **CompareEntryIDs** returns the value MAPI_E_UNKNOWN_ENTRYID.

IMAPISupport::CompleteMsg

[New - Windows 95]

Calls the MAPI spooler to complete message delivery processing. Used only by message store providers that are tightly coupled with transport providers.

Syntax

HRESULT CompleteMsg(**ULONG** *ulFlags*, **ULONG** *cbEntryID*, **LPENTRYID** *lpEntryID*)

Parameters

ulFlags
Reserved; must be zero.

cbEntryID
Input parameter containing the number of bytes in the entry identifier pointed to by the *lpEntryID* parameter.

lpEntryID
Input parameter pointing to the entry identifier of the message that needs to be delivered.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

Comments

Message store providers use the **IMAPISupport::CompleteMsg** method to instruct the MAPI spooler to complete the post-processing of a message. **CompleteMsg** is available to all message store providers, but should only be called by those stores that are tightly coupled with transport providers, and even then only in the following circumstances:

- When the message in question required preprocessing.
- When the message store provider can handle all recipients, the transport provider the message store is coupled with can handle all recipients of this message without involving the MAPI spooler, and the MAPI spooler's transport order indicates that the transport can handle all of the recipients.

IMAPISupport::CopyFolder

[New - Windows 95]

Copies or moves a folder to a destination folder.

Syntax

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 of the source folder.

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 number of bytes in the entry identifier pointed to by the *lpEntryID* parameter.

lpEntryID
Input parameter pointing to the entry identifier of the folder to be copied or moved.

lpInterface
Reserved; must be NULL.

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 a string containing the name to be given to the newly created or moved folder. If your 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 the dialog box is modal to. The *ulUIParam* parameter is ignored unless your 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 the *lpProgress* parameter, the progress information is provided by MAPI. The *lpProgress* parameter is ignored unless the FOLDER_DIALOG flag is set in the *ulFlags* parameter.

ulFlags
Input parameter containing a bitmask of flags used to control how the copy or move operation is accomplished. The following flags can be set:

COPY_SUBFOLDERS
Indicates that all subfolders are included in the copy operation. This is optional for copy operations and is implied for move operations.

FOLDER_DIALOG
Displays a progress-information user interface while the operation proceeds.

FOLDER_MOVE
Indicates that the folder is to be moved. If this flag is not set, the folder will be copied.

MAPI_UNICODE
Indicates the passed-in strings are in Unicode format. If the MAPI_UNICODE flag is not set, the

strings are in 8-bit 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 is the same as the source folder. The operation stops without completing.

MAPI_W_PARTIAL_COMPLETION
The call succeeded, but not everything was copied or moved.

Comments

Message store providers use 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 folder being copied or moved 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 the subfolders of the affected folder. To copy or move all subfolders nested within the copied or moved folder, your application passes the COPY_SUBFOLDERS flag in the *ulFlags* parameter.

To allow copy or move operations involving more than one folder to continue even if one or more folders specified for the operation do not exist or have already been moved elsewhere and thus cannot be copied or moved, a message store provider should attempt to 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 the value S_OK. If one or more folders cannot be copied or moved, **CopyFolder** returns the value 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 application cannot function on the assumption that an error return implies that no work was done.

If an entry identifier for a folder that doesn't exist is passed in the *lpEntryID* parameter, **CopyFolder** returns MAPI_W_PARTIAL_COMPLETION.

IMAPISupport::CopyMessages

[New - Windows 95]

Copies or moves messages from the current folder to another folder.

Syntax

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 of the source folder from which messages are being copied or moved.

lpSrcFolder
Input parameter pointing to the source folder from which messages are being copied or moved.

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 interface identifier for the destination folder to which messages are being copied or moved.

lpDestFolder
Input parameter pointing to the open destination folder where the messages identified in the *lpMsgList* parameter are to be copied or moved.

ulUIParam
Input parameter containing the handle of the window that the dialog box is modal to. The *ulUIParam* parameter is ignored unless your 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 the *lpProgress* parameter, the progress information is provided by MAPI. The *lpProgress* parameter is ignored unless the MESSAGE_DIALOG flag is set in the *ulFlags* parameter.

ulFlags
Input parameter containing a bitmask of flags used to control how the copy or move operation is accomplished. The following flags can be set:

MESSAGE_DIALOG
Displays a progress-information user interface as the operation proceeds.

MESSAGE_MOVE
Moves messages. If this flag is not set, the function 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 clicking the Cancel button in a dialog box.

Comments

Message store providers use the **IMAPISupport::CopyMessages** method to call MAPI to move or copy messages from one folder to another as specified by the 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

[New - Windows 95]

Creates an entry identifier for a custom recipient.

Syntax

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. *lpszName* can be NULL.

lpszAdrType
Input parameter pointing to a string containing the e-mail address type of the recipient; examples of e-mail address types are FAX, SMTP, X500, and so on. *lpszAdrType* cannot be NULL.

lpszAddress
Input parameter pointing to a string containing the e-mail address of the recipient. *lpszAddress* cannot be NULL.

ulFlags
Input parameter containing a bitmask of flags controlling 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 8-bit format.

lpcbEntryID
Output parameter pointing to a variable in which the **IMAPISupport::CreateOneOff** method returns the number of bytes in the entry identifier pointed to by the *lppEntryID* parameter.

lppEntryID
Output parameter pointing to a variable where the pointer to the entry identifier of the custom recipient address that has been created is stored.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

Comments

Service providers use the **IMAPISupport::CreateOneOff** method to create an entry identifier for a custom recipient. A *custom recipient* is a message recipient that the user does not choose from an address book but specifies by typing in the recipient name. This entry identifier can be used as a valid recipient on a message.

When the provider is done using the entry identifier returned by **CreateOneOff**, it should free the allocated memory by using the **MAPIFreeBuffer** function.

See Also

MAPIFreeBuffer function

IMAPISupport::Details

[New - Windows 95]

Displays a details dialog box on a particular entry in an address book.

Syntax

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 to the parent window for the dialog box.

lpfnDismiss
Input parameter pointing to the address of a function based on the **DISMISSMODELESS** function prototype that is called when the modeless variety of the details dialog box is dismissed.

lpvDismissContext
Input parameter that is passed to the function specified by the *lpfnDismiss* parameter.

cbEntryID
Input parameter containing the number of bytes in the *lpEntryID* parameter.

lpEntryID
Input parameter pointing to the entry identifier for which the object's details are displayed.

lpfButtonCallback
Input parameter pointing to a pointer to a button callback function that adds a button to a dialog box.

lpvButtonContext
Input parameter pointing to data used as a parameter for the button callback function pointed to by the *lpfButtonCallback* parameter.

lpszButtonText
Input parameter pointing to a string containing the text to be applied to the button mentioned previously 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 controlling 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 8-bit format.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

Comments

Providers use the **IMAPISupport::Details** methods to display a details dialog box on a particular entry in an address book. The *lpfButtonCallback*, *lpvButtonContext*, and *lpButtonText* parameters can be used to add a button it has defined to the addressing dialog box. When the button is chosen, MAPI calls the callback function pointed to by *lpfButtonCallback*, passing both the entry identifier of the chosen button and the data in *lpvButtonContext*. If an extensible button is not needed, *lpszButtonText* should be NULL.

The *lpfnDismiss* parameter is used to support the modeless version of the details address book. When the user dismisses the address book, MAPI calls **IMAPISupport::Details**. The dismiss function used with **Details** is based on the function prototype **DISMISSMODELESS**.

See Also

DISMISSMODELESS function prototype, **IMAPISupport::Address** method

IMAPISupport::Details, LPFNBUTTON

[New - Windows 95]

The **LPFNBUTTON** function prototype represents a service provider callback function that MAPI calls to hook to an optional button control on an address book dialog box. This button is typically a **Details** button.

Syntax

SCODE (STDMETHODCALLTYPE FAR * LPFNBUTTON)(ULONG *ulUIParam*, **LPVOID** *lpvContext*, **ULONG** *cbEntryID*, **LPENTRYID** *lpSelection*, **ULONG** *ulFlags***)**

Parameters

ulUIParam
Input parameter specifying an implementation-specific 32-bit value normally 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 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. Typically, for C++ code, *lpvContext* represents a pointer to the address of a C++ object.

cbEntryID
Input parameter specifying the size, in bytes, of the identifier indicated by *lpSelection*.

lpSelection
Input parameter specifying a pointer to an **ENTRYID** structure 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.

Comments

This hook function can be used by a service provider to define a button on the details dialog box. The provider passes a pointer to the callback function in calls to **IMAPISupport::Details**. When the dialog box is displayed and the user selects the defined button, MAPI calls this function.

See Also

ENTRYID structure, **IMAPISupport::Details** method

IMAPISupport::DoConfigPropsheet

[New - Windows 95]

Enables service providers to display property sheets using MAPI's default user interface. Primarily used for configuration.

Syntax

HRESULT DoConfigPropsheet(**ULONG** *ulUIParam*, **ULONG** *ulFlags*, **LPTSTR** *lpszTitle*,
LPMAPITABLE *lpDisplayTable*, **LPMAPIPROP** *lpConfigData*, **ULONG** *ulTopPage*)

Parameters

ulUIParam
Input parameter containing the handle of the window that the property sheets are modal to.

ulFlags
Reserved; must be zero.

lpszTitle
Input parameter pointing to the string that contains the title of the property-sheet dialog box.

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 that are used to build the display table in 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.

Comments

Service providers use the **IMAPISupport::DoConfigPropSheet** method to display configuration property sheets. **DoConfigPropSheet** can be called as part of the implementation of **IMAPIStatus::SettingsDialog** or from a button used to display details on properties. It can also be called from the entry point for a messaging service.

The display table that is passed in the *lpDisplayTable* parameter can be built using the **BuildDisplayTable** function, or using any other method convenient for the provider.

Microsoft strongly recommends that service providers use property sheets for their configuration user interface so that users can 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

[New - Windows 95]

Copies or moves a selected set of properties from the source object to a given destination object. (The source object is the object on which the call to the **IMAPISupport::DoCopyProps** method is made.)

Syntax

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 of the source object from which the properties are being copied or moved.

lpSrcObj
Input parameter pointing to the source object from which the properties are being copied or moved.

lpIncludeProps
Input parameter pointing to an **SPropTagArray** structure holding a counted array of property tags indicating the properties to be copied or moved. The *lpIncludeProps* parameter value cannot be NULL.

ulUIParam
Input parameter containing the handle of the window the progress-information user interface is modal to.

lpProgress
Input parameter pointing to a progress object that contains client- or provider-supplied progress information. If NULL is sent in the *lpProgress* parameter, the progress information is provided by MAPI. The *lpProgress* parameter is ignored unless the MAPI_DIALOG flag is set in the *ulFlags* parameter.

lpDestInterface
Input parameter pointing to the interface identifier for the destination object to which properties are being copied or moved.

lpDestObj
Input parameter pointing to the open destination object.

ulFlags
Input parameter containing a bitmask of flags used to control how the copy or move operation is to be performed. The following flags can be set:

MAPI_DIALOG
Displays a dialog box to provide status or prompt the user for additional information, such as recipients, sending options, or name resolution. If this flag is not set, no dialog box is displayed.

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 a **SPropProblemArray** structure is stored. If a value of 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_DECLINE_COPY
: Indicates that the provider has chosen not to implement the copy operation.

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 object might be partially modified.

MAPI_E_INTERFACE_NOT_SUPPORTED
: An appropriate interface cannot be obtained.

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.

The following values can be returned in the **SPropProblemArray** structure, but not as a return value 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 is not a severe error; allow the process to continue.

MAPI_E_INVALID_TYPE
: The property type is invalid.

MAPI_E_UNEXPECTED_TYPE
: The type of the property value is not the type expected by the calling application.

Comments

Message store providers use the **IMAPISupport::DoCopyProps** method to copy or move to the destination object those properties designated in the **SPropTagArray** structure passed in the *lpIncludeProps* parameter that are present in the source object. When copying properties between like objects, for example between two message objects, the interface identifiers and the object types must be the same for both the source and the destination object. 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 by copied or moved data 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, respectively. For folder objects, the PR_CONTAINER_HIERARCHY, PR_CONTAINER_CONTENTS, and PR_FOLDER_ASSOCIATED_CONTENTS properties can be included in the *lpIncludeProps* **SPropTagArray** structure to permit copying or moving of subfolders, messages, or associated objects from a source folder to a destination object. If subfolders are copied or moved, they are copied or moved in their entirety, regardless of the use of properties indicated by the **SPropTagArray** in the source object itself.

If the source object directly or indirectly contains the destination object, the overall call fails and returns the value MAPI_E_FOLDER_CYCLE. Some implementations, however, perform significant work before discovering this error and leave the source and destination objects partially modified, so applications should try to avoid such calls. If the same pointer is used for both the source and destination objects, MAPI_E_NO_ACCESS is returned.

The interface of the destination object, indicated in the *lpInterface* parameter, is usually exactly the same interface as that for the source object. If *lpInterface* is set to NULL, then the value MAPI_E_INVALID_PARAMETER is returned for **DoCopyProps**. If an acceptable interface is given in the *lpInterface* parameter but an invalid pointer is passed in the *lpDestObj* parameter, the results are unpredictable, with the most likely result being that the calling application stops.

If the MAPI_DIALOG flag is not set in the *ulFlags* parameter, then **DoCopyProps** ignores the *ulUIParam* and *lpProgress* parameters and no progress-information user interface is provided. If a client application sets the MAPI_DIALOG flag in the *ulFlags* parameter and passes a value of NULL in the *lpProgress* parameter, then the provider is responsible for generating a progress-information user interface. If a client application sets the MAPI_DIALOG flag in the *ulFlags* parameter and passes a progress object in the *lpProgress* parameter, 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 of the selected properties, the value S_OK is returned and an **SPropProblemArray** structure is returned 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, exactly 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, the PR_MESSAGE_RECIPIENTS or PR_MESSAGE_ATTACHMENTS property is returned in the **SPropProblemArray** structure.

The value returned in the *lppProblems* parameter 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 **IMAPISupport::GetLastError** to get the **MAPIERROR** structure for the error.

The calling application must free the returned **SPropProblemArray** structure by calling the **MAPIFreeBuffer** function, but this should only be done if **DoCopyProps** returns with S_OK.

See Also

IMAPISupport::CopyMessages method, **IMAPISupport::DoCopyTo** method, **SPropProblemArray** structure, **SPropTagArray** structure

IMAPISupport::DoCopyTo

[New - Windows 95]

Copies or moves all properties of the source object to a destination object, except for a given set of properties that are excluded from the copy or move operation. (The source object is the object on which the call to the **IMAPISupport::DoCopyTo** method is made.)

Syntax

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 of the source object from which properties and objects are being copied or moved.

lpSrcObj
Input parameter pointing to the source object from which the properties and objects are being copied or moved.

ciidExclude
Input parameter containing the number of interfaces to be excluded when copying or moving properties.

rgiidExclude
Input parameter containing an array of interface identifiers indicating interfaces that should not be used when copying or moving supplemental information to the destination object.

lpExcludeProps
Input parameter pointing to the **SPropTagArray** structure containing the property identifiers of the properties that should not be copied or moved to the destination object. Passing a value of NULL in the *lpExcludeProps* parameter indicates all properties are to be copied or moved. Passing zero in the **cValues** member of the *lpExcludeProps* **SPropTagArray** structure results in the value MAPI_E_INVALID_PARAMETER being returned.

ulUIParam
Input parameter containing the handle of the window the dialog box is modal to. In this case it is used to indicate the user interface where progress information is displayed.

lpProgress
Input parameter pointing to a progress object that contains client- or provider-supplied progress information. If NULL is sent in the *lpProgress* parameter, 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 interface identifier for the interface of the destination object to which properties are being copied.

lpDestObj
Input parameter pointing to the open destination object.

ulFlags
Input parameter containing a bitmask of flags used to control how the copy or move operation is to be performed. The following flags can be set:

MAPI_DECLINE_OK
Used by MAPI to limit recursion within MAPI's **CopyTo** implementation. Informs the provider that if it chooses not to implement **IMAPISupport::DoCopyTo**, that it should immediately return

MAPI_E_DECLINE_COPY.

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 a **SPropProblemArray** structure is stored. If a value of 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_DECLINE_COPY
Indicates that the provider has chosen not to implement the copy operation.

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 object might be partially modified.

MAPI_E_INTERFACE_NOT_SUPPORTED
An appropriate interface cannot be obtained.

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.

The following values can be returned in the **SPropProblemArray** structure, but not as a return value 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 is not a severe error; allow the process to continue.

MAPI_E_INVALID_TYPE
The property type is invalid.

MAPI_E_UNEXPECTED_TYPE
The type of the property value is not the type expected by the calling application.

Comments

Message store providers use 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 by copied or moved data 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 being copied or moved to the destination object. For example, passing in a value of 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, respectively. For folder objects, the PR_CONTAINER_HIERARCHY, PR_CONTAINER_CONTENTS, and PR_FOLDER_ASSOCIATED_CONTENTS properties can be included in the *lpExcludeProps* **SPropTagArray** structure to prevent copying or moving of subfolders, messages, or associated objects from the source folder to the destination object. If subfolders are copied or moved, they are copied or moved in their entirety, regardless of the use of properties indicated by the **SPropTagArray** in the source object itself.

Providers implementing **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.

If the source object directly or indirectly contains the destination object, the overall call fails and returns the value MAPI_E_FOLDER_CYCLE. Some implementations, however, perform significant work before discovering this error and leave the source and destination objects partially modified, so applications should try to avoid such calls. If the same pointer is used for both the source and destination objects, MAPI_E_NO_ACCESS is returned.

The interface of the destination object, indicated in the *lpInterface* parameter, is usually exactly the same interface as that for the source object. If *lpInterface* is set to NULL, then the value MAPI_E_INVALID_PARAMETER is returned for **DoCopyTo**. If an acceptable interface is given in the *lpInterface* parameter but an invalid pointer is passed in the *lpDestObj* parameter, the results are unpredictable, with the most likely result being that the calling application stops.

Some objects contain supplemental information beyond that, which can be accessed with the interface pointer in the *lpInterface* parameter. To copy or move such information, a provider's **IMAPISupport::DoCopyTo** implementation first uses **IUnknown::QueryInterface** on the destination object to see if it can accept the extra data. Conversely, if the calling application is aware of supplemental information and requires that **DoCopyTo** not copy or move it, the application can specify in the array passed in the *rgiidExclude* parameter interface identifiers for the properties that **DoCopyTo** should not copy or move. For example, if the application wants to copy messages, but not embedded objects within the messages, it can pass IID_IMessage in the exclude array. **DoCopyTo** ignores any interfaces listed in *rgiidExclude* that 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 **IMAPIProp** also excludes **IMAPIFolder**, **IMessage**, **IAttach**, 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 the *ulFlags* parameter, then **DoCopyTo** ignores the *ulUIParam* and *lpProgress* parameters and no progress-information user interface is provided. If a client application sets the MAPI_DIALOG flag in the *ulFlags* parameter and passes a value of NULL in the *lpProgress* parameter, then the provider is responsible for generating a progress-information user interface. If a client application sets the MAPI_DIALOG flag in the *ulFlags* parameter and passes a progress object in the *lpProgress* parameter, 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 of the selected properties, the value S_OK is returned and an **SPropProblemArray** structure is returned 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, exactly 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, the PR_MESSAGE_RECIPIENTS or PR_MESSAGE_ATTACHMENTS property is returned in the **SPropProblemArray** structure.

The value returned in the *lppProblems* parameter 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 **IMAPISupport::GetLastError** to get the **MAPIERROR** structure for the error.

If an error occurs on the **DoCopyTo** call, do not use or free the SPropProblemArray structure. Applications should ignore the **ulIndex** member in property problem sets returned by **DoCopyTo**.

The calling application must free the returned **SPropProblemArray** structure by calling the **MAPIFreeBuffer** function, but this should only be done if **DoCopyTo** returns with S_OK.

See Also

IMAPISupport::CopyFolder method, **IMAPISupport::CopyMessages** method, **SPropProblemArray** structure, **SPropTagArray** structure

IMAPISupport::DoProgressDialog

[New - Windows 95]

Provides a default implementation of the **IMAPIProgress** interface for use by service providers.

Syntax

HRESULT DoProgressDialog(**ULONG** *ulUIParam*, **ULONG** *ulFlags*, **LPMAPIPROGRESS FAR *** *lppProgress*)

Parameters

ulUIParam
: Input parameter containing the handle of the window the dialog box is modal to.

ulFlags
: Input parameter containing a bitmask of flags indicating how progress information is to be calculated. The following flag can be set:

  MAPI_TOP_LEVEL
  : Uses the values in the **IMAPIProgress::Progress** method's *ulCount* (the item being operated on) and *ulTotal* (total items to operate on) parameters to increment progress being made on the operation. For example, if you were copying a folder that contained 10 subfolders and the MAPI_TOP_LEVEL flag is set in the *lpulFlags* parameter, progress will increment 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.

Comments

Message store providers use 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

[New - Windows 95]

Generates a sentmail message for an open message. This method is only valid when called from within the MAPI spooler's process.

Syntax

HRESULT DoSentMail(**ULONG** *ulFlags*, **LPMESSAGE** *lpMessage*)

Parameters

ulFlags
Reserved; must be zero.

lpMessage
Input parameter pointing to the open message for which a sent-mail message should be generated.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

Comments

Message store providers call the **IMAPISupport::DoSentMail** method as part of the implementation of the **IMsgStore::FinishedMsg** method. **FinishedMsg** is called by the MAPI spooler when it has finished processing the message. **DoSentMail** should be one of the last lines of code in a message store's **FinishedMsg** implementation. **FinishedMsg** should unlock the message and only have one reference count to the message when this **DoSentMail** call is made. After **DoSentMail** returns, the message will have been fully released.

The MAPI spooler's implementation of **DoSentMail** does the following:

- Determines whether the message is to be deleted after sending by checking the message for the PR_DELETE_AFTER_SUBMIT property.
- Determines the location of the sent-mail folder.
- Initiates message hook processing for any hooks set on the sent-mail folder.
- Moves the message to the sent-mail folder, deleted-items folder, or to another folder as specified by any message hooks.
- Releases the message.

See Also

IMsgStore::FinishedMsg method

IMAPISupport::ExpandRecips

[New - Windows 95]

Completes all recipient lists and expands certain distribution lists for the transmission of a message.

Syntax

HRESULT ExpandRecips(**LPMESSAGE** *lpMessage*, **ULONG FAR** * *lpulFlags*)

Parameters

lpMessage
Input parameter pointing to an open message object that has read-write access.

lpulFlags
Output parameter pointing to a variable that receives a bitmask of flags controlling what occurs after recipient entries are resolved. The following flag can be set:

NEEDS_PREPROCESSING
Indicates the message needs preprocessing before sending.

NEEDS_SPOOLER
Indicates that the messages store should have the MAPI spooler send the message rather than the message store's tightly coupled transport.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

Comments

Message store providers use the **IMAPISupport::ExpandRecips** method to prompt MAPI to resolve 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 changed display names (for example, 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 e-mail 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, and if it does, sets the NEEDS_PREPROCESSING flag and returns it 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

[New - Windows 95]

Returns a **MAPIERROR** structure containing information about the last error that occurred for the support object.

Syntax

HRESULT GetLastError(**HRESULT** *hResult*, **ULONG** *ulFlags*, **LPMAPIERROR FAR** * *lppMAPIError*)

Parameters

hResult
: Input parameter containing the result returned for the last call on the session object that returned an error.

ulFlags
: Input parameter containing a bitmask of flags controlling the type of the returned strings. The following flag can be set:

  MAPI_UNICODE
  : Indicates the strings returned 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 8-bit 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.

Comments

Use the **IMAPISupport::GetLastError** method to retrieve information to display as a message to the user regarding the last error returned from a method call on the support object.

To release all the memory allocated by MAPI, providers need only call the **MAPIFreeBuffer** function for the **MAPIERROR** structure.

The return value from **GetLastError** must be S_OK for the provider to make use of the **MAPIERROR** structure. Even if the return value is S_OK, it is still possible that a **MAPIERROR** structure won't be returned. If the implementation cannot determine what the last error was, or if a MAPIERROR structure is not available for that error, a pointer to NULL is returned in the *lppMAPIError* parameter instead.

See Also

MAPIERROR structure, **MAPIFreeBuffer** function

IMAPISupport::GetMemAllocRoutines

[New - Windows 95]

Retrieves the addresses of the three MAPI memory allocation functions, **MAPIAllocateBuffer**, **MAPIAllocateMore**, and **MAPIFreeBuffer**.

Syntax

HRESULT GetMemAllocRoutines(**LPALLOCATEBUFFER FAR** * *lppAllocateBuffer*,
  LPALLOCATEMORE FAR * *lppAllocateMore*, **LPFREEBUFFER FAR** * *lppFreeBuffer*)

Parameters

lppAllocateBuffer
  Output parameter pointing to a variable where the pointer to the **MAPIAllocateBuffer** function is stored.

lppAllocateMore
  Output parameter pointing to a variable where the pointer to the **MAPIAllocateMore** function is stored.

lppFreeBuffer
  Output parameter pointing to a variable where the pointer to the **MAPIFreeBuffer** function is stored.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

Comments

Service providers use the **IMAPISupport::GetMemAllocRoutines** method to get the addresses of the three memory allocation functions passed in on your provider's initialization call. For the syntax of each function, see its reference entry.

See Also

MAPIAllocateBuffer function, **MAPIAllocateMore** function, **MAPIFreeBuffer** function

IMAPISupport::GetOneOffTable

[New - Windows 95]

Returns the table of custom recipient address templates that can be created onto a message.

Syntax

HRESULT GetOneOffTable(**ULONG** *ulFlags*, **LPMAPITABLE FAR *** *lppTable*)

Parameters

ulFlags
: Input parameter containing a bitmask of flags controlling the format of returned data. 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 8-bit 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.

Comments

Address book providers use 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. A custom recipient address, which is typed by the user, can be used once to send an individual message. 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
: Indicates the custom template name, such as Microsoft Fax Template or Internet Form.

PR_ENTRYID
: Indicates the entry identifier used to identify a particular template.

PR_DEPTH
: Indicates how to indent the display name of the template; its value is a LONG that is zero-based, with zero meaning no indentation.

PR_SELECTABLE
: A bitmask of selection flags indicating how the entry can be used.

PR_ADDRTYPE
: Indicates the custom recipient address types that are added to your provider's custom recipient 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 custom recipient table by the **IMAPITable::QueryColumns** method. The initial active columns for a custom recipient table are those columns the **QueryColumns** method returns before the application that contains the custom recipient 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 application that contains the custom recipient table calls the **IMAPITable::SetColumns** method.

- Controls the property types of the sort order specification returned by the **IMAPITable::QuerySortOrder** method before the application that contains the custom recipient table calls the **IMAPITable::SortTable** method.

If your provider registers for notification of changes to its custom recipient table, it will also receive notifications of changes made to other provider's custom recipient tables. Your provider can then 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

[New - Windows 95]

Creates a new support object for use by calls to a messaging service provider's configuration entry point function.

Syntax

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.

Comments

Use the **IMAPISupport::GetSvcConfigSupportObj** method to acquire a pointer to a support object that can be used for a messaging 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 messaging services to configure themselves or perform other actions when an application changes the profile. The primary action of such a function is to furnish a dialog box in which the user can change messaging service settings. A **MSGSERVICEENTRY**-based function can also support configuration by a client application through a property value array passed through the **IMsgServiceAdmin::ConfigureMsgService** method.

See Also

IMsgServiceAdmin::CreateMsgService method, **IProfAdmin : IUnknown** interface, **MSGSERVICEENTRY** function prototype

IMAPISupport::IStorageFromStream

[New - Windows 95]

Implements an IStorage object on an IStream object.

Syntax

HRESULT IStorageFromStream(**LPUNKNOWN** *lpUnkIn*, **LPCIID** *lpInterface*, **ULONG** *ulFlags*,
LPSTORAGE FAR * *lppStorageOut*)

Parameters

lpUnkIn
Input parameter pointing to the IUnknown object for the object that implements the stream object.

lpInterface
Input parameter pointing to the interface identifier 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 controlling how the storage is to be created relative to the stream; the default is that the storage has read access and occurs within the stream starting at position zero. The following flags can be set:

STGSTRM_CREATE
Creates a new storage for the stream object.

STGSTRM_CURRENT
Starts storage at the current position of the stream.

STGSTRM_MODIFY
Allows the calling implementation 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 IStorage object is stored.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

Comments

Providers that don't implement the OLE **IStorage** interface themselves use 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, the provider first opens an OLE stream object with read-write access for the property, then internally marks the property stream as an IStorage object, generates an **IStorage** interface from the support function, and returns the MAPI-generated **IStorage** interface to the calling application.

A newly created IStorage 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 IStorage 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 IStorage object from one container to another, but they must pass IID_IStreamDocfile in the **OpenProperty** method's *lpInterface* parameter. Messages 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 *OLE Programmer's Reference, Volume One* and *Volume Two.*

See Also

IMAPIProp::OpenProperty method

IMAPISupport::MakeInvalid

[New - Windows 95]

Invalidates an object derived from the **IUnknown** interface.

Syntax

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. Its interface must be derived from **IUnknown**.

ulRefCount
Input parameter indicating the object's present reference count.

cMethods
Input parameter indicating the number of methods in the object's vtable (that is, the table used for dispatching calls to object methods).

Return Values

S_OK
The call succeeded and has returned the expected value or values.

Comments

Service providers use the **IMAPISupport::MakeInvalid** method to invalidate an object, typically when a provider is shutting down. But 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 Your application 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 is the usual result of 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.

Providers can call **MakeInvalid** and then perform any shutdown work, such as discarding associated data structures, that it would normally do if it were to release the object. Code to support the object does not need to 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 forget about the invalidated object. Client applications are required to release any memory associated with the object.

See Also

MAPIAllocateBuffer function

IMAPISupport::ModifyProfile

[New - Windows 95]

Makes a message store provider's profile section permanent.

Syntax

HRESULT ModifyProfile(**ULONG** *ulFlags*)

Parameters

ulFlags

Input parameter containing a bitmask of flags controlling how the store logs onto the profile. The following flags can be set:

MDB_TEMPORARY

Advises MAPI that the store is temporary and should not be added to the message stores information table. S_OK is returned immediately.

Return Values

S_OK

The call succeeded and has returned the expected value or values.

Comments

Message store providers use 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 resource to be listed in the table returned by a call to the **IMAPISession::GetMsgStoresTable** method and makes it possible to open the resource 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

[New - Windows 95]

Creates the status table column values for the service providers.

Syntax

HRESULT ModifyStatusRow(**ULONG** *cValues*, **LPSPropValue** *lpColumnVals*, **ULONG** *ulFlags*)

Parameters

cValues
: Input parameter containing the number of columns in the status row to be passed to MAPI.

lpColumnVals
: Input parameter pointing to an **SPropValue** structure containing the property values used to define the columns in the status row to be passed to MAPI.

ulFlags
: Input parameter containing a bitmask of flags controlling how information defining the status row is to be processed. The following flag can be set:

  STATUSROW_UPDATE
  : Directs MAPI to merge the service providers' status information.

Return Values

S_OK
: The call succeeded and has returned the expected value or values.

Comments

Service providers call the **IMAPISupport::ModifyStatusRow** method to enter a row in the status table at logon. Providers should call **IMAPISupport::ModifyStatusRow** during their logon process. The STATUSROW_UPDATE flag should be set with each subsequent **ModifyStatusRow** call. 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 column values in the *lpColumnVals* parameter:

- PR_PROVIDER_DISPLAY
- PR_RESOURCE_METHODS
- PR_STATUS_CODE

The following column values are required if the service provider lists its user identity in the status table:

- PR_IDENTITY_ENTRYID
- PR_IDENTITY_DISPLAY
- PR_IDENTITY_SEARCH_KEY

The following column values are provided by the messaging subsystem; the provider should not pass them in *lpColumnVals*:

- PR_ENTRYID
- PR_OBJECT_TYPE
- PR_PROVIDER_DLL_NAME
- PR_RESOURCE_FLAGS

- PR_RESOURCE_TYPE
- PR_ROWID

The following optional column values can be passed by service providers 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 applications can retrieve them by opening the provider's status object.

IMAPISupport::NewEntry

[New - Windows 95]

Displays a dialog box for creating new entries within containers or custom recipient addresses within messages.

Syntax

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 to the window that the dialog box is modal to.

ulFlags
: Reserved; must be zero.

cbEIDContainer
: Input parameter containing the number of bytes in the entry identifier in the *lpEIDContainer* parameter.

lpEIDContainer
: Input parameter pointing to the container where the new custom recipient address will be added. If the value in *cbEIDContainer* is zero, **NewEntry** returns a recipient entry identifier and a list of templates as if your provider called **IMAPISupport::CreateOneOff**.

cbEIDNewEntryTpl
: Input parameter containing the number of bytes in the entry identifier in the *lpEIDNewEntryTpl* parameter.

lpEIDNewEntryTpl
: Input parameter pointing to a template to be used to create the new custom recipient address. If the value in *cbEIDNewEntryTpl* is zero, passing NULL in the *lpEIDNewEntryTpl* parameter displays a dialog box enabling the user to select an address-creation template from among the address types that can be created in the container.

lpcbEIDNewEntry
: Output parameter pointing to a variable containing the number of bytes in the entry identifier in the *lppEIDNewEntry* parameter.

lppEIDNewEntry
: Output parameter pointing to a variable where the pointer to the entry identifier for the new custom recipient address is stored.

Return Values

S_OK
: The call succeeded and has returned the expected value or values.

Comments

The **IMAPISupport::NewEntry** method can be used in several different modes. To add a custom recipient address directly to the open message and not to a modifiable container, your provider passes zero in the *cbEIDContainer* parameter and NULL in the *lpEIDContainer* parameter. To display a dialog box enabling the user to select a template for adding custom recipients to a modifiable container, your provider passes zero in the *cbEIDNewEntryTpl* parameter and NULL in the *lpEIDNewEntryTpl* parameter.

To create an entry in a modifiable address book and not get the entry identifier back, pass 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, your 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. Doing so opens the root folder of the address book container. Next, your provider calls the **IABContainer::OpenProperty** method and passes the PR_CREATE_TEMPLATES property in the *ulPropTag* parameter so that it can open the property. Doing this returns a table object that lists the types of objects that can be created in the address book container. Your provider finds in this table the entry identifier for the template you want new entries to be created with. Then, your provider calls **NewEntry** and passes NULL in the *lpEIDContainer* parameter and the entry identifier for the entry-creation template you want used in the *lpEIDNewEntryTpl* parameter.

Calls made to **NewEntry** return the entry identifier of the new custom recipient address in the *lppEIDNewEntry* parameter (unless your provider passed NULL in *lppEIDNewEntry*). Your 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

[New - Windows 95]

Returns a new, unique MAPI identifier (a MAPIUID) for an item.

Syntax

HRESULT NewUID(**LPMAPIUID** *lpMuid*)

Parameters

lpMuid
Points to the 16 bytes of memory where the new **MAPIUID** structure will be placed.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

Comments

Service providers use 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 your 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. Use the UUIDGEN.EXE utility program to generate the MAPIUID.

See Also

MAPIUID structure

IMAPISupport::Notify

[New - Windows 95]

Notifies interested applications about changes to an object that the provider owns.

Syntax

HRESULT Notify(**LPNOTIFKEY** *lpKey*, **ULONG** *cNotification*, **LPNOTIFICATION** *lpNotifications*, **ULONG FAR** * *lpulFlags*)

Parameters

lpKey
Input parameter pointing to a key for the object for which 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 used to control the notification. On input, the following flag can be set by the provider:

MAPI_UNICODE
Indicates the passed-in strings are in Unicode format. If the MAPI_UNICODE flag is not set, the strings are in 8-bit format.

On output, the following flag can be set by the MAPI notification engine:

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.

Comments

Service providers use the **IMAPISupport::Notify** method to call MAPI's notification engine. The notification engine is responsible for copying the structures indicated by the *lpNotifications* parameter into memory that is accessible to the process that requires notifications before calling the callback function that broadcasts the notifications, and for releasing that memory. 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 the reference entry for the **NOTIFICATION** structure.

The notification key passed for the object whose notification status is reported in **Notify**'s *lpKey* parameter should be the same as the key formerly passed for that object in the *lpKey* of the **IMAPISupport::Subscribe** method that set up notification subscription. This key is similar to PR_RECORD_KEY in that it is binary-comparable and can be used to ensure that the correct object is used. The notification engine 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 this object to provide an entry identifier useful to all applications registered for notifications.

When a provider instructs the notification engine to broadcast a notification held in an **ERROR_NOTIFICATION** or **NEWMAIL_NOTIFICATION** structure and the error or new message string indicated by 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 8-bit format, the provider should not set MAPI_UNICODE. The MAPI_UNICODE flag only applies to the strings within the notification structures.

For asynchronous notifications, **Notify** returns before the callback functions are made to the application registered for notifications. For synchronous notifications, **Notify** makes the callbacks before returning so the calling implementation can determine whether a process is active by checking whether the NOTIFY_CANCELED flag is set. If any callback function has returned CALLBACK_DISCONTINUE, the MAPI notification engine 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.

See Also

IMAPISupport::Subscribe method, **IMAPISupport::UnSubscribe** method, **NOTIFICATION** structure, **NOTIFKEY** structure, PR_RECORD_KEY property

IMAPISupport::OpenAddressBook

[New - Windows 95]

Opens the address book and returns a pointer that provides further access.

Syntax

HRESULT OpenAddressBook(**LPCIID** *lpInterface*, **ULONG** *ulFlags*, **LPADRBOOK FAR *** *lppAdrBook*)

Parameters

lpInterface
Input parameter pointing to the interface identifier for the address book. Passing NULL in the *lpInterface* parameter indicates that the return value is cast to the standard interface for the address book. The *lpInterface* parameter can also be set to an 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 of the address book providers could not be loaded. Use the HR_FAILED macro to test for this warning, although the call should be handled as a successful return.

Comments

Tightly coupled message store and transport providers use 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 addressing dialog boxes.

This method can return MAPI_W_ERRORS_RETURNED if it cannot load an address book provider. This is a warning, not an error, and should be handled as a successful return. 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, your provider can still use the **IAddrBook** interface, and must release it when done.

Your provider can call the **IMAPISupport::GetLastError** method on the support object to obtain a **MAPIERROR** structure containing information about the address book providers that failed to 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.

See Also

IAddrBook : IMAPIProp interface, **IMAPISession::OpenAddressBook** method

IMAPISupport::OpenEntry

[New - Windows 95]

Opens an object given its entry identifier.

Syntax

HRESULT OpenEntry(**ULONG** *cbEntryID*, **LPENTRYID** *lpEntryID*, **LPCIID** *lpInterface*, **ULONG** *ulOpenFlags*, **ULONG FAR *** *lpulObjType*, **LPUNKNOWN FAR *** *lppUnk*)

Parameters

cbEntryID
Input parameter containing the number of bytes in the entry identifier pointed to by the *lpEntryID* parameter.

lpEntryID
Input parameter pointing to the entry identifier of the object to be opened.

lpInterface
Input parameter pointing to the interface identifier for the indicated object. Passing NULL in the *lpInterface* parameter indicates that the return value is cast to the standard interface for the indicated object.

ulOpenFlags
Input parameter containing a bitmask of flags used to control how the object is opened. The following flags can be set:

MAPI_BEST_ACCESS
Indicates the object should be opened with the maximum privileges allowed to the user. For example, if the client application has write privilege, open the object with write privilege; if the client application has read-only privilege, open the object with read-only privilege. The client application can learn the privilege 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 application. 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 client applications should not assume that write access was granted.

lpulObjType
Output parameter pointing to a variable that receives the object type for the opened object.

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 to modify a read-only object or an attempt to access an object for which the user has insufficient permissions.

MAPI_E_UNKNOWN_ENTRYID
The object indicated in the *lpEntryID* parameter is not recognized. This return value is typically returned if the message store or address book provider that the object is contained within is not open.

MAPI_E_NOT_FOUND

The object indicated in the *lpEntryID* parameter does not exist.

Comments

Service providers use the **IMAPISupport::OpenEntry** method to open objects. 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. Unlike **IMAPISession::OpenEntry**, your provider cannot open the root folder by passing NULL in the *lpEntryID* parameter of **IMAPISupport::OpenEntry**.

The calling application should check the value returned in the *lpulObjType* parameter to determine that the object type returned is what was expected. Commonly, after the application 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, you must open the address book first.

If your application passes an entry identifier in the *lpEntryID* parameter that belongs to a container, such as a message store or address book, which is not currently open, the call might fail and return the value MAPI_E_UNKNOWN_ENTRYID.

IMAPISupport::OpenProfileSection

[New - Windows 95]

Opens a section of the current profile and returns a pointer that provides further access.

Syntax

HRESULT OpenProfileSection(**LPMAPIUID** *lpUid*, **ULONG** *ulFlags*, **LPPROFSECT FAR *** *lppProfileObj*)

Parameters

lpUid
Input parameter pointing to the **MAPIUID** structure holding the MAPI unique identifier (MAPIUID) that identifies 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 used to control how the profile section is opened. The following flags can be set:

MAPI_BEST_ACCESS
Indicates the object should be opened with the maximum privileges allowed to the user. For example, if the client application has write privilege, open the object with write privilege; if the client application has read-only privilege, open the object with read-only privilege. The client application can learn the privilege 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 application. 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 client applications should not assume that write access was 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 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 requested object does not exist.

MAPI_E_UNKNOWN_FLAGS
Reserved or unsupported flags were used, and therefore the operation did not complete.

Comments

Service providers use 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 applications. To do so, your provider passes a MAPIUID in the *lpUid* parameter for the profile section that is known to other applications or providers.

The default behavior for **OpenProfileSection** is to open the profile section object as read-only, unless an application sets the MAPI_MODIFY flag in the *ulFlags* parameter. If an **OpenProfileSection** call attempts to open a nonexistent section with read-only access, the value MAPI_E_NOT_FOUND is returned.

If an **OpenProfileSection** call opens a nonexistent profile section with read-write access by passing the MAPI_MODIFY flag in the *ulFlags* parameter, the call creates the section.

Profile section operations shouldn't be held open any longer than necessary, but a provider that is writing to a profile section can keep it open while displaying a configuration property sheet.

See Also

IMAPIProp : IUnknown interface, **IProfSect : IMAPIProp** interface, **MAPIUID** structure

IMAPISupport::OpenTemplateID

[New - Windows 95]

Allows runtime binding of one address book provider's code to the data in another address book provider. **OpenTemplateID** is only used for interaction between address book providers.

Syntax

HRESULT OpenTemplateID(**ULONG** *cbTemplateID*, **LPENTRYID** *lpTemplateID*, **ULONG** *ulTemplateFlags*, **LPMAPIPROP** *lpMAPIPropData*, **LPCIID** *lpInterface*, **LPMAPIPROP FAR** * *lppMAPIPropNew*, **LPMAPIPROP** *lpMAPIPropSibling*)

Parameters

cbTemplateID
Input parameter containing the number of bytes in the *lpTemplateID* parameter. This value is obtained from the address book entry's PR_TEMPLATEID property.

lpTemplateID
Input parameter pointing to the template identifier for the template to be used. This value is obtained from the address book entry's PR_TEMPLATEID property.

ulTemplateFlags
Input parameter containing a bitmask of flags controlling how the address book entry is opened. The following flag can be set:

FILL_ENTRY
Indicates this is the first time this entry is being created in your provider's address book and that the foreign address book provider should copy all relevant properties, including name to identifier mapping, to the object pointed to by the *lpMAPIPropData* parameter. The foreign provider must perform this operation.

lpMAPIPropData
Input parameter pointing to the property object that is linked to the data in your address book provider. This is the object that receives the template information from the foreign address book provider. This object must support the interface being requested in the *lpInterface* parameter and it must be set to read-write access.

lpInterface
Input parameter pointing to the interface identifier that is being used for both the property objects in *lppMAPIPropData* and *lppMAPIPropNew* parameters. Passing NULL in the *lpInterface* parameter indicates the return value is cast to the standard interface for the address-book entry, which can be assumed to be IID_IMailUser.

lppMAPIPropNew
Output parameter pointing to a variable where the pointer to the interface indicated by the *lpInterface* parameter is stored.

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 other service provider doesn't exist.

Comments

Address book providers use the **IMAPISupport**::**OpenTemplateID** method to bind code that exists in a foreign address book provider to data that is contained in your address book provider. Your address book provider only needs to call **OpenTemplateID** if the address book entry that was previously obtained from a foreign address book provider has the PR_TEMPLATEID property. The values contained within PR_TEMPLATEID are passed in the **OpenTemplateID** method's *cbTemplateID* and *lpTemplateID* parameter. The foreign address book provider uses those values to determine which code to bind to your data.

Any address book provider that can contain entries with template identifiers copied from other containers should check for the PR_TEMPLATEID property after calls to the **IABContainer::OpenEntry** or **IABLogon::CreateEntry** methods. If the address book entry contains the PR_TEMPLATEID property, **OpenTempalteID** should be called so that the foreign address book provider has the opportunity to update the entry. If the FILL_ENTRY flag is passed in the *ulFlags* parameter, the foreign address book provider sets all properties on the entry. If the provider updates an entry without performing other operations, it can return the same interface for the **OpenTemplateID** call that was passed to it in the *lpInterface* parameter.

If **OpenTemplateID** fails with MAPI_E_UNKNOWN_ENTRYID, your provider should try to continue by treating the object as read-only.

See Also

IABLogon::OpenTemplateID method, **IPropData : IMAPIProp** interface, **PR_TEMPLATEID** property

IMAPISupport::PrepareSubmit

[New - Windows 95]

Prepares a message for submission to the MAPI spooler.

Syntax

HRESULT PrepareSubmit(**LPMESSAGE** *lpMessage*, **ULONG FAR *** *lpulFlags*)

Parameters

lpMessage
Input parameter pointing to the message to prepare.

lpulFlags
On input, reserved; must be zero. On output, must be NULL.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

Comments

Message store providers use the **IMAPISupport::PrepareSubmit** method to prepare a message for submission to the MAPI spooler. Store providers call **PrepareSubmit** in response to a call to the **IMessage::SubmitMessage** method by a client application. The **PrepareSubmit** call should be made soon after the client application calls **IMessage::SubmitMessage**. Message store providers that need to ensure that the MAPI spooler is synchronized with the client application as part of their **PrepareSubmit** implementation should call **IMAPISupport::SpoolerNotify**, passing the NOTIFY_READYTOSEND flag in the *ulFlags* parameter.

Message store providers call the **IMAPIFolder::GetMessageStatus** method to check the status of a message. If a message has its status set to MSGSTATUS_RESEND, the recipient list is checked and the PR_RESPONSIBILITY bit is set to true for all recipients that don't have MAPI_SUBMITTED 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

[New - Windows 95]

Generates a read or nonread report for a message.

Syntax

HRESULT ReadReceipt(**ULONG** *ulFlags*, **LPMESSAGE** *lpReadMessage*, **LPMESSAGE FAR *** *lppEmptyMessage*)

Parameters

ulFlags
: Input parameter containing a bitmask of flags used to control how the read report is generated. The following flag can be set:

  MAPI_NON_READ
  : Indicates that a non-read 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 that will be used as the read or nonread report is stored. On input the message store provider passes in the newly created method. MAPI then 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.

Comments

Message store providers use the **IMAPISupport::ReadReceipt** method to call 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. **ReadReciept** is typically called in response to a message being read, but also when a message is moved or copied. **ReadReceipt** is not called when a message is deleted. Providers can also call **ReadReciept** in response to calls to **IMessage::SetReadFlag**, but the read or non-read report should only be sent once. Providers need to keep track of the read status and should not send multiple reports for a single message.

If the *lppEmptyMessage* parameter references a valid message when MAPI returns from **ReadReceipt**, the store provider should call **IMessage::SubmitMessage** followed by **Release** on the message.

Providers can select whether to hide or display read and nonread reports generated by stores in their folders. Storing read and nonread reports in hidden folders allows the provider to implement tighter security.

If **ReadReciept** fails, the message should be released without being submitted. Providers can choose to store the message's status and try to generate the read or nonread report later.

See Also

IMAPIFolder::DeleteMessages method, **IMessage::SubmitMessage** method, PR_READ_RECEIPT_REQUESTED property

IMAPISupport::RegisterPreprocessor

[New - Windows 95]

Registers a preprocessor function for a transport provider.

Syntax

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) that identifies the entry identifier that the function handles. *lpMuid* can be NULL.

lpszAdrType
Input parameter pointing to a string containing the e-mail address type for the messages the preprocessor function operates on; examples of e-mail address types are FAX, SMTP, X500, and so on. *lpszAdrType* can be NULL or empty.

lpszDLLName
Input parameter pointing to a string containing the name of the DLL containing the entry point for the preprocessor function.

lpszPreprocess
Input parameter pointing to a string containing the name of the preprocessor function. *lpszPreprocess* can be NULL.

lpszRemovePreprocessInfo
Input parameter pointing to a string containing the name of the function that removes preprocessor information. *lpszRemovePreprocessInfo* can be NULL.

ulFlags
Reserved; must be zero.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

Comments

Transport providers use the **IMAPISupport::RegisterPreprocessor** method to register their preprocessor and postprocessing functions. Preprocessor functions must be registered before they can be called by the MAPI spooler prior to allowing the message to be sent.

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 preprocessed.

The *lpszPreprocess*, *lpszRemovePreprocessInfo* and *lpszDLLName* parameters should all point to strings that can be used in conjunction with calls to the **GetProcAddress** function so that the preprocessor's DLL entry point is called correctly.

Preprocessors are transport-order specific; that is, if a transport ahead of your transport in MAPI's transport order is able to handle the message, then your provider's preprocessor won't get called.

See Also

MAPIUID method, **PreprocessMessage** function, **RemovePreprocessInfo** function

IMAPISupport::SetProviderUID

[New - Windows 95]

Informs MAPI of the MAPI unique identifier (MAPIUID) assigned to the service provider using this support object.

Syntax

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.

Comments

Service providers use the **IMAPISupport::SetProviderUID** method to inform MAPI that they handle all globally unique identifiers (GUID) 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 a particular 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**. This call 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

[New - Windows 95]

Informs the MAPI spooler that the provider needs servicing.

Syntax

HRESULT SpoolerNotify(**ULONG** *ulFlags*, **LPVOID** *lpvData*)

Parameters

ulFlags
Input parameter containing a bitmask of flags used to control how the transport provider is serviced. The following flags can be set:

NOTIFY_CONFIG_CHANGE
Indicates to the MAPI spooler that the transport's configuration has changed. Causes the MAPI spooler to call the transport'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 provider requires the MAPI spooler to stop message processing and deinitialize the transport. If the transport provider chooses to call the **IMAPISupport::SpoolerNotify** method with NOTIFY_CRITICAL_ERROR 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 code.

NOTIFY_CRITSEC
Indicates the transport provider cannot allow the MAPI spooler to automatically yield to Windows during processing of calls made by the transport provider to the MAPI spooler. The transport provider will either yield when it can or begin automatically yielding again later. The transport provider's **IXPLogon::Idle** and **IXPLogon::Poll** methods should not be called. The *lpvData* parameter is undefined and should be set to NULL.

NOTIFY_NEWMAIL
Indicates the transport provider has new incoming messages the MAPI spooler should request. The message download occurs at the next available time. The *lpvData* parameter is undefined and should be set to NULL.

NOTIFY_NEWMAIL_RECEIVED
Indicates the *lpvData* parameter points to a new mail notification structure. Used for transports that are tightly coupled with message stores. This flag is ignored if the message store logged on with the MAPI_NO_MAIL flags 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 (see the possible return values for the **IXPLogon::SubmitMessage** method) and is again ready to accept messages. When set by a message store provider, this flag indicates that the MAPI spooler should synchronize with the client application's session. The *lpvData* parameter 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** method call. The entry identifier of the deferred message is contained in an **SBinary** structure pointed to by the *lpvData* parameter.

lpvData
Input parameter pointing to a notification structure holding data associated with the notification. The meaning of the data in the *lpvData* parameter depends on what flags are set in the *ulFlags* parameter; for more information on what *lpvData* can hold, see the preceding descriptions for the *ulFlags* flags.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

Comments

Transport providers use the **IMAPISupport::SpoolerNotify** method to notify the MAPI spooler to handle processing of a new message. A transport provider can call **IMAPISupport::SpoolerNotify** at any time.

When a message store tightly coupled with a transport 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 used.

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 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 the *ulFlags* parameter so that the MAPI spooler can reconfigure the transport on **IXPLogon::AddressTypes** and **IMAPISession::MessageOptions** method calls.

Message store providers that need to ensure that the MAPI spooler is synchronized with the client application as part of their **IMAPISupport::PrepareSubmit** implementation should call **SpoolerNotify**, passing the NOTIFY_READYTOSEND flag in the *ulFlags* parameter.

See Also

IMsgStore::NotifyNewMail method, **IXPLogon::StartMessage** method, **IXPLogon::SubmitMessage** method

IMAPISupport::SpoolerYield

[New - Windows 95]

Allows the transport provider to permit the MAPI spooler to give processing time to Windows.

Syntax

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 that the user wants to stop the transfer of an incoming or outgoing message regardless of how many recipients might already have received it.

Comments

Transport providers use the **IMAPISupport::SpoolerYield** method to allow the MAPI spooler to give up processing to Windows. Transports typically make this call when they are performing lengthy operations and 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 that it signals to the transport 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. Your provider should return MAPI_E_USER_CANCEL to the calling process and exit if possible.

IMAPISupport::StatusRecips

[New - Windows 95]

Generates delivery reports and nondelivery reports on behalf of transport providers.

Syntax

HRESULT StatusRecips(**LPMESSAGE** *lpMessage*, **LPADRLIST** *lpRecipList*)

Parameters

lpMessage
Input parameter pointing to the message for which a report is to be 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.

Comments

Transport providers use the **IMAPISupport::StatusRecips** method to notify MAPI that delivery or nondelivery reports should be sent for a set of one or more recipients. A *delivery report* is a report that indicates a message has been delivered to a recipient; a *nondelivery report* is a report that a message could not be delivered to a recipient.

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
Message class. This information is added to the report name; for example, for subject class *X*, the nondelivery report (NDR) name is REPORT.*X*.NDR and the delivery report (DR) name is REPORT.*X*.DR.

PR_REPORT_TEXT
Optional text generated by the messaging system. For a nondelivery report, this text indicates the reason for nondelivery.

PR_REPORT_TIME
Usually, the date and time when the message was delivered or deemed to be undeliverable. The information provided by PR_REPORT_TIME depends on the implementation.

PR_ROWID
Row in the recipient table where the recipient of the message for which the report is being generated appears. The MAPI spooler copies this row into the recipient table of the report.

PR_SEARCH_KEY
Contains a binary-comparable key that uniquely identifies the message object.

The following additional property is required for delivery reports:

PR_MESSAGE_DELIVERY_TIME
  Contains the date and time the message was delivered.

The following additional property is required for nondelivery reports:

PR_NDR_DIAGNOSTIC_CODE
  Contains diagnostic values identifying delivery problems.

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.

See Also

ADRLIST structure, **IMAPISupport::Address** method, **IMAPISupport::SpoolerNotify** method, **IXPLogon::EndMessage** method, **MAPIAllocateBuffer** function, **MAPIAllocateMore** function, **MAPIFreeBuffer** function

IMAPISupport::StoreLogoffTransports

[New - Windows 95]

Specifies the orderly release of the message store to the MAPI spooler.

Syntax

HRESULT StoreLogoffTransports(**ULONG FAR** * *lpulFlags*)

Parameters

lpulFlags

Input-output parameter containing a bitmask of flags used to control how message store logoff occurs.

On input, all flags for this parameter are mutually exclusive; a client application can set only one per call. The following flags can be set on input for *lpulFlags*:

LOGOFF_ABORT
: Indicates any transport activity on this store should be stopped before logoff. Control is returned to the client application 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 behavior occurs. If transport activity is quiet, the MAPI spooler releases the store. This flag is set by the store provider.

LOGOFF_NO_WAIT
: Indicates the message store closing should not wait for messages from the 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 application immediately. This flag is set by the store provider.

LOGOFF_ORDERLY
: Indicates the message store closing should not wait for information from the 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. This flag is set by the store provider.

LOGOFF_PURGE
: Works the same as LOGOFF_NO_WAIT. LOGOFF_PURGE returns control to the client application after completion. This flag is set by the store provider.

LOGOFF_QUIET
: Indicates that if any transport provider activity is taking place, the logoff does not occur and the type of activity is returned as a flag on output. This flag is set by the store provider.

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 **Release** method should be called at this point. LOGOFF_COMPLETE is returned by the store provider.

LOGOFF_INBOUND
: Indicates a message is currently coming into the store from one or more transport providers. This flag is returned by the store provider.

LOGOFF_OUTBOUND
: Indicates a message is currently being sent from the store by one or more transport providers. This flag is returned by the store provider.

LOGOFF_OUTBOUND_QUEUE
Indicates there are currently messages in the outbound queue for the store. This flag is returned by the store provider.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

Comments

Message store providers use 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 application calls the **IMsgStore::StoreLogoff** method and sets flags in its *lpulFlags* parameter to indicate how the client application requires the message store to be shut down. A message store provider should use these **IMsgStore::StoreLogoff** flags set by the client as input values for the **StoreLogoffTransports** method's *lpulFlags* parameter. The status returned from a client application call to the **IMsgStore::StoreLogoff** method is usually the same status that the message store provider then sends on the call to **StoreLogoffTransports**.

If the client application calls the **IUnknown::Release** method on a message store object without calling the **IMsgStore::StoreLogoff** method first, the message store provider involved should set the LOGOFF_ABORT flag in the *lpulFlags* parameter 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 value 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

[New - Windows 95]

Sets up a subscription for notification events with the MAPI notification engine.

Syntax

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 types are combined in a bitmask to filter 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 data structures:

Notification event type	Corresponding data 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 used to control how notification occurs. The following flag can be set:

NOTIFY_SYNC
Indicates that all notification callbacks should be made before the **Notify** method returns to the calling implementation. If this flag is not set, notification callbacks are queued to the registered processes and started when those processes gain control of the CPU.

lpAdviseSink
Input parameter pointing to an advise sink object created by the calling application to be called when an event for the object occurs 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.

Comments

Service providers use the **IMAPISupport::Subscribe** method to receive notification of changes to objects from MAPI's notification engine. A provider can make one call to **Subscribe** for every call to its **HrAllocAdviseSink** function. The MAPI notification engine is a convenience for provider writers; providers can choose to use it as the basis for their implementation of **Advise** and **Unadvise** methods, or they can use their own.

To use the MAPI notification engine, providers must create a key for the object for which notifications should be generated and place that key in the *lpKey* parameter. The notification engine 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 registered; 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 sent in *lpKey*. A **NOTIFKEY** structure is used in *lpKey* so your 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 client application requested synchronous or asynchronous notifications. The notification engine 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 the value 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

[New - Windows 95]

Removes an object's subscription for notification of changes previously established with a call to the **IMAPISupport::Subscribe** method.

Syntax

HRESULT Unsubscribe(**ULONG** *ulConnection*)

Parameters

ulConnection
Input parameter containing the number of the registration connection previously returned by a call to the **IMAPISupport::Subscribe** method.

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.

Comments

Providers use 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 **IMAPIAdviseSink::OnNotify** on the advise sink object, the **Release** call is delayed until the **OnNotify** method returns.

The MAPI notification engine is a convenience for provider writers; providers can choose to use it as the basis for their implementation of **Advise** and **Unadvise** methods, or they can use their own.

See Also

IMAPIAdviseSink::OnNotify method, **IMAPISupport::Subscribe** method

IMAPISupport::WrapStoreEntryID

[New - Windows 95]

Maps the private entry identifier of a message store object to an entry identifier more useful to the messaging system.

Syntax

HRESULT WrapStoreEntryID(**ULONG** *cbOrigEntry*, **LPENTRYID** *lpOrigEntry*, **ULONG FAR *** *lpcbWrappedEntry*, **LPENTRYID FAR *** *lppWrappedEntry*)

Parameters

cbOrigEntry
Input parameter containing the number of bytes in the *lpOrigEntry* parameter.

lpOrigEntry
Input parameter pointing to the private entry identifier for the message store resource.

lpcbWrappedEntry
Output parameter pointing to a variable that contains the number of bytes in the *lppWrappedEntry* parameter.

lppWrappedEntry
Output parameter pointing to a pointer to the new identifier 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.

Comments

Message store providers use the **IMAPISupport::WrapStoreEntryID** method to have MAPI generate a wrapped entry identifier to allow the opening of an entry if the store is not opened.

To allow storage of a message store's PR_STORE_ENTRYID properties 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 property requests for PR_STORE_ENTRYID. In other words, when a client application calls the **IMAPIProps::GetProps** method to retrieve PR_STORE_ENTRYID, instead of returning the store resource's entry identifier directly, the provider calls **WrapStoreEntryID**.

One use of WrapStoreEntryID is to allow an entry in the store to be opened if the store is not already open. In such cases, a valid entry identifier for the entry 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 entry identifier.

See Also

IMAPIProp::GetProps method, **IMAPISupport::CompareEntryIDs** method, **IMSLogon::CompareEntryIDs** method, **IMSProvider::Logon** method, **MAPIFreeBuffer** function

IMAPITable : IUnknown

[New - Windows 95]

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 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 **Advise** method.
GetStatus	Returns the status and type of a table.
SetColumns	Sets the order of columns in table rows to be returned by the **IMAPITable::QueryRows** method.
QueryColumns	Returns either the current list of columns for a table or the full list of columns available for the table.
GetRowCount	Returns the total number of rows in a table.
SeekRow	Moves the cursor to a specific position in a table.
SeekRowApprox	Finds an approximate fractional position in a table.
QueryPosition	Retrieves the row within a table where the cursor is currently positioned, based on a fractional value indicating the cursor position.
FindRow	Finds the next row in the 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 that 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 an 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

[New - Windows 95]

Stops any asynchronous operations currently in progress for a table.

Syntax

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.

Comments

Use the **IMAPITable::Abort** method to stop any asynchronous operation that is in progress on a table. To detect asynchronous operations currently in progress, your implementation can call the **IMAPITable::GetStatus** method.

If a call to the **IMAPITable::Restrict** method is interrupted by a call to **Abort**, whatever is in the table at the time 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 background 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

[New - Windows 95]

Registers an implementation for notification on changes to a table object.

Syntax

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 for which MAPI will generate notifications to filter specific cases. Each event type has a structure associated with it that holds additional information about the event. The only possible event type for this parameter is fnevTableModified; the corresponding data structure is **TABLE_NOTIFICATION**.

lpAdviseSink
Input parameter pointing to an advise sink object to be called when a event for the session object occurs about which notification has been requested. This advise sink object must have already been allocated.

lpulConnection
Output parameter pointing to a variable that on 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.

Comments

Use the **IMAPITable::Advise** method to register a table object implemented in a service provider for notification callbacks. Whenever a change occurs to the indicated object, the provider checks to see what event mask bit has been set in the *ulEventMask* parameter and thus what type of change has 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, your client should use the **HrThisThreadAdviseSink** function.

To provide notifications, the service provider implementing **Advise** needs to keep a copy of the pointer to the advise sink object; to do so, it calls the **IUnknown::AddRef** method for the advise sink object to maintain the 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, client applications must be prepared for the advise sink object to be released. Clients should therefore release their advise sink object after **Advise** returns unless they have a specific long-term use for it.

Because of the asynchronous behavior of notification, implementations that change the column settings can receive notifications with information organized by the 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.

See Also

HrThisThreadAdviseSink function, **IMAPIAdviseSink::OnNotify** method, **IMAPITable::Unadvise** method, **NOTIFICATION** structure, **TABLE_NOTIFICATION** structure

IMAPITable::CollapseRow

[New - Windows 95]

Collapses a table category and removes it from the table view.

Syntax

HRESULT CollapseRow(**ULONG** *cbInstanceKey*, **LPBYTE** *pbInstanceKey*, **ULONG** *ulFlags*, **ULONG FAR *** *lpulRowCount*)

Parameters

cbInstanceKey
: Input parameter containing the number of bytes in 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 category row does not exist.

Comments

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 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 the table view.

See Also

IMAPITable::ExpandRow method, **IMAPITable::GetCollapseState** method, **IMAPITable::QuerySortOrder** method, **IMAPITable::SetCollapseState** method, **IMAPITable::SortTable** method, **SSortOrderSet** structure

IMAPITable::CreateBookmark

[New - Windows 95]

Marks the current position of the table cursor so that an implementation can return to that position even when the table is updated.

Syntax

HRESULT CreateBookmark(**BOOKMARK FAR *** *lpbkPosition*)

Parameters

lpbkPosition
Output parameter pointing to a variable that receives a 32-bit bookmark value that 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.

Comments

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 an application 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.

CreateBookmark can allocate memory for the bookmark it creates. An implementation must release the resources for the bookmark by calling the **IMAPITable::FreeBookmark** method.

See Also

IMAPITable::FreeBookmark method, **IMAPITable::SeekRow** method

IMAPITable::ExpandRow

[New - Windows 95]

Expands a collapsed table category and adds the rows of that category to the table view.

Syntax

HRESULT ExpandRow(**ULONG** *cbInstanceKey*, **LPBYTE** *pbInstanceKey*, **ULONG** *ulRowCount*, **ULONG** *ulFlags*, **LPSRowSet FAR** * *lppRows*, **ULONG FAR** * *lpulMoreRows*)

Parameters

cbInstanceKey
Input parameter containing the number of bytes in 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 the 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 row containing the instance key specified in the *pbInstanceKey* parameter doesn't exist.

Comments

Use the **IMAPITable::ExpandRow** method to expand a collapsed table category, adding its rows to the table view. The position of the bookmark value BOOKMARK_CURRENT is moved to the row immediately following the last row in the **SRowSet** structure returned in the *lppRows* parameter. 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 the *pbInstanceKey* parameter.

Providers must not generate notifications on rows that are collapsed out of the table view.

See Also

IMAPITable::CollapseRow method

IMAPITable::FindRow

[New - Windows 95]

Finds the next row in the table that contains a property matching the specified criteria.

Syntax

HRESULT FindRow(**LPSRestriction** *lpRestriction*, **BOOKMARK** *BkOrigin*, **ULONG** *ulFlags*)

Parameters

lpRestriction
Input parameter pointing to a **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 used:

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.

ulFlags
Input parameter containing a bitmask of flags controlling 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 on the same row as when it was last used, or if it hasn't been used, the bookmark is no longer in the same position as when it was created. Use the **HR_FAILED** macro to test for this warning, but the call should be handled as a successful return.

Comments

Service providers use 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, the beginning of the table, and the 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; a *restriction* is a test conducted against a table to filter the rows of the table according to the criteria of the test. The restriction that clients should use for a scrolling prefix is as follows: For forward searching on a column sorted in ascending order, and for backward searching on a column sorted in descending order, pass in the *lpRestriction* parameter an **SPropertyRestriction** structure with relation **RELOP_GE** and the appropriate property tag and prefix value using the format *tag* **GE** *prefix value*.

Note The type of prefix searching performed by **FindRows** 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 desired behavior, the comparison function implied by the **RELOP_GE** sent 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 used in *BkOrigin* is either BOOKMARK_BEGINNING or BOOKMARK_END, **FindRow** does not return an error value 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 the PR_INSTANCE_KEY.

See Also

IMAPITable::CreateBookmark method, **SPropertyRestriction** structure, **SRestriction** structure

IMAPITable::FreeBookmark

[New - Windows 95]

Releases a bookmark from memory.

Syntax

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.

Comments

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

[New - Windows 95]

Returns the data necessary to rebuild the current table view.

Syntax

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 *lpbInstanceKey* parameter.

lpbInstanceKey
Input parameter pointing to the instance key (that is, the PR_INSTANCE_KEY property) identifying the row location within the table at which the current collapsed or expanded state should be rebuilt. The *lpbInstanceKey* parameter cannot be NULL; a selected row must be passed in.

lpcbCollapseState
Output parameter pointing to a variable containing the size, in bytes, of 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.

Comments

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**, your 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

[New - Windows 95]

Returns a **MAPIERROR** structure containing information about the last error that occurred for a table object.

Syntax

HRESULT GetLastError(**HRESULT** *hResult*, **ULONG** *ulFlags*, **LPMAPIERROR FAR** * *lppMAPIError*)

Parameters

hResult
Input parameter containing the result returned for the last call on the table object that returned an error.

ulFlags
Input parameter containing a bitmask of flags controlling the type of the returned strings. The following flag can be set:

MAPI_UNICODE
Indicates the strings returned 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 8-bit 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.

Comments

Use the **IMAPITable::GetLastError** method to retrieve information to display as a message to the user regarding the last error returned from a method call on the table object.

To release all the memory allocated by MAPI for the **MAPIERROR** structure, client applications 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, it is still possible that a **MAPIERROR** structure won't be returned. If the implementation cannot determine what the last error was, or if a **MAPIERROR** structure is not available for that error, a pointer to NULL is returned in the *lppMAPIError* parameter instead.

See Also

MAPIERROR structure, **MAPIFreeBuffer** function

IMAPITable::GetRowCount

[New - Windows 95]

Returns the total number of rows in a table.

Syntax

HRESULT GetRowCount(**ULONG** *ulFlags*, **ULONG FAR** * *lpulCount*)

Parameters

ulFlags
Reserved; must be zero.

lpulCount
Output parameter pointing to a variable holding the total number of rows in the table view.

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. Use the **HR_FAILED** macro to test for this warning, but the call should be handled as a successful return.

Comments

Use the **IMAPITable::GetRowCount** method to find out how many rows a table 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 on the whole table. If the row count returned in *lpulCount* is greater than 20, the calling implementation might limit the rows to be returned with **QueryRows**.

Some providers do not support **GetRowCount** and return MAPI_E_NO_SUPPORT. If **GetRowCount** is not supported, the client application 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's exact row count (that is, if the value returned in the *lpulCount* parameter is approximate and not exact), it returns MAPI_W_APPROX_COUNT. This result can occur on systems in which an exact count expends too much memory to perform, for example in systems 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 a client application deduces from a return value of MAPI_E_BUSY that asynchronous operations are in progress, it can call the asynchronous method **IMAPITable::WaitForCompletion**, 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.

See Also

IMAPITable::GetStatus method, **IMAPITable::QueryPosition** method, **IMAPITable::QueryRows** method, **IMAPITable::WaitForCompletion** method

IMAPITable::GetStatus

[New - Windows 95]

Returns the status and type of a table.

Syntax

HRESULT GetStatus(**ULONG FAR *** *lpulTableStatus*, **ULONG FAR *** *lpulTableType*)

Parameters

lpulTableStatus
Output parameter pointing to a variable in which the status of the table is placed. 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 occurring as a result of calls that sort or restrict the table; rather, it indicates unexpected changes.

TBLSTAT_SORTING
A sort operation is in progress.

TBLSTAT_SORT_ERROR
An error occurred during sorting.

TBLSTAT_SETTING_COLS
A column-setting operation is in progress.

TBLSTAT_SETCOL_ERROR
An error occurred while columns were being set.

TBLSTAT_RESTRICTING
A restriction operation is in progress.

TBLSTAT_RESTRICT_ERROR
An error occurred during restricting.

lpulTableType
Output parameter pointing to a variable in which the type of the table is placed. There are three table types:

TBLTYPE_DYNAMIC
The table's contents 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.

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.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

Comments

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 (a *restriction* is a test conducted against a table to filter table rows according to the criteria of the test). In addition, **GetStatus** can be used with the **IMAPITable::SortTable** method to monitor the status of a sort operation and with the **IMAPITable::SetColumns** method to see when a column-setting operation has completed.

See Also

IMAPITable::Restrict method, **IMAPITable::SetColumns** method, **IMAPITable::SortTable** method

IMAPITable::QueryColumns

[New - Windows 95]

Returns either the current list of columns for a table or the full list of columns available for the table.

Syntax

HRESULT QueryColumns(**ULONG** *ulFlags*, **LPSPropTagArray FAR** * *lpPropTagArray*)

Parameters

ulFlags
Input parameter containing a bitmask of flags used to control 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 containing a counted array of property tags is placed. Each property 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.

Comments

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 the table view, your implementation should set the TBL_ALL_COLUMNS flag in the *ulFlags* parameter. Otherwise, **QueryColumns** returns the provider's current table columns, typically those columns that are cached. You can retrieve additional columns by calling **IMAPITable::SetColumns**.

To free the memory holding the structure returned in the *lpPropTagArray* parameter, your implementation uses the **MAPIFreeBuffer** function.

See Also

IMAPITable::SetColumns method, **MAPIFreeBuffer** function, **SPropTagArray** structure

IMAPITable::QueryPosition

[New - Windows 95]

Retrieves the row within a table where the cursor is currently positioned, based on a fractional value indicating the cursor position.

Syntax

HRESULT QueryPosition(**ULONG FAR *** *lpulRow*, **ULONG FAR *** *lpulNumerator*, **ULONG FAR *** *lpulDenominator*)

Parameters

lpulRow
Output parameter pointing to the variable in which the current row number is placed. The row number is zero-based, with the first row in the table being zero.

lpulNumerator
Output parameter pointing to the variable in which is placed the numerator of the fraction representing the table position.

lpulDenominator
Output parameter pointing to the variable in which is placed the denominator of the fraction representing the table position. The *lpulDenominator* parameter cannot be zero.

Comments

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 based on 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 **QueryPosition** return values 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 the *lpulRow* parameter; this result can occur on systems where the **QueryPosition** calculation requires too much memory to perform. Applications 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

[New - Windows 95]

Returns one or more rows from a table, beginning at the current cursor position.

Syntax

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 used to control how rows are returned. The following flag can be set:

  TBL_NOADVANCE
  : Prevents the cursor from advancing, so that the position value BOOKMARK_CURRENT is always returned.

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.

Comments

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 the *lRowCount* parameter is negative, the cursor position moves backward the indicated number of rows and then the 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 or end of the table. Fewer rows might be returned than are requested if memory or implementation limits are reached or 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.

QueryRows supports inclusion of one or more property tags of PR_NULL 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 **IMAPIProps::GetProps** method.

If the calling application requests zero rows, then **QueryRows** returns the value MAPI_E_INVALID_PARAMETER. If an asynchronous operation is in progress, a call to **QueryRows** can return MAPI_E_BUSY. If a client application receives MAPI_E_BUSY in such a situation, it can call the asynchronous method **IMAPITable::WaitForCompletion**, then once the asynchronous operation is complete retry the call to **QueryRows**.

For most complex MAPI data structures, your implementation uses the **MAPIFreeBuffer** function to free the structure by freeing the top level pointer. **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 client applications 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

[New - Windows 95]

Retrieves the current sort order for a table.

Syntax

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 another operation is attempted.

Comments

Use the **IMAPITable::QuerySortOrder** method to retrieve the current sort order for a table. The sort order is returned in an **SSortOrderSet** structure.

For the following cases, **QuerySortOrder** returns an **SSortOrderSet** structure holding zero columns:

- If the table is unsorted.
- If the provider does not have information on how the table is sorted.
- If the provider cannot express the sort order within the capabilities of an **SSortOrderSet** structure.

In cases where a client application has called **IMAPITable::SortTable** with a **SSortOrderSet** structure 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. More columns than are in the present view can be returned.

To free the returned **SSortOrderSet** structure, the application uses the **MAPIFreeBuffer** function.

See Also

IMAPITable::SortTable method, **MAPIFreeBuffer** function, **SSortOrderSet** structure

IMAPITable::Restrict

[New - Windows 95]

Applies a restriction to a table, reducing the rows visible to only those matching the restriction criteria.

Syntax

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 used to control how returns are made 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 another operation is attempted.

Comments

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 filters the data so only some rows are seen. To discard the current restriction without creating a new one, pass NULL in the *lpRestriction* parameter.

A restriction on a column of multivalued properties works just as does a restriction on a column of single-valued properties. To reference columns of multivalued properties on which a restriction is to operate, the restriction must have the MVI_FLAG flag set. This requirement seems straightforward, but it means that an **SPropertyRestriction** structure can have a **ulPropTag** member different from *prop.ulPropTag*.

Restrictions on a column of multivalued properties without the MVI_FLAG being set will 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 behavior in situations where asynchronous operations are in progress is indicated by the flag set in the *ulFlags* parameter. If the TBL_BATCH flag is set in *ulFlags*, an application can defer the restriction operation until it requires the results. If the TBL_ASYNC flag is set in *ulFlags*, the **Restrict** operation can begin synchronously and return before the operation completes.

Asynchronous calls in progress when a **Restrict** call is required can be stopped by using the **IMAPITable::Abort** method. Most implementations of **Restrict** return the value 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.

Note that 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

[New - Windows 95]

Moves the cursor to a specific position in a table.

Syntax

HRESULT SeekRow(**BOOKMARK** *bkOrigin*, **LONG** *lRowCount*, **LONG FAR *** *lplRowsSought*)

Parameters

bkOrigin
Input parameter indicating the bookmark position from which the seek operation starts seeking. A bookmark can be created using the **IMAPITable::CreateBookmark** method, or one of the following predefined values can be used:

BOOKMARK_BEGINNING
Seeks from the beginning of the table.

BOOKMARK_CURRENT
Seeks from the row in the table where the cursor is located.

BOOKMARK_END
Seeks 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 in which the number of rows actually searched through is returned. 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 another 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 on the same row as when it was last used, or if it hasn't been used, the bookmark is no longer in the same position as when it was created. Use the HR_FAILED macro to test for this warning, although the call should be handled as a successful return.

Comments

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 the *lRowCount* parameter should be kept to less than 50; use the **IMAPITable::SeekRowApprox** method if you need to seek past a larger number of rows.

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 the *lRowCount* parameter. To seek to the beginning of the table, pass zero in the *lRowCount* parameter and the flag BOOKMARK_BEGINNING in the *bkOrigin* parameter. If there are large numbers of rows in the table, the seek operation can be slow.

SeekRow returns the number of rows actually searched through, positive or negative, in the variable pointed to by the *lplRowsSought* parameter. In normal operation, it should return the same value for *lplRowsSought* as passed in for the *lRowCount* parameter, unless the search reached the beginning or end of the table.

The operation of **SeekRow** can be slower than otherwise if the calling application requires a number of rows to be returned in the *lplRowsSought* parameter. If the calling application does not require a return count, it should pass NULL for the *lplRowsSought* parameter.

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 the value 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 the value MAPI_W_POSITION_CHANGED.

A bookmark pointing to a row that is no longer in the table view can still be used. If an application 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 the value 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.

See Also

IMAPITable::CreateBookmark method, **IMAPITable::FindRow** method, **IMAPITable::QueryRows** method, **IMAPITable::SeekRowApprox** method

IMAPITable::SeekRowApprox

[New - Windows 95]

Moves the cursor to an approximate fractional position in a table.

Syntax

HRESULT SeekRowApprox(**ULONG** *ulNumerator*, **ULONG** *ulDenominator*)

Parameters

ulNumerator
: Input parameter pointing to the variable indicating 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 value of the fraction's denominator. If the *ulNumerator* value is equal to the value in the *ulDenominator* parameter, the cursor is positioned after the last table row.

ulDenominator
: Input parameter pointing to the variable in which the denominator of the fraction representing the table position is placed. 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 another operation is attempted.

Comments

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 application can model that action by calling **SeekRowApprox** and passing in an equivalent fractional value using the *ulNumerator* and *ulDenominator* parameters. The **SeekRowApprox** search is always absolute from the beginning of the table. To move to the end of the table, the values in the *ulNumerator* and *ulDenominator* parameters must be the same. Any numbering scheme can be used based on what is convenient for your application; 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 can 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

[New - Windows 95]

Reestablishes the expanded or collapsed state of the table view that was saved by a call to the **IMAPITable::GetCollapseState** method.

Syntax

HRESULT SetCollapseState(**ULONG** *ulFlags*, **ULONG** *cbCollapseState*, **LPBYTE** *pbCollapseState*, **BOOKMARK FAR *** *lpbkLocation*)

Parameters

ulFlags
Reserved; must be zero.

cbCollapseState
Input parameter containing the number of bytes in 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 or rows pointed to by the instance key held in the *lpbInstanceKey* parameter on the call to the **IMAPITable::GetCollapseState** method.

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.

Comments

Use the **IMAPITable::SetCollapseState** method to reestablish the expanded or collapsed state of the table view that was saved by a call to the **IMAPITable::GetCollapseState** method. Use **SetCollapseState** and **GetCollapseState** together to present to the user, upon opening a table in your application, 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 application must have previously used **GetCollapseState**. It is expected that the column state 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.

Note that service 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

[New - Windows 95]

Sets the order of columns in table rows to be returned by the **IMAPITable::QueryRows** method.

Syntax

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 the value MAPI_E_INVALID_PARAMETER.

ulFlags
Input parameter containing a bitmask of flags used to control 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 another operation is attempted.

Comments

Use the **IMAPITable::SetColumns** method to control the order of the property values returned for each table row by an **IMAPITable::QueryRows** call. Some service 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 properties that are not in the original column set.

SetColumns supports inclusion of one or more PR_NULL property types in the property tag array in the *lpPropTagArray* parameter. 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 applications 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**, your application applies the MVI_FLAG flag to a table column's property type. To do so, it specifies **MVI_PROP(ulPropTag)** 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 MVI_INSTANCE. MAPI ignores MVI_FLAG if it is applied to a single-valued property.

If a call to **SetColumns** changes the order of table columns that contain multivalued instance 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 rows in a table, all bookmarks for the table are discarded. If a call to **SetColumns** adds or removes one or more columns, the call can change the column set.

A call to **SetColumns** completes its operation before returning, unless different behavior in situations where asynchronous operations are in progress is indicated by the flag set in the *ulFlags* parameter. If the TBL_BATCH flag is set in *ulFlags*, an application 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 return before completing. When the TBL_BATCH flag is set on asynchronous operations, providers should return a property type of PT_ERROR and a 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 the value 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, providers must order the properties in their column set in the same order as the column set was in when the request for notification was created.

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

[New - Windows 95]

Sorts table rows based on the sort criteria provided.

Syntax

HRESULT SortTable(**LPSSortOrderSet** *lpSortCriteria*, **ULONG** *ulFlags*)

Parameters

lpSortCriteria
: Input parameter pointing to an **SSortOrderSet** structure containing sort criteria to be applied. Passing an **SSortOrderSet** containing zero columns indicates your application doesn't require the table to be sorted or doesn't care about the table's sort order.

ulFlags
: Input parameter containing a bitmask of flags used to control 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 another operation is attempted.

MAPI_E_NO_SUPPORT
: The service provider either does not support changes to its objects or does not support notification of changes.

MAPI_E_TOO_COMPLEX
: Although the client application has passed valid parameters, the specified sort operation, typically a sub-restriction, is too complex for the implementation and could not be performed.

Comments

Use the **IMAPITable::SortTable** method to reorder the rows in a table view.

MAPI does not require service providers to be able to sort tables. To indicate table sorting is unavailable, providers should return the value 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 sort operation is applied to a full table (that is, a table with no restrictions).

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 applications require that only active columns be sorted. If a sort operation is requested for a column that cannot be sorted, the value MAPI_E_TOO_COMPLEX is returned.

Calling **SortTable** with zero columns in the **SSortOrderSet** structure in the *lpSortCriteria* parameter indicates your application 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. When your application 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 MVI_FLAG being set will 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 **SortTable** completes its operation before returning, unless different behavior in situations where asynchronous operations are in progress is indicated by the flag set in the *ulFlags* parameter. If the TBL_BATCH flag is set in *ulFlags*, an application can defer the restriction 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 the value 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 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 the value MAPI_E_TOO_COMPLEX under any of the following conditions:

- The implementation does not support the sort order requested in the **ulOrder** member of the **SSortOrderSet**.
- The number of columns to be sorted, as specified in the **cSorts** member in the **SSortOrderSet**, is larger than the implementation can handle.
- A sort operation is requested for a property column that the implementation cannot sort.
- A sort operation is requested, as indicated by a property tag in the **SSortOrderSet**, based on a property that is not in the available or active set and the application 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 application cannot perform such a sort operation.
- A sort operation based on multivalued property columns is requested using the MVI flag and the application does not support sorting on multivalued properties.
- A property tag for a property in the *lpSortCriteria* **SSortOrderSet** specifies a property or type that the application does not support.
- A sort operation other than one that proceeds through the table from the PR_RENDERING_POSITION property forward is specified for an attachment table that only supports this type of sorting.

For best performance, implementations of **SortTable** should establish their column sets with **SetColumns** and then their restrictions with **Restrict**, before sorting the table.

See Also

IMAPITable::Abort method, **IMAPITable::GetRowCount** method, **IMAPITable::QueryColumns** method, **IMAPITable::QuerySortOrder** method, **IMAPITable::SetColumns** method, **SSortOrderSet** structure

IMAPITable::Unadvise

[New - Windows 95]

Removes a table's registration for notification changes previously established with a call to the **IMAPITable::Advise** method.

Syntax

HRESULT Unadvise(**ULONG** *ulConnection*)

Parameters

ulConnection
Input parameter containing the number of the registration connection previously returned by a call to **IMAPITable::Advise**.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

Comments

Use the **IMAPITable::Unadvise** method to cancel notifications for the table object. To do so, **Unadvise** releases the pointer to the advise sink object passed in the *lpAdviseSink* parameter in the previous call to **IMAPITable::Advise**. As part of discarding the pointer to the advise sink, the 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 **IMAPIAdviseSink::OnNotify** for the advise sink, the **Release** call is delayed until the **OnNotify** method returns.

See Also

IMAPIAdviseSink::OnNotify method, **IMAPITable::Advise** method

IMAPITable::WaitForCompletion

[New - Windows 95]

Suspends an application while asynchronous operations occur on a table.

Syntax

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 the value MAPI_E_TIMEOUT. If 0xFFFFFFFF is sent in the *ulTimeout* parameter, your application pauses until the operation completes, however long that takes.

lpulTableStatus
Output parameter pointing to a variable in which is placed the most recent status of the table for which **WaitForCompletion** is called. If your application passes NULL in the *lpulTableStatus* parameter, no status information is returned. If a nonzero HRESULT is returned, including MAPI_E_TIMEOUT, the contents of the *lpulTableStatus* parameter 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.

Comments

Use the **IMAPITable::WaitForCompletion** method to suspend your application as any asynchronous operations currently underway for a table are processed. **WaitForCompletion** can allow the asynchronous operations either to complete or to run for a number of milliseconds, as indicated by the *ulTimeout* parameter, before being interrupted. To detect asynchronous operations in progress, your application uses the **IMAPITable::GetStatus** method.

See Also

IMAPITable::GetRowCount method, **IMAPITable::GetStatus** method, **IMAPITable::Restrict** method, **IMAPITable::SetColumns** method, **IMAPITable::SortTable** method

IMAPIViewAdviseSink : IUnknown

[New - Windows 95]

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 a message composed using a particular form has been submitted to the MAPI spooler.
OnSaved	Notifies a form viewer that a message composed using a particular form has been saved.

IMAPIViewAdviseSink::OnShutdown

[New - Windows 95]

Notifies a form viewer that a form is being closed.

Syntax

HRESULT OnShutdown()

Parameters

None

Return Values

S_OK
    The call succeeded.

IMAPIViewAdviseSink::OnNewMessage

[New - Windows 95]

Notifies a form viewer that either a new or an existing message has been loaded in a form.

Syntax

HRESULT OnNewMessage()

Parameters

None

Return Values

S_OK
    The call succeeded.

Comments

A form calls the **IMAPIViewAdviseSink::OnNewMessage** method whenever a message is loaded in the form using either the **IPersistMessage::InitNew** or **IPersistMessage::Load** method. A viewer will often elect to release 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

[New - Windows 95]

Notifies a form viewer of the printing status of a form.

Syntax

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 showing the status of the print job. The following values can be used to indicate status:

  S_FALSE
    The printing job is 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.

Comments

A form calls 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

[New - Windows 95]

Notifies a form viewer that the message in the form object has been saved.

Syntax

HRESULT OnSaved()

Parameters

None

Return Values

S_OK
    The call succeeded and has returned the expected value or values.

Comments

A form object calls the **IMAPIViewAdviseSink::OnSaved** method after the current message has been successfully saved. Its doing so permits viewers to update their windows to reflect changes to the message.

IMAPIViewAdviseSink::OnSubmitted

[New - Windows 95]

Notifies a form viewer that the current message has been submitted to the MAPI spooler.

Syntax

HRESULT OnSubmitted()

Parameters

None

Return Values

S_OK
    The call succeeded and has returned the expected value or values.

Comments

A form object calls the **IMAPIViewAdviseSink::OnSubmitted** method after a call to **IMAPIMessageSite::SubmitMessage** has returned successfully. After **OnSubmitted** is called, viewers can work on the assumption the message has been updated and can update their windows.

See Also

IMAPIMessageSite::SubmitMessage method

IMAPIViewContext : IUnknown

[New - Windows 95]

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:	Form objects

Vtable Order

SetAdviseSink	Registers a form object for notifications of changes to a view's status state.
ActivateNext	Activates the next or previous message in a view.
GetPrintSetup	Retrieves the current print setup so as to print the current message.
GetSaveStream	Retrieves the stream to place a textized version of the current message.
GetViewStatus	Retrieves the current viewer status state.

IMAPIViewContext::ActivateNext

[New - Windows 95]

Activates the next or previous message in a view.

Syntax

HRESULT ActivateNext(**ULONG** *ulDir*, **LPCRECT** *prcPosRect*)

Parameters

ulDir
Input parameter containing a status value indicating which next 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 used to display the next message.

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.

Comments

Form objects call the **IMAPIViewContext::ActivateNext** method to change the current message being displayed to the user. The *ulDir* parameter is used to inform the form viewer of the reason for the change in the current message status. The VCDIR_NEXT and VCDIR_PREVIOUS flags correspond to users choosing the next or previous commands. These operations normally correspond to moving up or down one message in the form viewer's list of messages. VCDIR_DELETE and VCDIR_MOVE are set by the **IMAPIMessageSite::DeleteMessage** or **IMAPIMessagesSite::MoveMessage** methods. 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 allow users to specify the direction to be moved.

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 message is displayed, the form uses the window rectangle passed in the *prcPosRect* parameter.

See Also

IMAPIViewContext::GetViewStatus method

IMAPIViewContext::GetPrintSetup

[New - Windows 95]

Retrieves the current print setup so as to print the current message.

Syntax

HRESULT GetPrintSetup(**ULONG** *ulFlags*, **LPFORMPRINTSETUP FAR** * *lppFormPrintSetup*)

Parameters

ulFlags
Input parameter containing a bitmask of flags. The following flag can be set:

MAPI_UNICODE
Indicates the returned strings are to be in Unicode format. If the MAPI_UNICODE flag is not set, the strings are in 8-bit format.

lppFormPrintSetup
Output parameter pointing to where the pointer to the returned **FORMPRINTSETUP** structure holding information on the print setup is stored.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

Comments

Form objects call the **IMAPIViewContext::GetPrintSetup** method to retrieve the current print setup. A form object 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** structure must be freed by the calling form object using the **MAPIFreeBuffer** function.

See Also

FORMPRINTSETUP structure

IMAPIViewContext::GetSaveStream

[New - Windows 95]

Retrieves the stream to place a textized version of the current message.

Syntax

HRESULT GetSaveStream(**ULONG FAR *** *pulFlags*, **ULONG FAR *** *pulFormat*, **LPSTREAM FAR *** *ppstm*)

Parameters

pulFlags
Output parameter pointing to a bitmask of flags used to indicate the format of the saved text. The following flag can be used.

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 used to control additional formatting characteristics. The following flags can be set:

SAVE_FORMAT_RICHTEXT
The message is textized using rich text format.

SAVE_FORMAT_TEXT
The message is textized using plain text format.

ppstm
Output parameter pointing to a pointer to where the textized version of the message is to be written.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

Comments

The **IMAPIViewContext::GetSaveStream** method is used to support the Save As verb set for form viewers. A stream interface is returned to the form object. Form objects are not permitted to write any data before the seek pointer on entry, and must leave the seek pointer at the end of the textized message when done. The message should be fully textized into the stream prior to returning from the **IMAPIForm::DoVerb** call.

IMAPIViewContext::GetViewStatus

[New - Windows 95]

Retrieves the current viewer status state.

Syntax

HRESULT GetViewStatus(**ULONG FAR** * *lpulStatus*)

Parameters

lpulStatus
Output parameter pointing to a variable in which a bitmask of flags giving information on view status is returned. The following flags can be set:

VCSTATUS_DELETE
Indicates the form can delete the message.

VCSTATUS_INTERACTIVE
Indicates the form should suppress displaying user interface even in response to a verb which might otherwise cause user interface to be displayed.

VCSTATUS_MODAL
Indicates the form is to be modal to the viewer.

VCSTATUS_NEXT
Indicates there is a next form.

VCSTATUS_PREV
Indicates there is a previous form.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

Comments

Form servers 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 VCSTATUS_NEXT and VCSTATUS_PREVIOUS are valid parameters for **IMAPIViewContext::ActivateNext**. If VCSTATUS_DELETE is set, but not VCSTATUS_READONLY, then the message can be deleted using the **IMAPIMessageSite::DeleteMessage** method. Typically, form viewer applications disable menu commands and buttons if next and previous are not valid for the context. The *lpulStatus* values are dynamic and can be changed by the view context's calling the **IMAPIFormAdviseSink::OnChange** method.

The VCSTATUS_MODAL flag is set if the form must be modal to the window whose handle is passed in with the **IMAPIForm::DoVerb** call. If VCSTATUS_MODAL is set, the form can use the thread on which the former call to the **IMAPIForm::DoVerb** method was made until the form closes. If VCSTATUS_MODAL is not set, the form should not be modal to the window indicated by the passed handle 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

[New - Windows 95]

Registers a form object for notifications of changes to a view's status state.

Syntax

HRESULT SetAdviseSink(**LPMAPIFORMADVISESINK** *pmvns*)

Parameters

pmvns
Input parameter pointing to a form advise-sink object.

Return Values

S_OK
The call succeeded.

Comments

Form objects call the **IMAPIViewContext::SetNotifySink** method to register a form for notification on changes to that which is the next or previous message within a particular view context. When called with NULL in the *pmvns* parameter, **SetNotifySink** cancels a previous registration.

IMessage : IMAPIProp

[New - Windows 95]

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
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 in a message.
ModifyRecipients	Adds, deletes, or modifies recipients in a message.
SubmitMessage	Saves all changes to a message and marks the message as ready to be submitted to the MAPI spooler.
SetReadFlag	Sets or clears the read flags for a message, and manages the sending of read notifications.

IMessage::CreateAttach

[New - Windows 95]

Creates a new attachment in a message.

Syntax

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 for the *lpInterface* parameter 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 used to control 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 application. If the object is not accessible, some subsequent call to the object might return an error.

lpulAttachmentNum
Output parameter pointing to a variable receiving an index number identifying the newly created attachment. 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 Values

S_OK
The call succeeded and has returned the expected value or values.

Comments

Message store providers use 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

[New - Windows 95]

Deletes an attachment from a message.

Syntax

HRESULT DeleteAttach(**ULONG** *ulAttachmentNum*, **ULONG** *ulUIParam*, **LPMAPIPROGRESS** *lpProgress*, **ULONG** *ulFlags*)

Parameters

ulAttachmentNum
Input parameter containing the index number of the attachment to be deleted. 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 window the dialog box is modal to. 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 the *lpProgress* parameter, the progress information is provided by MAPI. The *lpProgress* parameter is ignored unless the ATTACH_DIALOG flag is set in the *ulFlags* parameter.

ulFlags
Input parameter containing a bitmask of flags used to control what happens when the attachment is deleted. The following flag can be set:

ATTACH_DIALOG
Causes a progress-information user interface to be displayed as the operation proceeds.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

Comments

Message store providers use the **IMessage::DeleteAttach** method to delete an attachment from within a message. Prior to calling **DeleteAttach** to delete the attachment, client applications 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

[New - Windows 95]

Returns the attachment table for a message.

Syntax

HRESULT GetAttachmentTable(**ULONG** *ulFlags*, **LPMAPITABLE FAR** * *lppTable*)

Parameters

ulFlags
Input parameter containing a bitmask of flags controlling 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 8-bit format.

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.

Comments

Message store providers use the **IMessage::GetAttachmentTable** method to acquire a pointer to a message's attachment table, which lists all the attachments within a 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 required property columns:

PR_ATTACH_NUM
The index number of the attachment within the message. It is used to identify the attachment when the attachment is opened with the **IMessage::OpenAttach** method. Within a client application session, an attachment's PR_ATTACH_NUM property remains constant as long as the message is open. However, PR_ATTACH_NUM is only valid during a message session and is not unique across sessions.

PR_RECORD_KEY
Uniquely identifies the attachment object within a message. It remains unique across sessions so that it stays the same after the message containing the attachment is closed and reopened.

PR_RENDERING_POSITION
Identifies where within the text of the message MAPI should render the attachment.

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 client applications should not be designed with the expectation that restrictions are supported in all cases.

Attachments don't necessarily appear in the attachment table until **IMAPIProp::SaveChanges** is called on the message. The attachment table can change while it is open if the application calls the **IMessage::CreateAttach** or **IMessage:: DeleteAttach** method to create or delete an attachment, or if an attachment is modified such that its properties change in the attachment table, 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 the **QueryColumns** method returns before the application that contains the attachment 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 application that contains the attachment table calls the **IMAPITable::SetColumns** method.
- Controls the property types of the sort order specification returned by the **IMAPITable::QuerySortOrder** method before the application that contains the attachment table calls the **IMAPITable::SortTable** method.

See Also

IMessage::CreateAttach method, **IMessage::DeleteAttach** method, **IMessage::OpenAttach** method

IMessage::GetRecipientTable

[New - Windows 95]

Returns the recipient table in a message.

Syntax

HRESULT GetRecipientTable(**ULONG** *ulFlags*, **LPMAPITABLE FAR** * *lppTable*)

Parameters

ulFlags
Input parameter containing a bitmask of flags used to control 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 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 8-bit format.

lppTable
Output parameter pointing to a variable where the pointer to the returned table object is stored.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

Comments

Message store providers use the **IMessage::GetRecipientsTable** 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 a received message or a message under composition contains one row for each recipient of the message.

The table will have at least the following columns:

PR_DISPLAY_NAME
The display name for the recipient.

PR_ENTRYID
The entry identifier for the recipient.

PR_RECIPIENT_TYPE
The recipient type, one of MAPI_TO, MAPI_CC, or MAPI_BCC.

PR_ROWID
Identifies a recipient in the table for modification or deletion operations. The PR_ROWID of a given recipient is only valid as long as the message is open.

Messages that have been sent also contain the following property columns:

PR_ADDRTYPE
The address type for the recipient.

PR_SENDER_NAME
The display name associated with the entry identifier in the PR_SENDER_ENTRYID property.

PR_SENDER_ENTRYID
   The entry identifier of the sender.

PR_CLIENT_SUBMIT_TIME
   The time that the message was submitted.

Depending on a provider's implementation, additional columns might be in the table.

Most messaging system providers 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 client applications 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. These 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. Recipients indicated by the MAPI_SUBMITTED type can also appear in a message when an effort is made to resend a message to one or more of the recipients named in a nondelivery report.

The initial active columns for a newly opened recipient table, that is those columns that **IMAPITable::QueryRows** returns in data if the application does not perform an **IMAPITable::SetColumns** call, include all available columns. Restrictions on the table are specific to a provider's implementation, so client applications should not work on the expectation that restrictions are supported in all cases. The application can change this table while it is open by calling the **IMessage::ModifyRecipients** method. When you are modifying the recipient list you must call the **IMAPITable::QueryColumns**, **IMAPITable::SetColumns**, and **IMAPITable::QueryRows** methods for each column that you want in the list – not just the modifications – because **ModifyRecipients** deletes from the recipient list all columns that you do not specify.

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 the **IMAPITable::QueryColumns** method. The initial active columns for a recipient's table are those columns the **QueryColumns** method returns before the application that contains the recipient's table calls the **IMAPITable::SetColumns** method.
- Sets the string type to Unicode for data returned for the initial active rows of the recipient table by the **IMAPITable::QueryRows** method. The initial active rows for a recipient table are those rows **QueryRows** returns before the application that contains the recipient table calls the **IMAPITable::SetColumns** method.
- Controls the property types of the sort order specification returned by the **IMAPITable::QuerySortOrder** method before the application that contains the recipient table calls the **IMAPITable::SortTable** method.

See Also

IMAPIProp::SaveChanges method, **IMAPITable::QueryRows** method, **IMessage::ModifyRecipients** method

IMessage::ModifyRecipients

[New - Windows 95]

Adds, deletes, or modifies recipients in a message.

Syntax

HRESULT ModifyRecipients(**ULONG** *ulFlags*, **LPADRLIST** *lpMods*)

Parameters

ulFlags
Input parameter containing a bitmask of flags used to control how the recipients 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 the *ulFlags* parameter:

MODRECIP_ADD
Adds all recipients to the current table.

MODRECIP_MODIFY
Replaces the entire row with the rows you specify.

MODRECIP_REMOVE
Removes the 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 Values

S_OK
The call succeeded and has returned the expected value or values.

Comments

Message store providers use 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; client applications should send the same type of table in the *lpMods* parameter of **ModifyRecipients** to modify, add to, or delete recipients from the existing recipient table. You can also pass in the recipient list returned from **IMessage::GetRecipientTable**. When you are modifying the recipient list, you must call the **IMAPITable::QueryColumns**, **IMAPITable::SetColumns**, and **IMAPITable::QueryRows** methods for each column that you want in the list – not just the modifications – because **ModifyRecipients** deletes from the recipient list all columns that you do not specify.

The default behavior of **ModifyRecipients** if all bits in the bitmask passed in the *ulFlags* parameter are set to zero is to replace the entire existing recipient table with the list passed in the *lpMods* parameter. When calling **ModifyRecipients** with the MODRECIP_MODIFY flag set in the *ulFlags* parameter, **ModifyRecipients** instead replaces each entire recipient row with the associated row in the **ADRLIST** structure passed in the *lpMods* parameter.

Following are some rules for setting the properties of the recipients in the table passed in the *lpMods* parameter:

- 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 the *ulFlags* parameter.
- The PR_ROWID property must not be provided for *lpMods* table recipients when the MODRECIP_ADD flag or zero is set in the *ulFlags* parameter.
- An unresolved recipient entry in *lpMods* must contain only the PR_DISPLAY_NAME and PR_RECIPIENT_TYPE properties.
- A resolved recipient entry 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.
- If the **lpProps** member of the **ADRENTRY** structure is NULL, the provider should ignore that entry.

A client application 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 the PR_ADDRTYPE property, giving a recipient's address type, or the PR_EMAIL_ADDRESS property, giving a recipient's messaging address, is not consistent with the address of the recipient identified by the PR_ENTRYID property, the address where the message with the modified recipient table is delivered is undefined. The message might be delivered to the addresses specified in the PR_ADDRTYPE and PR_EMAIL_ADDRESS properties or to the recipient identified by the PR_ENTRYID property, or it might be returned as undeliverable because of the ambiguity of the address information.

The **ADRLIST** structure passed in the *lpMods* parameter must be allocated as an **SRowSet** structure is. **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** with a new one that can later be freed by the calling application using the **MAPIFreeBuffer** function. The original **SPropValue** structure will also be freed by **MAPIFreeBuffer**.

See Also

ADRENTRY structure, **ADRLIST** structure, **IAddrBook::Address** method, **IMAPISupport::Address** method, **MAPIAllocateBuffer** function, **MAPIFreeBuffer** function, **SPropValue** structure

IMessage::OpenAttach

[New - Windows 95]

Opens an attachment.

Syntax

HRESULT OpenAttach(**ULONG** *ulAttachmentNum*, **LPCIID** *lpInterface*, **ULONG** *ulFlags*, **LPATTACH FAR *** *lppAttach*)

Parameters

ulAttachmentNum
Input parameter containing the index number of the attachment to be opened. 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 for the *lpInterface* parameter 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 used to control how the attachment is opened. The following flags can be set:

MAPI_BEST_ACCESS
Indicates the object should be opened with the maximum privileges allowed to the user. For example, if the client application has read-write privilege, the object is opened with read-write privilege; if the client application has read-only privilege, the object is opened with read-only privilege. The client application can retrieve the privilege 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 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 client applications should not work on the assumption that read-write access was granted.

lppAttach
Output parameter pointing to a variable where the pointer to the open attachment object is stored.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

Comments

Message store providers use the **IMessage::OpenAttach** method to open an attachment within a message. A client application can open an attachment by first locating the PR_ATTACH_NUM property for that attachment (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, which is only valid within the message. The PR_ATTACH_NUM value is only unique within a 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

[New - Windows 95]

Sets or clears the read flags for a message, and manages the sending of read notifications.

Syntax

HRESULT SetReadFlag(**ULONG** *ulFlags*)

Parameters

ulFlags

Input parameter containing a bitmask of flags used control 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 notifications. The following flags can be set:

CLEAR_READ_FLAG
: Resets the MSGFLAG_READ flag.

GENERATE_RECEIPT_ONLY
: Generates a read notification but does not change the state of the MSGFLAG_READ flag in PR_MESSAGE_FLAGS.

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.

SUPPRESS_RECEIPT
: Indicates the messaging system should not send a read notification for each message, but that each message should be marked as read.

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 notifications.

Comments

Message store providers use the **IMessage::SetReadFlag** method to mark a message as read and, optionally, to send a read notification for that message. A read notification is only sent if the user who sent the message requested it. Applications generally cannot determine if the originator requested a read notification. 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 none of the flags are set in the *ulFlags* parameter, the following rules apply:

- If the MSGFLAG_READ bit is already set, do nothing.
- If the PR_READ_RECEIPT_REQUESTED bit is set, send the read notification and set the MSGFLAG_READ bit.

MAPI_E_INVALID_PARAMETER is returned 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 SUPPRESS_RECEIPT and GENERATE_RECEIPT_ONLY are set in *ulFlags*, the PR_READ_RECEIPT_REQUESTED bit, if it is set, is turned off, and a read notification should not be sent.

Note Providers can optimize report behavior so that a client application's setting a message attribute to get a read notification or delivery report is only a request and the provider can support not sending read notifications or delivery reports. However, some message stores do not support the suppression of read notifications for some messages. If the client application calls **SetReadFlag** on such a message with the SUPPRESS_RECEIPT flag set in the *ulFlags* parameter, **SetReadFlag** returns the value MAPI_E_NO_SUPPRESS; in this case, MAPI does not mark the message as read and does not generate a report or notification.

See Also

IMAPIContainer::OpenEntry method, **IMAPIFolder::SetReadFlags** method, **IMAPIProp::GetProps** method, **IMAPIProp::SaveChanges** method, PR_MESSAGE_FLAGS property

IMessage::SubmitMessage

[New - Windows 95]

Saves all changes to a message and marks the message as ready for sending.

Syntax

HRESULT SubmitMessage(**ULONG** *ulFlags*)

Parameters

ulFlags
Reserved; must be zero.

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 list is empty.

Comments

Message store providers use 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 application 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, your 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 your client has this entry identifier pointer, it then passes it as the *lpEntryId* parameter of the **IMsgStore::AbortSubmit** method.

A message might stay in a message store for some time before the underlying messaging system can take responsibility for it. This occurs because MAPI passes messages to the underlying messaging system in the order in which they are marked for sending. However, the order of receipt at the destination is in the underlying messaging system's control and is not guaranteed to match the order in which messages were sent.

See Also

IMsgStore::AbortSubmit method

IMsgServiceAdmin : IUnknown

[New - Windows 95]

The **IMsgServiceAdmin** interface is used to make changes to a message service within a profile. You can get a pointer to an **IMsgServiceAdmin** interface in two ways: by calling the **IMAPISession::AdminServices** method or by calling the **IProfAdmin::AdminServices** method. If your application is primarily concerned with profile configuration, for example if it is a control panel, then **IProfAdmin::AdminServices** is the preferred way to get the **IMsgServiceAdmin** interface because it does not log the providers onto the MAPI session. If your application requires the ability to make changes to the active profile, then **IMAPISession::AdminServices** should be called to get the pointer to the **IMsgServiceAdmin** interface. Applications 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 an application from removing all of 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 information about the last error that occurred for the message-service administration object.
GetMsgServiceTable	Returns a table object 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 the profile.
CopyMsgService	Copies a message service into a profile.
RenameMsgService	Renames a message service that cannot be copied.
ConfigureMsgService	Enables the user to reconfigure the 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 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 object listing the service

providers installed in a profile.

IMsgServiceAdmin::AdminProviders

[New - Windows 95]

Enumerates and changes the service providers belonging to a message service.

Syntax

HRESULT AdminProviders(**LPMAPIUID** *lpUID*, **ULONG** *ulFlags*, **LPPROVIDERADMIN FAR *** *lppProviderAdmin*)

Parameters

lpUID
Input parameter pointing to the **MAPIUID** structure holding the unique identifier 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 controlling the type of the 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 8-bit 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** does not exist.

Comments

Use the **IMsgServiceAdmin::AdminProviders** method to get a pointer to a provider administration object so that the methods of the **IProviderAdmin** interface can be called for that object. Client applications cannot make changes to service providers; all a client application 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

[New - Windows 95]

Enables the user to reconfigure the message service using the service's configuration property sheet.

Syntax

HRESULT ConfigureMsgService(**LPMAPIUID** *lpUID*, **ULONG** *ulUIParam*, **ULONG** *ulFlags*, **ULONG** *cValues*, **LPSPropValue** *lpProps*)

Parameters

lpUID
Input parameter pointing to the **MAPIUID** structure holding the unique identifier for the message service to be configured.

ulUIParam
Input parameter containing a handle of the window the dialog box is modal to; in this case, the *ulUIParam* parameter is the handle for the window to serve as the parent window for the configuration property sheet.

ulFlags
Input parameter containing a bitmask of flags used to control 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 8-bit format.

MSG_SERVICE_UI_READ_ONLY
Indicates the service's configuration user interface 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 dialog box. If SERVICE_UI_ALWAYS is not set, a configuration dialog box 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 to allow a user interface to be displayed.

cValues
Input parameter containing the number of property values in the **SPropValue** structure pointed to by the *lpProps* parameter. The *cValues* parameter should be set to zero if there are no properties.

lpProps
Input parameter pointing to an **SPropValue** structure containing the property values of the properties to be displayed to the user in the property sheet. The *lpProps* parameter should be NULL if message service is being configured without displaying the property sheet to the user.

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. Call the **IMsgServiceAdmin::GetLastError** method to get the **MAPIERROR** structure describing the error.

MAPI_E_NOT_FOUND

The **MAPIUD** does not match an existing message service.

MAPI_E_NOT_INTIALIZED

The message service does not have a message-service entry function.

MAPI_E_USER_CANCEL

The user canceled the operation, typically by clicking the Cancel button in the dialog box.

Comments

Use the **IMsgServiceAdmin::ConfigureMsgService** method to display the configuration property sheet for a message service so that client application 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.

Client applications can also configure the message service programmatically, that is without displaying the property sheet to the user. This can only be done if the message service's property identifiers and property values are known in advance. If an application is programmatically configuring the service and the property sheets should not be displayed, then neither the SERVICE_UI_ALLOWED nor the SERVICE_UI_ALWAYS flag should be set in the *ulFlags* parameter. If an application is not doing programmatic configuration, or if the programmatic configuration does not completely configure the service, set the SERVICE_UI_ALLOWED flag in *ulFlags*. If an application is doing programmatic configuration but only to establish the default settings and the user will be able to change the settings, set the SERVICE_UI_ALWAYS flag in *ulFlags*.

To allow programmatic configuration, message services typically prepare a header file that includes constants for all of the required and optional properties and their values.

See Also

MAPIUID structure, **SPropValue** structure

IMsgServiceAdmin::CopyMsgService

[New - Windows 95]

Copies a message service into a profile.

Syntax

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 unique identifier for the message service to be copied.

lpszDisplayName
Input parameter pointing to a string containing the display name for the message service to be copied.

lpInterfaceToCopy
Input parameter pointing to the interface identifier (IID) for the profile section object into which the message service is to be copied. Passing NULL for the *lpInterfaceToCopy* parameter 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 for the *lpInterfaceDst* parameter 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 to the window the dialog box is modal to.

ulFlags
Input parameter containing a bitmask of flags controlling the type of the 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 strings are in 8-bit 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 **MAPIUID** does not refer to an existing message service.

MAPI_E_NO_ACCESS

The message service is already in the profile and does not allow multiple instances of itself.

Comments

Use the **IMsgServiceAdmin::CopyMsgService** method to copy a message service into a profile section. The client application must previously have obtained an interface to the target profile, but it does not have to be logged onto the profile. The target profile is not necessarily the same as the source profile. Client applications can copy messages services within a profile or from one profile to another. The message service's entry point function does not get called on either the source or the destination of the copy operation. After the copy operation, the configuration settings of the service remain unchanged; call the **IMsgServiceAdmin::ConfigureMsgService** method to configure the copied message service.

See Also

IMsgServiceAdmin::ConfigureMsgService method, **MAPIUID** structure

IMsgServiceAdmin::CreateMsgService

[New - Windows 95]

Adds a message service to the current profile.

Syntax

HRESULT CreateMsgService(**LPTSTR** *lpszService*, **LPTSTR** *lpszDisplayName*, **ULONG** *ulUIParam*, **ULONG** *ulFlags*)

Parameters

lpszService
Input parameter pointing to a string containing the message service name, using which client applications can find in the MAPISVC.INF file the information necessary to create the message service. 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. 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 window the dialog box is modal to.

ulFlags
Input parameter containing a bitmask of flags used to control the installation of the message service. The following flags can be set:

SERVICE_UI_ALLOWED
Displays the message service's configuration property sheet only if the service is not completely configured.

MAPI_UNICODE
Indicates the passed-in strings are in Unicode format. If the MAPI_UNICODE flag is not set, the strings are in 8-bit format.

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.

Comments

Use the **IMsgServiceAdmin::CreateMsgService** method to add a message service to the current profile. Your client application 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 user interface to enable the user to configure its settings. The list of providers and the properties for each provider that make up a message service 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 from the MAPISVC.INF file for that service into the profile, creating new sections for each service provider. After all of the information has been copied from MAPISVC.INF, the message service's entry 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 *ulUIParam* and *ulFlags* parameters are also passed when the service provider's message service entry function is called.

Service providers are expected to display their configuration property sheets so that the user can configure the message service.

CreateMsgService does not return the **MAPIUID** for the message service that was added to the profile. To retrieve the **MAPIUID**, call the **IMsgServiceAdmin::GetMsgServiceTable** method to get the message service administration table. Search for the message service by the PR_SERVICE_NAME property. Then retrieve the service's PR_SERVICE_UID property. You then pass that **MAPIUID** in the *lpUid* parameter when calling the **IMsgServiceAdmin::ConfigureMsgService** method to configure the service.

IMsgServiceAdmin::DeleteMsgService

[New - Windows 95]

Deletes a message service and its associated profile sections.

Syntax

HRESULT DeleteMsgService(**LPMAPIUID** *lpuid*)

Parameters

lpuid
Input parameter pointing to the **MAPIUID** structure holding the unique identifier for the message service being deleted.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

MAPI_E_NOT_FOUND
The **MAPIUID** does not match an existing message service.

Comments

Use the **IMsgServiceAdmin::DeleteMsgService** method to delete a message service from a profile. To retrieve the **MAPIUID** for the message service, call the **IMsgServiceAdmin::GetMsgServiceTable** method to get the message-service administration table. Search for the message service by the PR_SERVICE_NAME property. Then retrieve the service's PR_SERVICE_UID property. You then pass that **MAPIUID** in the *lpUid* parameter when calling **DeleteMsgService**. **DeleteMsgService** removes all profile sections related to the message service.

If the service has defined a message-service entry function, it is called with the MSG_SERVICE_DELETE value set in the *ulContext* parameter before the profile sections are removed so that the service can perform any service-specific tasks. First the service is deleted, then the service's profile section skeleton is deleted. The message service entry function is not called again after the service has been deleted.

See Also

MAPIUID structure

IMsgServiceAdmin::GetLastError

[New - Windows 95]

Returns information about the last error that occurred for the message-service administration object.

Syntax

HRESULT GetLastError(**HRESULT** *hResult*, **ULONG** *ulFlags*, **LPMAPIERROR FAR** * *lppMAPIError*)

Parameters

hResult
: Input parameter containing the result returned for the last call on the session object that returned an error.

ulFlags
: Input parameter containing a bitmask of flags controlling the type of the returned strings. The following flag can be set:

  MAPI_UNICODE
  : Indicates the strings returned 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 8-bit 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.

Comments

Use the **IMsgServiceAdmin::GetLastError** method to retrieve information to display as a message to the user regarding the last error returned from a method call on the message-service administration object.

To release all the memory allocated by MAPI, client applications need only call the **MAPIFreeBuffer** function for the **MAPIERROR** structure.

The return value from **GetLastError** must be S_OK for the application to make use of the **MAPIERROR** structure. Even if the return value is S_OK, it is still possible that a **MAPIERROR** structure won't be returned. If the implementation cannot determine what the last error was, or if a **MAPIERROR** structure is not available for that error, a pointer to NULL is returned in the *lppMAPIError* parameter instead.

See Also

MAPIERROR structure, **MAPIFreeBuffer** function

IMsgServiceAdmin::GetMsgServiceTable

[New - Windows 95]

Returns a table object listing the message services installed in a profile.

Syntax

HRESULT GetMsgServiceTable(**ULONG** *ulFlags*, **LPMAPITABLE FAR** * *lppTable*)

Parameters

ulFlags
Input parameter containing a bitmask of flags controlling 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 8-bit format.

lppTable
Output parameter pointing to a variable where the pointer to the returned table object is stored.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

Comments

Use the **IMsgServiceAdmin::GetMsgServiceTable** method to get a pointer to a table object that lists the message services currently installed in a profile. The columns of the message services table contain the current information for the following properties:

PR_DISPLAY_NAME
The name of the message service as given by the service. Some implementations enable the user to change this name.

PR_SERVICE_NAME
A particular message service's name as found in the MAPISVC.INF file (for example, "Microsoft Exchange Services"). This name, which is fixed and permanent, can be used to access entries in MAPISVC.INF.

PR_RESOURCE_FLAGS
Flags describing the message service's capabilities.

PR_SERVICE_DLL_NAME
The dynamic-link library (DLL) for a particular service's entry function.

PR_SERVICE_ENTRY_NAME
The name of a service's entry function, in 8-bit format.

PR_SERVICE_UID
A unique identifier for the profile section that contains any further properties maintained by this service. This is the property column containing the **MAPIUID** that the **IMsgServiceAdmin::DeleteMsgService** and **IMsgServiceAdmin::ConfigureMsgService** methods use to perform their operations.

PR_SERVICE_SUPPORT_FILES
A list of the names of the support files that are required for this message service.

PR_INSTANCE_KEY
A search key for an instance of a particular MAPI object.

Once a message-service administration table has been returned, it does not reflect changes being made to the profile, such as adding or deleting providers. Calls to the **IMAPITable::Advise** calls 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; a table object supporting the **IMAPITable** interface is returned. If you call the **IMAPITable::QueryRows** method on that table, zero rows are returned.

Client applications that are configuring or deleting messages services using the **IMsgServiceAdmin::ConfigureMsgService** or **IMsgService::DeleteMsgService** method typically call **GetMsgServicesTable** to get the **MAPIUID** structure 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 the **QueryColumns** method returns before the application that contains the message-service administration 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 application that contains the message-service administration table calls the **IMAPITable::SetColumns** method.
- Controls the property types of the sort order specification returned by the **IMAPITable::QuerySortOrder** method before the application that contains the message-service administration table calls the **IMAPITable::SortTable** method.

See Also

IMsgServiceAdmin::ConfigureMsgService method, **IMsgServiceAdmin::DeleteMsgService** method

IMsgServiceAdmin::GetProviderTable

[New - Windows 95]

Returns a table object listing the service providers installed in a profile.

Syntax

HRESULT GetProviderTable(**ULONG** *ulFlags*, **LPMAPITABLE FAR** * *lppTable*)

Parameters

ulFlags
Input parameter containing a bitmask of flags controlling 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 8-bit 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.

Comments

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
The provider's name as given by the provider.

PR_INSTANCE_KEY
A search key for an instance of a particular MAPI object.

PR_PROVIDER_DISPLAY
The name of the provider as given by the messaging service. This name is not localizable and is a reliable way to search on a particular provider.

PR_PROVIDER_DLL_NAME
The dynamic-link library (DLL) for a particular provider.

PR_PROVIDER_ORDINAL
A number computed by MAPI that identifies the order in which the providers are listed in the profile.

PR_PROVIDER_UID
Identifies a particular instance of a provider.

PR_RESOURCE_FLAGS
Flags describing the capabilities of the provider.

PR_RESOURCE_TYPE
An enumeration describing the type of the provider. This property can be used to determine all the providers of a particular type that are installed within a profile.

PR_SERVICE_NAME
The name of the message service as given by the message service. This name is not localizable and is a reliable way to search on a particular service.

PR_SERVICE_UID

Identifies a particular instance of a message service.

Providers that have been deleted, or are in use but have been marked for deletion, are not returned in the provider administration table. Once a provider administration table has been returned, it does not reflect changes being made to the profile, such as adding or deleting providers. Calls to the **IMAPITable::Advise** method for the provider administration table return S_OK, but no changes are made to the table. If no provider exists, **GetProviderTable** does not return an error; a table object supporting the **IMAPITable** interface is returned. If you call 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 your client application 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 order, restrict the table on the PR_RESOURCE_TYPE==MAPI Transport Provider. Then sort the table by PR_PROVIDER_ORDINAL. Then call **IMAPITable::QueryRows** to get the rows of the table. These can also all be made in a single call to the **HrQueryAllRows** function with all of the appropriate data structures passed in. This information for setting the transport order can only be obtained by calling **IMsgServiceAdmin::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 administration table by the **IMAPITable::QueryColumns** method. The initial active columns for a provider administration table are those columns the **QueryColumns** method returns before the application that contains the provider administration table calls the **IMAPITable::SetColumns** method.
- Sets the string type to Unicode for data returned for the initial active rows of the provider administration table by the **IMAPITable::QueryRows** method. The initial active rows for a provider administration table are those rows **QueryRows** returns before the application that contains the provider administration table calls the **IMAPITable::SetColumns** method.
- Controls the property types of the sort order specification returned by the **IMAPITable::QuerySortOrder** method before the application that contains the provider administration table calls the **IMAPITable::SortTable** method.

See Also

IMsgServiceAdmin::GetMsgServiceTable method, **IMsgServiceAdmin::MsgServiceTransportOrder** method, **IProviderAdmin::GetProviderTable** method

IMsgServiceAdmin::MsgServiceTransportOrder

[New - Windows 95]

Sets the order in which transport providers are called to deliver a message.

Syntax

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 unique identifiers that 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 transports actually in the profile.

MAPI_E_NOT_FOUND
: One or more of the UIDs passed in *lpUIDList* parameter does not refer to a transport currently in the profile.

Comments

Use the **IMsgServiceAdmin::MsgServiceTransportOrder** method to set the delivery order of transport providers within a profile. The *lpUIDList* parameter must contain the sorted list of transport-provider entry identifiers that was obtained from the PR_PROVIDER_UID property of the table returned from the **IMsgServiceAdmin::GetProviderTable** method. You must pass in the complete list.

SetTransportOrder overrides transport provider preferences such as the STATUS_XP_PREFER_LAST bit in the PR_RESOURCE_FLAGS property.

See Also

MAPIUID structure

IMsgServiceAdmin::OpenProfileSection

[New - Windows 95]

Opens a section of the current profile and returns a pointer that provides further access to the profile object.

Syntax

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 that identifies the profile section. The *lpUID* parameter must not be NULL.

lpInterface
Input parameter pointing to the interface identifier used to open the profile section. Passing NULL in the *lpInterface* parameter indicates that the return value is cast to the standard interface for a profile section. The *lpInterface* parameter can also be set to an appropriate interface identifier. Valid interface identifiers are IID_IMAPIProp and IID_IProfSect.

ulFlags
Input parameter containing a bitmask of flags used to control 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 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 client applications should not work on the assumption that read-write access was granted.

lppProfSect
Output parameter pointing to a variable that receives the pointer to the open profile object.

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 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.

Comments

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. The default behavior is to open the profile section as read-only, unless an application sets the MAPI_MODIFY flag in the *ulFlags* parameter. Profile sections belonging to service providers cannot be opened by calls to the **IMsgServiceAdmin::OpenProfileSection** method.

If an **OpenProfileSection** call opens a nonexistent profile section by passing the MAPI_MODIFY flag in the *ulFlags* parameter, the call creates the section. If an **OpenProfileSection** call attempts to open a nonexistent section with read-only access, MAPI_E_NOT_FOUND is returned.

All open operations should be as brief as possible, but an application that is writing to a profile section can keep it open while displaying a modification dialog box.

See Also

IMAPIProp : IUnknown interface, **IMAPISession::OpenProfileSection** method, **IProfSect : IMAPIProp** interface, **MAPIUID** structure

IMsgServiceAdmin::RenameMsgService

[New - Windows 95]

Renames a message service that cannot be copied.

Syntax

HRESULT RenameMsgService(**LPMAPIUID** *lpUID*, **ULONG** *ulFlags*, **LPTSTR** *lpszDisplayName*)

Parameters

lpUID
: Input parameter pointing to the **MAPIUID** structure holding the unique identifier 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.

Comments

Use the **IMsgServiceAdmin::RenameMsgService** method to rename a message service that cannot be copied.

See Also

MAPIUID structure

IMsgServiceAdmin::SetPrimaryIdentity

[New - Windows 95]

Designates each service provider belonging to the message service as being the supplier of the primary identifier for the profile.

Syntax

HRESULT SetPrimaryIdentity(**LPMAPIUID** *lpUID*, **ULONG** *ulFlags*)

Parameters

lpUID
  Input parameter pointing to the **MAPIUID** structure holding the unique identifier for the message service whose primary identity is to be set. This **MAPIUID** is typically obtained from the PR_RESOURCE_UID property column in the message-service administration table.

ulFlags
  Reserved; must be zero.

Return Values

S_OK
  The call succeeded and has returned the expected value or values.

MAPI_E_NO_ACCESS
  The method attempted to designate a service as primary which has the PR_NO_PRIMARY bit set in the PR_RESOURCE_FLAGS property.

Comments

Use the **IMsgServiceAdmin::SetPrimaryIdentity** method to set the primary identity for each provider in the messaging service indicated by the *lpUID* parameter. The primary identity for a message service is the identity of the user logged onto the message service and is represented by the entry identifier returned by a call to the **IMAPISession::QueryIdentity** method.

Each messaging service provider that MAPI has information about establishes an identity for each of its users. This identity can be established when a client application 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 bit in the PR_RESOURCE_FLAGS property. Client applications can call the **IMsgServiceAdmin::SetPrimaryIdentity** method to designate one of the many identities established for a user by messaging service providers as the primary identity for that user.

If the calling application passes NULL in the *lpUID* parameter, 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

[New - Windows 95]

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
Called by:	Client applications, the MAPI spooler, MAPI

Vtable Order

Advise	Registers your application for notifications on changes within the 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 object in the message store.
OpenEntry	Opens a folder or message in the message store given its entry identifier.
SetReceiveFolder	Sets the receive folder for a particular message class.
GetReceiveFolder	Obtains the receive folder that has been set to receive messages for a particular message class and related information about the message reception behavior of that message class.
GetReceiveFolderTable	Returns a table that shows property settings for all receive folders for the 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 the message store's outgoing queue.
GetOutgoingQueue	Returns the message store's outgoing queue table. This method is called only by the MAPI spooler.
SetLockState	Allows the MAPI spooler to lock or unlock a message.
FinishedMsg	Called only by the MAPI spooler when message processing is complete.
NotifyNewMail	Notifies the message store that a new message has arrived. This method is called only by the MAPI spooler.

IMsgStore::AbortSubmit

[New - Windows 95]

Attempts to remove a message from the message store's outgoing queue.

Syntax

AbortSubmit(**ULONG** *cbEntryID*, **LPENTRYID** *lpEntryID*, **ULONG** *ulFlags*)

Parameters

cbEntryID
: Input parameter containing the number of bytes in the entry identifier pointed to by the *lpEntryID* parameter.

lpEntryID
: Input parameter pointing to the entry identifier of the message that is to be removed 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.

Comments

Message store providers use 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

[New - Windows 95]

Registers your client application for notifications on changes within the message store.

Syntax

HRESULT Advise(**ULONG** *cbEntryID*, **LPENTRYID** *lpEntryID*, **ULONG** *ulEventMask*,
  LPMAPIADVISESINK *lpAdviseSink*, **ULONG FAR** * *lpulConnection*)

Parameters

cbEntryID
  Input parameter containing the number of bytes in 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 is set up to provide 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 for which MAPI will generate notifications to filter 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 data structures:

Notification event type	Corresponding data structure
fnevCriticalError	**ERROR_NOTIFICATION**
fnevNewMail	**NEWMAIL_NOTIFICATION**
fnevObjectCreated	**OBJECT_NOTIFICATION**
fnevObjectDeleted	**OBJECT_NOTIFICATION**
fnevObjectModified	**OBJECT_NOTIFICATION**
fnevObjectCopied	**OBJECT_NOTIFICATION**
fnevSearchComplete	**OBJECT_NOTIFICATION**

lpAdviseSink
  Input parameter pointing to the advise sink object to be called when an event for the object occurs 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.

Comments

Message store providers use the **IMsgStore::Advise** method to register an object implemented 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 has been set in the *ulEventMask* parameter and thus what type of change has 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, your client should use the **HrThisThreadAdviseSink** function.

To provide notifications, the service provider implementing **Advise** needs to keep a copy of the pointer to the advise sink object; to do so, it calls the **IUnknown::AddRef** method for the advise sink object to maintain the 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, client applications must be prepared for the advise sink object to be released. Clients should therefore release their advise sink object after **Advise** returns unless they have a specific long-term use for it.

See Also

HrThisThreadAdviseSink function, **IMAPIAdviseSink::OnNotify** method, **IMsgStore::Unadvise** method, **NOTIFICATION** structure

IMsgStore::CompareEntryIDs

[New - Windows 95]

Compares two entry identifiers to determine if they refer to the same service provider 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.

Syntax

HRESULT CompareEntryIDs(ULONG *cbEntryID1*, **LPENTRYID** *lpEntryID1*, **ULONG** *cbEntryID2*, **LPENTRYID** *lpEntryID2*, **ULONG** *ulFlags*, **ULONG FAR *** *lpulResult*)

Parameters

cbEntryID1
: Input parameter containing the number of bytes in 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 number of bytes in 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 that receives the result of the comparison; 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.

Comments

Message store providers use the **IMsgStore::CompareEntryIDs** method to compare two entry identifiers for a given message store entry and 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

[New - Windows 95]

Called only by the MAPI spooler when message processing is complete.

Syntax

HRESULT FinishedMsg(**ULONG** *ulFlags*, **ULONG** *cbEntryID*, **LPENTRYID** *lpEntryID*)

Parameters

ulFlags
Reserved; must be zero.

cbEntryID
Input parameter containing the number of bytes in 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.

Comments

The MAPI spooler calls the **IMsgStore::FinishedMsg** method when it has finished processing a message. When MAPI calls a message store provider's **FinishedMsg** implementation, the provider should unlock a message for which processing is complete and, if the PR_DELETE_AFTER_SUBMIT bit 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

[New - Windows 95]

Returns a pointer to the message store's outgoing queue table. This method is called only by the MAPI spooler.

Syntax

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 Values

S_OK
: The call succeeded and has returned the expected value or values.

Comments

The MAPI spooler uses 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
: The time when the message sender marked the message for submission.

PR_DISPLAY_BCC
: The display name of a message recipient of the blind carbon-copy type.

PR_DISPLAY_CC
: The display name of a carbon copy recipient.

PR_DISPLAY_TO
: The primary message recipients.

PR_ENTRYID
: The entry identifier for the object.

PR_MESSAGE_FLAGS
: Flags indicating the current state of the message.

PR_MESSAGE_SIZE
: The approximate size in bytes of the message if it is moved from one message store to another.

PR_PRIORITY
: The priority of the message.

PR_SENDER_NAME
: The message sender's display name. The property corresponds to the PR_SENDER_ENTRYID property.

PR_SUBJECT
: The subject of the message.

PR_SUBMIT_FLAGS
: The locked or preprocessing status of the message.

Due to the requirements that messages be preprocessed and that they be 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 client applications 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. Messages 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.

It is the responsibility of the message store provider to 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

[New - Windows 95]

Obtains the receive folder that has been set to receive messages for a particular message class and related information about the message reception behavior of that message class.

Syntax

HRESULT GetReceiveFolder(LPTSTR *lpszMessageClass*, **ULONG** *ulFlags*, **ULONG FAR *** *lpcbEntryID*, **LPENTRYID FAR *** *lppEntryID*, **LPTSTR FAR *** *lppszExplicitClass*)

Parameters

lpszMessageClass
Input parameter pointing to a string containing the name of 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 controlling the type of the passed 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 8-bit format.

lpcbEntryID
Output parameter pointing to a variable containing the number of bytes in the entry identifier in the *lppEntryID* parameter.

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 containing the name of the message class that explicitly sets as its receive folder the folder indicated by the entry identifier returned in the *lppEntryID* parameter. This message class name should either be that of the class named in the *lpszMessageClass* parameter or of a superclass of that class. Passing in 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 Values

S_OK
The call succeeded and has returned the expected value or values.

Comments

Message store providers use the **IMsgStore::GetReceiveFolder** method to obtain the entry identifier of the folder where a client application puts messages of a specific class when they are received. If the message class indicated in the *lpszMessageClass* parameter 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 an application 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 the *lppEntryID* parameter and IPM.Note in the *lppszExplicitClass* parameter.

If the client application calls **GetReceiveFolder** for a message class and has not set a receive folder for that message class, the value returned in the *lppszExplicitClass* is either a zero-length string, a string in Unicode format, or a string in 8-bit 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.

The client application should call the **MAPIFreeBuffer** function when it is done with the entry identifier returned in the *lppEntryID* parameter 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 the *lppszExplicitClass* parameter to free the memory that holds that string.

See Also

MAPIFreeBuffer function

IMsgStore::GetReceiveFolderTable

[New - Windows 95]

Returns a table that shows property settings for all receive folders for the message store, including which message classes the folders are associated with.

Syntax

HRESULT GetReceiveFolderTable(**ULONG** *ulFlags*, **LPMAPITABLE FAR** * *lppTable*)

Parameters

ulFlags
Input parameter containing a bitmask of flags used to control 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 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 8-bit format.

lppTable
Output parameter pointing to a variable where the pointer to the returned receive folder table is stored.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

Comments

Message store providers use the **IMsgStore::GetReceiveFolderTable** method to get a table showing the property settings for all receive folders in the store, including which message classes the receive folders are associated with. A *receive folder* is the folder where incoming messages are placed. The table returned contains information for one receive folder in each of its rows and contains the following property columns:

PR_MESSAGE_CLASS
The message class that explicitly sets this folder as its receive folder.

PR_RECORD_KEY
The record key of this receive folder, used in restrictions and comparisons. A *record key* is a binary-comparable unique identifier.

PR_ENTRYID
The entry identifier of this row's receive folder.

Your message store provider should implement the receive folder table such that it supports setting restrictions to filter for particular values of PR_RECORD_KEY to allow 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 the **QueryColumns** method returns before the application that contains the receive folder 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 application that contains the receive folder table calls the **IMAPITable::SetColumns** method.
- Controls the property types of the sort order specification returned by the **IMAPITable::QuerySortOrder** method before the application 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

[New - Windows 95]

Notifies the message store that a new message has arrived. This method is called only by the MAPI spooler.

Syntax

HRESULT NotifyNewMail(**LPNOTIFICATION** *lpNotification*)

Parameters

lpNotification
Input parameter pointing to the **NOTIFICATION** structure holding the new-message notification.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

Comments

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 that are 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 client applications that have registered for new mail notification (that is, that have included the fnevNewMail event type in the *ulEventMask* parameter passed in their **IMsgStore::Advise** calls) about the new message. 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-message 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

[New - Windows 95]

Opens a folder or a message in the message store given its entry identifier.

Syntax

HRESULT OpenEntry(**ULONG** *cbEntryID*, **LPENTRYID** *lpEntryID*, **LPCIID** *lpInterface*, **ULONG** *ulFlags*, **ULONG FAR** * *lpulObjType*, **LPUNKNOWN FAR** * *lppUnk*)

Parameters

cbEntryID
Input parameter containing the number of bytes in the entry identifier pointed to by the *lpEntryID* parameter.

lpEntryID
Input parameter pointing to the entry identifier of the folder or message to be opened. The root folder of the store can be opened either with its entry identifier, as with any other folder, or by passing NULL for the *lpEntryID* parameter.

lpInterface
Input parameter pointing to the interface identifier for the indicated object. Passing NULL in the *lpInterface* parameter indicates that the return value is cast to the standard interface for the indicated object. The *lpInterface* parameter can also be set to an appropriate interface identifier for the object being opened.

ulFlags
Input parameter containing a bitmask of flags used to control how the object is opened. The following flags can be set:

MAPI_BEST_ACCESS
Indicates the object should be opened with the maximum privileges allowed to the user. For example, if the client application has read-write privilege, the object is opened with read-write access; if the client application has read-only privilege, the object is opened with read-only access. The client application can retrieve the privilege 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 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 client applications should not work under the assumption that read-write access was granted.

lpulObjType
Output parameter pointing to a variable where the object type for 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 to modify a read-only object or an attempt to access an object for which the user has insufficient permissions.

Comments

Message store providers use the **IMsgStore::OpenEntry** method to open a folder or a message in the message store. The calling application 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 the client application can use **IMsgStore::OpenEntry** to open any folder or message, it is usually faster if the calling application instead uses the **IMAPIContainer::OpenEntry** method on the folder containing the object being opened.

The calling application should check the value returned in the *lpulObjType* parameter to determine that the object type returned is what was expected. Commonly, after the application 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

[New - Windows 95]

Allows the MAPI spooler to lock or unlock a message.

Syntax

HRESULT SetLockState(**LPMESSAGE** *lpMessage*, **ULONG** *ulLockState*)

Parameters

lpMessage
Input parameter pointing to the message to be locked or unlocked.

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 Values

S_OK
The call succeeded and has returned the expected value or values.

Comments

The MAPI spooler uses the **IMsgStore::SetLockState** method to lock a message while it is in the process of sending the message. The reason the MAPI spooler locks the message is so that calls to the **IMsgStore::AbortSubmit** method cannot succeed. **SetLockState** also provides the MAPI spooler with a mechanism to unlock messages that it has previously locked. If the MAPI spooler passes the MSG_LOCKED value in the *ulLockState* parameter, 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 uses **SetLockState** to lock a message, it only locks the oldest message – that is, the next message queued for it to send. If the oldest message in the queue is waiting for a temporarily unavailable transport provider, and the next message in the queue is to use a different transport provider, then 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 as part of its implementation prior to calling the **IMsgStore::FinishedMsg** method.

See Also

IMsgStore::AbortSubmit method, **IMsgStore::FinishedMsg** method

IMsgStore::SetReceiveFolder

[New - Windows 95]

Sets the receive folder for a particular message class.

Syntax

HRESULT SetReceiveFolder(**LPTSTR** *lpszMessageClass*, **ULONG** *ulFlags*, **ULONG** *cbEntryID*, **LPENTRYID** *lpEntryID*)

Parameters

lpszMessageClass
Input parameter pointing to a string containing the name of 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. The *receive folder* is the folder in which incoming messages are placed.

ulFlags
Input parameter containing a bitmask of flags controlling 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 8-bit format.

cbEntryID
Input parameter containing the number of bytes in the entry identifier pointed to by the *lpEntryID* parameter.

lpEntryID
Input parameter pointing to the entry identifier of the folder to be 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 Values

S_OK
The call succeeded and has returned the expected value or values.

Comments

Message store providers use 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 that has been defined or specify that incoming messages for multiple message classes all be directed to the same folder. An application 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 message store provider should place incoming faxes and one in which the store 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 the value S_OK even if there was no existing setting for the indicated message class.

IMsgStore::StoreLogoff

[New - Windows 95]

Enables the orderly logoff of a message store under client application control.

Syntax

HRESULT StoreLogoff(**ULONG FAR** * *lpulFlags*)

Parameters

lpulFlags

Input-output parameter containing a bitmask of flags used to control logoff from the message store. On input, all flags set for this parameter are mutually exclusive; your client application must specify only one flag per call. A client can set the following flags on input:

LOGOFF_ABORT

Indicates any transport activity on this store should be stopped before logoff. Control is returned to the client application after the 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. This flag is set by the client application.

LOGOFF_NO_WAIT

Indicates the message store closing should not wait for messages from the 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 application immediately.

LOGOFF_ORDERLY

Indicates the message store closing should not wait for information from the 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 application after completion.

LOGOFF_QUIET

Indicates that if any transport provider activity is taking place, the logoff does not occur.

Flags are not returned on output because the message store provider has already been released.

Return Values

S_OK

The call succeeded and has returned the expected value or values.

Comments

Client applications use 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 to either stop sending messages at message store release or to complete sending prior to message store release.

A client is only allowed to control transport provider logoff if it is the only application 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 application.

Message store providers use **StoreLogoff** as a way to record the flags that the client application requires passed to the **IMAPISupport::StoreLogoffTransports** method. However, 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.

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* that it passes on the call to **StoreLogoffTransports**.

See Also

IMAPIStatus::FlushQueues method, **IMAPISupport::StoreLogoffTransports** method, **IXPLogon::FlushQueues** method

IMsgStore::Unadvise

[New - Windows 95]

Removes an object's registration for notification of message store changes previously established with a call to the **IMsgStore::Advise** method.

Syntax

HRESULT Unadvise(**ULONG** *ulConnection*)

Parameters

ulConnection
Input parameter containing the number of the registration connection previously returned by a call to **IMsgStore::Advise**.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

Comments

Message store providers use the **IMsgStore::Unadvise** method to cancel notifications for a message store object. To do so, **Unadvise** releases the pointer to the advise sink object passed in the *lpAdviseSink* parameter in the previous call to **IMsgStore::Advise**. As part of discarding the pointer to the advise sink, the 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 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

[New - Windows 95]

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 information about the last error that occurred for the message-store logon object.
Logoff	Logs the message store provider off.
OpenEntry	Opens a folder or a message within the message store given its entry identifier.
CompareEntryIDs	Compares two entry identifiers to determine if they refer to the same object in the message store.
Advise	Registers a message store provider for notifications on 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

[New - Windows 95]

Registers a message store provider for notifications on changes within the message store.

Syntax

HRESULT Advise(**ULONG** *cbEntryID*, **LPENTRYID** *lpEntryID*, **ULONG** *ulEventMask*,
  LPMAPIADVISESINK *lpAdviseSink*, **ULONG FAR** * *lpulConnection*)

Parameters

cbEntryID
  Input parameter containing the number of bytes in 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 is set up to provide 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 for which MAPI will generate notifications to filter 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 data structures:

Notification event type	Corresponding data structure
fnevCriticalError	**ERROR_NOTIFICATION**
fnevNewMail	**NEWMAIL_NOTIFICATION**
fnevObjectCreated	**OBJECT_NOTIFICATION**
fnevObjectDeleted	**OBJECT_NOTIFICATION**
fnevObjectModified	**OBJECT_NOTIFICATION**
fnevObjectCopied	**OBJECT_NOTIFICATION**
fnevSearchComplete	**OBJECT_NOTIFICATION**
fnevStatusObjectModified	**STATUS_OBJECT_NOTIFICATION**

lpAdviseSink
  Input parameter pointing to an advise sink object to be called when a event for the session object occurs about which notification has been requested. This advise sink object must have already been allocated.

lpulConnection
  Output parameter pointing to a variable that on 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.

Comments

Message store providers use the **IMSLogon::Advise** method to register an object implemented 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 has been set in the *ulEventMask* parameter and thus what type of change has 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, your client should use the **HrThisThreadAdviseSink** function.

To provide notifications, the service provider implementing **Advise** needs to keep a copy of the pointer to the advise sink object; to do so, it calls the **IUnknown::AddRef** method for the advise sink object to maintain the 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. Providers should therefore release their advise sink object after **Advise** returns unless they have a specific long-term use for it.

See Also

HrThisThreadAdviseSink function, **IMAPIAdviseSink::OnNotify** method, **IMSLogon::Unadvise** method, **NOTIFICATION** structure

IMSLogon::CompareEntryIDs

[New - Windows 95]

Compares two entry identifiers to determine if they refer to the same service provider 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.

Syntax

HRESULT CompareEntryIDs(**ULONG** *cbEntryID1*, **LPENTRYID** *lpEntryID1*, **ULONG** *cbEntryID2*, **LPENTRYID** *lpEntryID2*, **ULONG** *ulFlags*, **ULONG FAR *** *lpulResult*)

Parameters

cbEntryID1
Input parameter containing the number of bytes in 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 number of bytes in 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 that receives the result of the comparison; 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.

Comments

Message store providers use the **IMSLogon::CompareEntryIDs** method to compare two entry identifiers for a given message store entry and 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

[New - Windows 95]

Returns a **MAPIERROR** structure containing information about the last error that occurred for the message store object.

Syntax

HRESULT GetLastError(**HRESULT** *hResult*, **ULONG** *ulFlags*, **LPMAPIERROR FAR** * *lppMAPIError*)

Parameters

hResult
Input parameter containing the result returned for the last call on the session object that returned an error.

ulFlags
Input parameter containing a bitmask of flags controlling the type of the returned strings. The following flag can be set:

MAPI_UNICODE
Indicates the strings returned 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 8-bit 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.

Comments

Use the **IMSLogon::GetLastError** method to retrieve information to display as a message to the user regarding the last error returned from a method call on the session object.

To release all the memory allocated by MAPI for the **MAPIERROR** structure, client applications need only call the **MAPIFreeBuffer** function.

The return value from **GetLastError** must be S_OK for the application to make use of the **MAPIERROR** structure. Even if the return value is S_OK, it is still possible that a **MAPIERROR** structure won't be returned. If the implementation cannot determine what the last error was, or if a **MAPIERROR** structure is not available for that error, a pointer to NULL is returned in the *lppMAPIError* parameter instead.

See Also

MAPIERROR structure, **MAPIFreeBuffer** function

IMSLogon::Logoff

[New - Windows 95]

Logs the message store provider off.

Syntax

HRESULT Logoff(**ULONG FAR** * *lpulFlags*)

Parameters

lpulFlags
Reserved; must be a pointer to zero.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

Comments

Message store providers use the **IMSLogon::Logoff** method to forcibly shut down the message store. **IMSLogon::Logoff** is called in the following situations:

- While MAPI is logging off a client application 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 a **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

[New - Windows 95]

Opens a folder or a message in the message store given its entry identifier.

Syntax

HRESULT OpenEntry(**ULONG** *cbEntryID*, **LPENTRYID** *lpEntryID*, **LPCIID** *lpInterface*, **ULONG** *ulOpenFlags*, **ULONG FAR *** *lpulObjType*, **LPUNKNOWN FAR *** *lppUnk*)

Parameters

cbEntryID
Input parameter containing the number of bytes in 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 to be opened.

lpInterface
Input parameter pointing to the interface identifier (IID) for the folder or object. Passing NULL for the *lpInterface* parameter indicates the MAPI interface for the object will be returned. The *lpInterface* parameter can also be set to an identifier for another appropriate interface for the object.

ulOpenFlags
Input parameter containing a bitmask of flags used to control how the object is opened. The following flags can be set:

MAPI_BEST_ACCESS
Indicates the object should be opened with the maximum privileges allowed to the user. For example, if the client application has read-write privilege, the object is opened with read-write access; if the client application has read-only privilege, the object is opened with read-only access. The client application can retrieve the privilege 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 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 client applications should not work on the assumption that read-write access was granted.

lpulObjType
Output parameter pointing to a variable where the object type for 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.

Comments

MAPI calls the **IMSLogon::OpenEntry** method to open a folder or a message in the message store. MAPI passes in the entry identifier of the object to open; your 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 the reference entry for the **IMAPISupport::SetProviderUID** method.

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 the client application. 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

[New - Windows 95]

Opens a status object.

Syntax

HRESULT OpenStatusEntry(**LPCIID** *lpInterface*, **ULONG** *ulFlags*, **ULONG FAR** * *lpulObjType*, **LPVOID FAR** * *lppEntry*)

Parameters

lpInterface
Input parameter pointing to the interface identifier (IID) to be used for the returned object. Passing NULL for the *lpInterface* parameter indicates the MAPI interface for the object will be returned, in this case the **IMAPIStatus** interface. The *lpInterface* parameter can also be set to an identifier for another appropriate interface for the object.

ulFlags
Input parameter containing a bitmask of flags used to control how the status entry 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 was granted.

lpulObjType
Output parameter pointing to a variable that holds the type of the opened object.

lppEntry
Output parameter pointing to a variable where the pointer to the returned object is stored.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

Comments

Message store providers use the **IMSLogon::OpenStatusEntry** method to open a status object. This status object is then used to enable client applications 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

[New - Windows 95]

Removes an object's registration for notification of message store changes previously established with a call to the **IMSLogon::Advise** method.

Syntax

HRESULT Unadvise(**ULONG** *ulConnection*)

Parameters

ulConnection
Input parameter containing the number of the registration connection previously returned by a call to **IMSLogon::Advise**.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

Comments

Message store providers use the **IMSLogon::Unadvise** method to cancel notifications for a message store object. To do so, **Unadvise** releases the pointer to the advise sink object passed in the *lpAdviseSink* parameter in the previous call to **IMSLogon::Advise**. As part of discarding the pointer to the advise sink, the 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 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

[New - Windows 95]

The **IMSProvider** interface provides access to a message store provider through a message-store provider object. This message-store provider object is returned by the **MSProviderInit** entry point function of the message store provider's dynamic-link library (DLL). The object is primarily used by client applications or the MAPI spooler to open message stores.

MAPI uses one message-store provider object for all the message stores opened by the provider implementing the provider 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:

- A pointer to the *lpMalloc* 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

Vtable Order

Shutdown	Closes down a message store provider in an orderly fashion.
Logon	Logs MAPI onto one instance of a message store provider.
SpoolerLogon	Logs the MAPI spooler onto a message store.
CompareStoreIDs	Compares two message store entry identifiers to determine if they refer to the same store object.

IMSProvider::CompareStoreIDs

[New - Windows 95]

Compares two message-store entry identifiers to determine if they refer to the same store object.

Syntax

HRESULT CompareStoreIDs(ULONG *cbEntryID1*, **LPENTRYID** *lpEntryID1*, **ULONG** *cbEntryID2*, **LPENTRYID** *lpEntryID2*, **ULONG** *ulFlags*, **ULONG FAR *** *lpulResult*)

Parameters

cbEntryID1
: Input parameter containing the number of bytes in 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 number of bytes in 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 that receives the result of the comparison; 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.

Comments

MAPI calls the **IMSProvider::CompareStoreIDs** method while processing a call to the **IMAPISession::OpenMsgStore** method; it calls **CompareStoreIDs** at this point to determine which section in the profile, 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 call made by the message store provider to the **IMAPISupport::OpenProfileSection** method.

The entry identifiers compared in the **CompareStoreIDs** call are both for the current message store provider's dynamic link library (DLL) and unwrapped store entry identifiers. For more information on wrapping store entry identifiers, see the reference entry for the **IMAPISupport::WrapStoreEntryID** method.

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

[New - Windows 95]

Logs MAPI onto one instance of a message store provider.

Syntax

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 MAPI support object for the message store.

ulUIParam
Input parameter containing the handle of the window the logon dialog box is modal to.

lpszProfileName
Input parameter pointing to a string containing the name of the profile being used for the 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 number of bytes in 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 used to control the logon. 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 application. 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 8-bit 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 the 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 to not 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 the store on so that information can be retrieved programmatically from the profile section. 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 be logged onto. Passing NULL for the *lpInterface* parameter indicates the MAPI interface for the message store will be returned − that is, the **IMsgStore** interface. The *lpInterface* parameter can also be set to an identifier for another appropriate interface for the message store, for example IID_IUnknown or IID_IMAPIProp.

lpcbSpoolSecurity
Output parameter pointing to the variable in which 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 validation data to be returned is stored. This validation data is provided so the **IMSProvider::SpoolerLogon** method can log the MAPI spooler onto the same store as the message store provider.

lppMAPIError
Output parameter pointing to a variable where the pointer to the returned **MAPIERROR** structure, if any, is stored. 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_LOGON_FAILED
A logon session could not be established.

MAPI_E_UNCONFIGURED
The profile does not contain enough information for the logon to complete. MAPI calls the provider's message service entry function.

MAPI_E_USER_CANCEL
The user canceled the operation, typically by clicking the Cancel button in the dialog box.

MAPI_W_ERRORS_RETURNED
The call succeeded, but the message store provider has error information available. Use the **HR_FAILED** macro to test for this warning, but the call should be handled as a successful return. Call **IMAPISession::GetLastError** to get the error information from the provider.

Comments

MAPI uses 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 use.

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 this store. The message-store logon object and the message store object should be tightly linked inside your 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 functions 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 appropriate for 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 the *ulFlags* parameter contains the MDB_NO_DIALOG flag. If a user interface is called but *ulFlags* does not allow it, or if for some other reason the user interface cannot be displayed, the provider should return the error value MAPI_E_LOGON_FAILED. If **Logon** displays a dialog box and the user cancels logon, typically by clicking the Cancel button in the dialog box, your provider should return the value 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 application as a PR_STORE_ENTRYID property.
- It can be an entry identifier that the provider previously wrapped and returned to a calling application 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.

Within this entry identifier, the current store provider should have put all 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 selection 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 the message service's profile section at configuration.

If your provider finds that all the required information is not in the profile, it should return MAPI_E_UNCONFIGURED so that the provider's message service entry function is called by MAPI to enable the user to select the 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 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 the MAPI message store interface 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 the MAPI message store interface. When IID_IUnknown is passed, after the open operation succeeds the calling application 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 any user interface. 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, your provider shouldn't put addresses into the buffer. For more information on MAPI spooler logon, see the reference entry for the **IMSProvider::SpoolerLogon** method.

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 the reference entry for the **IProfSect : IMAPIProp** interface.

In addition to any private data in properties with identifiers 0x6600 through 0x67FF, the store provider should provide information for the PR_RESOURCE_DISPLAY and PR_DISPLAY_NAME properties in its profile section. It should place in PR_RESOURCE_DISPLAY the display name of the provider itself − for example, Microsoft Personal Information Store. In PR_DISPLAY_NAME, the provider places an identifying string displayed to users so they can distinguish this message store from others they might have access to. Commonly, server names, user account names, or paths are used in PR_DISPLAY_NAME.

Some profile section properties are visible in the message stores 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 the reference entry for **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 use **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 by the client.

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, your provider should check information returned to see whether a store has been opened a second time. In addition, your 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. Your 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 your store supports, your provider should return the error value MAPI_E_INTERFACE_NOT_SUPPORTED.
2. Your provider returns a pointer to the required interface of the existing store object in *lppMDB*.
3. Your provider returns NULL in *lppMSLogon*.
4. Your provider should not open the profile for the support object passed in on the call. Neither should it register a provider unique identifier (UID), register a status row, nor return MAPI spooler logon data.
5. Your 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, your provider sets the members in the **MAPIERROR** structure. MAPI looks for, uses, and releases the **MAPIERROR** structure if it is returned by your provider.

Memory for this **MAPIERROR** structure 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 *ulFlags;* otherwise, they should be in the appropriate 8-bit string character set for the platform.

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

[New - Windows 95]

Closes down a message store provider in an orderly fashion.

Syntax

HRESULT Shutdown(**ULONG FAR *** *lpulFlags*)

Parameters

lpulFlags
Reserved; must be a pointer to zero.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

Comments

MAPI calls the **IMSProvider::Shutdown** method just prior to releasing the message-store provider object. MAPI releases all logon objects for a provider before calling **Shutdown** for that provider.

IMSProvider::SpoolerLogon

[New - Windows 95]

Logs the MAPI spooler onto a message store.

Syntax

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 window the logon dialog box is modal to.

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 number of bytes in 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 used to control the logon. 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 application. 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 8-bit 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 the 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 be logged onto. Passing NULL for the *lpInterface* parameter indicates the MAPI interface for the message store will be returned – that is, the **IMsgStore** interface. The *lpInterface* parameter can also be set to an identifier for another appropriate interface for the message store, for example IID_IUnknown or IID_IMAPIProp.

cbSpoolSecurity

Input parameter pointing to the variable in which the store provider returns the size, in bytes, of validation data in the *lppbSpoolSecurity* parameter.

lpbSpoolSecurity
: Input parameter pointing to a variable where the pointer to validation data to be returned is stored. 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 variable where the pointer to the returned **MAPIERROR** structure, if any, is stored. 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. MAPI calls the provider's message service entry function.

MAPI_W_ERRORS_RETURNED
: The call succeeded, but the message store provider has error information available. Use the **HR_FAILED** macro to test for this warning, but the call should be handled as a successful return. Call **IMAPISession::GetLastError** to get the error information from the provider.

Comments

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.

See Also

IMSProvider::Logon method, **MAPIERROR** structure

IPersistMessage : IUnknown

[New - Windows 95]

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 working with storage objects in general, see *Inside OLE, Second Edition,* and *OLE Programmer's Reference, Volume One*.

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

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 or submission operation.
HandsOffMessage	Causes a message to release its storage object.

IPersistMessage::GetClassID

[New - Windows 95]

Returns a form's message class identifier.

Syntax

HRESULT GetClassID(**LPCLSID** *lpClassID*)

Parameters

lpClassID
Input parameter pointing to the returned class identifier.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

Comments

For more information on working with class identifiers of storage objects, see the documentation for the **IPersistStorage** methods in *OLE Programmer's Reference, Volume One.*

IPersistMessage::HandsOffMessage

[New - Windows 95]

Causes a form to release its message object.

Syntax

HRESULT HandsOffMessage()

Parameters

None

Return Values

S_OK
: The call succeeded and has returned the expected value or values.

Comments

For more information on working with the hands-off state of storage objects, see *OLE Programmer's Reference, Volume One.*

IPersistMessage::InitNew

[New - Windows 95]

Provides a form with a base message on which to build a new message.

Syntax

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.

Comments

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 in a form with **InitNew**, it can be assumed that the following required properties, and no others, have been set by form server implementations:

PR_ORIGINATOR_DELIVERY_REPORT_REQUESTED, PR_READ_RECEIPT_REQUESTED, PR_IMPORTANCE, PR_PRIORITY, PR_SENSITIVITY, PR_SENTMAIL_ENTRYID, PR_DELETE_AFTER_SUBMIT.

For more information on initializing new storage objects, see *OLE Programmer's Reference, Volume One.*

IPersistMessage::IsDirty

[New - Windows 95]

Checks a form for changes made since the form was last saved.

Syntax

HRESULT IsDirty()

Parameters

None

Return Values

S_OK
  The form is dirty.

S_FALSE
  The form is not dirty.

Comments

For more information, see the documentation for the **IPersistStorage** methods in *OLE Programmer's Reference, Volume One.*

IPersistMessage::Load

[New - Windows 95]

Loads a form from a specified message.

Syntax

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 application - or service 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 indicates the current state of the message.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

Comments

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 *OLE Programmer's Reference, Volume One.*

See Also

PR_MESSAGE_FLAGS property, PR_MSG_STATUS property

IPersistMessage::Save

[New - Windows 95]

Saves a revised form back to the message from which it was loaded or created.

Syntax

HRESULT Save(**LPMESSAGE** *pMessage*, **ULONG** *fSameAsLoad*)

Parameters

pMessage
Input parameter pointing to a message.

fSameAsLoad
Input parameter indicating whether the message the *pMessage* parameter points to is the message from which the form was loaded or created.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

Comments

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 the **save** method.

For more information on saving storage objects, see the documentation on the **IPersistStorage** methods in *OLE Programmer's Reference, Volume One.*

IPersistMessage::SaveCompleted

[New - Windows 95]

Returns a message to a form after a save or submission operation.

Syntax

HRESULT SaveCompleted(**LPMESSAGE** *pMessage*)

Parameters

pMessage
Input parameter pointing to the message.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

Comments

Form viewers call the **IPersistMessage::SaveCompleted** method at the end of a save request or a failed dismissive operation such as save or delete. **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 was held for the message at the beginning of the save 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 *OLE Programmer's Reference, Volume One.*

IProfAdmin : IUnknown

[New - Windows 95]

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 information about the last error that occurred for the 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 changes the message services in a profile.

IProfAdmin::AdminServices

[New - Windows 95]

Enumerates and changes the message services in a profile.

Syntax

HRESULT AdminServices(**LPTSTR** *lpszProfileName*, **LPTSTR** *lpszPassword*, **ULONG** *ulUIParam*, **ULONG** *ulFlags*, **LPSERVICEADMIN FAR** * *lppServiceAdmin*)

Parameters

lpszProfileName
Input parameter pointing to a string containing the profile name. This parameter must not be NULL.

lpszPassword
Input parameter pointing to a string containing the profile 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 window the dialog box is modal to.

ulFlags
Input parameter containing a bitmask of flags. 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 8-bit format.

MAPI_DIALOG
Displays a dialog box prompting the user for the correct password. If this flag is not set, no dialog box is displayed.

lppServiceAdmin
Output parameter pointing to a variable where the pointer to the message-service administration object is stored. The message-service administration object is used to change message service settings in the profile indicated in the *lpszProfileName* parameter.

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 the MAPI_DIALOG flag was not set in the *ulFlags* parameter.

MAPI_E_USER_CANCEL
The user canceled the operation, typically by clicking the Cancel button in the dialog box.

Comments

Use the **IProfAdmin::AdminServices** method to obtain a message-service administration object so as to make configuration changes to the message services within a profile. Applications 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, applications should use **IMAPISession::AdminServices**.

Applications calling **IProfAdmin::AdminServices** must specify an existing profile in *lpszProfileName*.

When 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 in length. 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 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™ and Windows 95 do not support passwords; Windows version 3.1 does.

See Also

IMAPISession::AdminServices method

IProfAdmin::ChangeProfilePassword

[New - Windows 95]

Changes the password for a profile.

Syntax

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 controlling the type of the 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 8-bit 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 platform does not support passwords.

Comments

Use the **IProfAdmin::ChangeProfilePassword** method to replace one profile password string with another. **ChangeProfilePassword** never displays a user interface.

Profile providers are not required to implement support for profile passwords. MAPI_E_NO_SUPPORT is returned if the operating system platform does not support passwords. Currently, Windows NT and Windows 95 do not support passwords; Windows version 3.1 does.

IProfAdmin::CopyProfile

[New - Windows 95]

Copies a profile.

Syntax

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 be copied.

lpszOldPassword
Input parameter pointing to a string containing the password of the profile to be copied.

lpszNewProfileName
Input parameter pointing to a string containing the name of the new profile to be created.

ulUIParam
Input parameter containing the handle of the window the dialog box is modal to.

ulFlags
Input parameter containing a bitmask of flags used to control how the profile is copied. The following flags can be set:

MAPI_DIALOG
Displays a dialog box prompting the user for the correct 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 8-bit 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 an existing profile.

MAPI_E_LOGON_FAILED
The old password is incorrect, and a dialog box could not be displayed to the user requesting the correct password, because the MAPI_DIALOG flag 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 the dialog box.

Comments

The **IProfAdmin::CopyProfile** method makes a copy of the profile indicated in the *lpszOldProfileName* parameter and 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 in length. 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 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 and Windows 95 do not support passwords; Windows version 3.1 does.

If a client application passes NULL in *lpszOldPassword* and the profile 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 the MAPI_DIALOG flag is not set in the *ulFlags* parameter, the call returns MAPI_E_LOGON_FAILED instead of prompting the user to provide the correct password.

IProfAdmin::CreateProfile

[New - Windows 95]

Creates a new profile.

Syntax

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 to the window the dialog box is modal to.

ulFlags
Input parameter containing a bitmask of flags used to control how the profile is created. The following flags can be set:

MAPI_DEFAULT_SERVICES
Indicates that 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 8-bit format.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

MAPI_E_NO_ACCESS
The specified profile already exists.

Comments

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, the message-service entry function is called for each message service in the [Default Services] section in the MAPISVC.INF file. The message service entry function is called with the MSG_SERVICE_CREATE value set in the *ulContext* parameter . 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 service provider's message-service entry function is called. Service providers should display their configuration property sheets so that the user can configure the message service. The message-service entry function is only called after all available information from the MAPISVC.INF file has been added to the profile.

Profile providers are required to support names and passwords up to 64 characters in length. 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 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 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

[New - Windows 95]

Deletes a profile.

Syntax

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 controlling the type of the 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 strings are in 8-bit 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.

Comments

Use the **IProfAdmin::DeleteProfile** method to delete profiles. If the profile indicated in the *lpszProfileName* parameter does not exist, MAPI_E_NOT_FOUND is returned. If the profile to be deleted is in use by an application when **DeleteProfile** is called, S_OK is returned, but the profile is not deleted immediately. Instead, MAPI marks the profile for deletion and deletes it after all applications have logged off the profile.

The message-service entry function is called for each message service in the MAPISVC.INF file before each service is removed from the profile. The message-service entry function is called with the MSG_SERVICE_DELETE value set in the *ulContext* parameter . First the service is deleted, and then the service's profile section skeleton is deleted. The message-service entry 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

[New - Windows 95]

Returns a **MAPIERROR** structure containing information about the last error that occurred for the profile administration object.

Syntax

HRESULT GetLastError(**HRESULT** *hResult*, **ULONG** *ulFlags*, **LPMAPIERROR FAR** * *lppMAPIError*)

Parameters

hResult
Input parameter containing the result returned for the last call on the session object that returned an error.

ulFlags
Input parameter containing a bitmask of flags controlling the type of the returned strings. The following flag can be set:

MAPI_UNICODE
Indicates the strings returned 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 8-bit 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.

Comments

Use the **IProfAdmin::GetLastError** method to retrieve information to display as a message to the user regarding the last error returned from a method call on the profile administration object.

To release all the memory allocated by MAPI for the **MAPIERROR** structure, client applications need only call the **MAPIFreeBuffer** function.

The return value from **GetLastError** must be S_OK for the application to make use of the **MAPIERROR** structure. Even if the return value is S_OK, it is still possible that a **MAPIERROR** structure won't be returned. If the implementation cannot determine what the last error was, or if a **MAPIERROR** structure is not available for that error, a pointer to NULL is returned in the *lppMAPIError* parameter instead.

See Also

MAPIERROR structure, **MAPIFreeBuffer** function

IProfAdmin::GetProfileTable

[New - Windows 95]

Returns a table listing all profiles associated with a particular client application.

Syntax

HRESULT GetProfileTable(**ULONG** *ulFlags*, **LPMAPITABLE FAR** * *lppTable*)

Parameters

ulFlags
Input parameter containing a bitmask of flags controlling the type of the returned strings 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 8-bit format.

lppTable
Output parameter pointing to a variable where the pointer to the returned table object is stored.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

Comments

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
The name of a particular profile. To log onto this profile, your client passes this name to the **MAPILogonEx** function.

PR_DEFAULT_PROFILE
A value indicating whether this profile is the default profile. If this column is present, its value must also be TRUE.

Profiles that have been deleted, or 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 adding or deleting profiles. **IMAPITable::Advise** calls on the profile table return S_OK, but no changes are made to the table. If no profile exists, **GetProfileTable** does not return an error; a table object supporting the **IMAPITable** interface is returned. If you call the **IMAPITable::QueryRows** method 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 profile table by the **IMAPITable::QueryColumns** method. The initial active columns for a profile table are those columns the **QueryColumns** method returns before the application that contains the profile 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 application that contains the profile table calls the **IMAPITable::SetColumns** method.
- Controls the property types of the sort order specification returned by the **IMAPITable::QuerySortOrder** method before the application that contains the profile table calls the

IMAPITable::SortTable method.

See Also

IMAPITable : IUnknown interface, **MAPILogonEx** function

IProfAdmin::RenameProfile

[New - Windows 95]

Renames a profile.

Syntax

HRESULT RenameProfile(**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 be renamed.

lpszOldPassword
Input parameter pointing to a string containing the password of the profile to be renamed. If NULL is passed in the *lpszOldPassword* parameter, and the profile requires a password, the profile provider will display a dialog box prompting the user for the password.

lpszNewProfileName
Input parameter pointing to a string containing the name of the profile to be renamed.

ulUIParam
Input parameter containing the handle of the window the dialog box is modal to.

ulFlags
Input parameter containing a bitmask of flags controlling the rename operation. The following flags can be set:

MAPI_DIALOG
Displays a dialog box prompting the user for the correct 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 8-bit format.

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 the dialog box.

MAPI_E_LOGON_FAILED
The password is incorrect.

Comments

The **IProfAdmin::RenameProfile** method 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 the password. If the profile to be renamed is in use by an application when **RenameProfile** is called, S_OK is returned, but the profile is not renamed immediately. Instead, MAPI marks the profile for renaming and renames it after all applications have logged off the profile.

Profile providers are required to support names and passwords up to 64 characters in length. 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 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 and Windows 95 do not support passwords; Windows version 3.1 does.

If a client application passes NULL in *lpszOldPassword* and the profile 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 the MAPI_DIALOG flag is not set in the *ulFlags* parameter, the call returns MAPI_E_LOGON_FAILED instead of prompting the user to provide the correct password.

IProfAdmin::SetDefaultProfile

[New - Windows 95]

Sets the default profile for a client application.

Syntax

HRESULT SetDefaultProfile(**LPTSTR** *lpszProfileName*, **ULONG** *ulFlags*)

Parameters

lpszProfileName
Input parameter pointing to a string containing the name of the new default profile. If your client application passes either NULL or a pointer to a zero-length string in the *lpszProfileName* parameter, there will not be any default profile and any existing default profile setting is removed.

ulFlags
Input parameter containing a bitmask of flags controlling the type of the 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 8-bit 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.

Comments

Use the **IProfAdmin::SetDefaultProfile** method to set the name of the profile to be used as the default profile when a particular client application logs onto a MAPI session. This is the profile that is used when applications 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, **MAPILogon** function, **MAPILogonEx** function, PR_DEFAULT_PROFILE property

IProfSect : IMAPIProp

[New - Windows 95]

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 restrictions.

The profile section object does not support a transaction model, so all changes made to a profile section following calls to the **IProfSect::CopyProps** and **IProfSect::CopyTo** methods occur immediately. Calls to the **IProfSect::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. Rather than providing a direct implementation of this sort, service providers should instead implement their property sheets and dialog boxes using copies of the profile section object as following:

1. Open the profile section with the **IMAPISupport::OpenProfileSection** method.
2. Get an IPropData object.
3. Use the **IProfSect::CopyTo** method to copy properties from the profile section to the IPropData object.
4. Use the IPropData object as the source object for the data in the *lpConfigData* parameter of the **IMAPISupport::DoConfigPropSheet** method used to display your configuration user interface.
5. When the user saves changes to the property sheet, call the **IMAPIProp::CopyTo** method to copy the properties from the IPropData object to the profile section.

Profile section objects also do not support named properties; both the **IProfSect::GetIDsFromNames** and **IProfSect::GetNamesFromIDs** methods return MAPI_E_NO_SUPPORT if called on a profile section object.

Calls to the **IProfSect::SetProps** method on properties in the range above x8000 return PT_ERROR as the property type.

Profile sections also reserve the range 0X67F0 to 0X67FF as 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 **IProfSect::GetProps** method, nor are they returned in the *lppPropTagArray* parameter of the **IProfSect::GetPropList** method. Secure properties must be requested specifically by their identifier.

For more information on working with profile section objects, see *MAPI Programmer's Guide*.

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
Called by:	Client applications

Vtable Order

No unique methods.

IPropData : IMAPIProp

[New - Windows 95]

Client applications and service providers use the **IPropData** interface to add properties to an object and to get and set access rights for objects and properties. The **IPropData** interface is derived from **IMAPIProp** and does not have an object type of its own. To get an **IPropData** interface, call the **QueryInterface** method passing in an interface identifier of IID_IPropData. For additional information about using the **IPropData** interface to manage data associated with **IMAPIProp** interfaces, see *MAPI Programmer's Guide*.

At a Glance

Specified in header file:	MAPIUTIL.H
Object that supplies this interface:	This interface has no object type of its own.
Corresponding pointer type:	LPPROPDATA
Implemented by:	MAPI
Called by:	Client applications, service providers

Vtable Order

HrSetObjAccess	Sets the access rights for an object.
HrSetPropAccess	Sets the access rights for the specified properties, and also sets the state of the object relative to the IPROP_CLEAN or IPROP_DIRTY flags.
HrGetPropAccess	Returns the access rights currently set for the specified properties.
HrAddObjProps	Adds properties of type PT_OBJECT to an object.

IPropData::HrAddObjProps

[New - Windows 95]

Adds properties of type PT_OBJECT to an object.

Syntax

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 be added to the object.

lppProblems
: Output parameter pointing to a variable where the pointer to the returned **SPropProblemArray** structure is stored. This structure indicates the 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, then no **SPropProblemArray** is returned even if some properties were not added to the object.

Return Values

S_OK
: The call succeeded and has returned the expected value or values.

MAPI_E_NO_ACCESS
: The object has been set to not allow read-write access.

MAPI_E_INVALID_TYPE
: A property type other than PT_OBJECT was passed in the **SPropTagArray** structure.

MAPI_W_PARTIAL_COMPLETION
: Some, but not all of the properties, were added.

Comments

Use 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 the *lppProblems* parameter, 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 has previously been set to disallow read-write access, MAPI_E_NO_ACCESS is returned. To obtain read-write access, 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

[New - Windows 95]

Returns the access rights currently set for the specified properties.

Syntax

HRESULT HrGetPropAccess(**LPSPropTagArray FAR** * *lppPropTagArray*, **ULONG FAR * FAR** * *lprgulAccess*)

Parameters

lppPropTagArray
Input-output parameter that on input contains an **SPropTagArray** structure indicating the property tags for which to return access rights. If a pointer to NULL is passed in *lppPropTagArray,* then access rights for all properties on the object are sought. On output, *lppPropTagArray* points to a variable where the returned **SPropTagArray** structure is stored. This returned **SPropTagArray** structure 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**. For each property tag, the following flags can be returned:

IPROP_CLEAN
Indicates that the property hasn't been modified.

IPROP_DIRTY
Indicates that the property has been modified.

IPROP_READONLY
Indicates that the property is read-only.

IPROP_READWRITE
Indicates that the property is read-write.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

Comments

Use 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 you requested has already been deleted, it will not be returned in the **SPropTagArray** structure; zero is returned instead. If you passed a pointer to NULL in the *lppPropTagArray* parameter, the deleted property will not be returned in the array. If a property has been modified, its IPROP_DIRTY flag will be set.

See Also

SPropTagArray structure

IPropData::HrSetObjAccess

[New - Windows 95]

Sets the access rights for an object.

Syntax

HRESULT HrSetObjAccess(**ULONG** *ulAccess*)

Parameters

ulAccess
Input parameter containing a bitmask of flags used to set the access rights of the object. One of the following flags can be set:

IPROP_READONLY
Indicates that the properties on the object are read-only.

IPROP_READWRITE
Indicates that the properties on the object are read-write.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

Comments

Use the **IPropData::HrSetObjAccess** method to set the access rights for an object. By default, all MAPI objects have read-write access set when they are created. If you want to set access rights on individual properties, first set the access rights on the object by calling **HrSetObjAccess** with the IPROP_READWRITE flag set in the *ulAccess* parameter. Then call **IPropData::HrSetPropAccess** to set the access for the properties.

See Also

IPropData::HrGetPropAccess method, **IPropData::HrSetPropAccess** method

IPropData::HrSetPropAccess

[New - Windows 95]

Sets the access rights for the specified properties, and also sets the state of the object relative to the IPROP_CLEAN or IPROP_DIRTY flags.

Syntax

HRESULT HrSetPropAccess(LPSPropTagArray *lpPropTagArray*, **ULONG FAR *** *rgulAccess*)

Parameters

lpPropTagArray
: Input parameter pointing to an **SPropTagArray** structure containing an array of property tags indicating 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 to modify a read-only object or an attempt to access an object for which the user has insufficient permissions.

Comments

Use the **IPropData::HrSetPropAccess** method to set the access rights for each individual property in the **SPropTagArray** structure in the *lpPropTagArray* parameter. These rights are set based on the flags set in the corresponding bitmasks in the *rgulAccess* parameter. By default, all MAPI properties have read-write access set when they are created.

The valid flag combinations for the bitmask in the *rgulAccess* parameter are shown in the following table.

IPROP_READONLY	IPROP_READWRITE	IPROP_CLEAN	IPROP_DIRTY
•		•	
•			•
	•	•	
	•		•

The flags IPROP_READONLY and IPROP_READWRITE are mutually exclusive, as are the flags

IPROP_CLEAN and IPROP_DIRTY. If any flag combinations are set other than those listed directly previously, **HrSetPropAccess** returns MAPI_E_INVALID_PARAMETER.

See Also

IPropData::HrSetPropAccess method, **SPropTagArray** structure

IProviderAdmin : IUnknown

[New - Windows 95]

The **IProviderAdmin** interface is used to manage the service providers within a message service. You can get a pointer to an **IProviderAdmin** interface in two ways: by calling the **IMsgServices::AdminProviders** method or from within a provider's message-service entry 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 function. Most message services do not allow providers to be added or deleted while the profile is in use. The results are unpredictable if you make 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, message-service entry function

Vtable Order

GetLastError	Returns information about the last error that occurred for a provider administration object.
GetProviderTable	Returns a table object 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

[New - Windows 95]

Adds a service provider to a message service.

Syntax

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 be added to the message service.

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 window the dialog box is modal to. The *ulUIParam* parameter is used if the MAPI_DIALOG flag is set in the *ulFlags* parameter.

ulFlags
Input parameter containing a bitmask of flags controlling the type of the 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 8-bit format.

lpUID
Output parameter pointing to the **MAPIUID** structure holding the unique identifier for the provider to be added.

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 the dialog box.

Comments

Use the **IProviderAdmin::CreateProvider** method to add a provider to a message service. The string in the *lpszProvider* parameter must be for a provider that belongs to the message service. MAPI does not verify that they do in fact match; if they don't, 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 function for the message service is called with the MSG_SERVICE_PROVIDER_CREATE value set in the *ulContext* parameter. If the MAPI_DIALOG flag is set in the **CreateProvider** method's *ulFlags* parameter, then the values in the *ulUIParam* and *ulFlags* parameters are also passed when the service provider's message-service entry function is called. Service providers should display their configuration property sheets so that the user can configure the message service.

The message-service entry 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

[New - Windows 95]

Deletes a service provider from a message service.

Syntax

HRESULT DeleteProvider(**LPMAPIUID** *lpUID*)

Parameters

lpUID
Input parameter pointing to the **MAPIUID** structure holding the unique identifier for the provider being deleted.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

MAPI_E_NOT_FOUND
The **MAPIUID** was not recognizable.

Comments

Use 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 be deleted is in use by an application when **DeleteProvider** is called, S_OK is returned, but the provider is not deleted immediately. Instead, MAPI marks the provider for deletion and deletes it after all applications have logged off the provider.

The message-service entry function is called for the message service before the provider is removed from the service. The message-service entry function is called with the MSG_SERVICE_PROVIDER_DELETE value set in the *ulContext* parameter . First the provider is deleted, then the provider's profile section skeleton is deleted. The message service entry function is not called again after the provider has been deleted.

See Also

MAPIUID structure, **MSGSERVICEENTRY** function prototype

IProviderAdmin::GetLastError

[New - Windows 95]

Returns a **MAPIERROR** structure containing information about the last error that occurred for the provider administration object.

Syntax

HRESULT GetLastError(**HRESULT** *hResult*, **ULONG** *ulFlags*, **LPMAPIERROR FAR *** *lppMAPIError*)

Parameters

hResult
Input parameter containing the result returned for the last call on the provider administration object that returned an error.

ulFlags
Input parameter containing a bitmask of flags controlling the type of the returned strings. The following flag can be set:

MAPI_UNICODE
Indicates the strings returned 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 8-bit 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.

Comments

Use the **IProviderAdmin::GetLastError** method to retrieve information to display as a message to the user regarding the last error returned from a method call on the provider administration object.

To release all the memory allocated by MAPI for the **MAPIERROR** structure, client applications need only call the **MAPIFreeBuffer** function.

The return value from **GetLastError** must be S_OK for the application to make use of the **MAPIERROR** structure. Even if the return value is S_OK, it is still possible that a **MAPIERROR** structure won't be returned. If the implementation cannot determine what the last error was, or if a **MAPIERROR** structure is not available for that error, a pointer to NULL is returned in the *lppMAPIError* parameter instead.

See Also

MAPIERROR structure, **MAPIFreeBuffer** function

IProviderAdmin::GetProviderTable

[New - Windows 95]

Returns a table object listing the service providers in a message service.

Syntax

HRESULT GetProviderTable(**ULONG** *ulFlags*, **LPMAPITABLE FAR** * *lppTable*)

Parameters

ulFlags
Input parameter containing a bitmask of flags controlling the strings in the default column set of the table. The following flag can be set:

MAPI_UNICODE
Indicates the returned strings are to be in Unicode format. If the MAPI_UNICODE flag is not set, the strings are in 8-bit format.

lppTable
Output parameter pointing to a variable where the pointer to the returned table object is stored.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

Comments

Use 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
The provider's name as given by the provider.

PR_INSTANCE_KEY
A search key for an instance of a particular MAPI object.

PR_PROVIDER_DISPLAY
The name of the provider as given by the message service. This name is not localizable and is a reliable way to search on a particular provider.

PR_PROVIDER_DLL_NAME
The dynamic-link library (DLL) for a particular provider.

PR_PROVIDER_UID
Identifies a particular instance of a provider.

PR_RESOURCE_TYPE
An enumeration describing the type of the provider. This property can be used to determine all the providers of a particular type that are installed within a profile.

PR_SERVICE_NAME
The name of the message service as given by the message service. This name is not localizable and is a reliable way to search on a particular service.

PR_SERVICE_UID
Identifies a particular instance of a message service.

The following properties are computed only for transport providers:

PR_PROVIDER_ORDINAL

A number computed by MAPI that identifies the order in which the providers are listed in the profile.

PR_RESOURCE_FLAGS
: Flags describing the capabilities of the provider.

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 your client application 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 administration table. Once a provider administration table has been returned, it does not reflect changes being made to the profile, such as adding or deleting providers. **IMAPITable::Advise** calls on the provider administration table return S_OK, but no changes are made to the table. If no provider exists, **GetProviderTable** does not return an error; a table object supporting the **IMAPITable** interface is returned. If you call **IMAPITable::QueryRows** 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 administration table by the **IMAPITable::QueryColumns** method. The initial active columns for a provider administration table are those columns the **QueryColumns** method returns before the application that contains the provider administration table calls the **IMAPITable::SetColumns** method.
- Sets the string type to Unicode for data returned for the initial active rows of the provider administration table by the **IMAPITable::QueryRows** method. The initial active rows for a provider administration table are those rows **QueryRows** returns before the application that contains the provider administration table calls the **IMAPITable::SetColumns** method.
- Controls the property types of the sort order specification returned by the **IMAPITable::QuerySortOrder** method before the application that contains the provider administration table calls the **IMAPITable::SortTable** method.

See Also

IMAPITable::QueryColumns method, **IMAPITable::QueryRows** method, **IMAPITable::QuerySortOrder** method, **IMAPITable::SetColumns** method, **IMsgServiceAdmin::GetProviderTable** method

IProviderAdmin::OpenProfileSection

[New - Windows 95]

Opens a section of the current profile and returns a pointer that provides further access to the profile object.

Syntax

HRESULT OpenProfileSection(**LPMAPIUID** *lpUID*, **LPCIID** *lpInterface*, **ULONG** *ulFlags*,
LPPROFSECT FAR * *lppProfSect*)

Parameters

lpUID
Input parameter pointing to the **MAPIUID** structure holding the unique identifier for the profile section. For client applications, the *lpUID* parameter must not be NULL. Providers can pass NULL to get the **MAPIUID** when calling from their message-service entry function.

lpInterface
Input parameter pointing to the interface identifier (IID) for the profile section object. Passing NULL for the *lpInterface* parameter indicates the identifier for the MAPI profile-section object interface, IID_IProfSect, is used. The *lpInterface* parameter can also be set to an identifier for an appropriate interface, for example IID_IMAPIProp or IID_IProfSect.

ulFlags
Input parameter containing a bitmask of flags used to control 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 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; providers should not work under the assumption that read-write access was granted. Client applications are not allowed read-write access to provider sections of the profile.

lppProfSect
Output parameter pointing to a variable where the pointer to the open 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 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.

Comments

Use 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. The default behavior is to open the profile section as read-only, unless an application sets the MAPI_MODIFY flag in the *ulFlags* parameter. Client applications cannot open profile sections belonging to service 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 application has the profile section open, a read-write open operation fails and returns the value 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 the MAPI_MODIFY flag in the *ulFlags* parameter, the call creates the section. If an **OpenProfileSection** call attempts to open a nonexistent section with read-only access, MAPI_E_NOT_FOUND is returned.

All open operations should be as brief as possible, but an application that is writing to a profile section can keep it open while displaying a modification dialog box.

See Also

IMAPIProp : IUnknown interface, **IProfSect : IMAPIProp** interface, **MAPIUID** structure

ISpoolerHook : IUnknown

[New - Windows 95]

The **ISpoolerHook** interface enables your message 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:	Message-hook provider object
Corresponding pointer type:	LPSPOOLERHOOK
Implemented by:	Message-hook providers
Called by:	The MAPI spooler

Vtable Order

InboundMsgHook	Informs the MAPI spooler that it should allow the message hook provider to reroute a message from the default Inbox to another folder.
OutboundMsgHook	Informs the MAPI spooler that it should allow the message hook provider to reroute a message from the default Sent Items folder to another folder.

ISpoolerHook::InboundMsgHook

[New - Windows 95]

Informs the MAPI spooler that it should allow the message hook provider to reroute a message from the default Inbox to another folder.

Syntax

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 be rerouted.

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 used to control 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 your hook closes, moves, or deletes the message, you should set this flag.

  HOOK_DELETE
  : Indicates that 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 of the folder where the message will be moved.

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.

Comments

Message hook providers use the **ISpoolerHook::InboundMsgHook** method to reroute a message from the default Inbox to another folder. Before moving the message to another folder or message store, your message hook provider should call the **IUnknown::QueryInterface** method on the message object to make sure your provider can get an interface for the message that is compatible with your provider's implementation.

Before rerouting messages, message hook providers must replace the passed-in entry identifier in the *lppEntryID* parameter with the entry identifier of the new target folder. The MAPI spooler moves the message to the indicated folder for your provider, unless another hook function replaces your folder entry identifier. If your hook requires that its operation be the final action on the message, it can set the HOOK_CANCEL flag in *lpulFlags* before returning. If your provider replaces the *lppEntryID* entry identifier, it must call the **MAPIFreeBuffer** function to free the previous one. The copy of the entry identifier that your provider stores in *lppEntryID* should be allocated using the **MAPIAllocateBuffer** function.

If the message hook provider's implementation must move the message to another folder itself, it should close the message, place zero in the *lpcbEntryID* parameter, and free the entry identifier in the *lppEntryID* parameter, if *lppEntryID* is not already NULL. The hook then sets *lppEntryID* to NULL and then places a pointer to the message's new parent folder in *lpFolder*. Message hook providers that move the message must set the HOOK_CANCEL flag in the *lpulFlags* parameter.

If your 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 the HOOK_CANCEL flag in the *lpulFlags* parameter. 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 message-hook provider object at session shutdown. If your 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

[New - Windows 95]

Informs the MAPI spooler that it should allow the message hook provider to reroute a message from the default Sent Items folder to another folder.

Syntax

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 be rerouted.

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 used to control 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 your hook closes, moves, or deletes the message, you should set this flag.

HOOK_DELETE
Indicates that 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 of the folder where the message will be moved.

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.

Comments

Message hook providers use the **ISpoolerHook::OutboundMsgHook** method to reroute a message from the default Sent Items folder to another folder. Before moving the message to another folder or message store, your message hook provider should call the **IUnknown::QueryInterface** method on the message object to make sure your provider can get an interface for the message that is compatible with your provider's implementation.

Before rerouting messages, message hook providers must replace the passed-in entry identifier in the *lppEntryID* parameter with the entry identifier of the new target folder. The MAPI spooler moves the message to the indicated folder for your provider, unless another hook function replaces your folder entry identifier. If your hook requires that its operation be the final action on the message, it can set the HOOK_CANCEL flag in *lpulFlags* before returning. If your provider replaces the *lppEntryID* entry identifier, it must call the **MAPIFreeBuffer** function to free the previous one. The copy of the entry identifier that your provider stores in *lppEntryID* should be allocated using the **MAPIAllocateBuffer** function.

If the message hook provider's implementation must move the message to another folder itself, it should close the message, place zero in the *lpcbEntryID* parameter, and free the entry identifier in the *lppEntryID* parameter, if *lppEntryID* is not already NULL. The hook then sets *lppEntryID* to NULL and then places a pointer to the message's new parent folder in *lpFolder*. Message hook providers that move the message must set the HOOK_CANCEL flag in the *lpulFlags* parameter.

If your 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 the HOOK_CANCEL flag in the *lpulFlags* parameter. 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 message-hook provider object at session shutdown. If your 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

[New - Windows 95]

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 and 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	Delete 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 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

[New - Windows 95]

Deletes a row from a table.

Syntax

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 be deleted. This property value must contain the same index column as was used 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 requested does not match a corresponding row in the table.

Comments

The **ITableData::HrDeleteRow** method is used to delete a table row. To perform this deletion, **HrDeleteRow** takes in the *lpSPropValue* parameter the property value for the property indicating the row's index number and uses this index value 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 as was used for the *ulPropTagIndexColumn* parameter of the **CreateTable** call when the table was created. If no row with that index number exists, MAPI_E_NOT_FOUND is returned.

After the row is deleted, notifications are sent to all applications 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

[New - Windows 95]

Deletes multiple rows from a table.

Syntax

HRESULT HrDeleteRows(**ULONG** *ulFlags,* **LPSRowSet** *lprowsetToDelete*, **ULONG FAR** * *cRowsDeleted*)

Parameters

ulFlags
Input parameter containing a bitmask of flags used to control 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 be deleted. Each row must have an index property that is unique among all rows for this table to be deleted. This parameter can be NULL if the TAD_ALL_ROWS flag is set in the *ulFlags* parameter.

cRowsDeleted
Output parameter containing a variable that holds the number of rows deleted.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

Comments

The **ITableData::HrDeleteRows** method finds and deletes multiple table rows. To perform this deletion, **HrDeleteRows** locates and deletes each row corresponding to an index property in the **SRowSet** passed in the *lprowsetToDelete* parameter. These property values must contain the same index columns as were used 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, a value of zero is returned in *cRowsDeleted*. If the TAD_ALL_ROWS flag is set in the *ulFlags* parameter, all rows in the table are deleted.

After the rows are deleted, a single notification is sent to each application 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

[New - Windows 95]

Returns the properties contained in a row of a table.

Syntax

HRESULT HrEnumRow(ULONG *ulRowNumber*, **LPSRow FAR** * *lppSRow*)

Parameters

ulRowNumber
Input parameter indicating the row number. 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 the pointer to a 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.

Comments

The **ITableData::HrEnumRow** method returns 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 the *ulRowNumber* parameter does not exist, NULL is returned in the **SRow** structure 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.

See Also

MAPIAllocateBuffer function, **MAPIFreeBuffer** function, **SRow** structure

ITableData::HrGetView

[New - Windows 95]

Creates a new view for a table.

Syntax

HRESULT HrGetView(LPSSortOrderSet *lpSSortOrderSet*, **CALLERRELEASE FAR *** *lpfCallerRelease*, **ULONG** *ulCallerData*, **LPMAPITABLE FAR *** *lppMAPITable*)

Parameters

lpSSortOrderSet
Input parameter pointing to the **SSortOrderSet** structure holding the sort order to be used as 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 be called 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 the 32-bit data that the calling process requires to be 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. This 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.

Comments

The **ITableData::HrGetView** method creates a new view for a table. All rows and columns of the table are initially visible in the view; no restriction is 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 is released, the callback function – based on the **CALLERRELEASE** function prototype – specified in the *lpfCallerRelease* parameter is called with the data contained in the *ulCallerData* parameter, a pointer to the table data object to which the view applies, and the pointer to the table object being released.

See Also

CALLERRELEASE function prototype, **IMAPITable : IUnknown** interface, **SSortOrderSet** structure

ITableData::HrGetView, CALLERRELEASE

[New - Windows 95]

The **CALLERRELEASE** function prototype defines a client-application callback function that releases a table data object.

Syntax

void CALLERRELEASE(ULONG *ulCallerData*, **LPTABLEDATA** *lpTblData*, **LPMAPITABLE** *lpVue***)**

Parameters

ulCallerData
Input parameter specifying data about the calling client application.

lpTblData
Input parameter specifying a pointer to the **ITableData** interface for the table data object.

lpVue
Input parameter specifying a pointer to the **IMAPITable** interface that provides the display table. This is the table object passed in the **ITableData::HrGetView** method in the *lppMAPITable* parameter.

Return Values

None

Comments

Client applications use the **CALLERRELEASE** function prototype to create a callback that allows them to release views on the underlying table data object without having to keep track of that object. After all of the views on that table data object are released, implementations of **ITableData::HrGetView** call the client application's **CALLERRELEASE** function and then release the table data object underlying the views.

See Also

IMAPITable : IUnknown interface, **ITableData::HrGetView** method

ITableData::HrInsertRow

[New - Windows 95]

Inserts a row into a table.

Syntax

HRESULT HrInsertRow(**ULONG** *uliRow*, **LPSRow** *lpSRow*)

Parameters

uliRow
: Input parameter indicating the number of the table row before which the new row will be 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 end of the table.

lpSRow
: Input parameter pointing to the **SRow** structure containing the set of properties for the row being inserted in the table.

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.

Comments

The **ITableData::HrInsertRow** method inserts 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 the value for the index column already exists, MAPI_E_INVALID_PARAMETER is returned. If there is no current row with that value, **HrInsertRow** adds the new row. The property columns in the **SRow** 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 applications 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

[New - Windows 95]

Modifies a row in a table, or adds a row to a table.

Syntax

HRESULT HrModifyRow(LPSRow *lpSRow*)

Parameters

lpSRow
Input parameter pointing to the **SRow** structure containing the set of properties for the row to be added or modified. This row must contain the same index column as was used 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_INVALID_PARAMETER
The passed-in row does not have an index column.

Comments

The **ITableData::HrModifyRow** method modifies one row of a table or adds one row to a table. The **SRow** structure specified in the *lpSRow* parameter contains the properties for the row to be added or modified.

One of the property columns in the **SRow** must be a column holding the index column of the row to be modified or added, this index column must be the same one as was used for the *ulPropTagIndexColumn* parameter of the **CreateTable** call when the table was created. **HrModifyRow** replaces with the new row any row in the table that has the same index value as that in the **SRow**. If there is no current row with the index value, **HrModifyRow** adds the new row to the end of the table.

The property columns in the **SRow** do not have to be in the same order as the property columns in the table. The **SRow** can also include properties for which there are no current columns in the table; MAPI adds new columns to the table as needed.

Rows are modified and 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** includes properties for which there are no current columns become available to existing views but are not included in the 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 applications 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 application table must be in the client's character set, whether 8-bit or Unicode. MAPI's table implementation does not handle character set conversions.

See Also

SRow structure, **TABLE_NOTIFICATION** structure

ITableData::HrModifyRows

[New - Windows 95]

Modifies multiple rows in a table, or adds multiple rows to a table.

Syntax

HRESULT HrModifyRows(**ULONG** *ulFlags*, **LPSRowSet** *lpSRowSet*)

Parameters

ulFlags
: Reserved; must be zero.

lpSRowSet
: Input parameter pointing to the **SRowSet** structure containing the set of rows to be added or modified.

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.

Comments

The **ITableData::HrModifyRows** method modifies 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 rows to the table.

Each row in the **SRowSet** 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 columns as were used for the *ulPropTagIndexColumn* parameter of the call to the **CreateTable** function when the table was created. If one of the rows has a different index column, MAPI_E_INVALID PARAMETER is returned and no rows are changed. **HrModifyRows** replaces with the **SRowSet** 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 end of the table.

The property columns in the **SRowSet** do not have to be in the same order as the property columns in the table. The **SRowSet** 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** includes properties for which there are no current columns become available to existing views but are not included in the 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 application 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 application table must be in the client's character set, whether 8-bit or Unicode. MAPI's table implementation does not handle character set conversions.

MAPI copies appropriate data from the specified **SRowSet** structure.

See Also

SRowSet structure

ITableData::HrNotify

[New - Windows 95]

Finds a particular row so as to send a notification about that row.

Syntax

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 used to identify a particular row, whose properties must exactly match those passed in the **SPropValue**.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

Comments

Calling the **ITableData::HrNotify** method results in 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, your client application should reload the data associated with the control.

See Also

SPropValue structure, **TABLE_NOTIFICATION** structure

ITableData::HrQueryRow

[New - Windows 95]

Returns all properties of a specified row in a table and its row index in that table.

Syntax

HRESULT HrQueryRow(**LPSPropValue** *lpSPropValue*, **LPSRow FAR** * *lppSRow*, **ULONG FAR** * *lpuliRow*)

Parameters

lpSPropValue
Input parameter pointing to an **SPropValue** structure that holds an index property specifying the row for which to return properties.

lppSRow
Output parameter pointing to a variable where the pointer to a returned **SRow** structure holding all properties for the specified row is stored.

lpuliRow
Output parameter pointing to the address where the index number of the row is returned. This is the row's index number in the table. If no row 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 the index column.

Comments

The **ITableData::HrQueryRow** method returns the full property set for the row indicated by the index property in the *lpSPropValue* parameter, and it returns the row index that this row occupies in the table.

MAPI uses, but does not modify, the passed-in **SPropValue** structure. If the calling application allocated memory for this structure, the calling application must free it after the **HrQueryRow** call returns.

MAPI allocates memory for the returned **SRow** structure using the **MAPIAllocateBuffer** function when the table is created. The calling application must release this memory by calling the **MAPIFreeBuffer** function when done.

See Also

MAPIAllocateBuffer function, **MAPIFreeBuffer** function, **SPropValue** structure, **SRow** structure

ITnef : IUnknown

[New - Windows 95]

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 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, and gateways

Vtable Order

AddProps	Allows the calling implementation to add properties to be included 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 decorated message 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	Forces the encoding of the recipient table in the TNEF data stream for a message.
FinishComponent	Processes individual components from a message one component at a time into a TNEF stream.

ITnef::AddProps

[New - Windows 95]

Allows the calling implementation to add properties to be included in the encapsulation of a message or an attachment.

Syntax

HRESULT AddProps(**ULONG** *ulFlags*, **ULONG** *ulElemID*, **LPVOID** *lpvData*, **LPSPropTagArray** *lpPropList*)

Parameters

ulFlags
: Input parameter containing a bitmask of flags used to control how properties are included in or excluded from encapsulation. The following flags can be set:

TNEF_PROP_ATTACHMENTS_ONLY
: Applies 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 will be 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 the *ulElemID* parameter. The value in the *lpvData* parameter must be an ISTREAMTNEF pointer.

TNEF_PROP_EXCLUDE
: Encodes all properties not specified in the *lpPropList* parameter.

TNEF_PROP_INCLUDE
: Encodes all properties specified in the *lpPropList* parameter.

TNEF_PROP_MESSAGE_ONLY
: Encodes only those properties specified in *lpPropList* that are part of the message itself.

ulElemID
: Input parameter indicating the value of an attachment's PR_ATTACH_NUM property, which contains 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 the TNEF_PROP_CONTAINTAINED_TNEF flag was set in the *ulFlags* parameter.

lpvData
: Input parameter pointing to a buffer holding attachment data that is used to replace the data of the attachment specified in the *ulElemID* parameter. The *lpvData* parameter should be NULL unless the TNEF_PROP_CONTAINED or the TNEF_PROP_CONTAINTAINED_TNEF flag was set in the *ulFlags* parameter.

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.

Comments

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.

ITnef::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

[New - Windows 95]

Forces the encoding of the recipient table in the TNEF data stream for a message.

Syntax

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 to be encoded. The *lpRecipientTable* parameter can be NULL.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

Comments

Transport providers, message store providers, and gateways call the **ITenf::EncodeRecips** method to perform TNEF encoding for a particular recipient-table view. Such encoding is useful, for instance, if your transport provider, message store provider, or gateway requires a particular column set, sort order, or restriction for the recipient table.

Your provider or gateway passes the table view to be encoded in the *lpRecipientTable* parameter. The TNEF implementation encodes the recipient table with the given view, using the given column set, sort order, restrictions, and position. If your 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 current column set.

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 the *ulFlags* parameter and the PR_MESSAGE_RECIPIENTS property specified in the *lpPropList* parameter.

See Also

IMessage::GetRecipientTable method, **ITnef::AddProps** method, PR_MESSAGE_RECIPIENTS property

ITnef::ExtractProps

[New - Windows 95]

Extracts the properties from a TNEF encapsulation.

Syntax

HRESULT ExtractProps(**ULONG** *ulFlags*, **LPSPropTagArray** *lpPropList*, **LPSTnefProblemArray FAR *** *lpProblems*)

Parameters

ulFlags
Input parameter containing a bitmask of flags used to control 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 the *lpPropList* parameter.

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 the pointer to a **STnefProblemArray** structure is stored. If NULL is passed in the *lppProblems* parameter, no property problem array is returned. The **STnefProblemArray** indicates which properties were not encoded properly.

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.

Comments

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 will go back to the message and extract the new list of properties.

Unlike with 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 on objects opened with the TNEF_DECODE flag set on **OpenTnefStream** or the **OpenTnefStreamEx** function.

The TNEF implementation reports TNEF stream encoding problems without halting the **ExtractProps** process. The **STnefProblemArray** structure returned in the *lppProblems* parameter indicates which TNEF attributes or MAPI properties, if any, could not be processed. The value returned in the **scode** member of the **STnefProblemArray** structure indicates the specific problem. Your 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 report for this type of problem contains a value of 0L for the **ulPropTag** member, a value of attMAPIProps or attAttachment for the **ulAttribute** member, and a value of 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 your 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 the *lppProblems* parameter is only valid if the call returns S_OK. When S_OK is returned, 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 **STnefProblemArray** structure.

Your 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

[New - Windows 95]

Finishes processing for all TNEF operations that are queued and waiting.

Syntax

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. 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 the pointer to a **STnefProblemArray** structure is stored. If NULL is passed in the *lppProblems* parameter, no property problem array is returned. The **STnefProblemArray** indicates which properties were not encoded properly.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

Comments

Transport providers, message store providers, and gateways call the **ITnef::Finish** method to perform the encoding of all the properties requested in calls to the **ITnef::AddProps** and **ITnef::SetProps** methods. If the TNEF object was opened with the TNEF_ENCODE flag, all properties processed on the **Finish** call are encoded into the encapsulation stream passed to the TNEF 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 the TNEF_ENCODE flag set for the **OpenTnefStream** or **OpenTnefStreamEx** function.

The TNEF implementation reports TNEF stream encoding problems without halting the **Finish** process. The **STnefProblemArray** structure returned in the *lppProblems* parameter indicates which TNEF attributes or MAPI properties, if any, could not be processed. The value returned in the **scode** member of the **STnefProblemArray** structure indicates the specific problem. Your 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 your 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 the *lppProblems* parameter is only valid if the call returns S_OK. When S_OK is returned, 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 **STnefProblemArray** structure.

Your 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

[New - Windows 95]

Processes individual components from a message one component at a time into a TNEF stream.

Syntax

HRESULT FinishComponent(ULONG *ulFlags*, **ULONG** *ulComponentID*, **LPSPropTagArray** *lpCustomPropList*, **LPSPropValue** *lpCustomProps*, **LPSPropTagArray** *lpPropList*, **LPSTnefProblemArray FAR *** *lppProblems*)

Parameters

ulFlags

Input parameter containing a bitmask of flags used to determine for which component to finish processing. One of the other of the following flags must be set:

TNEF_COMPONENT_ATTACHMENT

Indicates that an attachment object is encoded; the value in the *ulComponentID* parameter refers to the PR_ATTACH_NUM property of the attachment being processed.

TNEF_COMPONENT_MESSAGE

Indicates that the message-level properties are encoded.

ulComponentID

Input parameter containing either zero to indicate processing for a message or the value of the PR_ATTACH_NUM property of the attachment to be processed. If TNEF_COMPONENT_MESSAGE is set in the *ulFlags* parameter, *ulComponentID* must be zero.

lpCustomPropList

Input parameter pointing to a **SPropTagArray** structure that identifies the properties passed in the *lpCustomProps* parameter. There must be a one-to-one correspondence between each of the property values in *lpCustomProps* and a property tag in the *lpCustomPropList* parameter.

lpCustomProps

Input parameter pointing to an **SPropValue** structure containing the properties to include in the encapsulation object.

lpPropList

Input parameter pointing to an **SPropTagArray** structure containing the properties that are to be encoded.

lppProblems

Output parameter pointing to a variable where the pointer to a **STnefProblemArray** structure is stored. If NULL is passed in the *lppProblems* parameter, no property problem array is returned. The **STnefProblemArray** indicates which properties were not encoded properly.

Return Values

S_OK

The call succeeded and has returned the expected value or values.

Comments

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 either the **OpenTnefStream** function or the **OpenTnefStreamEx** function.

Passing values in the *lpCustomPropList* and *lpCustomProps* parameters 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 the *lppProblems* parameter indicates which TNEF attributes or MAPI properties, if any, could not be processed. The value returned in the **scode** member of the **STnefProblemArray** structure indicates the specific problem. Your 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 your 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 the *lppProblems* parameter is only valid if the call returns S_OK. When S_OK is returned, 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 **STnefProblemArray** structure.

Your 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

[New - Windows 95]

Opens a stream interface on the decorated message text of an encapsulated message.

Syntax

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 on the call to the **OpenTnefStream** or **OpenTnefStreamEx** function.

ulFlags
: Input parameter containing a bitmask of flags used to control the opening of the stream interface. 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 you set the MAPI_CREATE flag, you should also set the MAPI_MODIFY flag.

  MAPI_MODIFY
  : Requests read-write access. The default interface is read-only. MAPI_MODIFY must be set when MAPI_CREATE is also set.

lppStream
: Output parameter pointing to a variable where the pointer is stored 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.

Comments

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 that is sequential, 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 stream is passed a message that has tags, 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

[New - Windows 95]

Sets the value of one or more properties for an encapsulated message or attachment without modifying the original message or attachment.

Syntax

HRESULT SetProps(ULONG *ulFlags*, **ULONG** *ulElemID*, **ULONG** *cValues*, **LPSPropValue** *lpProps*)

Parameters

ulFlags
Input parameter containing a bitmask of flags used to control 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 indicating the value of an attachment's PR_ATTACH_NUM property.

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 be set.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

Comments

Transport providers, message store providers, and gateways call the **ITnef::SetProps** method to set properties to be included 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 on TNEF objects that have previously been 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

[New - Windows 95]

The **IXPLogon** interface is used to provide the MAPI spooler access into 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 transport provider has 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 the transport provider that the MAPI spooler has a message for the provider to deliver.
EndMessage	Informs the transport provider that the MAPI spooler has completed its send pass for this provider.
Poll	Allows a transport provider to indicate when it has one or more incoming messages available. This method is called periodically by the MAPI spooler.
StartMessage	Indicates that the transport provider should start downloading into a message.
OpenStatusEntry	Returns an IMAPIStatus object. This method is called by the MAPI spooler when a client application calls an **OpenEntry** method against the entry identifier in the transport provider's subsystem status-table row.
ValidateState	Has a transport provider check its external status. This method is called by the MAPI spooler.
FlushQueues	Requests that transport operations occur quickly. This method is called by the MAPI spooler.

IXPLogon::AddressTypes

[New - Windows 95]

Indicates to the MAPI spooler what types of recipients a transport provider can handle.

Syntax

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. 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 8-bit 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.

Comments

The MAPI spooler calls the **IXPLogon::AddressTypes** method immediately after a transport provider returns from a call to the **IXPProvider::TransportLogon** method so that the transport 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 instead 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 array referred to by *lpppUIDArray* to identify those recipient entry identifiers that are directly handled by the transport or by the messaging system to which the transport 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 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, the transport can handle that recipient.

In the event multiple transport providers can handle the same type of recipient, MAPI selects a transport based on the transport priority order indicated in the client application's profile. To determine which transport 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 to match a particular recipient in this scan gets the first opportunity to handle this recipient. If that transport does not handle the recipient, the MAPI spooler continues the scan so as to find a transport 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 needs to dynamically construct 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 structures 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 should be installed last in the MAPI and MAPI spooler priority order of transports 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 the *lpppAdrTypeArray* parameter. If a transport provider doesn't support anything, then it should pass NULL for the **MAPIUID** and an empty string for the address type.

For more information on working with address types, see *MAPI Programmer's Guide*.

See Also

IXPLogon::TransportLogoff method, **IXPProvider::TransportLogon** method, **MAPIUID** structure

IXPLogon::EndMessage

[New - Windows 95]

Informs the transport provider that the MAPI spooler has completed its send pass for this provider.

Syntax

HRESULT EndMessage(**ULONG** *ulMsgRef*, **ULONG FAR *** *lpulFlags*)

Parameters

ulMsgRef
: Input parameter containing a 32-bit reference value, specific to this message, obtained in an earlier call to the **IXPLogon::SubmitMessage** method.

lpulFlags
: Output parameter containing a bitmask of flags used to indicate 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 a **IMessage::SubmitMessage** method call.

Return Values

S_OK
: The call succeeded and has returned the expected value or values.

Comments

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 considered to be no longer valid. The transport provider can reuse the same value on a future message.

All objects opened by the transport during the transfer of a message should be released before returning from the **EndMessage** call, with the exception of the IMessage object the MAPI spooler passes to the transport. The IMessage 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

[New - Windows 95]

Requests that transport operations occur quickly. This method is called by the MAPI spooler.

Syntax

HRESULT FlushQueues(**ULONG** *ulUIParam*, **ULONG** *cbTargetTransport*, **LPENTRYID** *lpTargetTransport*, **ULONG** *ulFlags*)

Parameters

ulUIParam
Input parameter containing the handle to the window that the dialog box is modal to.

cbTargetTransport
Reserved; must be zero.

lpTargetTransport
Reserved; must be NULL.

ulFlags
Input parameter containing a bitmask of flags used to control message queue flushing. 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 user interface components.

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.

Comments

The MAPI spooler calls **IXPLogon::FlushQueues** to advise the transport provider that the MAPI spooler is about to begin processing messages. The transport provider should call **IMAPISupport::ModifyStatusRow** 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 will then start sending messages with the operation being synchronous to the MAPI spooler.

To support its implementation of **IMAPIStatus::FlushQueues**, the MAPI spooler calls **IXPLogon::FlushQueues** on all logon objects for all active transport providers running in a profile session. When a transport provider's **FlushQueues** method is call as a result of a client application calling **IMAPIStatus::FlushQueues**, the message processing occurs asynchrousously to the client application.

See Also

IMAPIStatus::FlushQueues method

IXPLogon::Idle

[New - Windows 95]

Calls a transport provider at a point when the system is idle to perform low-priority operations.

Syntax

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.

Comments

If requested, by passing the XP_LOGON_SP flag in the call to the **IXPProvider::TransportLogon** method that opened the current session, the MAPI spooler periodically calls the **IXPLogon::Idle** method during times when the system is idle. At such a time, the transport provider can perform background operations that are not appropriate during other calls, or which need to occur on a regular basis.

See Also

IXPProvider::TransportLogon method

IXPLogon::OpenStatusEntry

[New - Windows 95]

Opens a status object to get information about or modify the transport-provider logon session. This method is called by the MAPI spooler.

Syntax

HRESULT OpenStatusEntry(**LPCIID** *lpInterface*, **ULONG** *ulFlags*, **ULONG FAR** * *lpulObjType*, **LPMAPISTATUS FAR** * *lppEntry*)

Parameters

lpInterface
: Input parameter pointing to the interface identifier (IID) for the transport logon object. Passing NULL for the *lpInterface* parameter indicates the MAPI interface for the object will be returned, in this case the **IMAPIStatus** interface. The *lpInterface* parameter can also be set to an identifier for another appropriate interface for the object.

ulFlags
: Input parameter containing a bitmask of flags used to control how the status entry 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 that holds the type of the opened object.

lppEntry
: Output parameter pointing to a variable where the pointer to the returned status object.

Return Values

S_OK
: The call succeeded and has returned the expected value or values.

Comments

The **IXPLogon::OpenStatusEntry** method opens an object with the **IMAPIStatus** interface associated with this particular transport 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

[New - Windows 95]

Checks whether a transport provider has one or more incoming messages available. This method is called periodically by the MAPI spooler.

Syntax

HRESULT Poll(**ULONG FAR *** *lpulIncoming*)

Parameters

lpulIncoming
Output parameter 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.

Comments

The MAPI spooler periodically calls the **IXPLogon::Poll** method if the transport indicates it must be polled for new messages 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 transport to process the first incoming message. The transport indicates incoming messages by setting the value in **Poll**'s *lpulIncoming* parameter to nonzero.

See Also

IXPLogon::StartMessage method, **IXPProvider::TransportLogon** method

IXPLogon::RegisterOptions

[New - Windows 95]

Informs the messaging system about the options provided by a transport provider for a messaging address type.

Syntax

HRESULT RegisterOptions(**ULONG FAR** * *lpulFlags*, **ULONG FAR** * *lpcOptions*, **LPOPTIONDATA FAR** * *lppOptions*)

Parameters

lpulFlags
Output parameter containing a bitmask of flags. 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 8-bit 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 the pointer to the returned **OPTIONDATA** structure is stored. The **OPTIONDATA** structure contains information for this 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.

Comments

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 message 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 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 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 of type **OPTIONCALLBACK** that the transport provider supplies; this callback function retrieves 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::RegisterOptions, OPTIONCALLBACK

[New - Windows 95]

The **OPTIONCALLBACK** function prototype is used by a transport provider to implement a callback function that MAPI calls to retrieve a wrapped **IMAPIProp** interface that manages the provider's properties.

Syntax

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* for this transport provider's dynamic link library (DLL) as returned from the **LoadLibrary** function call made by MAPI.

lpMalloc
Input parameter specifying a pointer to a standard memory allocator.

ulFlags
Input parameter specifying a bitmask of flags indicating the type of options being 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 transport-provider option data for the recipient or message. This data is contained in the **OPTIONDATA** structure that was passed in the **IXPLogon::RegisterOptions** call that registered the options.

lpMAPISup
Input parameter pointing to a MAPI support object that the provider can use to call the methods of the **IMAPISupport** interface.

lpDataSource
Input parameter pointing to an **IMAPIProp** 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 variable where the pointer to a returned **MAPIERROR** structure, if any, is stored.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

Comments

The transport provider's **OPTIONCALLBACK** function is called by MAPI 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 **IMAPProp** interface in the *lpDataSource* parameter. The transport 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 an **IMAPIProp::OpenProperty** call on 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 and how to implement **OPTIONCALLBACK**, see *MAPI Programmer's Guide*.

See Also

IMAPIProp : IUnknown interface, **IMAPISupport : IUnknown** interface, **IXPLogon::RegisterOptions** method

IXPLogon::StartMessage

[New - Windows 95]

Initiates the transfer of an incoming message from the transport provider to the MAPI spooler.

Syntax

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 to access and manipulate the message. This object remains valid until after the transport returns from the call to the **IXPLogon::StartMessage** method.

lpulMsgRef
Output parameter pointing to a variable in which the transport returns the 32-bit reference value it assigned to this message. This value is initialized to 1 by the MAPI spooler before calling the transport.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

Comments

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 starts to use the message object, it should store a message reference into the unsigned long pointed to by the *lpulMsgRe* 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 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 for the message must contain all required addressing properties. If necessary, the transport provider can construct a custom recipient to represent a particular message 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 has enough information about an address book provider's recipient format that it can build a valid entry identifier for a recipient for that address book provider, it should build the entry identifier.

If any nontransmittable properties are received, the transport should not store them in the new message. However, the transport 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 is unable to use the **IMAPISupport::StatusRecips** method to generate the report from the original message, the transport 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 calls the **IMAPIProp::SaveChanges** method. If the transport provider doesn't have any messages, it can stop the incoming message by returning from the **StartMessage** call without calling **SaveChanges**.

All objects opened by the transport during a **StartMessage** call should be released before returning. However, the transport should not release the message object originally passed by the MAPI spooler in *lpMessage*.

If an error is returned from **StartMessage**, the message in process is released without having changes saved and is lost. The transport provider 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 the transport provider is in a severe error condition.

For more information on interacting with the MAPI spooler, see *MAPI Programmer's Guide*.

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

[New - Windows 95]

Indicates to the transport provider that the MAPI spooler has a message for the provider to deliver.

Syntax

HRESULT SubmitMessage(**ULONG** *ulFlags*, **LPMESSAGE** *lpMessage*, **ULONG FAR** * *lpulMsgRef*, **ULONG FAR** * *lpulReturnParm*)

Parameters

ulFlags
Input parameter containing a bitmask of flags used to control how a 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 (by 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 be sent, which is used by the transport to access and manipulate the message. This object remains valid until after the transport returns from a subsequent call to the **IXPLogon::EndMessage** method.

lpulMsgRef
Output parameter pointing to a variable in which the transport 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 calling the transport.

lpulReturnParm
Output parameter pointing to a variable in which corresponds to the MAPI_E_WAIT or MAPI_E_NETWORK_ERROR error values returned by **SubmitMessage**.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

MAPI_E_BUSY
The transport can't handle the message because it is performing another operation. Use this return value to indicate that no processing occurred and that the MAPI spooler should not call **IXPLogon::EndMessage**. The MAPI spooler will try again later.

MAPI_E_CANCEL
On a previous call to the **IXPLogon::SpoolerNotify** method, the transport requested that the MAPI spooler resubmit the message. Since then, conditions have changed, and the message should not be resent. The MAPI spooler will go on to handle something else.

MAPI_E_NETWORK_ERROR
A network error prevented successful completion of the operation. The value in 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. Use this return value to indicate that no processing occurred and that the MAPI spooler should not call **IXPLogon::EndMessage**.

MAPI_E_WAIT
A temporary problem prevents the transport from handling the message. The value in

lpulReturnParm should be set to the number of seconds before the MAPI spooler resubmits the message.

Comments

The MAPI spooler calls the **IXPLogon::SubmitMessage** method when it has a message for the transport to carry. The message is passed to the transport provider using the *lpMessage* parameter.

If the transport 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 transport is not prepared to handle the transfer, it should return an error value and, optionally, another MAPI return value in the *lpulReturnParm* parameter 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 to be sent 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 transport can lock the resource, prepare the message, and submit it. If the resource is busy, the transport 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 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 the *lpulMsgRef* parameter, 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 transport that it should release any open objects and stop message transfer.

The transport 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 transport 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.

During a **SubmitMessage** call, the MAPI spooler processes methods for objects opened during the transfer of the message and 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 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. A PR_RESPONSIBILITY already set to TRUE means that another transport provider handles that recipient. When the transport 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 to TRUE in the passed message. Usually, the transport makes this determination after message delivery is complete.

Typically, the transport 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 transport 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 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 based on the value returned in the *lpulRetunParm* parameter. 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

[New - Windows 95]

Terminates a transport provider session with the MAPI spooler.

Syntax

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.

Returning any value other than S_OK causes the transport to be logged off anyway.

Comments

The MAPI spooler calls the **IXPLogon::TransportLogoff** method to terminate a transport session for a particular user. Before calling **TransportLogoff**, the MAPI spooler discards any data about supported messaging address types for this session that had been 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 transport should stop the sending process.

The transport should release all resources allocated for the current transport 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 to satisfy calls to the **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 transport'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

[New - Windows 95]

Signals in a transport provider session the occurrence of an event for the MAPI spooler about which the transport provider has requested notification.

Syntax

HRESULT TransportNotify(**ULONG FAR** * *lpulFlags*, **LPVOID FAR** * *lppvData*)

Parameters

lpulFlags

Input-output parameter containing a bitmask of flags used to signal notification events. The following flags can be set on input and must be returned unchanged on output:

NOTIFY_ABORT_DEFERRED

Notifies the transport that a message for which it accepted responsibility is being canceled. Only transports that support deferral need to support this flag. The *lppvData* parameter points to the entry identifier of the 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 session should be updated before returning by calling the **IMAPISupport::ModifyStatusRow** method. The NOTIFY_BEGIN_INBOUND flag is mutually exclusive with NOTIFY_END_INBOUND.

NOTIFY_BEGIN_INBOUND_FLUSH

Signals the transport provider to cycle through inbound mail as quickly as possible. This flag is set by the MAPI spooler. To comply with this notification, the transport provider should set the flag STATUS_INBOUND_FLUSH bit in the PR_STATUS_CODE property of its status table row as soon as it can, using the **IMAPISupport::ModifyStatus Row** method. Until the end of this inbound messaging cycle, that is when the transport determines it has downloaded all it can or when it has received an **IXPLogon::TransportNotify** method call with the flag NOTIFY_END_INBOUND_FLUSH, the transport 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 all incoming messages. At the end of the inbound flush operation, the transport should clear the STATUS_INBOUND_FLUSH bit in the PR_STATUS_CODE property of its status row using the **IMAPISupport::SpoolerNotify** method.

NOTIFY_BEGIN_OUTBOUND

Indicates outbound operations can now occur for this transport provider session. If there are any messages to be set for this transport provider, a call to the **IXPLogon::SubmitMessage** method follows. The status table row for the transport session should be updated before returning. The NOTIFY_BEGIN_OUTBOUND flag is mutually exclusive with NOTIFY_END_OUTBOUND.

NOTIFY_BEGIN_OUTBOUND_FLUSH

Works similarly to NOTIFY_BEGIN_INBOUND_FLUSH but refers to outbound mail, and the appropriate status flag is STATUS_OUTBOUND_FLUSH.

NOTIFY_CANCEL_MESSAGE

Indicates the MAPI spooler must cancel transfer of a message for which the *lppvData* parameter points to the *ulMsgRef* parameter (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 **IXPLogon::StartMessage**,

IXPLogon::SubmitMessage, 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 in at the next convenient time. The message object data already stored for the incoming message is discarded.

If the transport provider did not update the *ulMsgRef* parameter at **StartMessage** or **SubmitMessage** time, *ulMsgRef* has a value of 0 for outbound messages (set with **SubmitMessage**) or 1 for inbound messages (set with **StartMessage**).

NOTIFY_END_INBOUND

Indicates inbound operations must cease for this transport provider session. The MAPI spooler ceases to use the **IXPLogon::Poll** method and ignores the NOTIFY_NEWMAIL flag set with the **IMAPISupport::SpoolerNotify** method for this session. In-process messages should not be aborted. The status table row for the transport session should be updated by calling the **IMAPISupport::ModifyStatusRow** method before returning. The NOTIFY_END_INBOUND flag is mutually exclusive with NOTIFY_BEGIN_INBOUND.

NOTIFY_END_INBOUND_FLUSH

Notifies the transport provider to come out of flush mode. This flag is set by the MAPI spooler. Setting this flag does not indicate the transport provider should stop downloading, but that downloading should be done in the background. The status table row for the transport session should be updated by calling the **IMAPISupport::ModifyStatusRow** method 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 the **IXPLogon::SubmitMessage** method and ignores the NOTIFY_READYTOSEND flag set with the **IMAPISupport::SpoolerNotify** method. 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 the transport session should be updated by calling the **IMAPISupport::ModifyStatusRow** method before returning. The NOTIFY_END_INBOUND flag is mutually exclusive with NOTIFY_BEGIN_OUTBOUND.

NOTIFY_END_OUTBOUND_FLUSH

Works similarly to NOTIFY_END_INBOUND_FLUSH but refers to outbound mail, and the appropriate status flag is STATUS_OUTBOUND_FLUSH.

lppvData

Output parameter pointing to a variable where the pointer to event-specific data is stored. For more information on what the *lppvData* parameter holds, see the descriptions for the flags for the *lpulFlags* parameter.

Return Values

S_OK

The call succeeded and has returned the expected value or values.

Comments

The MAPI spooler calls the **IXPLogon::TransportNotify** method to signal to the transport 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 either an incoming message queue or an 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 flag and the NOTIFY_CANCEL_MESSAGE flag in the *ulFlags* parameter. If the MAPI spooler is logging off and still has deferred messages in the queue, it passes only the NOTIFY_ABORT_DEFERRED flag 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

[New - Windows 95]

Has a transport provider check its external status. This method is called by the MAPI spooler.

Syntax

HRESULT ValidateState(**ULONG** *ulUIParam*, **ULONG** *ulFlags*)

Parameters

ulUIParam
Input parameter containing the handle to the window that the dialog box is modal to.

ulFlags
Input parameter containing a bitmask of flags used to control the status check for the transport. The following flags can be set:

ABORT_XP_HEADER_OPERATION
Indicates that the user canceled the operation, typically by pressing the Cancel button in a dialog box. The transport has the option to continue working on the operation, or the transport can abort the operation and return MAPI_E_USER_CANCELED.

CONFIG_CHANGED
Validates the state of currently loaded transports by causing the MAPI spooler to call the transport's **IXPLogon::AddressTypes** and **IMAPISsession::MessageOptions** methods. This flag also provides the MAPI spooler an opportunity to correct critical transport failures without forcing the client applications to log off and then log on again.

FORCE_XP_CONNECT
Indicates that 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 that 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 as MSGSTATUS_REMOTE_DOWNLOAD should be downloaded, and that all messages marked as MSGSTATUS_REMOTE_DELETE 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 mailbag headers should be downloaded, and that all message status marks should be cleared.

SUPPRESS_UI
Prevents the provider from displaying user interface components.

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 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 clicking the Cancel button in a dialog box.

Comments

The MAPI spooler calls the **IXPLogon::ValidateState** method to support calls to the **IMAPIStatus::ValidateState** method, for the status object. The transport 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 this profile session.

See Also

IMAPISession::MessageOptions method, **IMAPIStatus::ValidateState** method, **IXPLogon::AddressTypes** method

IXPProvider : IUnknown

[New - Windows 95]

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

[New - Windows 95]

Closes down a transport provider in an orderly fashion.

Syntax

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.

Comments

The MAPI spooler calls the **IXPProvider::Shutdown** method just prior to releasing a transport provider object. MAPI will have released all logon objects on a provider before calling **Shutdown**.

See Also

XPProviderInit function

IXPProvider::TransportLogon

[New - Windows 95]

Establishes a session with the MAPI spooler for the transport provider.

Syntax

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 and the MAPI spooler for this session. This object remains valid until the transport provider object releases it.

ulUIParam
Input parameter containing the handle to the window the dialog box is modal to. The *ulUIParam* parameter can be non-null, for example when called with the LOGON_SETUP flag.

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 used to control 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 onto 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 that 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 usually sets this flag. The MAPI spooler can use **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 the 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 usually sets this flag. The MAPI spooler can use the **IXPLogon::TransportNotify** method 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 8-bit format.

The following flags can be set by the transport provider on output:

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 to indicate it should do so. 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 covered 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 8-bit format.

lppMAPIError
Output parameter pointing to a variable where the pointer to the returned **MAPIERROR** structure, if any, is stored. The **MAPIERROR** structure contains 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 logon object 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. MAPI calls the provider's message-service entry function.

MAPI_E_USER_CANCEL
The user canceled the operation, typically by clicking the Cancel button in the dialog box.

Comments

The MAPI spooler calls the **IXPProvider::TransportLogon** method to establish a logon session with 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, or credentials. Using this method 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 that data related to this user context is separated from data for other contexts.

The transport provider must call **IUnknown::AddRef** on the support object and keep a copy of the pointer to this object as part of the logon object.

The display name of the user's profile in the *lpszProfileName* parameter is provided so that 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.

Transports 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, transports are opened when the user first logs onto 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 the *lpulFlags* parameter. Later, when the appropriate message stores are available in the profile session, the MAPI spooler calls the **IXPLogon::TransportNotify** method to initiate inbound and outbound operations for the transport.

Passing the LOGON_NO_CONNECT flag in the *lpulFlags* parameter signals offline operation of the transport. This value indicates that 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 the **IXPLogon::Idle** method when system idle time occurs to initiate such operations. The MAPI spooler does not call **Idle** at a regular interval; rather, it is called only during true idle time, so providers should not make any assumptions about how frequently their **Idle** method will be called. Providers that support such operations should supply the correct configuration user interface in their provider property sheet.

If the transport logon succeeds, the transport provider should return in the *lppXPLogon* parameter a pointer to a logon object for the MAPI spooler to use for further transport-provider access. If **TransportLogon** displays a logon dialog box and the user cancels logon, typically by clicking the Cancel button in the dialog box, your provider should return MAPI_E_USER_CANCEL.

If your 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 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

IAddrBook : IMAPIProp

[New - Windows 95]

The **IAddrBook** interface supports operations such as displaying address-book dialog boxes, opening containers within the address book, and resolving display names to messaging recipients.

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, messaging user, or distribution list object.
CompareEntryIDs	Compares two address book entry identifiers to determine if they refer to the same object.
Advise	Registers for notifications on changes to objects within the address book.
Unadvise	Removes a registration for notification on address book changes previously established with a call to the **IAddrBook::Advise** method.
CreateOneOff	Creates an entry identifier for a custom recipient address.
NewEntry	Displays a dialog box for creating new entries within containers or custom recipient addresses within messages.
ResolveName	Displays a dialog box for locating a recipient given a partial name.
Address	Displays an addressing dialog box.
Details	Displays a details dialog box on a particular entry in an address book.
RecipOptions	Displays a modal per-recipient options dialog box on a particular recipient.
QueryDefaultRecipOpt	Returns the available recipient options for a particular messaging address type, with defaults.
GetPAB	Returns the directory designated as the personal address book (PAB).
SetPAB	Designates a particular container to be the personal address book.
GetDefaultDir	Returns the default directory for the address book container.
SetDefaultDir	Chooses a different directory as the default directory for the address book container.

GetSearchPath	Returns the search path that is used in **ResolveNames** methods.
SetSearchPath	Sets the path that is used in **ResolveNames** methods.
PrepareRecips	Prepares a recipient list for later use by the messaging system.

IAddrBook::Address

[New - Windows 95]

Displays an addressing dialog box.

Syntax

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. A window handle must always be passed in on input. If the DIALOG_SDI flag is set in the **ADDRPARM** structure, then on output 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 addressing dialog box.

lppAdrList
Input-Output parameter pointing to a variable where the pointer to an **ADRLIST** structure holding the recipient list updated by **IAddrBook::Address** is stored. *lppAdrList* can be NULL on input.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

Comments

Use the **IAddrBook::Address** method to pass the current list of recipients in a message and return a list updated using the MAPI addressing dialog box. A client application passes the current list of recipients for a message, possibly empty, in the *lppAdrList* parameter when calling **Address**, 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 one holding an individual recipient that is organized by which type of recipient it holds, as indicated by the recipient's PR_RECIPIENT_TYPE property. The possible types are MAPI_TO (that is, a primary recipient), MAPI_CC (that is, a recipient that receives a copy of a message), and MAPI_BCC (that is, a recipient that receives a copy of a message without other recipients being made aware).

ADRLIST structures can hold both resolved and unresolved recipient entries. An *unresolved entry* does not have a PR_ENTRYID property. Applications that enable users to type recipient names directly into a message, in addition to choosing names from a predetermined list, create unresolved entries by creating an ADRENTRY with just the PR_DISPLAY_NAME and the PR_RECIPIENT_TYPE properties. A *resolved entry* will at least contain the following properties: PR_ENTRYID, PR_RECIPIENT_TYPE, PR_DISPLAY_NAME, PR_ADDRTYPE, and PR_DISPLAY_TYPE.

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 is passed in by the calling application, or if NULL is passed in the *lppAdrList* parameter on input, then **Address** allocates a larger buffer for the **ADRLIST** structure it returns using **MAPIAllocateBuffer** and returns this buffer's address in the *lppAdrList* parameter. **Address** frees the old buffer by using the **MAPIFreeBuffer** function.

Each **SPropValue** property value structure is also 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 was 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

[New - Windows 95]

Registers an object for notifications about changes within the address book.

Syntax

HRESULT Advise(**ULONG** *cbEntryID*, **LPENTRYID** *lpEntryID*, **ULONG** *ulEventMask*,
LPMAPIADVISESINK *lpAdviseSink*, **ULONG FAR** * *lpulConnection*)

Parameters

cbEntryID
Input parameter containing the number of bytes in 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.

ulEventMask
Input parameter containing an event mask of the types of notification events occurring for the object for which MAPI will generate notifications to filter 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 data structures:

Notification event type	Corresponding data structure
fnevCriticalError	**ERROR_NOTIFICATION**
fnevObjectCreated	**OBJECT_NOTIFICATION**
fnevObjectDeleted	**OBJECT_NOTIFICATION**
fnevObjectModified	**OBJECT_NOTIFICATION**
fnevObjectCopied	**OBJECT_NOTIFICATION**
fnevTableModified	**TABLE_NOTIFICATION**
fnevStatusObjectModified	**STATUS_OBJECT_NOTIFICATION**

lpAdviseSink
Input parameter pointing to the advise sink object to be called when a event for the object occurs about which notification has been requested.

lpulConnection
Output parameter pointing to a variable that holds the connection number for this notification registration upon a successful return. 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 the *lpEntryID* parameter.

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 could not be found to handle the entry identifier.

Comments

Use the **IAddrBook::Advise** method to register an address book object for notification callbacks. MAPI will forward this call to the service provider that is active on the session and is responsible for the object indicated by the entry identifier in the *lpEntryID* parameter. Whenever a change occurs to the indicated object, the provider checks to see what event mask bit has been set in the *ulEventMask* parameter and thus what type of change has 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, your client should use the **HrThisThreadAdviseSink** function.

To provide notifications, the service provider implementing **Advise** needs to keep a copy of the pointer to the advise sink object; to do so, it calls the **IUnknown::AddRef** method for the advise sink object to maintain the object pointer until notification registration is canceled with a call to the **IAddrBook::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, client applications must be prepared for the advise sink object to be released. Clients should therefore release their advise sink object after **Advise** returns, unless they have a specific long-term use for it.

See Also

HrThisThreadAdviseSink function, **IAddrBook::Unadvise** method, **IMAPIAdviseSink::OnNotify** method, **NOTIFICATION** structure

IAddrBook::CompareEntryIDs

[New - Windows 95]

Compares two entry identifiers to determine if they refer to the same service provider 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.

Syntax

HRESULT CompareEntryIDs(ULONG *cbEntryID1*, **LPENTRYID** *lpEntryID1*, **ULONG** *cbEntryID2*, **LPENTRYID** *lpEntryID2*, **ULONG** *ulFlags*, **ULONG FAR *** *lpulResult*)

Parameters

cbEntryID1
Input parameter containing the number of bytes in the *lpEntryID1* parameter.

lpEntryID1
Input parameter pointing to the first entry identifier to be compared.

cbEntryID2
Input parameter containing the number of bytes in 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 that receives the result of the comparison; 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.

Comments

Use the **IAddrBook::CompareEntryIDs** method to compare two entry identifiers for a given address book provider object and 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 could occur, for example, after a new version of a service provider is installed.

If **CompareEntryIDs** returns an error, the calling application should make no assumptions about the comparison and should take the most conservative approach based on what your application is trying to do. **CompareEntryIDs** might fail if, for example, no provider has registered for one of the entry identifiers. If your application compares message-store entry identifiers when one or both of the stores has not yet opened, **CompareEntryIDs** returns the value MAPI_E_UNKNOWN_ENTRYID.

IAddrBook::CreateOneOff

[New - Windows 95]

Creates an entry identifier for a custom recipient address.

Syntax

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. *lpszName* can be NULL.

lpszAdrType
Input parameter pointing to a string containing the e-mail address type of the recipient; examples of e-mail address types are FAX, SMTP, X500, and so on. *lpszAdrType* cannot be NULL.

lpszAddress
Input parameter pointing to a string containing the e-mail address of the recipient. *lpszAddress* cannot be NULL.

ulFlags
Input parameter containing a bitmask of flags controlling 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 8-bit format.

lpcbEntryID
Output parameter pointing to a variable in which the **IAddrBook::CreateOneOff** method returns the number of bytes in the entry identifier pointed to by the *lppEntryID* parameter.

lppEntryID
Output parameter pointing to a variable where the pointer to the entry identifier of the custom recipient address that has been created is stored.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

Comments

Service providers use the **IAddrBook::CreateOneOff** method to create an entry identifier for a custom recipient. A *custom recipient* is a message recipient that the user does not choose from an address book but specifies by typing in the recipient name. This entry identifier can be used as a valid recipient on a message.

When the provider is done using the entry identifier returned by **CreateOneOff**, it should free the allocated memory by using the **MAPIFreeBuffer** function.

See Also

MAPIFreeBuffer function

IAddrBook::Details

[New - Windows 95]

Displays a details dialog box on a particular entry in an address book.

Syntax

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 to the parent window for the dialog box.

lpfnDismiss
Input parameter pointing to the address of a function based on the **DISMISSMODELESS** function prototype that is called when the modeless variety of the details dialog box is dismissed.

lpvDismissContext
Input parameter that is passed to the function specified by the *lpfnDismiss* parameter.

cbEntryID
Input parameter containing the number of bytes in the *lpEntryID* parameter.

lpEntryID
Input parameter pointing to the entry identifier for which the object's details are displayed.

lpfButtonCallback
Input parameter pointing to a pointer to a button callback function that adds a button to a dialog box.

lpvButtonContext
Input parameter pointing to data used as a parameter for the button callback function pointed to by the *lpfButtonCallback* parameter.

lpszButtonText
Input parameter pointing to a string containing the text to be applied to the button mentioned previously 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 controlling 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 8-bit format.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

Comments

Client applications use the **IMAPISupport::Details** methods to display a details dialog box on a particular entry in an address book. The *lpfButtonCallback*, *lpvButtonContext*, and *lpButtonText* parameters can be used to add a button it has defined to the addressing dialog box. When the button is chosen, MAPI calls the callback function pointed to by *lpfButtonCallback*, passing both the entry identifier of the chosen button and the data in *lpvButtonContext*. If an extensible button is not needed, *lpszButtonText* should be NULL.

The *lpfnDismiss* parameter is used to support the modeless version of the details address book. When the user dismisses the address book, MAPI calls **IAddrBook::Details**. The dismiss function used with **Details** is based on the function prototype **DISMISSMODELESS**.

See Also

DISMISSMODELESS function prototype, **IAddrBook::Address** method, **IAddrBook::Details** method, **LPFNBUTTON** function prototype

IAddrBook::Details, LPFNBUTTON

[New - Windows 95]

The **LPFNBUTTON** function prototype represents a client application callback function that MAPI calls to hook to an optional button control on an address book dialog box. This button is typically a **Details** button.

Syntax

SCODE (STDMETHODCALLTYPE FAR * LPFNBUTTON)(ULONG *ulUIParam*, **LPVOID** *lpvContext*, **ULONG** *cbEntryID*, **LPENTRYID** *lpSelection*, **ULONG** *ulFlags***)**

Parameters

ulUIParam
Input parameter specifying an implementation-specific 32-bit value normally 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 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. Typically, for C++ code, *lpvContext* represents a pointer to the address of a C++ object.

cbEntryID
Input parameter specifying the size, in bytes, of the identifier indicated by *lpSelection*.

lpSelection
Input parameter specifying a pointer to an **ENTRYID** structure 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.

Comments

This hook function can be used by a client application to define a button on the details dialog box. The client passes a pointer to the callback function in calls to **IAdrBook::Details**. When the dialog box is displayed and the user selects the defined button, MAPI calls this function.

See Also

BuildDisplayTable function, **ENTRYID** structure, **IAddrBook::Details** method

IAddrBook::GetDefaultDir

[New - Windows 95]

Returns the default directory for the address book container.

Syntax

HRESULT GetDefaultDir(**ULONG FAR** * *lpcbEntryID*, **LPENTRYID FAR** * *lppEntryID*)

Parameters

lpcbEntryID
Output parameter pointing to a variable containing the number of bytes in the *lpEntryID* parameter.

lppEntryID
Output parameter pointing to a variable that receives a pointer to the default directory's entry identifier.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

Comments

The default directory is the directory the user sees displayed in the address book when the address book is first opened. MAPI determines the default directory if one has never been set by a previous call to the **IAddrBook::SetDefaultDir** method. MAPI selects the default directory by locating the first entry that contains names and is not the personal address book. If there are none, then the personal address book is set as the default directory.

Calls made to **GetDefaultDir** return a pointer in the *lppEntryID* parameter to the entry identifier of the default directory. MAPI allocates memory for this entry identifier with the **MAPIAllocateBuffer** function and your application must release it with the **MAPIFreeBuffer** function when done.

To set the default directory, your application calls the **IAddrBook::SetDefaultDir** method. Your application doesn't need to 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

[New - Windows 95]

Returns the directory designated as the personal address book (PAB).

Syntax

HRESULT GetPAB(**ULONG FAR *** *lpcbEntryID*, **LPENTRYID FAR *** *lppEntryID*)

Parameters

lpcbEntryID
Output parameter pointing to a variable containing the number of bytes in the entry identifier pointed to by the *lppEntryID* parameter.

lppEntryID
Output parameter pointing to a variable that receives a pointer to the personal address book's entry identifier. Returns a value of zero if no containers can be established as the personal address book.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

Comments

MAPI establishes a default personal address book if one has never been set in the profile. The default is determined by choosing the first container in the hierarchy that allows modifications.

Calls made to **IAddrBook::GetPAB** return the entry identifier of the personal address book's container in the *lppEntryID* parameter. MAPI allocates the memory for this entry identifier with the **MAPIAllocateBuffer** function and your application must release it with the **MAPIFreeBuffer** function when done.

See Also

MAPIAllocateBuffer function, **MAPIFreeBuffer** function, PR_CONTAINER_FLAGS property

IAddrBook::GetSearchPath

[New - Windows 95]

Returns the search path that is used in **ResolveNames** methods.

Syntax

HRESULT GetSearchPath(**ULONG** *ulFlags*, **LPSRowSet FAR** * *lppSearchPath*)

Parameters

ulFlags
: Input parameter containing a bitmask of flags used to control 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 8-bit format.

lppSearchPath
: Output parameter pointing to the variable that receives an **SRowSet** structure containing information about the search path. If there are no containers in the hierarchy, zero is returned in the **SRowSet**.

Return Values

S_OK
: The call succeeded and has returned the expected value or values.

Comments

Use the **IAddrBook::GetSearchPath** method to get the search path that is used in **ResolveNames** methods. If **SetSearchPath** has never been called, **GetSearchPath** build a search path by working through the address book's hierarchy tables (this is the default behavior of **GetSearchPath**). The search path starts with the first address book with read-write access, usually the user's personal address book, followed by every address book container that has a PR_DISPLAY_TYPE property set to the DT_GLOBAL flag. (If a container's PR_DISPLAY_TYPE is set to DT_GLOBAL, it indicates that it contains a global address list which holds recipients.) If **GetSearchPath** does not find a global address book using this search path, then the default directory is added to the search path list, providing that the default directory is not the first container with read-write access.

If **SetSearchPath** has been called, **GetSearchPath** forms a search path using the address book containers that have been stored in the user's profile. The search path obtained from the user's profile is validated against the hierarchy table for the session's address book before **GetSearchPath** returns the search path to the calling application. After the first call to **SetSearchPath**, subsequent calls to **SetSearchPath** must be used to modify the search path returned by **GetSearchPath**. In other words, your application does not receive the default search path after the first call to **SetSearchPath**.

See Also

IAddrBook::SetSearchPath method, **SRowSet** structure

IAddrBook::NewEntry

[New - Windows 95]

Displays a dialog box for creating new entries within containers or custom recipient addresses within messages.

Syntax

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 to the window that the dialog box is modal to.

ulFlags
: Reserved; must be zero.

cbEIDContainer
: Input parameter containing the number of bytes in the entry identifier in the *lpEIDContainer* parameter.

lpEIDContainer
: Input parameter pointing to the container where the new custom recipient address will be added. If the value in *cbEIDContainer* is zero, **NewEntry** returns a recipient entry identifier and a list of templates as if your application called **IAddrBook::CreateOneOff**.

cbEIDNewEntryTpl
: Input parameter containing the number of bytes in the entry identifier in the *lpEIDNewEntryTpl* parameter.

lpEIDNewEntryTpl
: Input parameter pointing to a template to be used to create the new custom recipient address. If the value in *cbEIDNewEntryTpl* is zero, passing NULL in the *lpEIDNewEntryTpl* parameter displays a dialog box enabling the user to select an address-creation template from among the address types that can be created in the container.

lpcbEIDNewEntry
: Output parameter pointing to a variable containing the number of bytes in the entry identifier in the *lppEIDNewEntry* parameter.

lppEIDNewEntry
: Output parameter pointing to a variable where the pointer to the entry identifier for the new custom recipient address is stored.

Return Values

S_OK
: The call succeeded and has returned the expected value or values.

Comments

The **IAddrBook::NewEntry** method can be used in several different modes. To add a custom recipient address directly to the open message and not to a modifiable container, your provider passes zero in the *cbEIDContainer* parameter and NULL in the *lpEIDContainer* parameter. To display a dialog box enabling the user to select a template for adding custom recipients to a modifiable container, your application passes zero in the *cbEIDNewEntryTpl* parameter and NULL in the *lpEIDNewEntryTpl* parameter.

To create an entry in a modifiable address book and not get the entry identifier back, pass 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, your application 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. Doing so opens the root folder of the address book container. Next, your application calls the **IABContainer::OpenProperty** method and passes the PR_CREATE_TEMPLATES property in the *ulPropTag* parameter so that it can open the property. Doing this returns a table object that lists the types of objects that can be created in the address book container. Your application finds in this table the entry identifier for the template you want new entries to be created with. Then, your application calls **NewEntry** and passes NULL in the *lpEIDContainer* parameter and the entry identifier for the entry-creation template you want used in the *lpEIDNewEntryTpl* parameter.

Calls made to **NewEntry** return the entry identifier of the new custom recipient address in the *lppEIDNewEntry* parameter (unless your application passed NULL in *lppEIDNewEntry*). Your client application 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

[New - Windows 95]

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.

Syntax

HRESULT OpenEntry(**ULONG** *cbEntryID*, **LPENTRYID** *lpEntryID*, **LPCIID** *lpInterface*, **ULONG** *ulFlags*, **ULONG FAR *** *lpulObjType*, **LPUNKNOWN FAR *** *lppUnk*)

Parameters

cbEntryID
Input parameter containing the number of bytes in the *lpEntryID* parameter.

lpEntryID
Input parameter pointing to the address of the entry identifier for the opened object.

lpInterface
Input parameter pointing to the interface identifier (IID) for the object. The value NULL indicates the MAPI interface for the given object will be returned. The parameter can also be set to an identifier for another appropriate interface for the object.

ulFlags
Input parameter containing a bitmask of flags used to control how the object is opened. The following flags can be set:

MAPI_BEST_ACCESS
Indicates the object should be opened with the maximum privileges allowed to the user. For example, if the client application has write privilege, open the object with write privilege; if the client application has read-only privilege, open the object with read-only privilege. The client application can learn the privilege 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 application. 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 client applications should not assume that write access was granted.

lpulObjType
Output parameter pointing to a variable where the object type for 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 to modify a read-only object or an attempt to access an object for which the user has insufficient permissions.

MAPI_E_UNKNOWN_ENTRYID
The object indicated in the *lpEntryID* parameter is not recognized. This return value is typically returned if the address book provider that the object is contained within is not open.

MAPI_E_NOT_FOUND
The object indicated in the *lpEntryID* parameter does not exist.

Comments

Use the **IAddrBook::OpenEntry** method to open a container or recipient. 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 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 on the opened object. Passing NULL in *lpInterface* indicates the standard MAPI interface for that type of object should be used.

The calling application should check the value returned in the *lpulObjType* parameter to determine that the object type returned is what was expected. Commonly, after the application 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.

IAddrBook::PrepareRecips

[New - Windows 95]

Prepares a recipient list for later use by the messaging system. Converts recipients' short-term entry identifiers to long-term entry identifiers, updates those recipients that belong to this address book provider, and, if necessary, retrieves those recipients' permanent entry identifiers along with any additional properties requested.

Syntax

HRESULT PrepareRecips(**ULONG** *ulFlags*, **LPSPropTagArray** *lpPropTagArray*, **LPADRLIST** *lpRecipList*)

Parameters

ulFlags
Reserved; must be zero.

lpPropTagArray
Input parameter pointing to a counted array of property tags for the properties that require updating by the calling application. 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.

Comments

The **IAddrBook::PrepareRecips** method is called to ensure that all recipients your address book provider recognizes have permanent entry identifiers and that they have all the properties requested in the *lpPropTagArray* parameter. 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 recognized by your provider, it 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 property is not available to the calling application.

Like the **ADRLIST** structure, each **SPropValue** property value structure passed in the *lpPropTagArray* parameter must be separately allocated using **MAPIAllocateBuffer** and **MAPIAllocateMore** 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

IAddrBook::QueryDefaultRecipOpt

[New - Windows 95]

Returns the available recipient options for a particular messaging address type, with defaults.

Syntax

HRESULT QueryDefaultRecipOpt(**LPTSTR** *lpszAdrType*, **ULONG** *ulFlags*, **ULONG FAR *** *lpcValues*, **LPSPropValue FAR *** *lppOptions*)

Parameters

lpszAdrType
Input parameter pointing to a string containing the messaging address type for which the options dialog box should be displayed; examples of e-mail address types are FAX, SMTP, X500, and so on. The *lpszAdrType* parameter's value must not be NULL.

ulFlags
Input parameter containing a bitmask of flags used to control 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 8-bit 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.

Comments

Use the **IAddrBook::QueryDefaultRecipOpt** method to return the recipient options available for a particular messaging address type and set those options to the default values. Recipient options are the properties of a recipient that govern its behavior after an application submits a message. Recipient options are usually, but not always, specific to a particular address type.

Client applications 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 client applicants want to display a dialog box to enable the user to select recipient options, 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, see the **IMAPISession::MessageOptions** method.

See Also

IAddrBook::RecipOptions method, **IMAPISession::MessageOptions** method, **IMAPISession::QueryDefaultMessageOpt** method

IAddrBook::RecipOptions

[New - Windows 95]

Displays a modal per-recipient options dialog box on a particular recipient.

Syntax

HRESULT RecipOptions(**ULONG** *ulUIParam*, **ULONG** *ulFlags*, **LPADRENTRY** *lpRecip*)

Parameters

ulUIParam
    Input parameter containing the handle of the window the dialog box is modal to.

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. Use the HR_FAILED macro to test for this warning, but the call should be handled as a successful return. .

Comments

Use 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 the application submits a message. 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.

If you want to get 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 recipients, such options can be applied to an entire message, as a default for all the message's recipients; for more information, see the **IMAPISession::MessageOptions** method.

See Also

ADRENTRY structure, **IAddrBook::QueryDefaultRecipOpt** method, **IMAPISession::MessageOptions** method, **IMAPISession::QueryDefaultMessageOpt** method, **SPropValue** structure

IAddrBook::ResolveName

[New - Windows 95]

Displays a dialog box for locating a recipient given a partial name.

Syntax

HRESULT ResolveName(**ULONG** *ulUIParam*, **ULONG** *ulFlags*, **LPTSTR** *lpszNewEntryTitle*,
LPADRLIST *lpAdrList*)**;**

Parameters

ulUIParam
Input parameter whose usage depends on the platform. On Windows platforms, the *ulUIParam* parameter is the handle to the main window of the calling application, cast to a ULONG.

ulFlags
Input parameter that is a bitmask of flags that determine whether a name-resolution dialog box is displayed. The following flag can be set:

MAPI_DIALOG
Displays a dialog box to provide status or prompt the user for additional information, such as recipients, sending options, or name resolution. If this flag is not set, no dialog box is displayed.

lpszNewEntryTitle
Input parameter pointing to a string containing text for the title of 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 a list of names to be matched to recipients.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

MAPI_E_AMBIGUOUS_RECIP
The requested recipient matched more than one entry identifier. Usually this value is returned when a dialog box allowing the user to select the correct recipient 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.

Comments

The **IAddrBook::ResolveName** method goes through the address list, and finds all the names not yet resolved, resolves them, and returns the appropriately modified recipient address list. The address list can be one that was originally created using the **IAddrBook::Address** method.

Entries in the *lpAdrList* parameter that are lacking the PR_ENTRYID property are considered unresolved.

If a name cannot be resolved because there are multiple matches and because the client application has instructed MAPI not to display a name-resolution dialog box (by not setting the MAPI_DIALOG flag in the *ulFlags* parameter), **ResolveName** returns the value MAPI_E_AMBIGUOUS_RECIP in the *lpAdrList* parameter. If a name cannot be resolved because there are no matches for it, **ResolveName** returns the value MAPI_E_NOT_FOUND in *lpAdrList*. If there are some names with multiple matches and some names without matches, **ResolveName** can return either error value.

See Also

ADRLIST structure, **IABContainer::ResolveNames** method, **IAddrBook::Address** method

IAddrBook::SetDefaultDir

[New - Windows 95]

Chooses a different directory as the default directory for the address book container.

Syntax

HRESULT SetDefaultDir(**ULONG** *cbEntryID*, **LPENTRYID** *lpEntryID*)

Parameters

cbEntryID
Input parameter containing the number of bytes in the *lpEntryID* parameter.

lpEntryID
Input parameter pointing to the entry identifier of the default directory.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

Comments

The default directory is the directory the user sees displayed in the address book when the address book is first opened. The **IAddrBook::SetDefaultDir** method sets the default directory in the profile. The directory is saved within the profile between instances of a session, so after a call to the **IMAPISession::Logoff** method, subsequent calls to the **MAPILogonEx** function during the same session return the same default directory as previously set, as long as that directory still exists.

Your application doesn't need to call the **IAddrBook::SaveChanges** method to make the directory change permanent.

See Also

IAddrBook::GetDefaultDir method, **IAddrBook::GetSearchPath** method, **IMAPISession::Logoff** method, **MAPILogonEx** function

IAddrBook::SetPAB

[New - Windows 95]

Designates a particular container to be the personal address book.

Syntax

HRESULT SetPAB(**ULONG** *cbEntryID*, **LPENTRYID** *lpEntryID*)

Parameters

cbEntryID
Input parameter containing the number of bytes in 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 the value MAPI_E_INVALID_ENTRYID.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

Comments

Use the **IAddrBook::SetPAB** method to designate a particular container to be the personal address book. This is the container where new entries are added. The directory is saved between instances of a session, so after a call to the **IMAPISession::Logoff** method, subsequent calls to the **MAPILogonEx** function during the same session return the same default directory as previously set, as long as that directory still exists.

Your application doesn't need to 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

[New - Windows 95]

Sets the path that is used in **ResolveNames** methods.

Syntax

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 must contain the PR_ENTRYID property as the first column in the row set.

Comments

The **IAddrBook::SetSearchPath** method saves changes made to the container search order that is used for **ResolveNames** methods. The directory is saved between instances of a session, so after a call to the **IMAPISession::Logoff** method, subsequent calls to the **MAPILogonEx** function during the same session return the same default directory as previously set, as long as that directory still exists.

Your application doesn't need to 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

[New - Windows 95]

Removes an object's registration for notification of address book changes previously established with a call to the **IAddrBook::Advise** method.

Syntax

HRESULT Unadvise(**ULONG** *ulConnection*)

Parameters

ulConnection
Input parameter containing the number of the registration connection returned by **IAddrBook::Advise**.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

Comments

Client applications use the **IAddrBook::Unadvise** method to release the pointer to the advise sink object that was passed in the *lpAdviseSink* parameter in the previous call to **IAddrBook::Advise**, thereby canceling a notification registration. As part of discarding the pointer to 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 **IMAPIAdviseSink::OnNotify** on the advise sink object, the **Release** call is delayed until the **OnNotify** method returns.

See Also

IAddrBook::Advise method, **IMAPIAdviseSink::OnNotify** method

Introduction

This chapter covers concepts integral to working with Extended MAPI, particularly Extended MAPI functions and structures. It does not provide a full explanation of MAPI architecture and concepts; for a more complete conceptual discussion of MAPI, see *MAPI Programmer's Guide*.

Essential Functions vs. Additional Functions

Functions documented in Chapter 4, "Essential Extended MAPI Functions" are those functions that are basic to MAPI development and without which MAPI-based programs cannot be created. Functions documented in Chapter 7, "Additional Extended MAPI Functions," are listed there as general information for developers but are rarely used in development with MAPI.

Some Conventions for Working with Extended MAPI Functions and Structures

The following topics provide information on some conventions common to Extended MAPI functions and to Extended MAPI structures. Learning about these similarities helps in learning and applying the MAPI programming model consistently throughout your program.

Function Parameter Commonalities

Many MAPI functions take similar parameters as arguments. Learning about these similarities helps in learning and applying the MAPI programming model. Shown here are the more commonly used parameters and the way in which each is used throughout the MAPI functions.

Parameter	**Usage**
lpInterface	Pointer to the interface to be used.
ulUIParam	Handle of the window where the action is taking place.
ulFlags	Flags passed in.
lpulFlags	Flags passed in and passed back on return.
lpMAPIError	Pointer to a **MAPIERROR** structure.
lpMessage	Pointer to a message object.
lpProblems	Pointer to an array of problems returned on operations involving properties.
lpProgress	Pointer to a progress object.
lpTable	Pointer to a table object.
lpUnk	Pointer to an object returned from a call to a **QueryInterface** method.
lpPropTagArray	Pointer to an array of property tags.
lpEntryID	Pointer to an entry identifier for an object.
lpulConnection	Pointer to the connection number used to register a notification.

Input, Output, and Input-Output Parameters

In descriptions of function parameters, *MAPI Programmer's Reference, Volume 2,* has adopted the convention of indicating whether a parameter is an input, output, or input-output parameter. The distinction is important for knowing how to efficiently manage memory.

As discussed in *MAPI Programmer's Reference, Volume 1*, the memory management model employed in MAPI is an enhancement of the rules specified in OLE's component object model. When internally allocating and freeing memory not related to MAPI processes within your client application or service provider, you can use whatever mechanism makes sense.

Parameters fall into one of three groups. They may be *input* parameters, set by the calling implementation with information to be used by the called function, *output* parameters, set by the called function and returned to the calling implementation, or *input-output* parameters, a combination of the two groups. The MAPI memory management model dictates how memory for these groups of parameters is allocated and freed as follows:

Parameter group	**Allocating memory**	**Freeing memory**
Input	Calling implementation is responsible for allocating memory and can use any mechanism.	Calling implementation is responsible for freeing memory and can use any mechanism.
Output	Called implementation is responsible for allocating memory and must use the **MAPIAllocateBuffer** function.	Called implementation is responsible for freeing memory and must use the **MAPIFreeBuffer** function.
Input-output	Calling implementation is responsible for the initial allocation; called implementation can reallocate memory if necessary using the **MAPIAllocateMore** function.	Called implementation is responsible for initial memory freeing if reallocation is necessary. Calling implementation must free the final return value.

When a process fails, providers need to pay attention to output and input-output parameters because the calling implementation generally has no way to release the memory for the failed process. If a function returns a value indicating failure, then each output or input-output parameter must either be left at the value initialized by the calling implementation or set to a value for which memory will be released without any action on the part of the calling application. For example, a provider must leave an output pointer parameter of void ** ppv as it was on input; otherwise, it must set the returned pointer to NULL (*ppv = NULL).

Flag Usage for Functions and Structures

MAPI functions and structures (as well as MAPI methods) make extensive use of bitmasks to pass flags so that bits can be set to control how an operation is performed. The most common usage of bitmasks is in *ulFlags* parameters, but you will find them used in other parameters and structure members as well.

MAPI has predefined all of the allowable flags as constants. These constants take the form of all uppercase letters with each word separated by an underscore, for example MAPI_UNICODE. Your client or provider should only set the bits in the bitmask by passing in the constants allowed for that parameter; it should not set the bits itself or define its own flags. The only valid flags for a particular parameter of a given function, or a particular member of a given structure, are the ones that are listed under the parameter or member description. Passing any other flags, or setting any other bits by hand, will result in the error code MAPI_E_UNKNOWN_FLAGS being returned.

For example, **IMessage::SetReadFlag** takes a single parameter, *ulFlags*. The allowable flags are CLEAR_READ_FLAG, MAPI_DEFERRED_ERRORS, and SUPPRESS_RECEIPT. You set the bit for CLEAR_READ_FLAG like this:

```
pMessage->SetReadFlag(CLEAR_READ_FLAG);
```

Some functions take a *ulFlags* parameter in which all of the bits are reserved for MAPI's use. In such cases, the parameter description will read "Reserved; must be zero." Passing any value other than zero in such parameters results in the MAPI_E_UNKNOWN_FLAGS error being returned. If more than one flag is defined for a parameter, any combination of flags can be passed using the logical-OR operator, unless the documentation for that parameter specifically mentions that some flags are mutually exclusive. For example, you set all of the flags for **SetReadFlag** like this:

```
pMessage->SetReadFlag(CLEAR_READ_FLAG | MAPI_DEFERRED_ERRORS |
SUPPRESS_RECEIPT);
```

The called function then examines each relevant bit of the *ulFlags* parameter to see if that bit is set.

Common Messaging Calls (CMC)

The Common Messaging Calls (CMC) applications programming interface (API) is designed to provide simple and convenient set of functions to applications that need basic messaging services. CMC provides for all the major messaging functions 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 isolates the calling application from needing to know anything 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 that the CMC implementation uses to perform the tasks needed by the calling application. Output parameters are used by the CMC implementation to pass information back 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 discussed in this document. The C language definitions of them can be found in the following header files:

Header file name	**Header file contents**
XCMC.H	CMC data structure and symbolic constants.
XCMCEXT.H	Common CMC extension data structures and symbolic contstants.
XCMCMSXT.H	Microsoft CMC extension data structures and symbolic constants.

CMC 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 discussed later in this document.

cmc_act_on

[New - Windows 95]

Performs the specified operation on a message.

Syntax

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 identifier 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 code instead.

ui_id
Input parameter containing a handle that **cmc_act_on** uses to present a user interface to resolve 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 that the caller has no extensions for **cmc_act_on** and is expecting no extensions.

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 code.

CMC_E_INSUFFICIENT_MEMORY
Insufficient memory was available to complete the requested operation.

CMC_E_INVALID_ENUM
A **CMC_enum** value is 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 identifier 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 cannot be completed at this time because the message is 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_free

[New - Windows 95]

Frees memory allocated by the messaging system through another CMC function.

Syntax

CMC_return_code cmc_free (CMC_buffer *memory***)**

Parameters

memory
Input parameter pointing to memory previously allocated by the message service through a CMC function. 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 code.

CMC_E_INVALID_MEMORY
A memory pointer passed is invalid.

Comments

Results of the **cmc_free** function are unpredictable if your 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 your application with a base pointer to a complex structure containing several levels of pointers. Your 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

[New - Windows 95]

Lists summary information for messages that meet application-specified criteria.

Syntax

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_result FAR * FAR** **result*, **CMC_extension FAR** **list_extensions***)**

Parameters

session
Input parameter containing an opaque session identifier that represents a MAPI session object indicating a session with the messaging system. If the session identifier 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 code 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, the function 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.

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 specifies 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 user interface 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 code.

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 identifier 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.

Comments

Your application can specify a **cmc_list** search to start with a certain message or to start at the beginning of 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, your 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

[New - Windows 95]

Logs a client application off a service provider.

Syntax

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 identifier that represents a MAPI session object indicating a session with the messaging system. If the session identifier is invalid, the **cmc_logoff** function returns CMC_E_INVALID_SESSION_ID. After **cmc_logoff** returns, the session identifier is invalid.

ui_id
: Input parameter containing the handle of a user interface 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 code instead.

  CMC_LOGOFF_UI_ALLOWED
  : Indicates **cmc_logoff** can display a user interface 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 code.

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 identifier 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 is not logged on and the CMC_LOGON_UI_ALLOWED flag is not set.

See Also

CMC_extension structure

cmc_logon

[New - Windows 95]

Logs a client application onto a service provider.

Syntax

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 user interface.

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 user interface 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 user interface to prompt for a password.

character_set
Input parameter pointing to an object identifier for the character set used by the client application. The 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 user interface 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 length-first. 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 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 if the user does not supply enough information.

session
Output parameter pointing to the location to which **cmc_logon** writes an opaque session identifier.

This identifier represents a MAPI session object indicating a session with the messaging system.

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 code.

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 were 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 is unavailable.

CMC_E_UNSUPPORTED_CHARACTER_SET
The character set requested is not supported.

CMC_E_UNSUPPORTED_FLAG
The flag requested is not supported.

CMC_E_UNSUPPORTED_FUNCTION_EXT
The function extension requested is not supported.

CMC_E_UNSUPPORTED_VERSION
The version specified in the call cannot be supported by the current implementation.

Comments

The **cmc_logon** function can, at the client application's option, either prompt through a user interface or proceed without any user interaction. It writes a session identifier that the application can use in subsequent calls to the CMC implementation.

See Also

CMC_extension structure, **cmc_query_configuration** function

cmc_look_up

[New - Windows 95]

Looks up addressing information in a directory provided by a specified service provider.

Syntax

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 identifier that represents a MAPI session object that represents a session with the messaging system. If the session identifier is invalid, the **cmc_look_up** function returns the CMC_E_INVALID_SESSION_ID error code .

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 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 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 application has set the CMC_LOOKUP_ADDRESSING_UI flag, **cmc_look_up** displays the recipients specified in the recipient array in the address-list user interface.

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 length-first. 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 code 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 if the user does not supply enough information.

CMC_LOOKUP_ADDRESSING_UI

Displays a user interface 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 user interface. 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 user interface for the recipient pointed to in the *recipient_in* parameter. This user interface 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 user interface and returns the error CMC_E_AMBIGUOUS_RECIPIENT.

CMC_LOOKUP_RESOLVE_IDENTITY

Returns a recipient record for the identity of the current user of the messaging system. 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 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 CMC_E_AMBIGUOUS_RECIPIENT for services that must resolve to a single name. Services that can return multiple names return a list as indicated by other function parameters. Support for this flag is optional.

ui_id

Input parameter containing the handle of a user interface 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 specifies 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 is ambiguous. Multiple matches were found.

CMC_E_FAILURE

There was a general failure that does not fit the description of any other return code.

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 identifier 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_NOT_SUPPORTED
The operation requested is not supported by the current implementation.

CMC_E_RECIPIENT_NOT_FOUND
One or more of the specified recipients were not found.

CMC_E_UNSUPPORTED_DATA_EXT
The data extension requested is not supported.

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_CANCEL
The operation was canceled by the user.

CMC_E_USER_NOT_LOGGED_ON
The user is not logged on and the CMC_LOGON_UI_ALLOWED flag is not set.

Comments

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 user interface 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, your 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

[New - Windows 95]

Determines configuration information for the installed CMC implementation.

Syntax

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 identifier that represents a MAPI session object indicating a session with the messaging system. 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 pointers for *reference* depending on the value of the *item* parameter. 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 returns a pointer to the array of object identifier strings that indicate character sets for the current implementation. 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 this object identifier at logon to specify that the implementation use a different character set than the default.

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 service name, if available, followed by a null character. The **cmc_query_configuration** function returns NULL 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 asking the user for the service name, user name, and password. The string is returned in the implementation default character set.

CMC_CONFIG_DEFAULT_USER

Indicates the *reference* parameter should be a pointer to a CMC_string data type, into which **cmc_query_configuration** returns a pointer to the default user name, if available, followed by a null character. The **cmc_query_configuration** function returns NULL 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 asking the user for the provider name, user name, and password. The string is returned in the implementation default character set.

CMC_CONFIG_LINE_TERM

Indicates the *reference* parameter should be a pointer to a CMC_enum variable, which is set to a value of 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 a value of 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 a value of 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 a value of 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** 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 user interface 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 the buffer 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 code.

CMC_E_INSUFFICIENT_MEMORY
Insufficient memory was available to complete the requested operation.

CMC_E_INVALID_ENUM
A **CMC_enum** value is invalid.

CMC_E_INVALID_PARAMETER
    A function parameter was invalid.

CMC_E_NOT_SUPPORTED
    The operation requested is not supported by the current implementation.

CMC_E_UNSUPPORTED_FUNCTION_EXT
    The function extension requested is not supported.

Comments

Your application must cast the *reference* parameter to the CMC_buffer type before calling **cmc_query_configuration**. The application must allocate sufficient memory to contain the information passed in the *item* parameter. When this memory is no longer needed, your application should free this memory with a call to the **cmc_free** function.

See Also

CMC_extension structure, **cmc_free** function, **cmc_read** function

cmc_read

[New - Windows 95]

Reads a specified message.

Syntax

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 identifier that represents a MAPI session object indicating a session with the messaging system. 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 passing the CMC_CONFIG_SUP_NOMKMSGREAD value in 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 code instead.

CMC_MSG_AND_ATT_HDRS_ONLY
Indicates that the *attach_filename* fields areundefined when **cmc_read** returns and that they should be ignored. The calling client or provider can be used to reduce the amount of data transferred. If clear, the *attach_filename* fields are returned as usual.Note that if the CMC_MSG_TEXT_NOTE_AS_FILE value is set in the message to indicate the first attachment contains the text note, **cmc_read** returns the *attach_filename* field for that attachment regardless of the setting of this 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 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 user interface 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 code.

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 identifier 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 CMC_DO_NOT_MARK_AS_READ flag cannot be supported.

CMC_E_UNSUPPORTED_FLAG
The flag requested is not supported.

CMC_E_UNSUPPORTED_FUNCTION_EXT
The function extension requested is not supported.

Comments

The **cmc_read** function only reads the first message in the mailbox if the application passes a null message-reference value.

After processing, **cmc_read** writes the data from the message into the **CMC_message** structure. Unless your 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 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, your 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_message** structure, **CMC_message_reference** structure, **cmc_query_configuration** function

cmc_send

[New - Windows 95]

Sends a message.

Syntax

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** **send_extensions***)**

Parameters

session
Input parameter containing an opaque session identifier that represents a MAPI session object indicating a session with the messaging system. If the value provided for the *session* parameter is invalid, the **cmc_send** function returns the CMC_E_INVALID_SESSION_ID.

message
Input parameter pointing to a **CMC_message** structure identifying the message to be sent. If the application has not set the flag CMC_SEND_UI_REQUESTED in the *send_flags* parameter, the message structure must specify at least one primary, 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 length-first. 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 code 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 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** does not display a dialog box but must specify at least one recipient.

ui_id
Input parameter containing the handle of a user interface for **cmc_send** to present to help resolve processing questions, prompt the user for additional information, or verify 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 is 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
: 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 code.

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 identifier 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 data extension requested is not supported.

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_CANCEL
: The operation was canceled by the user.

CMC_E_USER_NOT_LOGGED_ON
: The user is not logged on and the CMC_LOGON_UI_ALLOWED flag is not set.

Comments

The **cmc_send** function can, at the 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.

Your application can optionally provide recipient list, subject text, attachments, and note text for the message. If your application does not provide the required message elements, the **cmc_send** function can prompt the user for them if a user interface is available. If the application provides one or more recipients, the function can send the message without prompting the user. If the application 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 your 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 your application does pass a non-null value to indicate a text note 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 your 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 your application specifies only a name, **cmc_send** resolves the name to an address using name resolution rules defined by the CMC implementation. If your application specifies only an address, **cmc_send** uses this address for delivery and for the recipient display name. Finally, if your application 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 your 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 user can freely change or delete attachment files 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 is ignored and the text note is contained in the file referred to by the first attachment. If this flag is clear, the text note is contained in the **text_note** member.

See Also

CMC_extension structure, **CMC_message** structure, **CMC_recipient** structure

cmc_send_documents

[New - Windows 95]

Sends a document.

Syntax

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 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 application can also use the prefix TO: for consistency. 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 controlling 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 length-first. 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 code instead.

CMC_FIRST_ATTACH_AS_TEXT_NOTE
Sends the first attachment as the message text note. If 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 name for the attachment file. When the application specifies multiple path names, 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

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 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 user interface 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

CMC_E_FAILURE
There was a general failure that does not fit the description of any other return code.

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 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_FLAG
The flag requested is not supported.

CMC_E_USER_CANCEL
The operation was canceled by the user.

CMC_E_USER_NOT_LOGGED_ON
The user is not logged on and the CMC_LOGON_UI_ALLOWED flag is not set.

Comments

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 user interface. If this is not possible, it prompts for logon information to establish a session. Before the function returns, it closes the session.

CMC Structures and Simple Data Types

The following data structures and simple data types are used by the CMC implementation for passing information into and out of CMC functions. Wherever possible, you should use these types to maintain compatibility with different CMC implementations.

BYTE

[New - Windows 95]

BYTE is an unsigned character data type for binary data.

Syntax

```
typedef unsigned char  BYTE;
```

CMC_attachment

[New - Windows 95]

A **CMC_attachment** structure holds a CMC message attachment.

Syntax

```
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
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** designates an attachment of an undefined type.

attach_filename
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 that describe attachment options. The following flags can be set:

CMC_ATT_APP_OWNS_FILE
Indicates on output that the 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 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 clear for any structure, that structure is not the last array element.

attach_extensions
Pointer to the first **CMC_extension** structure in an array of per-attachment extensions.   A pointer value of NULL indicates that no extensions are present.

Comments

A **CMC_message** structure, which holds information about a CMC message, contains a pointer to an array of one or more **CMC_attachment** structures defining attachments for the message. The last attachment structure should have the CMC_ATT_LAST_ELEMENT flag set in its **attach_flags** member.

See Also

CMC_extension structure, **CMC_message** structure

CMC_boolean

[New - Windows 95]

CMC_boolean is a CMC unsigned integer data type that is a Boolean value.

Syntax

```
typedef    CMC_uint16  CMC_boolean;
```

Comments

The CMC_boolean data type can hold 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 0, in line with C interface usage. However, CMC_TRUE specifically refers to the integer 1.

CMC_buffer

[New - Windows 95]

CMC_buffer is a CMC data type that is a pointer to a memory storage location of an undefined type and size.

Syntax

```
typedef    void *CMC_buffer;
```

Comments

The size of a void pointer is specific to the platform.

CMC_counted_string

[New - Windows 95]

A **CMC_counted_string** structure is an optional structure that supports character sets that allow embedded null characters. The structure contains a counted string and explicitly defines the length of the string.

Syntax

```
typedef struct {
     CMC_uint32  length;
     char   string[1];
} CMC_counted_string;
```

Members

length
Length, in bytes, of the string specified by the **string** member.

string
An array of characters that make up the string. CMC does not require the string to be null-terminated.

Comments

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 characters in a string, the CMC implementation looks at the session context. CMC implementations always attempt to map all strings passed to the client application to the character set for the session. If no session context has been created, CMC interprets the string by using its default character set.

See Also

CMC_string data type

CMC_enum

[New - Windows 95]

CMC_enum is a CMC data type that is an enumerated data value.

Syntax

```
typedef    CMC_sint32        CMC_enum;
```

Comments

A variable of this type contains a value selected from an enumeration.

CMC_extension

[New - Windows 95]

A **CMC_extension** structure holds 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.

Syntax

```
typedef struct {
     CMC_uint32 item_code;
     CMC_uint32 item_data;
     CMC_buffer item_reference;
     CMC_flags extension_flags;
} CMC_extension;
```

Members

item_code
Code that uniquely identifies extended functionality for a function or data structure. The **item_code** member is the dispatch mechanism for the extension that is invoked. The client application puts the extension code in this member before calling a CMC function. 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 identifier CMC_XC_COM. For definitions of these extensions, see CMC Data Extensions.

item_data
Item data for the extension. Depending on the value of **item_code**, the **item_data** member might hold 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
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 clear 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 clear, the implementation has not allocated memory for the extension that the application needs to free. This flag is always clear for structure extensions.

CMC_EXT_REQUIRED
Indicates an error is returned if this extension cannot be supported. If this flag is clear, it enables the CMC implementation to provide any level of support, including no support, of the extension.

Comments

Extensions are used to add functionality to the CMC API. For example, a client application might 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 application might 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 extension item code. For input extensions, the 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 the allocated storage with a call to the **cmc_free** function. A client must call **cmc_free** once for each pointer returned in the **item_reference** member of every **CMC_extension** structure in the array.

CMC does not require explicit release of a data extension structure, because CMC releases such structures along with the structures that contains 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** message structure.

See Also

cmc_act_on function, **cmc_free** function, **CMC_message** structure, **cmc_read** function

CMC_flags

[New - Windows 95]

CMC_flags is a CMC data type that is a bitmask of flags. These flags' meanings depend on the context in which the client application uses the flags.

Syntax

```
typedef            CMC_uint32  CMC_flags;
```

Comments

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 clear (that is, set to zero).

CMC_message

[New - Windows 95]

A **CMC_message** structure holds information about a CMC message.

Syntax

```
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
Pointer 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
Pointer to a string that identifies the type of the message. Three different string identifiers are available:

- Object identifiers, which are used for types identified by object identifiers, as defined in *CCITT Recommendation X.208*.
- CMC registered values, which are used for types defined in the CMC specification.
- Bilaterally defined values, which are used for types that are unregistered. CMC does not ensure that bilaterally defined values are unique.

For a complete list of CMC message types, see "Comments."

subject
Pointer to a string describing the subject of the message.

time_sent
CMC_time structure containing the date and time when the client application submits the message to the CMC implementation.

text_note
Pointer to the text note string of the message. If the value of this member is NULL, there is no text note. If the CMC_MSG_TEXT_NOTE_AS_FILE flag is set for the **message_flags** member, the text note is in the first attachment to the message. For information on text note format, see "Comments."

recipients
Pointer to the first **CMC_recipient** structure in an array of structures defining the message recipients.

attachments
Pointer to the first **CMC_attachment** structure in an array of 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 structure 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 clear for any structure, that structure is not the last array element.

CMC_MSG_READ
Indicates that the message has been read. If the flag is clear, the message has not been read.

CMC_MSG_TEXT_NOTE_AS_FILE
Indicates that the **text_note** member is ignored and the text note is contained in the file referred to by the first attachment. If this flag is clear, the text note is contained in the **text_note** member.

CMC_MSG_UNSENT
Indicates that the 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 clear, the application has sent the message.

message_extensions
Pointer to the first structure in an array of **CMC_extension** structures representing the message extensions.

Comments

The format of each CMC message type possible in the **message_type** member is shown in the following syntax. In this syntax, white space can be any combination of tabs or spaces. An asterisk (*) indicates one or more of the denoted tokens (separated by white space) is valid. Strings within quotes are case-insensitive.

```
message_type_value            ::= oid | cmc_reg | bilat_def
oid                                 ::= "OID:" object_identifier
cmc_reg                             ::= "CMC:" cmc_registered_value
bilat_def                     ::= "BLT:" string
object_identifier       ::= object_id_component*
object_id_component           ::= integer
cmc_registered_value          ::= "IPM"| "IP RN" | "IP NRN" | "DR" | "NDR"
```

The registered values provided for cmc_registered_value are defined as follows:

CMC: 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.

CMC: IP RN
Receipt notification for an interpersonal message. A receipt notification indicates the recipient has read a message.

CMC: IP NRN
Nonreceipt notification for an interpersonal message. A nonreceipt notification indicates a message has been removed from the recipient's mailbox without being read. For instance, the service or user has discarded the message or it has been automatically forwarded to another recipient.

CMC: DR
Delivery report. A delivery report indicates the service was able to deliver a message to its recipient.

CMC: NDR
Nondelivery report. A nondelivery report indicates the messaging service was not able to deliver a message to its recipient.

As the syntax preceding indicates, the oid type identifier indicates a type identified by object identifier, the cmc_reg type identifier indicates a type identified by a CMC registered value, and the bilat_def 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
```

Developers can format type identifiers as they want; the CMC implementation also defines a type identifier format that allows a client application to easily compare strings. The CMC implementation always returns type identifiers in this format, which does the following:

- Separates all tokens with a whitespace.
- Converts all white space to a single space.
- Converts the type identifiers OID, CMC, and BLT to uppercase.

The CMC implementation 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 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 message types correspond to X.400 message types, but non-X.400 message services can use them. Thus, the CMC implementation applies these types generically.

Note that 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 text note 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). Note that there is no guaranteed format fidelity. For example, the CMC read functions 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

[New - Windows 95]

A **CMC_message_reference** structure is a **CMC_counted_string** structure containing a message reference, which is the mailbox identifier for a message.

Syntax

```
typedef          CMC_counted_string      CMC_message_reference;
```

Comments

The CMC implementation only guarantees the message reference to be valid for the life of the session. The message reference is specific to the mailbox; CMC does not guarantee that the reference has any correspondence to a message identifier used by the messaging system. At any time during the current session, the client application can copy the message reference.

CMC_message_summary

[New - Windows 95]

A **CMC_message_summary** structure holds a CMC message summary

Syntax

```
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
Pointer 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
Pointer to a string that identifies the type of the message. See CMC_message for details.Three different string identifiers are available:

- Object identifiers, which are used for types identified by object identifiers, as defined in *CCITT Recommendation X.208*.
- CMC registered values, which are used for types defined in the CMC specification.
- Bilaterally defined values, which are used for types that are unregistered. CMC does not ensure that bilaterally defined values are unique.

subject
Pointer to a string describing the subject of the message.

time_sent
CMC_time structure containing the date and time when the client application submits the message to the CMC implementation.

byte_length
Message size, in bytes. The value should include the size of all the associated features of the message – attachments, envelope and heading fields, and so on. A client application can supply an approximate message size or, if the message size is unknown or unavailable, the value CMC_LENGTH_UNKNOWN.

originator
Pointer 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 clear for any structure, that structure is not the last array element.

CMC_SUM_READ
Indicates that the message has been read. If the flag is clear, the message has not been read.

CMC_SUM_UNSENT

Indicates that the application has not sent the message, for example when the message is a draft. If this flag is clear, the application has sent the message.

message_summary_extensions

Pointer 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

[New - Windows 95]

CMC_object_identifier is a data type that is a CMC object identifier string.

Syntax

```
typedef          CMC_string  CMC_object_identifier;
```

Comments

An identifier of this type is globally unique. Its syntax must match the number form defined in *CCITT Recommendation X.208*. This syntax is:

```
object_identifier      ::= object_id_component*
object_id_component          ::= integer
```

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 the reference entry for the **CMC_message** structure.

See Also

CMC_message structure

CMC_recipient

[New - Windows 95]

A **CMC_recipient** structure holds information about a messaging user, either a message recipient or the message sender.

Syntax

```
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
Pointer 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.

name_type
Enumeration that indicates whether the structure holds information for a message recipient or a message sender. Enumeration possibilities include:

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 (0)
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 to optimize resolution of the name.

address
Pointer to a recipient or sender address string in a format recognized by the 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. Enumeration possibilities include:

CMC_ROLE_AUTHORIZING_USER
Indicates the authorizing user of the message.

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 the messaging system should ignore the specified recipient. This flag is useful for reusing an incoming message's recipient list for a reply. If this flag is clear, 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 clear for any structure, that structure is not the last array element.

CMC_RECIP_LIST_TRUNCATED

Indicates that the messaging system 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 clear, **cmc_look_up** has written a complete recipient array.

recip_extensions

Pointer to the first structure in an array of **CMC_extension** structures that hold the recipient or sender data extensions, if any.

Comments

If a message service does not support carbon copy recipients, it can convert such a recipient to a primary recipient. Services that cannot support blind carbon-copy recipients should reject associated messages. If a user designates the same recipient in more than one role, the message service requires multiple recipient entries, each differing from the others in role.

On output, the CMC implementation writes an array of **CMC_recipient** structures in a certain 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

[New - Windows 95]

CMC_return_code is a data type that is a 32-bit value that a CMC function returns.

Syntax

```
typedef     CMC_uint32       CMC_return_code;
```

Comments

A nonzero return value for a CMC function indicates an error and is associated with one of the defined CMC return codes. A return value of zero for a function indicates success. The CMC implementation reserves values in the low-order 16 bits of the return code for standard CMC-defined error codes. The CMC implementation reserves values in the high-order 16 bits for error codes that it defines specifically.

CMC client applications can resolve errors within the scope of the CMC implementation. For example, an 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 error to indicate that a dialog box regarding the error has already been displayed.

CMC_session_id

[New - Windows 95]

CMC_session_id is a data type that is a 32-bit CMC session identifier.

Syntax

```
typedef    uint32            CMC_session_id;
```

Comments

The context identified by the session identifier contains per-session information, 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 identifier, and the **cmc_logoff** function deletes it.

See Also

cmc_logoff function, **cmc_logon** function

CMC_string

[New - Windows 95]

CMC_string is a CMC data type that is a pointer to a character string.

Syntax

```
typedef     char       *CMC_string;
```

Comments

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 session context. CMC always attempts to map all strings passed to the client application to the character set for the session. If there is no session context created before the client application call, the CMC implementation interprets the string by using its default character set.

See Also

CMC_counted_string structure, **cmc_logon** function

CMC_time

[New - Windows 95]

A **CMC_time** structure holds a time value in CMC-compatible form for use in a message.

Syntax

```
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
Seconds; possible values range from 0 through 59.

minute
Minutes; possible values range from 0 through 59.

hour
Hours since midnight; possible values range from 0 through 23.

day
Day of the month; possible values range from 1 through 31.

month
Months since January; possible values range from 0 through 11.

year
Years since 1900.

isdst
Value for daylight savings time. A nonzero value means daylight savings time is in force.

unused1
Reserved. Do not use.

tmzone
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.

Comments

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

[New - Windows 95]

CMC_ui_id is a data type that is a CMC user interface handle.

Syntax

```
typedef CMC_uint32           CMC_ui_id;
```

Comments

The CMC implementation uses a data value of this type for passing user interface information to CMC functions. For example, in a Windows-based 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 appropriate default behavior.

CMC_X_COM_configuration

[New - Windows 95]

A **CMC_X_COM_configuration** structure holds configuration data written by the **cmc_query_configuration** function for the CMC_X_COM_CONFIG_DATA data extension.

Syntax

```
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
CMC specification version number.

ver_implem
CMC version number multiplied by 100. For example, version 1.00 is represented as 100.

character_set
Pointer to a **CMC_object_identifier** structure.

line_term
Enumerated value that indicates the type of line delimiter for the message.

default_service
Pointer to a string identifying the default message service.

default_user
Pointer to a string identifying the default user name.

req_password
Enumerated value that indicates if a password is required.

req_service
Enumerated value that indicates if the message service name is required for logon.**req_user**
Enumerated value that indicates if the messaging user name is required for logon.**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 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 is TRUE if the **cmc_logon** function supports the CMC_COUNTED_STRING_TYPE flag and FALSE otherwise.

Comments

The definition for each of the structure members corresponds to the data that the **cmc_query_configuration** function writes to its *reference* parameter for the corresponding value of its *item* parameter. The client application can free the **CMC_X_COM_configuration** structure with one call to the **cmc_free** function.

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

[New - Windows 95]

A **CMC_X_COM_support** structure holds information about MAPI support for a particular CMC data extension or extension set.

Syntax

```
typedef struct {
     CMC_uint32 item_code;
     CMC_flags flags;
} CMC_X_COM_support;
```

Members

item_code

Code for the CMC data extension whose support the application is querying about. The client application sets this member to the item code of the extension the application is querying during a call to the **cmc_logon** function. 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

flags

Bitmask of extension code flags. The following flags can be set:

CMC_X_COM_SUPPORTED
Indicates the CMC implementation supports this extension. For extension sets, this flag indicates the CMC implementation supports the required function and structure extensions in the set.

CMC_X_COM_NOT_SUPPORTED
Indicates the CMC implementation does not support this extension. If this flag applies to an extension set containing both function and structure extensions, it indicates the CMC implementation does not support some 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 the current session.

CMC_X_COM_DATA_EXT_SUPPORTED
Indicates the CMC implementation supports all required structure extensions for an extension set, but not all required function extensions. If **cmc_logon** sets this flag, which applies to extension sets only, as opposed to individual extensions.

CMC_X_COM_FUNC_EXT_SUPPORTED
Indicates the CMC implementation supports all required function extensions for a set, but not all required structure extensions. The client application must explicitly request them.

See Also

CMC_extension structure, **cmc_logon** function, CMC_X_COM_SUPPORT_EXT extension

LONG

[New - Windows 95]

LONG is a data type that is a 32-bit signed integer.

Syntax

```
typedef long             LONG;
```

CMC 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 set 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, holds 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.

This section documents the CMC common extension set, the extensions that are common to most message services 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_X_COM_ATTACH_CHARPOS

[New - Windows 95]

CMC_X_COM_ATTACH_CHARPOS is a CMC extension that supports display of a representation of a graphical attachment in message text note. 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

Comments

At logon, CMC passes the CMC_X_COM_ATTACH_CHARPOS code and any flags in a **CMC_X_COM_support** structure. Doing so indicates that CMC supports the corresponding extension and that the client application can attach the **extension** 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

[New - Windows 95]

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, the **cmc_look_up** function's *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
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 message service 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

[New - Windows 95]

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
All flags used with the **CMC_extension** structure are valid. No further flags are defined.

Output Usage

item_data
Unchanged.

item_reference
Pointer 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 flagCMC_EXT_OUTPUT.

Adds to

The **cmc_query_configuration** function

See Also

CMC_extension structure, **cmc_query_configuration** function, **CMC_X_COM_configuration** structure

CMC_X_COM_PRIORITY

[New - Windows 95]

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 low message priority.

CMC_X_COM_NORMAL
Indicates a normal priority for the message.

CMC_X_COM_URGENT
Indicates high message priority.

item_reference
NULL.

extension_flags
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 low message priority.

CMC_X_COM_NORMAL
Indicates a normal priority for the message.

CMC_X_COM_URGENT
Indicates high message priority.

item_reference
Unchanged.

extension_flags
Unchanged.

Adds to

The **CMC_message** and **CMC_message_summary** structures

Comments

At logon, the CMC implementation passes CMC_X_COM_PRIORITY in the **item_code** member of a **CMC_X_COM_support** structure to indicate that the extension should be attached to the **CMC_message** structure during the session.

See Also

CMC_extension structure, **CMC_message** structure, **CMC_message_summary** structure, **CMC_X_COM_support** structure

CMC_X_COM_RECIP_ID

[New - Windows 95]

CMC_X_COM_RECIP_ID is a CMC extension that adds a unique recipient identifier to a **CMC_recipient** structure for the recipients of a message.

Input Usage

item_data
Length of the recipient identifier.

item_reference
Pointer to the **CMC_recipient** structure holding the recipient identifier.

extension_flags
All flags used with the **CMC_extension** structure are valid. No further flags are defined.

Output Usage

item_data
Length of the recipient identifier.

item_reference
Pointer to the **CMC_recipient** structure holding the recipient identifier.

extension_flags
Unchanged.

Adds to

The **CMC_recipient** structure

Comments

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 in some message services. In this way the client application can reuse the recipient identifier returned by a previous call to **cmc_look_up** by attaching it to a recipient structure that the message service would otherwise try to resolve.

At logon, the client application passes the CMC_X_COM_RECIP_ID **item_code** in the CMC_X_COM_SUPPORT_EXT array to indicate 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

[New - Windows 95]

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
Pointer 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 this structure. To indicate that the operation to be performed 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
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.

Output Usage

item_data
Unchanged.

item_reference
Pointer to the message reference of the message saved to the receive folder (the Inbox). The client application must free this pointer 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

[New - Windows 95]

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
All flags used with the **CMC_extension** structure are valid. No further flags are defined.

Output Usage

item_data
Unchanged.

item_reference
Pointer 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
If the *item_reference* parameter contains a pointer to a message, the CMC_EXT_OUTPUT flag is set.

Adds to

The **cmc_send** function

Comments

CMC_X_COM_SAVE_MESSAGE is used to obtain information when the **CMC_message** structure is set by CMC rather than by the client application.

See Also

CMC_extension structure, **cmc_free** function, **CMC_message** structure, **cmc_send** function

CMC_X_COM_SUPPORT_EXT

[New - Windows 95]

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
Count of items in an array pointed to by **item_reference**.

item_reference
Pointer to the first element in an array of **CMC_X_COM_support** structures listing extensions the application requests the CMC implementation to support. The flag for the structures used on input is CMC_X_COM_SUP_EXCLUDE, defined for **CMC_X_COM_support**.

extension_flags
All flags used with CMC are valid. No further flags are defined.

Output Usage

item_data
Unchanged.

item_reference
The CMC implementation sets the flags in the structures to indicate support for the extension. The implementation does not set these flags if CMC_X_COM_SUP_EXCLUDE was set on input.

extension_flags
Unchanged.

Adds to

cmc_query_configuration, **cmc_logon**

Comments

Client applications can use this extension before establishing a session to get preliminary information about support before logging on. When an 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

[New - Windows 95]

CMC_X_COM_TIME_RECEIVED is a CMC extension that provides a **CMC_time** structure that holds the delivery time of a message.

Input Usage

CMC ignores this extension on input.

Output Usage

item_data
Zero.

item_reference
Pointer to the **CMC_time** structure that holds the time the message was received.

extension_flags
NULL.

Adds to

The **CMC_message** and **CMC_message_summary** structures

Comments

At logon, the CMC implementation passes the CMC_X_COM_TIME_RECEIVED extension in the **item_code** member of a **CMC_X_COM_support** structure to indicate that the 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_time** structure, **CMC_X_COM_support** structure

CMC_X_MS_ADDRESS_UI

[New - Windows 95]

CMC_X_MS_ADDRESS_UI is a CMC extension that adds options to the address-book dialog box.

Input Usage

item_data
Number of edit boxes in the dialog box.

item_reference
Pointer to an array of two **strings** that are the caption for the address-book dialog box To not provide a label, the client application sets the label string to NULL. and the label for the recipient box if there is only one recipient list.

extension_flags
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

[New - Windows 95]

CMC_X_MS_ATTACH_DATA is a CMC extension that holds 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.

item_reference
Unchanged.

extension_flags
All flags used with the **CMC_extension** structure are valid. No further flags are defined.

Output Usage

item_data
Bitmask of attachment flags. The messaging system 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.

item_reference
Unchanged.

extension_flags
Unchanged.

Adds to

The **CMC_attachment** structure

See Also

CMC_attachment structure, **CMC_extension** structure

CMC_X_MS_FUNCTION_FLAGS

[New - Windows 95]

CMC_X_MS_FUNCTION_FLAGS is a CMC extension that holds bitmasks 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
Indicates the user should not modify the address book.

CMC_X_MS_LIST_GUARANTEE_FIFO
Indicates the message store lists message identifiers in order by time received.

CMC_X_MS_READ_BODY_AS_FILE
Indicates the recipient should read the body of the message as a file.

CMC_X_MS_READ_ENV_ONLY
Indicates the recipient is to read the message envelope only.

item_reference
NULL.

extension_flags
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

[New - Windows 95]

CMC_X_MS_MESSAGE_DATA is a CMC extension that holds a bitmask of flags that provide extra data regarding 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
Pointer to the conversation thread.

extension_flags
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
Pointer to the location of the conversation thread.

extension_flags
Unchanged.

Adds to

The **CMC_message** structure

See Also

CMC_extension structure, **CMC_message** structure

CMC_X_MS_SESSION_FLAGS

[New - Windows 95]

CMC_X_MS_SESSION_FLAGS is a CMC extension that holds a bitmask of flags used to provide information about session logon and logoff.

Input Usage

item_data
Bitmask of logon and logoff flags. The following flags can be set:

CMC_X_MS_FORCE_DOWNLOAD
Indicates the message sender requests a forced download. This flag is only valid for logon.

CMC_X_MS_LOGOFF_UI
Indicates the message sender requests a logon dialog box.

item_reference
NULL.

extension_flags
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_list**, **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

CMC_XS_COM

[New - Windows 95]

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 message services but are not in the CMC base specification. For a full list of common extension declarations, see the CMC header file.

See Also

cmc_logon function, **cmc_query_configuration** function, CMC_X_COM_SUPPORT_EXT extension

CMC_XS_MS

[New - Windows 95]

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_EXT.

See Also

cmc_logon function, **cmc_query_configuration** function, CMC_X_COM_SUPPORT_EXT extension

Simple MAPI

Simple MAPI is a set of functions and related data structures that help you 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. This chapter describes the functions and structures for C and C++ applications; the next chapter describes the functions and structures for Visual Basic applications.

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.

Simple MAPI Functions for C and C++

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 **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.

MAPIAddress

[New - Windows 95]

The **MAPIAddress** function creates or modifies a set of address list entries.

Syntax

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 *lhSession* parameter is zero, MAPI logs on the user and creates a session that exists only for the duration of the call. The temporary session may 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 is displayed, it is application modal. If *ulUIParam* contains a parent window handle, it is of type HWND (cast to a ULONG). If no dialog box is displayed during the call, the *ulUIParam* parameter is ignored.

lpszCaption
Input parameter specifying either a pointer to the caption for the address list dialog box, NULL, or an empty string. When lpszCaption is NULL or an points to an empty string, the **MAPIAddress** function 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 *nEditFields* is 4, each recipient class supported by the underlying messaging system has an edit control.If the *nEditFields* parameter is set to zero, only address list browsing is allowed. Values of 1, 2, or 3 control the number of edit controls present. If *nEditFields* is 4, each recipient class supported by the underlying messaging system has an edit control.

However, if the number of recipient classes in the *lpRecips* array 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 *nEditFields* parameter is set to 1 and more than one kind of entry exists in *lpRecips*, then *lpszLabels* is ignored.

Entries selected for the different controls are differentiated by the *ulRecipClass* field in the returned recipient structure.

lpszLabels
Input parameter specifying a pointer to a string to be used as an edit control label in the address list dialog box. When *nEditFields* is set to any value other than 1, this parameter is ignored and should be NULL or point to an empty string. Also, if the caller requires the default control label "To," the *lpszLabels* parameter should be NULL or point to an empty string.

nRecips
Input parameter specifying the number of entries in the array indicated by *lpRecips.* If the *nRecips* parameter is zero, the *lpRecips* parameter is ignored.

lpRecips
Input parameter specifying a pointer 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 specifying a bitmask of option flags. The following flags can be set:

MAPI_LOGON_UI

Indicates that a dialog box should be displayed to prompt for logon if required. When the MAPI_LOGON_UI flag is not set, the application does not display a logon dialog box and returns an error 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, the function uses an existing shared session.

ulReserved

Reserved for future use; must be zero.

lpnNewRecips

Output parameter specifying a pointer to the number of entries in the *lppNewRecips* recipient output array. If the *lpnNewRecips* parameter is zero, the *lppNewRecips* parameter is ignored.

lppNewRecips

Output parameter specifying a pointer to an array of **MapiRecipDesc** structures containing the the final list of recipients. This array is allocated by **MAPIAddress**, can not 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

The following table return values are possible from the **MAPIAddress** function. If any error occurs, no memory was allocated and you do not need to call **MAPIFreeBuffer**.

MAPI_E_FAILURE

One or more unspecified errors occurred while addressing the message.No list of recipient entrieswas 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 function successfully return a list of recipient entries.

Comments

This 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 the **MAPIAddress** function 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. Clients must call the **MAPIFreeBuffer** function to free this memory after the **MAPIAddress** function returns.

See Also

MAPIFreeBuffer function
MAPILogon function
MapiRecipDesc structure

MAPIDeleteMail

[New - Windows 95]

The **MAPIDeleteMail** function deletes a message.

Syntax

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 is displayed, it is application modal. If *ulUIParam* contains a parent window handle, it is of type HWND (cast to a ULONG). If no dialog box is displayed during the call, the *ulUIParam* parameter 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 the **MAPIDeleteMail** function successfully returns.

flFlags
Reserved for future use; must be zero.

ulReserved
Reserved for future use; must be zero.

Return Values

The possible return values for the **MAPIDeleteMail** function and their meanings are as follows:

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 *lpszMessageID* parameter. No message was deleted.

MAPI_E_INVALID_SESSION
An invalid session handle was passed in for the *lhSession* parameter. No message was deleted.

SUCCESS_SUCCESS
The function successfully deleted the message.

Comments

To find the message to be deleted, call the **MAPIFindNext** function before calling **MAPIDeleteMail**. Because message identifiers are opaque, messaging system-specific, and can be invalidated at any time, the **MAPIDeleteMail** function considers a message identifier to be valid only for the current session. The **MAPIDeleteMail** function handles invalid message identifiers by returning the MAPI_E_INVALID_MESSAGE value.

See Also

MAPIFindNext function

MAPILogon function
MAPISaveMail function

MAPIDetails

[New - Windows 95]

The **MAPIDetails** function displays a dialog box containing the details of a selected address list entry.

Syntax

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 *lhSession* parameter is zero, MAPI logs on the user and creates a session that exists only for the duration of the call. The temporary session may 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 is displayed, it is application modal. If *ulUIParam* contains a parent window handle, it is of type HWND (cast to a ULONG). If no dialog box is displayed during the call, the *ulUIParam* parameter is ignored.

lpRecip
Input parameter specifying a pointer to the recipient for which details are to be displayed. The **MAPIDetails** function ignores all members of this **MapiRecipDesc** structure except *ulEIDSize* and *lpEntryID*. If *ulEIDSize* is non-zero, the **MAPIDetails** function resolves the recipient entry. If *ulEIDSize* is zero, the **MAPIDetails** function returns MAPI_E_AMBIGUOUS_RECIP.

flFlags
Input parameter specifying a bitmask of option flags. The following flags can be set:

MAPI_AB_NOMODIFY
Indicates the caller is requesting that the dialog be read-only, prohibiting changes. **MAPIDetails** may or may not honor the request.

MAPI_LOGON_UI
Indicates that a dialog box should be displayed to prompt for logon if required. When the MAPI_LOGON_UI flag is not set, the application does not display a logon dialog box and returns an error 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, the function uses an existing shared session.

ulReserved
Reserved for future use; must be zero.

ReturnValues

The possible return values for the **MAPIDetails** function and their meanings are as follows:

MAPI_E_AMBIGUOUS_RECIPIENT
The dialog box could not be displayed because the *ulEIDSize* member of 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 *lpRecips* 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 function successfully displayed the details dialog box.

Comments

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 may 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 *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

[New - Windows 95]

The **MAPIFindNext** function retrieves the next (or first) message identifier of a specified type of incoming message.

Syntax

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 is displayed, it is application modal. If *ulUIParam* contains a parent window handle, it is of type HWND (cast to a ULONG). If no dialog box is displayed during the call, the *ulUIParam* parameter is ignored.

lpszMessageType
Input parameter specifying a pointer to a string identifying the message class to search. To find an interpersonal message (IPM), specify NULL in *lpszMessageType* or have it point to an empty string. Messaging systems whose only supported message class is IPM may ignore this parameter.

lpszSeedMessageID
Input parameter specifying a pointer to a string containing the message identifier seed for the request. If the *lpszSeedMessageID* parameter is NULL or points to an empty string, the **MAPIFindNext** function retrieves the first message that matches the type specified in the *lpszMessageType* parameter.

flFlags
Input parameter specifying 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 may take longer if this flag is set. Some implementations cannot honor this request and return MAPI_E_NO_SUPPORT.

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.

Older versions of MAPI supported smaller message identifiers 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. When this flag is not set, **MAPIFindNext** can return any message of the specified type.

ulReserved
Reserved for future use; 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 *flFlags*. A smaller buffer will only be sufficient if the returned message

identifier will always be 64 characters or less.

Return Values

The possible return values for the **MAPIFindNext** function and their meanings are as follows:

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 for *lpszSeedMessageID*. No message was found.

MAPI_E_INVALID_SESSION
: An invalid session handle was passed in for *lhSession*. No message was found.

MAPI_E_NO_MESSAGES
: A matching message could not be found.

SUCCESS_SUCCESS
: **MAPIFindNext** successfully returned the message identifier.

Comments

The **MAPIFindNext** function allows a client application to enumerate messages of a given type. **MAPIFindNext** can be called repeatedly to list all messages in the folder. Message identifiers returned from the **MAPIFindNext** function can be used in other Simple MAPI calls to retrieve message contents and delete messages. **MAPIFindNext** is for processing incoming messages, not for managing received messages.

The **MAPIFindNext** function 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 **MAPILogon**.

When *lpszSeedMessageID* is NULL or points to an empty string, **MAPIFindNext** returns the message identifier for the first message of the type specified with *lpszMessageType.* When *lpszSeedMessageID* contains a valid identifier, **MAPIFindNext** returns the next matching message of the type specified with *lpszMessageType*. Repeated calls to **MAPIFindNext** ultimately result in a return of MAPI_E_NO_MESSAGES, 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 the *lpszMessageType* parameter) 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 with *lpszSeedMessageID* is invalid, **MAPIFindNext** returns MAPI_E_INVALID_MESSAGE.

See Also

MAPILogon function

MAPIFreeBuffer

[New - Windows 95]

The **MAPIFreeBuffer** function frees memory allocated by the messaging system.

Syntax

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

The possible return values for the **MAPIFreeBuffer** function and their meanings are as follows:

MAPI_E_FAILURE
One or more unspecified errors occurred and the memory could not be freed.

SUCCESS_SUCCESS
MAPIFreeBuffer successfully freed the memory.

See Also

MAPILogoff function

MAPILogoff

[New - Windows 95]

The **MAPILogoff** function ends a session with the messaging system.

Syntax

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 **MAPILogon** 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 is displayed, it is application modal. If *ulUIParam* contains a parent window handle, it is of type HWND (cast to a ULONG). If no dialog box is displayed during the call, the *ulUIParam* parameter is ignored.

flFlags
Reserved for future use; must be zero.

ulReserved
Reserved for future use; must be zero.

Return Values

The possible return values for the **MAPILogoff** function and their meanings are as follows:

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 *lhSession* parameter. The session was not terminated.

SUCCESS_SUCCESS
MAPILogoff successfully ended the session.

See Also

MAPILogon function

MAPILogon

[New - Windows 95]

The **MAPILogon** function begins a Simple MAPI session, loading the default message store and address book providers.

Syntax

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 is displayed, it is application modal. If *ulUIParam* contains a parent window handle, it is of type HWND (cast to a ULONG). If no dialog box is displayed during the call, the *ulUIParam* parameter is ignored.

lpszProfileName
Input parameter specifying a pointer 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, the **MAPILogon** function displays a logon dialog box with an empty name field.

lpszPassword
Input parameter specifying a pointer 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, *lpszPassword* should be NULL or point to an empty string. When the user must enter credentials, *flFlags* must be set to MAPI_LOGON_UI to allow a logon dialog box to be displayed.

flFlags
Input parameter specifying 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 may be downloaded in the background after the function call returns.

MAPI_LOGON_UI
Indicates that a logon dialog box should be displayed to prompt for logon information. If the user needs to provide information to enable a successful log on, 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, the function uses an existing shared session.

ulReserved
Reserved for future use; must be 0.

lplhSession
Output parameter specifying a Simple MAPI session handle.

Return Values

The possible return values for the **MAPILogon** function and their meanings are as follows:

MAPI_E_FAILURE
One or more unspecified errors occurred during sign-on. 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
    A Simple MAPI session was successfully established.

Comments

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 sign-in dialog box if the credentials presented fail to validate the session, set the *flFlags* parameter to MAPI_LOGON_UI.

Your application can test for an existing session by calling MAPILogon with a **NULL** value for *lpszProfileName*, a **NULL** value for *lpszPassword* and not setting the MAPI_LOGON_UI flag in *flFlags*. If there is an existing session, the call will succeed and will return a valid LHANDLE for the session. Otherwise, the call will fail.

See Also

MAPILogoff function

MAPIReadMail

[New - Windows 95]

The **MAPIReadMail** function retrieves a message for reading.

Syntax

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 is displayed, it is application modal. If *ulUIParam* contains a parent window handle, it is of type HWND (cast to a ULONG). If no dialog box is displayed during the call, the *ulUIParam* parameter is ignored.

lpszMessageID
Input parameter specifying a pointer to a message identifier string for the message to be read. The string is allocated by the caller.

flFlags
Input parameter specifying 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 or 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** structure. The function ignores this flag if the calling application has set the MAPI_ENVELOPE_ONLY flag. Setting the MAPI_SUPPRESS_ATTACH flag is a performance enhancement.

ulReserved
Reserved for future use; must be zero.

lppMessage
Output parameter specifying a pointer to the location where the message is written. Messages are written to a **MapiMessage** structure; which can be freed with a single call to **MAPIFreeBuffer**.

When neither MAPI_ENVELOPE_ONLY or MAPI_SUPPRESS_ATTACH is set, attachments are written to temporary files pointed to by the *lpFiles* **MapiMessage** structure member. It is the caller's responsibility to delete these files when they are no longer needed.

Return Values

The possible return values for the **MAPIReadMail** function and their meanings are as follows:

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 left 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 for the *lpszMessageID* parameter.

MAPI_E_INVALID_SESSION
  An invalid session handle was passed in for 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 message was successfully read.

Comments

MAPIReadMail 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 the function 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 **MAPIFindNext** 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, the **MAPIReadMail** function considers a message identifier to be valid only for the current MAPI session.

See Also

MAPIFreeBuffer function
MAPILogon function
MapiMessage structure

MAPIResolveName

[New - Windows 95]

The **MAPIResolveName** function transforms a message recipient's name as entered by a user to an unambiguous address list entry.

Syntax

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 *lhSession* parameter is zero, MAPI logs on the user and creates a session that exists only for the duration of the call. The temporary session may 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 is displayed, it is application modal. If *ulUIParam* contains a parent window handle, it is of type HWND (cast to a ULONG). If no dialog box is displayed during the call, the *ulUIParam* parameter is ignored.

lpszName
Input parameter specifying a pointer to the name to be resolved.

flFlags
Input parameter specifying a bitmask of option flags. The following flags can be set:

MAPI_AB_NOMODIFY
Indicates the caller is requesting that the dialog 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, the function returns the MAPI_E_AMBIGUOUS_RECIPIENT value.

MAPI_LOGON_UI
Indicates that a dialog box should be displayed to prompt for logon if required. When the MAPI_LOGON_UI flag is not set, the application does not display a logon dialog box and returns an error 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, the function uses an existing shared session.

ulReserved
Reserved for future use; must be zero.

lppRecip
Output parameter specifying a pointer 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 **MAPIFreeBuffer**.

Return Values

If **MAPIResolveName** returns an error, it is not necessary to deallocate memory with **MAPIFreeBuffer**. The possible return values for the **MAPIResolveName** function and their meanings are as follows:

MAPI_E_AMBIGUOUS_RECIPIENT

The recipient requested has not or could not be resolved to a unique address list entry.

MAPI_E_FAILURE
: One or more unspecified errors occurred and 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 name was successfully resolved.

Comments

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 the **MAPIRecipDesc** structure at some point by calling the **MAPIFreeBuffer** function**.**

See Also

MAPIFreeBuffer function
MAPILogon function
MapiRecipDesc structure

MAPISaveMail

[New - Windows 95]

The **MAPISaveMail** function saves a message into the message store.

Syntax

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 *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 *lhSession* is zero, MAPI logs on the user and creates a session that exists only for the duration of the call. The temporary session may 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 is displayed, it is application modal. If *ulUIParam* contains a parent window handle, it is of type HWND (cast to a ULONG). If no dialog box is displayed during the call, the *ulUIParam* parameter is ignored.

lpMessage
Input parameter specifying a pointer to a **MapiMessage** structure containing the contents of the message to be saved. The *lpOriginator* member is ignored. Applications can either ignor the *flFlags* member, or, if the message has never been saved, can set the MAPI_SENT and MAPI_UNREAD bits.

flFlags
Input parameter specifying a bitmask of option flags. The following flags can be set:

MAPI_LOGON_UI
Indicates that a dialog box should be displayed to prompt for logon if required. When the MAPI_LOGON_UI flag is not set, the application does not display a logon dialog box and returns an error if the user is not logged on. The **MAPISaveMail** function 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, the function uses an existing shared session.

ulReserved
Reserved for future use; must be 0.

lpszMessageID
Input-output parameter specifying a either a pointer to 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

The possible return values for the **MAPISaveMail** function and their meanings are as follows:

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 for the *lpszMessageID* parameter; no message was saved.

MAPI_E_INVALID_SESSION
: An invalid session handle was passed for 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
: **MAPISaveMail** saved the message successfully.

Comments

MAPISaveMail saves a message, optionally replacing an existing message. Before calling **MAPISaveMail**, use **MAPIFindNext** 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 the *lpszMessageID* parameter 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 message system does not support message storage, the **MAPISaveMail** function returns the MAPI_E_NOT_SUPPORTED value.

Because message identifiers are system-specific and opaque and can be invalidated at any time, the **MAPISaveMail** function considers a message identifier to be valid only for the current MAPI session. The **MAPISaveMail** function handles invalid message identifiers by returning the MAPI_E_INVALID_MESSAGE value.

See Also

MAPILogon function
MAPIMessage data type

MAPISendDocuments

[New - Windows 95]

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 **MAPISendMail** in that it allows less flexibility in message generation.

Syntax

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 is displayed, it is application modal. If *ulUIParam* contains a parent window handle, it is of type HWND (cast to a ULONG). If no dialog box is displayed during the call, the *ulUIParam* parameter is ignored.

lpszDelimChar
Input parameter specifying a pointer 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 specifying a pointer to a string containing a list of full pathnames (including drive letters) to attachment files. This list is formed by concatenating correctly formed filepaths 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 specifying a pointer 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, the **MAPISendDocuments** function sets the filenames set to the filename values indicated by *lpszFullPaths*.

ulReserved
Reserved for future use; must be zero.

Return Values

The possible return values for the **MAPISendDocuments** function and their meanings are as follows:

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
: **MAPISendDocuments** sent the message successfully.

Comments

The **MAPISendDocuments** function sends a standard message, always displaying a cover note dialog box so that the user can provide recipients and other sending options.The function tries to establish a session using the messaging system's shared session. If no shared session exists, it prompts for logon information to establish a session. Before the function returns, it closes the session.

Message attachments can include the active document or all the currently open documents in the client application that called **MAPISendDocuments**. The function is used primarily for calls from a macro or scripting language, often found in applications such as spreadsheet or word-processing programs.

The **MAPISendDocuments** function creates as many file attachments as there are paths specified by the *lpszFullPaths* parameter in spite of the fact that there may be different numbers of filepaths and filenames. The function caller is responsible for deleting temporary files created when using this function.

See Also

MAPISendMail function

MAPISendMail

[New - Windows 95]

The **MAPISendMail** function sends a message. **MAPISendMail** differs from **MAPISendDocuments** in that it allows greater flexibility in message generation.

Syntax

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 *lhSession* parameter is zero, MAPI logs on the user and creates a session that exists only for the duration of the call. The temporary session may 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 is displayed, it is application modal. If *ulUIParam* contains a parent window handle, it is of type HWND (cast to a ULONG). If no dialog box is displayed during the call, the *ulUIParam* parameter is ignored.

lpMessage
Input parameter specifying a pointer 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 ask for a read report. All other members are ignored and unused pointers should be NULL.

flFlags
Input parameter specifying 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 specifieid.

MAPI_LOGON_UI
Indicates that a dialog box should be displayed to prompt for logon if required. When the MAPI_LOGON_UI flag is not set, the application does not display a logon dialog box and returns an error if the user is not logged on. The **MAPISaveMail** function 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, the function uses an existing shared session.

ulReserved
Reserved for future use; must be zero.

Return Values

The possible return values for the **MAPISendMail** function and their meanings are as follows:

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 open; 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_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 to sent; the message was not 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
MAPISendMail successfully sent the message.

Comments

MAPISendMail 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 the **MAPISendMail** function does not necessarily imply recipient validation. The message may not have been sent to all recipients. Depending on the transport provider, recipient validation may be a lengthy process.

A NULL value for *lpMessage->lpszSubject* indicates that there is no text for the subject of the message and a NULL value for *lpMessage->lpszNoteText* indicates that there is no text for the body of the message. Some client applications may 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). The **MAPISendMail** function wraps lines as appropriate. If the text exceeds system limits, the function returns the MAPI_E_TEXT_TOO_LARGE value.

The *lpMessage->lpszMessageType* parameter member 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 may be limited in some messaging systems. If the limit is exceeded, the error MAPI_E_TOO_MANY_FILES is returned. If no files are specified, a pointer value of NULL should be assigned to the *lpFiles* member of the structure pointed to be *lpMessage*. File attachments are copied to the message before the function 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 body.

Some messaging systems may limit the number of recipients per message. A pointer value of NULL for *lpMessage->lpRecips* indicates no recipients. If your application passes a non-NULL value indicating a number of recipients exceeding the system limit, the **MAPISendMail** function returns MAPI_E_TOO_MANY_RECIPIENTS.

Note that *lpMessage->lpRecips* 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 may want to avoid name resolution. Such clients should set *lpMessage->lpRecips-> lpszAddress* to the custom address.

The **MAPISendMail** function does not require an originator-type recipient to send a message.

See Also

MAPILogon function
MAPIMessage data type
MapiRecipDesc structure

Simple MAPI Structures for C and C++

This section contains a reference entry for each structure.

MapiFileDesc

[New - Windows 95]

A **MapiFileDesc** structure holds information about 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..

Syntax

```
typedef struct {
     ULONG ulReserved;
     ULONG flFlags;
     ULONG nPosition;
     LPTSTR lpszPathName;
     LPTSTR lpszFileName;
     LPVOID lpFileType;
} MapiFileDesc, FAR *lpMapiFileDesc;
```

Members

ulReserved
Reserved; must be zero.

flFlags
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
Integer used to indicate where in the message body to render the attachment. Attachments replace the character found at a certain position in the message body. That is, attachments replace character in the **MapiMessage** structure field **lpszNoteText[nPosition]**. A value of - 1 (0xFFFFFFFF) means attachment position is not indicated; the application will have to provide a way for the user to access the attachment.

lpszPathName
Pointer to the fully qualified path of the attached file. This path should include the disk drive letter and directory name.

lpszFileName
Pointer 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
Pointer to the attachment filetype, which can be represented with a **MapiFileTagExt** structure. A value of NULL indicates an unknown filetype or a filetype determined by the operating system.

Comments

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 body. 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 what kind a particular attachment is. Embedded messages can be identified by a .MSG extension in the lpszFileName member.

OLE (object linking and embedding) object files are file representations of OLE object streams. You can re-create an OLE object from the file by calling the **OleLoadFromStream** function 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**, your application should not place two attachments in the same location. Applications may not display file attachments at positions beyond the end of the message text.

See Also

MapiFileTagExt structure¤

MapiFileTagExt

[New - Windows 95]

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.

Syntax

```
typedef struct {
     ULONG ulReserved;
     ULONG cbTag;
     LPBYTE lpTag;
     ULONG cbEncoding;
     LPBYTE lpEncoding
} MapiFileTagExt, FAR *lpMapiFileTagExt;
```

Members

ulReserved
Reserved; must be zero.

cbTag
Size, in bytes, of the value defined by the **lpTag** member.

lpTag
Pointer to an X.400 object identifier indicating the type of the attachment in its original form, for example "Microsoft Excel worksheet".

cbEncoding
Size, in bytes, of the value defined by the **lpEncoding** member.

lpEncoding
Pointer to an X.400 object identifier indicating the form in which the attachment is currently encoded, for example MacBinary, UUENCODE, or binary.

Comments

MapiFileTagExt 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. For more information on using MAPI with X.400 messaging, see Using MAPI with X.400 Message Systems.

See Also

MapiFileDesc structure¤

MapiMessage

[New - Windows 95]

A **MapiMessage** structure holds information about a message.

Syntax

```
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
Pointer 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
Pointer to a string containing the message text. If this member is empty or NULL, there is no message text.

lpszMessageType
Pointer 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 ignor the **lpszMessageType** member when reading messages and set it to empty or NULL when sending messages.

lpszDateReceived
Pointer 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
Pointer to a string identifying the conversation thread o which the message belongs. Some messaging systems may ignore and not return this member.

flFlags
Bitmask of message status flags. The following flags can be set:

MAPI_RECEIPT_REQUESTED
Indicates a receipt notification is requested. Client applications set this bit when sending a message.

MAPI_SENT
Indicates the message has been sent.

MAPI_UNREAD
Indicates the message has not been read.

lpOriginator
Pointer to a **MapiRecipDesc** structure holding information about the sender of the message.

nRecipCount
Number of message recipient structures in the array pointed to by the **lpRecips** member. A value of zero indicates no recipients are included.

lpRecips
Pointer to an array of **MapiRecipDesc** structures, each holding information about a message recipient.

nFileCount
Number of structures that describe file attachments in the array pointed to by the **lpFiles** member. A value of zero indicates no file attachments are included.

lpFiles
Pointer to an array of **MapiFileDesc** structures, each holding information about a file attachment.

See Also

MapiFileDesc structure, **MapiRecipDesc** structure¤

MapiRecipDesc

[New - Windows 95]

A **MapiRecipDesc** structure holds information about a message sender or recipient.

Syntax

```
typedef struct {
     ULONG ulReserved
     ULONG ulRecipClass;
     LPTSTR lpszName;
     LPTSTR lpszAddress;
     ULONG ulEIDSize;
     LPVOID lpEntryID;
} MapiRecipDesc, FAR *lpMapiRecipDesc;
```

Members

ulReserved
Reserved; must be zero.

ulRecipClass
Numeric value that indicates the type of recipient. Possible values for the **ulRecipClass** member 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
Pointer 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 i [*address type*][*e-mail address*]. Examples of valid addresses are FAX:206-555-1212 and SMTP:M@X.COM.

ulEIDSize
Size, in bytes, of the entry identifier pointed to by the **lpEntryID** member.

lpEntryID
Pointer 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.

Simple MAPI 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 run.
- 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. The following section describes these functions.

MAPIAddress

[New - Windows 95]

The **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.

Syntax

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 *Session* parameter is zero, MAPI logs on the user and creates a session that exists only for the duration of the call. The temporary session may 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 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 to 4are valid. If *nEditFields* is 4, each recipient class supported by the underlying messaging system has an edit control.If the *nEditFields* parameter is set to zero, only address list browsing is allowed. Values of 1, 2, or 3 control the number of edit controls present. If *nEditFields* is 4, each recipient class supported by the underlying messaging system has an edit control.

However, if the number of recipient classes in the *lpRecips* array 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 *nEditFields* parameter is set to 1 and more than one kind of entry exists in *lpRecips*, then *lpszLabels* is ignored.

Entries selected for the different controls are differentiated by the *ulRecipClass* field in the returned recipient structure.

Label
Input parameter specifying an edit control label in the address-list dialog box. This argument is ignored and should be an empty string except when the *EditFields* parameter is 1. If you want a default control label "To:", the *Label* parameter should be an empty string.

RecipCount
Input parameter specifying the number of entries in the *Recipients* parameter. If the *RecipCount*

parameter is zero, the *Recipients* parameter 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 *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 specifying a bitmask of flags. The following flags can be set:

MAPI_LOGON_UI
Indicates that a dialog box should be displayed to prompt for logon if required. When the MAPI_LOGON_UI flag is not set, the application does not display a logon dialog box and returns an error 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, the function uses an existing shared session.

Reserved
Reserved for future use;must be zero.

Return Values

The possible return values for the **MAPIAddress** function and their meanings are as follows:

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* parameterwas outside the range of 0 through 4. No list of recipients was returned.

MAPI_E_INVALID_RECIPIENTS
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 may have been returned.

MAPI_E_USER_ABORT
The user canceled one of the dialog boxes. No list of recipients was returned.

SUCCESS_SUCCESS
MAPIAddress successfully returned a list of address entries.

Comments

With the **MAPIAddress** function, users can 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.

MAPIDeleteMail

[New - Windows 95]

The **MAPIDeleteMail** function deletes a message.

Syntax

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 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** or **MAPISaveMail** functions return message identifiers.

Flags
Reserved for future use; must be zero.

Reserved
Reserved for future use;must be zero.

Return Values

The possible return values for the **MAPIDeleteMail** function and their meanings are as follows:

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 function successfully deleted the message.

Comments

To find the message to be deleted, call the **MAPIFindNext** function before calling **MAPIDeleteMail**.

MAPIDetails

[New - Windows 95]

The **MAPIDetails** function displays a dialog box containing the details of a selected address-list entry.

Syntax

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 *Session* parameter is zero, MAPI logs on the user and creates a session that exists only for the duration of the call. The temporary session may 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 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. The **MAPIDetails** function ignores all fields of the **MapiRecip** type except *EIDSize* and *EntryID*. If the field *EIDSize* is non-zero, the **MAPIDetails** function resolves the recipient entry. If *EIDSize* is zero, the MAPI_E_AMBIGUOUS_RECIPIENT value is returned.

Flags
Input parameter specifying a bitmask of flags. The following flags can be set:

MAPI_AB_NOMODIFY
Indicates the caller is requesting that the dialog be read-only, prohibiting changes. **MAPIDetails** may or may not honor the request.

MAPI_LOGON_UI
Indicates that a dialog box should be displayed to prompt for logon if required. When the MAPI_LOGON_UI flag is not set, the application does not display a logon dialog box and returns an error 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, the function uses an existing shared session.

Reserved
Reserved for future use; must be zero.

Return Values

The possible return values for the **MAPIDetails** function and their meanings are as follows:

MAPI_E_AMBIGUOUS_RECIPIENT
The recipient requested has not 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 *Recipient* 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 details 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 function successfully displayed the details dialog box.

Comments

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 may 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* field of the **MapiRecip** type is nonzero. Entries are resolved when they are returned by the **MAPIAddress** or **MAPIResolveName** functions and as the result being recipients of read mail.

MAPIFindNext

[New - Windows 95]

The **MAPIFindNext** function retrieves the next (or first) message identifier of a specified type of incoming message.

Syntax

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 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, the **MAPIFindNext** function retrieves the first message that matches the type specified in the *MessageType* parameter.

Flags
Input parameter specifying 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 may take longer if this flag is set. Some implementations cannot honor this request and return MAPI_E_NO_SUPPORT.

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, the function 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 for future use;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 will only be sufficient if the returned message identifier will always be 64 characters or

less.

Return Values

The possible return values for the **MAPIFindNext** function and their meanings are as follows:

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 for *SeedMessageID*. No message was found.

MAPI_E_INVALID_SESSION
: An invalid session handle was passed in for *lhSession*. No message was found.

MAPI_E_NO_MESSAGES
: A matching message could not be found.

SUCCESS_SUCCESS
: **MAPIFindNext** successfully returned the message identifier.

Comments

The **MAPIFindNext** function allows a client application to enumerate messages of a given type. **MAPIFindNext** can be called repeatedly to list all messages in the folder. Message identifiers returned from the **MAPIFindNext** function can be used in other Simple MAPI calls to retrieve message contents and delete messages. **MAPIFindNext** is for processing incoming messages, not for managing received messages.

When *SeedMessageID* is NULL or empty, **MAPIFindNext** returns the message identifier for the first message of the type specified with *MessageType.* When *SeedMessageID* contains a valid identifier, **MAPIFindNext** returns the next matching message of the type specified with *MessageType*. Repeated calls to **MAPIFindNext** ultimately result in a return of MAPI_E_NO_MESSAGES, 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 MAPI_E_INVALID_MESSAGE.

Message type matching is done against message class strings. All message types whose names match, up to the length specified in the *MessageType* parameter are returned.

MAPILogoff

[New - Windows 95]

The **MAPILogoff** function ends a session with the messaging system.

Syntax

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 is displayed, it is application modal. If no dialog box is displayed during the call, the *UIParam* parameter is ignored.

Flags
Reserved for future use;must be zero.

Reserved
Reserved for future use;must be zero.

Return Values

The possible return values for the **MAPILogoff** function and their meanings are as follows:

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
MAPILogoff successfully ended the session.

MAPILogon

[New - Windows 95]

The **MAPILogon** function begins a Simple MAPI session, loading the default message store and address book providers.

Syntax

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 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, the **MAPILogon** function 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, *Password* should be empty. When the user must enter credentials, *Flags* must be set to MAPI_LOGON_UI to allow a logon dialog box to be displayed.

Flags
Input parameter specifying 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 may be downloaded in the background after the function call returns.

MAPI_LOGON_UI
Indicates that a logon dialog box should be displayed to prompt for logon information. If the user needs to provide information to enable a successful log on, 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, the function uses an existing shared session.

Reserved
Reserved for future use; must be zero.

Session
Output parameter specifying a Simple MAPI session handle.

Return Values

The possible return values for the **MAPILogon** function and their meanings are as follows:

MAPI_E_FAILURE
: One or more unspecified errors occurred during sign-on. 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
: A session was successfully established.

Comments

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 sign-in dialog box if the credentials presented fail to validate the session, set the *Flags* parameter to MAPI_LOGON_UI.

MAPIReadMail

[New - Windows 95]

The **MAPIReadMail** function retrieves a message for reading.

Syntax

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 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 be obtained from the **MAPIFindNext** and **MAPISaveMail** functions.

Flags
Input parameter specifying 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 or 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** structure. The function ignores this flag if the calling application has set the MAPI_ENVELOPE_ONLY flag. Setting the MAPI_SUPPRESS_ATTACH flag is a performance enhancement.

Reserved
Reserved for future use;must be zero.

Message
Output parameter specifying a type set by the **MAPIReadMail** function 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 the **MAPIReadMail** function 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

The possible return values for the **MAPIReadMail** function and their meanings are as follows:

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 for the *MessageID* parameter.

MAPI_E_INVALID_SESSION
An invalid session handle was passed in for 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 message was successfully read.

Comments

The **MAPIReadMail** function returns one message, breaking the message content into the same parameters and types used in the **MAPISendMail** function. The **MAPIReadMail** function 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 the function 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 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.

MAPIResolveName

[New - Windows 95]

The **MAPIResolveName** function transforms a message recipient's name as entered by a user to an unambiguous address list entry.

Syntax

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 *Session* parameter is zero, MAPI logs on the user and creates a session that exists only for the duration of the call. The temporary session may 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 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 specifying a bitmask of option flags. The following flags can be set:

MAPI_AB_NOMODIFY
Indicates the caller is requesting that the dialog 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, the function returns the MAPI_E_AMBIGUOUS_RECIPIENT value.

MAPI_LOGON_UI
Indicates that a dialog box should be displayed to prompt for logon if required. When the MAPI_LOGON_UI flag is not set, the application does not display a logon dialog box and returns an error 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, the function uses an existing shared session.

Reserved
Reserved for future use;must be zero.

Recipient
Output parameter specifying a recipient-type set returned by the **MAPIResolveName** function 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

The possible return values for the **MAPIResolveName** function and their meanings are as follows:

MAPI_E_AMBIGUOUS_RECIPIENT
: The recipient requested has not or could not be resolved to a unique address list entry.

MAPI_E_FAILURE
: One or more unspecified errors occurred and 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 name was successfully resolved.

Comments

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.

MAPISaveMail

[New - Windows 95]

The **MAPISaveMail** function saves a message.

Syntax

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 *Session* parameter is zero, MAPI logs on the user and creates a session that exists only for the duration of the call. The temporary session may 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 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. Applications can either ignor the *Flags* member, or, if the message has never been saved, can set the MAPI_SENT and MAPI_UNREAD bits.

Recips
Input parameter specifying the first element in an array of recipients. When the *RecipCount* field 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* field 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 may be limited in some systems. If the limit is exceeded, the MAPI_E_TOO_MANY_FILES value is returned. When the *FileCount* field 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 body.

Flags
Input parameter specifying a bitmask of option flags. The following flags can be set:

MAPI_LOGON_UI

Indicates that a dialog box should be displayed to prompt for logon if required. When the MAPI_LOGON_UI flag is not set, the application does not display a logon dialog box and returns an error if the user is not logged on. The **MAPISaveMail** function 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, the function uses an existing shared session.

Reserved

Reserved for future use;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 the **MAPISaveMail** function, or a null string. If the *MessageID* parameter contains a valid message identifier, the message is overwritten. If the parameter contains a null string, a new message is created.

Return Values

The possible return values for the **MAPISaveMail** function and their meanings are as follows:

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 for the *MessageID* parameter; no message was saved.

MAPI_E_INVALID_SESSION

An invalid session handle was passed for 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

MAPISaveMail saved the message successfully.

Comments

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 the *MessageID* parameter 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 the *MessageID* parameter. New messages are saved in the folder appropriate for incoming messages of that class. The new message identifier is returned in the *MessageID* parameter on completion.

The *MessageID* parameter must be a variable-length string. The elements of the message identified by the *MessageID* parameter are replaced by the elements in the *Message* parameter. If the *MessageID* parameter is empty, a new message is created.

The Visual Basic **MAPISaveMail** function 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.

MAPISendDocuments

[New - Windows 95]

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.

Syntax

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 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 filepaths 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 this parameter are added to the message as file attachments. If this parameter 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, the **MAPISendDocuments** function sets the filenames set to the filename values indicated by *FullPaths*.

Reserved
Reserved for future use;must be zero.

Return Values

The possible return values for the **MAPISendDocuments** function and their meanings are as follows:

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_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
: **MAPISendDocuments** sent the message successfully.

Comments

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 the **MAPISendDocuments** function. The 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 this function 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 function caller is responsible for deleting temporary files created when using this function.

MAPISendMail

[New - Windows 95]

The **MAPISendMail** function sends a standard message.

Syntax

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 *Session* parameter is zero, MAPI logs on the user and creates a session that exists only for the duration of the call. The temporary session may 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 specifies 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 may 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 to ask for a read report.

Recips
Input parameter specifying the first element of an array of recipients. When the *RecipCount* field in the *Message* parameter is zero, this 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* field is specified, the **MAPISendMail** function 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 the **MAPISendMail** function 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 message MAPI_E_TOO_MANY_FILES is returned. When the *FileCount* field in the **MapiMessage** type pointed to by the *Message* parameter 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 body.

Flags

Input parameter specifying 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 the function should display a dialog box to prompt for log on. When this flag is not set, the function does not display a dialog box and returns a message if the user is not signed in.

MAPI_LOGON_UI
: Indicates that a dialog box should be displayed to prompt for logon if required. When the MAPI_LOGON_UI flag is not set, the application does not display a logon dialog box and returns an error 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, the function uses an existing shared session.

Reserved
: Reserved for future use; must be zero.

Return Values

The possible return values for the **MAPISendMail** function and their meanings are as follows:

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 open; 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_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 to sent; the message was not 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
: **MAPISendMail** successfully sent the message.

Comments

MAPISendMail sends a standard message, with or without any user interaction. If recipient names, file attachments, or message text are provided, the **MAPISendMail** function 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 the function returns; therefore, later changes to the files do not affect the contents of the message. The files must be closed when they are copied.

The Visual Basic **MAPISendMail** function 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 your application's operating system process.

Simple MAPI Data Types for Visual Basic

Visual Basic uses a different set of calling and programming conventions than C and C++ do. 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 NULLvalue.
- 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 following section describes the three Simple MAPI types for Visual Basic.

MapiFile

[New - Windows 95]

The **MapiFile** type contains file attachment information.

Syntax

```
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 for future use; 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 a static OLE object file attachment.

Position
An integer describing where the attachment should be placed in the message body. Attachments replace the character found at a certain position in the message body; 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 body.

PathName
The full path name of the attached file. The file should be closed before this call is made.

FileName
The filename seen by the recipient. This name can differ from the filename in *PathName* if temporary files are being used. If *FileName* 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 tells 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 "" for this parameter.

Comments

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 linking and embedding) object files are file representations of OLE object streams. You can re-create an OLE object from the file by calling the **OleLoadFromStream** function 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

[New - Windows 95]

The **MAPIMessage** type contains message information.

Syntax

```
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 for future use. This field must be 0.

Subject
The subject text, limited to 256 characters or less. (Messages saved with **MAPISaveMail** are not limited to 256 characters.) An empty string indicates no subject text.

NoteText
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
A message type string used by applications other than interpersonal electronic mail. An empty string indicates an interpersonal message (IPM) type.

DateReceived
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
A string indicating the conversation thread ID to which this message belongs.

Flags
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
A **MapiFile** type that describes the sender of the message.

RecipCount
A count of the recipient descriptor types. A value of 0 indicates that no recipients are included.

FileCount

A count of the file attachment descriptor types. A value of 0 indicates that no file attachments are included.

MapiRecip

[New - Windows 95]

The **MAPIRecip** type contains recipient information.

Syntax

```
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 for future use. This member must be zero.

RecipClass
Classifies the recipient of the message. (Messages can be sorted by recipient class.) This field can also contain information about the originator of an inbound message.

Name
The name of the recipient that is displayed by the messaging system.

Address
Provider-specific message delivery data. This can be used by the message system to identify custom recipients who are not in an address list.

EIDSize
The size (in bytes) of the data in *EntryID*.

EntryID
A string used by the messaging system to uniquely identify the recipient. Unlike the *Address* field, this data is opaque and is not printable. The messaging system returns valid *EntryIDs* for recipients or senders included in the address list.

Essential Extended MAPI Functions

This chapter contains a reference entry for each function.

ABProviderInit

[New - Windows 95]

Initializes an address book provider for operation.

Syntax

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 an instance of the address book provider's dynamic-link library (DLL) that MAPI used when it linked.

lpMalloc
: Input parameter pointing to a standard memory allocator.

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
: Reserved; do not use.

ulMAPIVer
: Input parameter containing the version of the service provider interface that MAPI.DLL uses. For the current version number, see the MAPISPI.H header file.

lpulProviderVer
: Input parameter pointing to the version of the service provider interface that the message store uses.

lppABProvider
: Output parameter pointing to a variable where the initialized address-book provider object returned is stored

Return Values

S_OK
: The call succeeded and has returned the expected value or values.

Comments

To initialize an address book provider, MAPI calls the function named **ABProviderInit**, based on the **ABPROVIDERINIT** function prototype defined in MAPISPI.H, from the address book provider's DLL. The address book provider must use its implementation of **ABProviderInit** to respond to the MAPI initialization call.

The address book provider must also define **ABProviderInit** 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 **APPROVIDERINIT** prototype to help define **ABProviderInit** as CDECL. The **APPROVIDERINIT** prototype has a standard MAPI initialization call type, STDMAPIINITCALLTYPE.

The input parameters *lpAllocateBuffer*, *lpAllocateMore*, and *lpFreeBuffer* point to the **MAPIAllocateBuffer**, **MAPIAllocateMore**, and **MAPIFreeBuffer** functions, respectively, for use by the address-book 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 client applications in calling object interfaces. For example, two provider interface methods that typically allocate memory are **IMAPIProp::GetProps** and **IMAPITable::QueryRows**.

The address book 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 on using **ABProviderInit**, see the information on using the **MSProviderInit**, **ABProviderInit**, and **XPProviderInit** functions in *MAPI Programmer's Guide*.

The **ABProviderInit** function is defined in MAPISPI.H.

See Also

HPProviderInit function, **IMAPIProp::GetProps** method, **IMAPITable::QueryRows** method, **MAPIAllocateBuffer** function, **MAPIAllocateMore** function, **MAPIFreeBuffer** function, **MSProviderInit** function, **XPProviderInit** function

BuildDisplayTable

[New - Windows 95]

Creates a display table from the property page data contained in a **DTPAGE** structure.

Syntax

BuildDisplayTable(LPALLOCATEBUFFER *lpAllocateBuffer*, **LPALLOCATEMORE** *lpAllocateMore*, **LPFREEBUFFER** *lpFreeBuffer*, **LPVOID** *lpvReserved*, **HINSTANCE** *hInstance*, **UINT** *cPages*, **LPDTPAGE** *lpPage*, **ULONG** *ulFlags*, **LPMAPITABLE*** *lppTable*, **LPTABLEDATA*** *lppTblData***)**

Parameters

lpAllocateBuffer
Input parameters 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 the **BuildDisplayTable** function 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 hold 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 8-bit 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**.

Comments

The input parameters *lpAllocateBuffer*, *lpAllocateMore*, and *lpFreeBuffer* point to the **MAPIAllocateBuffer**, **MAPIAllocateMore**, and **MAPIFreeBuffer** functions, respectively. If a client application calls **BuildDisplayTable**, it passes in these parameters 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.

The **BuildDisplayTable** function is defined in MAPIUTIL.H.

See Also

DTPAGE structure, **IMAPISupport::GetMemAllocRoutines** method, **IMAPITable : IUnknown** interface, **ITableData : IUnknown** interface, **MAPIAllocateBuffer** function, **MAPIAllocateMore** function, **MAPIFreeBuffer** function

CheckParameters

[New - Windows 95]

Checks debugging parameters on methods called by MAPI.

Syntax

HRESULT CheckParameters(METHODS *eMethod*, **LPVOID** *First*)

Parameters

eMethod
Specifies the method to validate.

First
Specifies the address of the beginning of the method.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

Comments

In contrast to the **ValidateParameters** function, **CheckParameters** does not perform a full parameter validation. Parameters passed between MAPI and providers are assumed to be correct; therefore this function performs a debug validation only.

For more information on parameter validation, see *MAPI Programmer's Guide*.

The **CheckParameters** function is defined in MAPIVAL.H

See Also

ulValidateParameters function, **ValidateParameters** function

CreateIProp

[New - Windows 95]

Creates a property data object (that is, an IPropData object).

Syntax

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.

Comments

The input parameters *lpAllocateBuffer*, *lpAllocateMore*, and *lpFreeBuffer* point to the **MAPIAllocateBuffer**, **MAPIAllocateMore**, and **MAPIFreeBuffer** functions, respectively. If a client application calls **CreateIProp**, 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.

The **CreateIProp** function is defined in MAPIUTIL.H.

See Also

Chapter 5, "Extended MAPI Properties," **IMAPISupport::GetMemAllocRoutines** method, **IPropData : IMAPIProp** interface, **MAPIAllocateBuffer** function, **MAPIAllocateMore** function, **MAPIFreeBuffer** function

CreateTable

[New - Windows 95]

Creates a table data object (that is, an ITableData object) that can be used to create table views.

Syntax

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 the 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_SNAPSHOT
The table is static and the contents do not change when 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.

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.

Comments

The input parameters *lpAllocateBuffer*, *lpAllocateMore*, and *lpFreeBuffer* 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.

The **CreateTable** function is defined in MAPIUTIL.H.

See Also

IMAPISupport::GetMemAllocRoutines method, **IMAPITable::GetStatus** method, **IMAPITable : IUnknown** interface, **MAPIAllocateBuffer** function, **MAPIAllocateMore** function, **MAPIFreeBuffer** function, **SPropTagArray** structure

FreePadrlist

[New - Windows 95]

Destroys an **ADRLIST** structure and frees associated memory, including memory allocated for all member structures.

Syntax

void FreePadrlist(LPADRLIST *padrlist***)**

Parameters

padrlist
Input parameter pointing to the **ADRLIST** structure to be destroyed.

Comments

Usually, MAPI assigns the **MAPIAllocateBuffer** function as the function to free memory, but it can designate any other comparable function if necessary.

The **FreePadrlist** function is defined in MAPIUTIL.H.

See Also

ADRLIST structure, **MAPIAllocateBuffer** function

FreeProws

[New - Windows 95]

Destroys an **SRowSet** structure and frees associated memory, including memory allocated for all member structures.

Syntax

void FreeProws(LPSRowSet *prows***)**

Parameters

prows
Input parameter pointing to the **SRowSet** structure to be destroyed.

Comments

Usually, MAPI assigns the **MAPIAllocateBuffer** function as the function to free memory, but it can designate any other comparable function if necessary.

The **FreeProws** function is defined in MAPIUTIL.H.

See Also

MAPIAllocateBuffer function, **SRowSet** structure

GetAttribIMsgOnIStg

[New - Windows 95]

Retrieves the attributes of the properties of a particular object.

Syntax

GetAttribIMsgOnIStg(LPVOID *lpObject*, **LPSPropTagArray** *lpPropTagArray*, **LPSPropAttrArray FAR*** *lppPropAttrArray***)**

Parameters

lpObject
Input parameter 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 holds 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.

Comments

MAPI provides this function because there is no method of the **IMAPIProp** interface that enables the retrieval of attributes.

The **GetAttribIMsgOnIStg** function is defined in IMESSAGE.H.

See Also

IMAPIProp : IUnknown interface, **SetAttribIMsgOnIStg** function, **SPropAttrArray** structure, **SPropTagArray** structure

HPProviderInit

[New - Windows 95]

Initializes a message hook provider for operation.

Syntax

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. The **HPProviderInit** function 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.

lppSpoolerHook
Output parameter pointing to a variable where the initialized message-hook provider returned is stored.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

Comments

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** prototype to help define **HPProviderInit** as CDECL. The **HPPROVIDERINIT** prototype has a standard MAPI initialization call type, STDMAPIINITCALLTYPE.

In some respects, a message hook provider resembles a client application extension to the MAPI spooler more closely than it resembles a service provider, because it is an active process. The MAPI spooler, the process that sends and delivers messages through transport providers, is the only process not driven directly by client application actions.

Another difference between the message hook provider and other service providers is that **HPProviderInit** creates a session object. In contrast, other providers' initialization functions usually create support objects.

The input parameters *lpAllocateBuffer*, *lpAllocateMore*, and *lpFreeBuffer* point to the **MAPIAllocateBuffer**, **MAPIAllocateMore**, and **MAPIFreeBuffer** functions, respectively, for use by the message-hook 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 client applications in calling object interfaces. For example, two provider interface methods that typically allocate memory are **IMAPIProp::GetProps** and **IMAPITable::QueryRows**.

The message hook provider should retain information on the allocator pointers passed to it in *lpAllocateBuffer*, *lpAllocateMore*, and *lpFreeBuffer*. If the message hook provider will use a memory allocator later, it should call the **IUnknown::AddRef** method for the allocation object pointed to by the *lpMalloc* parameter.

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 **ABProviderInit**, see the information on using the **MSProviderInit**, **ABProviderInit**, and **XPProviderInit** functions in *MAPI Programmer's Guide*.

The **GetAttribIMsgOnIStg** function is defined in MAPIHOOK.H.

See Also

ABProviderInit function, **IMAPIProp::GetProps** method, **IMAPISession : IUnknown** interface, **IMAPISupport::OpenProfileSection** method, **IMAPITable::QueryRows** method, **MAPIAllocateBuffer** function, **MAPIAllocateMore** function, **MAPIFreeBuffer** function, **MSProviderInit** function, **XPProviderInit** function

HrAddColumnsEx

[New - Windows 95]

Adds or moves columns to the beginning of an existing table.

Syntax

HRESULT HrAddColumnsEx(LPMAPITABLE *lptbl*, **LPSPropTagArray** *lpproptagColumnsNew*, **LPALLOCATEBUFFER** *lpAllocateBuffer*, **LPFREEBUFFER** *lpFreeBuffer*, **void (FAR*** *lpfnFilterColumns*) **(LPSPropTagArray** *ptaga***)**

Parameters

lptbl
: Input parameter pointing to the MAPI table affected.

lpproptagColumnsNew
: Input parameter holding an **SPropTagArray** structure containing an array of property tags indicating the properties to be added or moved to the beginning of the table.

lpAllocateBuffer
: Input parameter pointing to the **MAPIAllocateBuffer** function, to be used to allocate memory.

lpFreeBuffer
: Input parameter pointing to the **MAPIFreeBuffer** function, to be used to free memory.

lpfnFilterColumns
: Input parameter pointing to a callback function provided by the caller. If the *lpfnFilterColumns* parameter is set to NULL, no callback is made.

Return Values

S_OK
: The call succeeded and the specified columns were moved or added.

Comments

HrAddColumnsEx differs from **HrAddColumns** in that it allows the caller to filter the original property tags. For example, the caller might want to use **HrAddColumnsEx** to convert strings from property type PT_UNICODE to PT_STRING8.

The properties passed to this function 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 input parameters *lpAllocateBuffer* and *lpFreeBuffer* point to the **MAPIAllocateBuffer** and **MAPIFreeBuffer** functions, respectively. If a client application calls the **HrAddColumnsEx** function, 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.

The **HrAddColumnsEx** function is defined in MAPIUTIL.H.

See Also

IMAPISupport::GetMemAllocRoutines method, **IMAPITable::QueryColumns** method, **IMAPITable::QueryRows** method, **IMAPITable::SetColumns** method, **MAPIAllocateBuffer** function, **MAPIFreeBuffer** function, **SPropTagArray** structure

HrIStorageFromStream

[New - Windows 95]

Layers an **IStorage** interface onto an **IStream** object.

Syntax

HRESULT HrIStorageFromStream(**LPUNKNOWN** *lpUnkIn*, **PIID** *lpInterface*, **ULONG** *ulFlags*, **LPSTORAGE FAR*** *lppStorageOut*)

Parameters

lpUnkIn
Input parameter pointing to the **IUnknown** object for the object that implements the stream object.

lpInterface
Input parameter pointing to the interface identifier for the object in the *lpUnkIn* parameter. The **IID_IStream** interface identifier should be used.

ulFlags
Reserved; must be zero.

lppStorageOut
Output parameter pointing to a variable where the pointer to the returned **IStorage** object is stored.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

Comments

Store providers support **HrIStorageFromStream** using the **IStorage** interface for attachments. Store providers must implement the **IStream** interface. This function provides the **IStorage** interface for the **IStream** object.

HrIStorageFromStream is defined in MAPIUTIL.H.

HrQueryAllRows

[New - Windows 95]

Retrieves all rows of a table.

Syntax

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, the default set of columns for the table is retrieved.

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 this value is zero, no limit on the number of rows retrieved is set.

pprows
Output parameter pointing to the returned **SRowSet** structure, which contains the retrieved table rows.

Comments

When querying all rows of a table, your client application or service provider should call **HrQueryAllRows** instead of the **IMAPITable::QueryRows** method.

The **HrQueryAllRows** function is defined in MAPIUTIL.H

See Also

IMAPITable : IUnknown interface, **IMAPITable::QueryRows** method, **SPropTagArray** structure, **SRestriction** structure, **SRowSet** structure, **SSortOrderSet** structure

HrThisThreadAdviseSink

[New - Windows 95]

Creates an advise sink that wraps another advise sink.

Syntax

HrThisThreadAdviseSink(**LPMAPIADVISESINK** *lpAdviseSink*, **LPMAPIADVISESINK FAR*** *lppAdviseSink*)

Parameters

lpAdviseSink
Input parameter pointing to the advise sink to be wrapped.

lppAdviseSink
Output parameter pointing to a new advise sink that wraps the one pointed to by the *lpAdviseSink* parameter.

Comments

The purpose of the wrapper is to ensure that the original advise sink is called on the same thread that called the **HrThisThreadAdviseSink** function. It is used to protect notification callbacks that must run on a particular thread when OLE is used. Remoted OLE interfaces must be called on their own threads, and each OLE thread must have a message loop.

Applications should use **HrThisThreadAdviseSink** to restrict when notifications are generated (that is, when calls are made to the **IMAPIAdviseSink::OnNotify** method of the advise sink object passed by the client application in a previous **Advise** call). If notifications can be generated arbitrarily, a notification implementation might make an application 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, an application is difficult to thoroughly test and is prone to error.

By implementing **HrThisThreadAdviseSink**, service providers can ensure that calls to clients' **OnNotify** methods can avoid causing inappropriate multithreading by occurring at only the following 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 *MAPI Programmer's Guide*. For more information on the **Advise** method, see the reference entries under the **IABLogon**, **IAddrBook**, **IMAPIForm**, **IMAPISession**, **IMAPITable**, **IMsgStore**, and **IMSLogon** interfaces.

The **HrThisThreadAdviseSink** function is defined in MAPIUTIL.H.

See Also

IMAPIAdviseSink::OnNotify method

LaunchWizardEntry

[New - Windows 95]

Starts the Profile Wizard application.

Syntax

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, *hParentWnd* 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. MAPI_PW_PROVIDER_UI_ONLY can only be set if MAPI_PW_ADD_SERVICE_ONLY is set.

lppszServiceNameToAdd
Input parameter specifying a pointer to a string containing the service name to be added, if *ulFlags* is set to MAPI_PW_ADD_SERVICE_ONLY.

cbBufferMax
Input parameter specifying the size of the buffer pointed to by *lpszNewProfileName.*

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.

Comments

LaunchWizardEntry is the entry point into the MAPIef 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 code 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.

The **LaunchWizard** function is defined in MAPIWZ.H.

See Also

MSGSERVICEENTRY function **SERVICEWIZARDENTRY** function

MAPIAdminProfiles

[New - Windows 95]

Creates a profile administration object.

Syntax

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 8-bit format.

lppProfAdmin
Output parameter pointing to a variable where a pointer to the returned new profile administration object is stored.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

Comments

The **MAPIAdminProfiles** function is defined in MAPIX.H.

See Also

IProfAdmin::CreateProfile method

MAPIAllocateBuffer

[New - Windows 95]

Allocates a memory buffer.

Syntax

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.

Comments

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**, and any buffers linked to it by calling the **MAPIAllocateMore** function, when the memory is no longer needed.

The **MAPIAllocateBuffer** function is defined in MAPIX.H

See Also

MAPIAllocateMore function, **MAPIFreeBuffer** function

MAPIAllocateMore

[New - Windows 95]

Allocates a memory buffer that is linked to another buffer previously allocated with the **MAPIAllocateBuffer** function.

Syntax

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.

Comments

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** allows **MAPIFreeBuffer** to release both buffers on a single call.

The **MAPIAllocateMore** function is defined in MAPIX.H

See Also

MAPIAllocateBuffer function, **MAPIFreeBuffer** function

MAPIFreeBuffer

[New - Windows 95]

Frees a memory buffer allocated with a call to the **MAPIAllocateBuffer** function or the **MAPIAllocateMore** function.

Syntax

ULONG MAPIFreeBuffer(LPVOID *lpBuffer***)**

Parameters

lpBuffer
Input parameter pointing to a previously allocated memory buffer. If NULL is passed in the *lpBuffer* parameter, the **MAPIFreeBuffer** function does nothing.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

Comments

Usually, when a client or 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 application 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 the buffer. Simply calling the **MAPILogoff** function at the end of a MAPI session does not automatically release memory buffers.

A client application or service provider should work 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 it 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 Extended MAPI. For Simple MAPI, it is an entry point function.

The **MAPIFreeBuffer** function is defined in MAPIX.H

See Also

IMAPISupport::GetMemAllocRoutines method, **MAPIAllocateBuffer** function, **MAPIAllocateMore** function, **MAPILogoff** function

MAPIGetDefaultMalloc

[New - Windows 95]

Retrieves the address of the default MAPI memory allocation function.

Syntax

LPMALLOC MAPIGetDefaultMalloc( )

Parameters

None.

Comments

MAPIGetDefaultMalloc is defined in MAPIUTIL.H

Return Values

This function returns a pointer to the default MAPI memory allocation function.

MAPIInitialize

[New - Windows 95]

Increments the reference count and initializes per-instance global data for the Extended MAPI DLL.

Syntax

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 call succeeded and has returned the expected value or values.

Comments

The **MAPIInitialize** function increments the MAPI reference count for the MAPI DLL, 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.

Your client application or service provider must call **MAPIInitialize** before making any other Extended MAPI call. Failure to do so causes client application or service provider calls to return MAPI_E_NOT_INITIALIZED.

MAPIInitialize is similar to the OLE **CoInitialize** function. One of the main things **CoInitialize** does is set up the OLE task allocator. Clients and providers that are OLE-aware should call the **CoInitialize** or **OleInitialize** function before calling **MAPIInitialize** to use any OLE task allocator other than the default. For more information on OLE programming, see *Inside OLE, Second Edition*, by Kraig Brockschmidt, and *OLE Programmer's Reference, Volume One* and *Volume Two.*

When calling **MAPIInitialize** from a multi-threaded application, set *lpMapiInit* 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.

The **MAPIInitialize** function is defined in MAPIX.H.

See Also

MAPIUninitialize function

MAPILogonEx

[New - Windows 95]

Logs a client application onto a session with the message system.

Syntax

ULONG MAPILogonEx(ULONG *ulUIParam*, **LPTSTR** *lpszProfileName*, **LPTSTR** *lpszPassword*, **FLAGS** *flFlags*, **ULONG** *ulReserved*, **LPLHANDLE** *lplhSession*)

Parameters

ulUIParam
Input parameter containing the handle to the window that the logon dialog box is modal to. If no dialog box is displayed during the call, the *ulUIParam* parameter is ignored.

lpszProfileName
Input parameter pointing to a string containing the name of the profile to use when logging on. This string is typically limited to 256 characters. If the *lpszProfileName* parameter is NULL or points to an empty string, and the *flFlags* parameter is set to the MAPI_LOGON_UI flag, the **MAPILogonEx** function generates a logon dialog box with an empty field for the profile name.

lpszPassword
Input parameter pointing to a string containing the password of the profile. This string is typically limited to 256 characters. The *lpszPassword* parameter can be NULL whether or not *lpszProfileName* is NULL.

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 applications to acquire the session without providing any user credentials.

MAPI_EXPLICIT_PROFILE
Indicates the messaging subsystem should substitute the profile name of the default profile for *lpszProfileName*. The MAPI_EXPLICIT_PROFILE flag is ignored unless *lpszProfileName* is NULL or empty.

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 application does not display a logon dialog box and returns an error if the user is not logged on. MAPI_LOGON_UI cannot be set if the MAPI_PASSWORD_UI flag is set because the calling client can only present one of the two dialog boxes.

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, the function uses an existing shared session.

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. A calling client sets this flag when either configuration work must be done or the client is browsing the available message stores.

MAPI_NT_SERVICE

Indicates that a Windows NT client is logging on, signaling that MAPI should operate within a Windows NT service. This flag is only valid for Windows NT.

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.

MAPI_SERVICE_UI_ALWAYS

Indicates that **MAPILogonEx** should display a configuration dialog box for each message service in the profile. The dialog box should be displayed after the profile has been chosen, but before any message service is logged on. The MAPI common dialog box for logon contains a check box that sets this flag.

MAPI_USE_DEFAULT

Indicates that the default profile is used for logon.

ulReserved

Reserved; must be zero.

lplhsession

Output parameter pointing to a handle indicating a MAPI session.

Return Values

MAPI_E_FAILONEPROVIDER

The service provider had not previously been configured and attempts to configure the provider failed.

MAPI_E_STRING_TOO_LONG

The string is longer than the maximum allowable length.

MAPI_E_TOO_MANY_SESSIONS

The user had too many sessions open simultaneously. No session handle was returned.

MAPI_E_UNCONFIGURED

A service provider has not been configured, and therefore the operation did not complete.

MAPI_E_USER_CANCEL

The user canceled the operation, typically by choosing the Cancel button in a dialog box.

Comments

Extended MAPI client applications call the **MAPILogonEx** function to log onto a session with the messaging system. **MAPILogonEx** does not start any service providers; it is up to the client to perform those calls. All strings passed 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 process.

The **MAPILogonEx** function is defined in MAPIX.H.

See Also

IMAPISession::GetMsgStoresTable method, **IMAPISession::OpenMsgStore** method

MAPIOpenFormMgr

[New - Windows 95]

Opens an **IMAPIFormMgr** interface on a form-registry provider object in the context of an existing session.

Syntax

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.

Comments

After a client application makes a call to the **MAPIOpenFormMgr** function, most subsequent forms-related interactions take place through the form registry provider or an interface returned by the form registry provider. Among other things, the **IMAPIFormMgr** interface allows the client to work with message handlers and perform resolutions between message classes and form registries.

The **MAPIOpenFormMgr** function is defined in MAPIFORM.H.

See Also

IMAPIFormMgr : IUnknown interface

MAPIOpenLocalFormContainer

[New - Windows 95]

Returns an interface pointer to the local form registry.

Syntax

MAPIOpenLocalFormContainer(LPMAPIFORMCONTAINER FAR* *ppfcnt***)**

Parameters

ppfcnt
Output parameter pointing to a pointer to the local form registry interface.

Comments

The interface to which a pointer is returned can be used by third-party installation programs to install application-specific forms into the registry without the programs' first having to log onto MAPI.

The **MAPIOpenLocalFormContainer** function is defined in MAPIFORM.H.

MAPIUninitialize

[New - Windows 95]

Decrements the reference count, cleans up, and deletes per-instance global data for the Extended MAPI DLL.

Syntax

void MAPIUninitialize(   )

Comments

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 Extended MAPI calls can be made by the client application.

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.

The **MAPIUninitialize** function is defined in MAPIX.H.

See Also

MAPIInitialize function

MSProviderInit

[New - Windows 95]

Initializes a message store provider for operation..

Syntax

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 standard memory allocator.

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.

ulFlagslpFreeBuffer
Input parameter pointing to the **MAPIFreeBuffer** function, to be used to free memory.

Reserved; do not use. If any flags are set, the **MSProviderInit** function fails and returns the value MAPI_E_UNKNOWN_FLAGS.

ulMAPIVer
Input parameter containing the version of the service provider interface that MAPI.DLL uses. For the current version number, see the MAPISPI.H header file.

lpulProviderVer
Input parameter pointing to the version of the service provider interface that the message store uses.

lppMSProvider
Output parameter pointing to a variable where the initialized message-store provider object returned is stored.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

Comments

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 is to help define **MSProviderInit** as CDECL. The **MSPROVIDERINIT** prototype has a standard MAPI initialization call type, STDMAPIINITCALLTYPE.

The input parameters *lpAllocateBuffer, lpAllocateMore and lpFreeBuffer* 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 client applications 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 **IUnknown::AddRef** method 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 *MAPI Programmer's Guide.*

The **MSProviderInit** function is defined in MAPISPI.H.

See Also

ABProviderInit function, **HPProviderInit** function, **IMAPIProp::GetProps** method, **IMAPITable::QueryRows** method, **MAPIAllocateBuffer** function, **MAPIAllocateMore** function, **MAPIFreeBuffer** function, **XPProviderInit** function

OpenIMsgOnIStg

[New - Windows 95]

Creates internal memory structures and an object handle for creation of a new message object.

Syntax

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 provider to allocate additional memory where required.

lpFreeBuffer
Input parameter pointing to the **MAPIFreeBuffer** function, to be used by the provider to free memory.

lpMalloc
Input parameter pointing to a standard OLE memory allocator, as described in *OLE Programmer's Reference, Volume One.*

lpMapiSup
Input parameter pointing to an optional MAPI support object used when a service provider calls the **OpenIMsgOnIStg** function.

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 *lpfMsgCallRelease*.

ulFlags
Input parameter containing a bitmask of flags used to control whether the OLE method **IStorage::Commit** is called when the client 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 **IMessage::SaveChanges**.

lppMsg
Output parameter pointing to a variable where the returned message object is stored.

Return Values

S_OK

The call succeeded and has returned the expected value or values.

Comments

Warning The correct process for defining an attachment to a message is to call the **IMAPIProp::OpenProperty** method with a source interface of **IMessage**. **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 application 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 input parameters *lpAllocateBuffer*, *lpAllocateMore*, and *lpFreeBuffer* 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 client applications 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** 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 the *lpfMsgCallRelease* parameter 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.

The **OpenIMsgOnIStg** function is defined in IMESSAGE.H.

See Also

IMessage : IMAPIProp interface, **IMessage::ModifyRecipients** method, **IMessage::OpenAttach** method, **IMAPIProp::OpenProperty** method, **MAPIAllocateBuffer** function, **MAPIAllocateMore** function, **MAPIFreeBuffer** function, **MSGCALLRELEASE** function prototype, **OpenIMsgSession** function

OpenIMsgOnIStg, **MSGCALLRELEASE**

[New - Windows 95]

The MSGCALLRELEASE prototype function represents a callback function that enables a service provider to free the **IStorage** interface after the final release of a top-level message that was opened with **OpenIMsgOnIStg**.

Syntax

void (ULONG *ulCallerData*,   **LPMESSAGE** *lpMessage***)**

Parameters

ulCallerData
  An input parameter that contains calling application information about the **IMessage** interface.

lpMessage
  An input parameter that is a pointer 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.

Comments

The **MSGCALLRELEASE** function prototype is defined in IMESSAGE.H.

See Also

OpenIMsgOnIStg function

MSGSERVICEENTRY

[New - Windows 95]

The **MSGSERVICEENTRY** prototype function represents an optional service provider entry point to support message service configuration. MAPI calls this entry point from the Control Panel application.

Syntax

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 specifying a pointer to a standard OLE memory allocator, as described in the OLE 2 documentation.

lpMAPISup
Input parameter specifying a pointer to an **IMAPISupport** 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, *ulUIParam* 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 8-bit 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 SERVICE_UI_ALLOWED 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 UI_SERVICE_ALWAYS 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 display a configuration dialog box. If UI_SERVICE_ALWAYS is not set, a configuration dialog may still be displayed if SERVICE_UI_ALLOWED 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 *lpProps* 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 goes away and then return with a success code. 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 *lpProps*. The *cValues* parameter is zero if MAPI is passing no property values.

lpProps
: Input parameter specifying a pointer 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 *ulContext* 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 *ulFlags*, *lpProps* must be zero.

lpProviderAdmin
: Input parameter specifying a pointer to an **IProviderAdmin** interface that the function can use to locate profile sections for a specific provider in the current message service.

lppMapiError
: Output parameter specifying a pointer to a MAPIERROR structure. The structure is allocated with **MAPIAllocateBuffer**. All members are optional, although most structures will contain a valid error message string in the *lpszError* member. If the *lpszComponent* and/or *lpszError* members of the structure are present, their memory must eventually be freed by a single **MAPIFreeBuffer** call 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.

Comments

A function defined using the **MSGSERVICEENTRY** prototype allows 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 *lpProps*. Programmatic configuration is optional unless the service supports the Profile Wizard. Support for the Profile Wizard requires programmatic configuration.

The MSGSERVICEENTRY function is typically called in response to a user's request either from the client application or the Profile Provider. 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 your 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 may be named **ServiceEntry**.

MAPI allows your message service to use Windows 95-style property sheets for its configuration dialog boxes. It 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 service entry 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 codes listed. MAPI forwards this code when responding to a client application 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.

The **MSGSERVICEENTRY** function prototype is defined in MAPISPI.H.

See Also

BuildDisplayTable function, **IMAPISupport::DoConfigPropsheet** method, **IMsgServiceAdmin::ConfigureMsgService** method, **IMalloc**

OpenStreamOnFile

[New - Windows 95]

Allocates and initializes a stream object to hold the contents of a file.

Syntax

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 OLE 2.0 **IStream::Read** and **IStream::Write** methods, 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 the file 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.

Comments

A client application or service provider calls **OpenStreamOnFile** 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 **IStream** interface. To free the stream object, the application or provider must call the OLE **IStream::Release** method, 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 input parameters *lpAllocateBuffer* and *lpFreeBuffer* 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 client applications in calling object interfaces. For example, two provider interface methods that typically allocate memory are **IMAPIProp::GetProps** and **IMAPITable::QueryRows**.

SOF_UNIQUEFILENAME is used to create a temporary filename that is unique to the message system.   If this flag is set, the *szFileName* parameter specifes the file name 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*.

The **OpenStreamOnFile** function is defined in MAPIUTIL.H.

See Also

IMAPIProp::GetProps method, **IMAPITable::QueryRows** method, **MAPIAllocateBuffer** function, **MAPIFreeBuffer** function

OpenTnefStream

[New - Windows 95]

Called by a transport provider to initiate a MAPI Transport Neutral Encapsulation Format (TNEF) session.

Syntax

HRESULT OpenTnefStream(LPMAPISUP *lpMapiSup*, **LPSTREAM** *lpStream*, **LPTSTR** *lpszStreamName*, **ULONG** *ulFlags*, **LPMESSAGE** *lpMessage*, **WORD** *wKey*, **LPITNEF FAR*** *lppTNEF***)**

Parameters

lpvsup
Passes a support object or passes in NULL. If NULL, *lpadrbook* 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 specifying a pointer 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 specifying a pointer 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 may 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 body. 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.

Comments

A TNEF object created by **OpenTnefStream** 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.

This function 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 kept around to allow compatibility for those already using TNEF.

The **OpenTnefStream** function is defined in TNEF.H.

See Also

IMAPISupport : IUnknown interface, **ITnef::ExtractProps** method, **IXPProvider::TransportLogon** method **OpenTnefStreamEx** function

OpenTnefStreamEx

[New - Windows 95]

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.

Syntax

HRESULT OpenTnefStreamEx(LPVOID *lpvSupport*, **LPSTREAM** *lpStreamName*, **LPTSTR** *lpszStreamName*, **ULONG** *ulFlags*, **LPMESSAGE** *lpMessage*, **WORD** *wKeyVal*, **LPADRBOOK** *lpAdressBook*, **LPITNEF FAR*** *lppTNEF*);

Parameters

lpvsup
Passes a support object or passes in NULL.   If NULL *lpadrbook* 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 specifying a pointer 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, the attributes attMAPIProps, attAttachment, attRenddata, and attRecipTable. This mode is ideal for use when no backwards compatibility is required.

lpMessage
Input parameter specifying a pointer 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 may 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 body. 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.

Comments

A TNEF object created by **OpenTnefStream** 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.

This function allocates and initializes a TNEF object for the provider to use in encoding a MAPI message   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.

The **OpenTnefStreamEx** function is defined in TNEF.H.

See Also

IMAPISupport : IUnknown interface, **ITnef::ExtractProps** method, **IXPProvider::TransportLogon** method **OpenTnefStream** function

RTFSync

[New - Windows 95]

Ensures that the rich text format (RTF) body of a message matches the plain text body.   It is necessary to call this function before reading the RTF body and after modifying the RTF body.

Syntax

HRESULT RTFSync(LPMESSAGE *lpMessage*, **ULONG** *ulFlags*, **BOOL FAR*** *lpfMessageUpdated***)**

Parameters

lpMessage
Input parameter specifying a pointer to the message to be updated.

ulFlags
Input parameter containing a bitmask of flags used to indicate the RTF or body of the message has changed. The following flags can be set:

RTF_SYNC_RTF_CHANGED
Indicates the RTF has changed.

RTF_SYNC_BODY_CHANGED
Indicates the plain text body of the message has changed.

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.

Comments

This function should be called with RTF_SYNC_BODY_CHANGED before reading the RTF stream. This function should be called with RTF_SYNC_RTF_CHANGED after modifying the RTF body of a message. If both the RTF body and plain text body are modified, this function should be called with RTF_SYNC_RTF_CHANGED | RTF_SYNC_BODY_CHANGED.

If the value of the *lpfMessageUpdated* parameter is set to TRUE, then **IMAPIProp::SaveChanges** should be called for the message. **RTFSync** does not call **SaveChanges** as part of its implementation. If **SaveChanges** is not called the modifications will not be saved in the message.

The **RTFSync** function is defined in MAPIUTIL.H.

See Also

WrapCompressedRTFStream

ScDupPropset

[New - Windows 95]

Duplicates a property value array in a single block of MAPI memory. The **ScDupPropset** function combines the operations of the **ScCopyProps** and **ScCountProps** functions.

Syntax

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.

Comments

The **ScDupPropset** function is defined in MAPIUTIL.H.

See Also

MAPIAllocateBuffer function, **ScCopyProps** function, **ScCountProps** function, **SPropValue** structure

SERVICEWIZARDDLGPROC

[New - Windows 95]

The **SERVICEWIZARDDLGPROC** prototype function represents a callback invoked by the Profile Wizard to allow the provider to react to user events when the provider's property sheets, or pages, are shown.

Syntax

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 *wMsgID*.

lParam
Input parameter whose contents depend on the message specified in *wMsgID*.

Return Values

The value returned is dependent on the message type sent. The recommended practice is to return TRUE if you process the message and FALSE if you don't process the message.

Comments

When the user selects the Next button, SERVICEWIZARDDLGPROC is called with the WM_COMMAND message and ID_NEXT in the *wParam* parameter. The following steps describe what occurs between the time when the user chooses Next and when 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 the SERVICEWIZARDDLGPROC function, sending the WM_INITDIALOG message, so that the provider can initialize the controls.
4. The Profile Wizard calls the SERVICEWIZARDDLGPROC function, sending the WIZ_QUERYNUMPAGES message. The provider returns the number of pages that it will be showing.
5. The Profile Wizard calls the SERVICEWIZARDDLGPROC function, 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 pressed). 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 the 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 **SERVICEWIZARDDLGPROC** function prototype is defined in MAPIWIZ.H.

SERVICEWIZARDENTRY

[New - Windows 95]

The **SERVICEWIZARDENTRY** prototype function represents a service provider entry point for the Profile Wizard. The Profile Wizard calls this entry point to retrieve enough information to display the provider's configuration property sheets.

Syntax

ULONG SERVICEWIZARDENTRY (HINSTANCE *hProviderDLLInstance,* **LPCSTR FAR*** *lpcsResourceName*, **DLGPROC FAR*** *lpDlgProc*, **LPMAPIPROP** *lpMAPIProp***)**

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.

lpDlgProc
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 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**.

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.

Comments

The Profile Wizard calls the SERVICEWIZARDENTRY function when it is ready to display the provider's configuration user interface. The pointer to the property object should be stored by the provider for future reference; this pointer allows the provider access to the profile. At some point 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 SERVICEWIZARDENTRY function must be placed in the PR_SERVICE_WIZARD_ENTRY_NAME property in MAPISVC.INF.

The resource name is the name of the dialog resource that will be rendered in the Profile Wizard's pane. 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 0, 0 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 this function, see the *MAPI Programmer's Guide* for more information on this function.

The **SERVICEWIZARDENTRY** prototype function is defined in MAPIWIZ.H.

SetAttribIMsgOnIStg

[New - Windows 95]

Sets property attributes for properties of a particular object.

Syntax

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 the location to which this function writes an **SPropAttrArray** defining the property attributes that it has set.

lppPropProblems
Output parameter pointing to a variable where the returned **SPropProblemArray** structure holding a set of property problems is stored. This structure identifies problems encountered if the **SetAttribIMsgOnIStg** function 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.

Comments

A message store provider usually calls **SetAttribIMsgOnIStg**.

The **SetAttribIMsgOnIStg** function is defined in IMESSAGE.H.

See Also

GetAttribIMsgOnIStg function, **SPropAttrArray** structure, **SPropProblemArray** structure, **SPropTagArray** structure

ulValidateParameters

[New - Windows 95]

Checks the parameters MAPI and providers receive from clients.

Syntax

HRESULT ulValidateParameters(METHODS *eMethod,* **LPVOID** *First)*

Parameters

eMethod
: Specifies the method to validate.

First
: Specifies the address of the beginning of the method.

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.

Comments

The **ulValidateParameters** function is called differently depending on whether the calling code is C or C++. This function is used to validate parameters for the few **IUnknown** and MAPI methods that return ULONG rather than HRESULT values; **ValidateParameters** works for all others.

Parameters passed between MAPI and providers are assumed to be correct and are not checked. Providers should check all parameters passed in by clients, but clients should assume that MAPI and provider parameters are correct. Use the HR_FAILED macro to test return values.

For more information on parameter validation, see *MAPI Programmer's Guide*.

The **ulValidateParameters** function is defined in MAPIVAL.H.

See Also

CheckParameters function   **ValidateParameters** function

ValidateParameters

[New - Windows 95]

Checks the parameters providers receive from clients.

Syntax

HRESULT ValidateParameters(METHODS *eMethod*, **LPVOID** *First)*

Parameters

eMethod
Specifies the method to validate.

First
Specifies the address of the beginning of the method.

Return Values

S_OK
All of the parameters are valid.

MAPI_E_CALL_FAILED
One or more of the parameters are not valid.

Comments

Parameters passed between MAPI and providers are assumed to be correct and are not checked. Providers should check all parameters passed in by clients, but clients should assume that MAPI and provider parameters are correct. Use the HR_FAILED macro to test the return values.

ValidateParameters 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 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 **ValidateParameter** 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** are the same.

For more information on parameter validation, see *MAPI Programmer's Guide*.

The **ValidateParameters** function is defined in MAPIVAL.H.

See Also

CheckParameters   **ulValidateParameters**

WrapCompressedRTFStream

[New - Windows 95]

Returns a stream containing the uncompressed RTF body of a message.

Syntax

HRESULT WrapCompressedRTFStream(LPSTREAM *lpCompressedRTFStream*, **ULONG** *uls*, **LPSTREAM FAR*** *lpUncompressedRTFStream***)**

Parameters

lpCompressedRTFStream
Input parameter specifying a pointer 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 flags can be set:

MAPI_WRITE
Returns a stream for reading the RTF.

MAPI_CREATE
Returns a stream for writing the RTF.

lpUncompressedRTFStream
Output parameter specifying a pointer to the location in which this function returns a stream for the uncompressed RTF.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

Comments

The **WrapCompressedRTFStream** function is defined in MAPIUTIL.H.

See Also

RTFSync function

WrapStoreEntryID

[New - Windows 95]

Initializes a message-store entry identifier.

Syntax

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 passed-in strings are in Unicode format. If the MAPI_UNICODE flag is not set, the strings are in 8-bit 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 location where the returned size, in bytes, of the new entry identifier is stored.

lppWrappedEntry
Output parameter pointing to a variable where the returned entry identifier is stored.

Comments

If the calling client application or service provider supplies an internal entry identifier, the **WrapStoreEntryID** function wraps it and forms a new identifier.

The **WrapStoreEntryID** function is defined in MAPIUTIL.H.

See Also

ENTRYID structure   **IMAPISession::OpenEntry** interface

XPProviderInit

[New - Windows 95]

Initializes a transport provider for operation.

Syntax

HRESULT XPProviderInit(HINSTANCE *hInstance*, **LPVOID** *lpvReserved*, **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.

lpvReserved
Reserved, must be zero.

lpAllocateBuffer
Input parameters pointing to the **MAPIAllocateBuffer** function, to be used to allocate memory.

lpAllocateMore
Input parameters pointing to the **MAPIAllocateMore** function, to be used to allocate additional memory where required.

lpFreeBuffer
Input parameters pointing to the **MAPIFreeBuffer** function, to be used to free memory.

ulFlags
Reserved; do not use. If any flags are set, the **XPProviderInit** function fails and returns the value MAPI_E_UNKNOWN_FLAGS.

ulMAPIVer
Input parameter containing the version of the service provider interface that MAPI.DLL uses. For the current version number, see the MAPISPI.H header file.

lpulProviderVer
Input parameter pointing to the version of the service provider interface that the message store uses.

lppXPProvider
Output parameter pointing a variable where the initialized transport provider object returned is stored.

Return Values

S_OK
The call succeeded and has returned the expected value or values.

Comments

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 input parameters *lpAllocateBuffer*, *lpAllocateMore*, and *lpFreeBuffer* point to the **MAPIAllocateBuffer**, **MAPIAllocateMore**, and **MAPIFreeBuffer** functions, respectively, for use by the transport 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 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 on using **XPProviderInit**, see the information on using the **MSProviderInit**, **ABProviderInit**, and **XPProviderInit** functions in *MAPI Programmer's Guide.*

The **XPProviderInit** function is defined in MAPISPI.H.

See Also

ABProviderInit function, **HPProviderInit** function, **IMAPIProp::GetProps** method, **IMAPITable::QueryRows** method, **MAPIAllocateBuffer** function, **MAPIAllocateMore** function, **MAPIFreeBuffer** function, **IXPProvider : IUnknown** interface, **MSProviderInit** function

Extended MAPI Properties

This chapter provides reference information for MAPI properties. This reference provides information about the property identifier, property type, and the value of the property's data for each property. It also provides comments describing each property's usage and limitations.

For definitions and a general discussion of how MAPI works with properties and a list of the properties available on each object, see the *MAPI Programmer's Guide.* For information on the property-related macros **PROP_ID**, **PROP_TAG**, **CHANGE_PROP_TYPE**, and **PROP_TYPE**, see *MAPI Programmer's Reference*.

Extended MAPI Property Types

The following table describes property types supported by MAPI 1.0. For a complete description of properties and multivalued properties, see the *MAPI Programmer's Guide*.

Property type	**Multivalued type**	**Underlying data type**
PT_APPTIME	PT_MV_APPTIME	Double. Time; compatible with the Visual Basic time representation.
PT_BINARY	PT_MV_BINARY	SBinary. Counted byte array.
PT_BOOLEAN	(None)	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. Same as several OLE types, including VT_IUNKNOWN.
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.

See Also

SBinary structure

Required Properties

This section describes required properties that must be present in the MAPI table objects. Additional columns may be in the table depending on a provider's implementation.

MAPI Table	Required Properties
AB and DL Contents Table	PR_ADDRTYPE, PR_DISPLAY_NAME, PR_DISPLAY_TYPE, PR_ENTRYID, PR_INSTANCE_KEY, PR_OBJECT_TYPE
AB Display Table	PR_CONTROL_FLAGS, PR_CONTROL_ID, PR_CONTROL_STRUCTURE, PR_CONTROL_TYPE, PR_DELTAX, PR_DELTAY, PR_XPOS, PR_YPOS
AB Hierarchy Table	PR_AB_PROVIDER_ID, PR_CONTAINER_FLAGS, PR_DEPTH, PR_DISPLAY_NAME, PR_DISPLAY_TYPE, PR_ENTRYID, PR_INSTANCE_KEY, PR_OBJECT_NAME, PR_OBJECT_TYPE
AB One-Off Table	PR_ADDRTYPE, PR_DEPTH, PR_DISPLAY_NAME, PR_DISPLAY_TYPE, PR_ENTRYID, PR_INSTANCE_KEY, PR_SELECTABLE
Attachment Table	PR_ATTACH_NUM, PR_RECORD_KEY, PR_RENDERING_POSITION
Message Store Folder Contents Table	PR_CLIENT_SUBMIT_TIME, PR_DISPLAY_TO, PR_ENTRYID, PR_HASATTACH, 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
Message Store Hierarchy Table	PR_COMMENT, PR_DEPTH, PR_DISPLAY_NAME, PR_ENTRYID, PR_FOLDER_TYPE, PR_STATUS, PR_SUBFOLDERS
Message Services Table	PR_DISPLAY_NAME, PR_INSTANCE_KEY, PR_SERVICE_NAME, PR_SERVICE_UID
Message Stores Table	PR_DEFAULT_STORE, PR_DISPLAY_NAME, PR_ENTRYID, PR_INSTANCE_KEY, PR_PROVIDER_DISPLAY, PR_RECORD_KEY, PR_RESOURCE_TYPE
Outgoing Message 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
Profile Table	PR_DISPLAY_NAME
Provider Table	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
Receive Folder Table	PR_ENTRYID, PR_MESSAGE_CLASS, PR_RECORD_KEY
Recipients Table	PR_ADDRTYPE, PR_DISPLAY_NAME, PR_ENTRYID, PR_OBJECT_TYPE, PR_RECIPIENT_TYPE, PR_ROWID
Status Table	PR_DISPLAY_NAME, PR_ENTRY_ID,

PR_IDENTITY_SEARCH_KEY, PR_INSTANCE_KEY, PR_OBJECT_TYPE, PR_PROVIDER_DISPLAY, PR_PROVIDER_DLL_NAME, PR_RESOURCE_FLAGS, PR_RESOURCE_METHODS, PR_RESOURCE_TYPE, PR_ROWID, PR_STATUS_CODE

You can get pointers to the table objects using Extended MAPI interface methods, as described in the table below:

Table	Table Access Method
AB and DL Contents Table	IMAPIContainer::GetContentsTable
AB Display Table	BuildDisplayTable
AB Hierarchy Table	IMAPIContainer::GetHierarchyTable
AB One-Off Table	IABLogon::GetOneOffTable, IMAPISupport::GetOneOffTable
Attachment Table	IMessage::GetAttachmentTable
Message Store Folder Contents Table	IMAPIFolder::GetContentsTable
Message Store Hierarchy Table	IMAPIFolder::GetHierarchyTable
Message Services Table	IMsgServiceAdmin::GetMsgServiceTable
Message Stores Table	IMAPISession::GetMsgStoresTable
Outgoing Message Queue Table	IMsgStore::GetOutgoingQueue
Profile Table	IProfAdmin::GetProfileTable
Provider Table	IMsgServiceAdmin::GetProviderTable, IProviderAdmin::GetProviderTable
Receive Folder Table	IMsgStore::GetReceiveFolderTable
Recipient Table	IMessage::GetRecipientTable
Status Table	IMAPISession::GetStatusTable

MAPI Property Identifier Ranges

MAPI property identifiers are grouped into the following ranges. The property identifiers 0x0000 and 0xFFFF are reserved and no object should use these property identifiers.

From	To	Property Category
0x0000	0x0000	Reserved for the special property PROP_ID_NULL
0x0001	0x3FFF	MAPI-defined properties
0x4000	0x57FF	Message envelope properties, defined by transport providers.
0x5800	0x5FFF	Message recipient properties, defined by transport providers.
0x6000	0x65FF	Message properties, defined by client applications.
0x6600	0x67FF	Message properties, defined by service providers.
0x6800	0x7BFF	Message contents properties, defined by the message class.
0x7C00	0x7FFF	Message properties, defined by the message class.
0x8000	0xFFFE	Named properties. Application-defined or provider-defined properties identified only by name through the property name-to-identifier mapping facility of the **IMAPIProp** interface.
0xFFFF	0xFFFF	Reserved for the special property PROP_ID_INVALID

Developing Portable Property-Related Code

To develop portable code for a client application, use the type PT_TSTRING for all string-related properties and conditionally compile your client application for your target platform.

The PT_TSTRING property type is conditionally compiled to the PT_UNICODE type for a Unicode platform and is conditionally compiled to PT_STRING8 for non-Unicode platforms.

For example, consider the property PR_MESSAGE_CLASS, which has three distinct definitions in the file MAPITAGS.H:

```
#define PR_MESSAGE_CLASS        PROP_TAG( PT_TSTRING, 0x001A)
#define PR_MESSAGE_CLASS_W    PROP_TAG( PT_UNICODE, 0x001A)
#define PR_MESSAGE_CLASS_A    PROP_TAG( PT_STRING8, 0x001A)
```

Note that all three properties use the same property identifier, 0x001A. The compiler defines PR_MESSAGE_CLASS as either PR_MESSAGE_CLASS_W or PR_MESSAGE_CLASS_A, depending on your target platform. The Unicode version of a string property (the name ending in "_W") is used when you define the symbolic constant UNICODE and compile your code.

Providers should explicitly handle either the UNICODE string type (the property name that ends in "_W") or the ANSI string type (the property name that ends in "_A"), or both.

Property Reference

The following section contains a reference entry for each Extended MAPI property.

The values for PT_TSTRING displayed in the property tags in this chapter are the values for PT_STRING8, which has the value 0x001E. To obtain the UNICODE version of the property tag, replace the PT_STRING8 value 0x001E with the PT_UNICODE value 0x001F in the low-order 16 bits of the property tag.

For example, the property tag for the PT_STRING8 version of PR_ADDRTYPE is shown as the PT_STRING8 value, 0x3002001E. 0x3002001F is the property tag for the UNICODE version of PR_ADDRTYPE, obtained by replacing the low-order word value 0x001E with the value 0x001F.

PR_AB_DEFAULT_DIR

[New - Windows 95]

Reserved for use by MAPI.

Details

Identifier 0x3D06; property type PT_BINARY; property tag 0x3D060102

Comments

Do not use this property. It is reserved for use by MAPI only.

See Also

PR_AB_SEARCH_PATH property

PR_AB_DEFAULT_PAB

[New - Windows 95]

Reserved for use by MAPI.

Details

Identifier 0x3D07; property type PT_BINARY; property tag 0x3D070102

Comments

Do not use this property. It is used by MAPI only.

See Also

PR_AB_PROVIDER_ID property

PR_AB_PROVIDER_ID

[New - Windows 95]

Contains a MAPI unique identifier value that identifies the address book provider.

Details

Identifier 0x3615; property type PT_BINARY; property tag 0x36150102

Comments

The uid identifies which provider supplies this particular container in the hierarchy. The value is generated by, and is unique to, each provider.

A provider can provide more than one identifier. For example, a provider that supplies two very different containers can publish unique identifiers for each container. This property represents a documented place where the provider can publish the identifier.

This property is analogous to the PR_MDB_PROVIDER property for stores.

This property appears in the hierarchy tables. Address book containers and hierarchy tables should provide this property. Your application can use it to find related rows in an address book hierarchy table.

See Also

MAPIUID structure, PR_MDB_PROVIDER property

PR_AB_PROVIDERS

[New - Windows 95]

Reserved for use by MAPI.

Details

Identifier 0x3D01; property type PT_BINARY; property tag 0x3D010102

Comments

Do not use this property. It is reserved for use by MAPI only.

See Also

PR_AB_PROVIDER_ID property

PR_AB_SEARCH_PATH

[New - Windows 95]

Reserved for use by MAPI.

Details

Identifier 0x3D05; property type PT_MV_BINARY; property tag 0x3D051102

Comments

Do not use this property. It is used by MAPI only.

See Also

PR_AB_DEFAULT_DIR property

PR_AB_SEARCH_PATH_UPDATE

[New - Windows 95]

Reserved for use by MAPI.

Details

Identifier 0x3D11; property type PT_BINARY; property tag 0x3D110102

Comments

Do not use this property. It is used by MAPI only. Contains an entry identifier for the address book container. Used in conjunction with PR_AB_SEARCH_PATH.

See Also

PR_AB_SEARCH_PATH property

PR_ACCESS

[New - Windows 95]

Contains a bitmask of flags indicating the level at which the client application can access the open object.

Details

Identifier 0x0FF4, Type: PT_LONG; property tag 0x0FF40003

Comments

Zero or more of the following flags can be set:

Value	Description
MAPI_ACCESS_CREATE_ASSOCIATED	The client can create the 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.

See Also

PR_ACCESS_LEVEL property

PR_ACCESS_LEVEL

[New - Windows 95]

Contains a bitmask of flags indicating the level at which the client application can access the open object.

Details

Identifier 0x0FF7; property type PT_LONG; property tag 0x0FF70003

Comments

Use PR_ACCESS_LEVEL to determine whether write access was granted. Calls to IMsgStore::OpenEntry can request the best available access. By default, access is read-only, although the access granted is defined by the provider. A generous provider can offer higher access than that requested. For example, the provider may choose to grant read and write access when the client requested read-only access.

The following flag can be set:

Value	Description
MAPI_MODIFY	Write access.

See Also

PR_ACCESS property

PR_ACCOUNT

[New - Windows 95]

Contains the messaging user's account name.

Details

Identifier 0x3A00; property type PT_TSTRING; property tag 0x3A00001E

Comments

Many properties are available for the messaging user, including several properties that identify the user's   addresses and several telephone numbers.

See Also

PR_EMAIL_ADDRESS property

PR_ACKNOWLEDGEMENT_MODE

[New - Windows 95]

Contains the identifier of the mode for message acknowledgment.

Details

Identifier 0x0001; property type PT_LONG; property tag 0x00010003

Comments

Corresponds to the interpersonal notification IM_ACKNOWLEDGEMENT_MODE attribute as defined by the X.400 message-handling standard.

X.400 defines the following values:

Value	Description
0	Manual acknowledgment
1	Automatic acknowledgment

See Also

PR_MESSAGE_CLASS property

PR_ADDRTYPE

[New - Windows 95]

Contains the recipient's e-mail address type, such as "SMTP".

Details

Identifier 0x3002; property type PT_TSTRING; property tag 0x3002001E

Comments

PR_ADDRTYPE specifies whichh messaging service the MAPI system uses to handle a given message. This property also determines the format of the address string. Examples include X400, FAX, MHS, and PROFS.

PR_ADDRTYPE qualifies the PR_EMAIL_ADDRESS property. The string provided by PR_ADDRTYPE can contain only the uppercase alphabetic characters A through Z and the numbers 0 through 9.

Distribution list objects and mail user objects must furnish PR_ADDRTYPE.

See Also

PR_EMAIL_ADDRESS property

PR_ALTERNATE_RECIPIENT

[New - Windows 95]

Contains an entry identifier defining an alternate recipient chosen by a message sender to receive a message if it cannot be delivered to the original recipient.

Details

Identifier 0x3A01; property type PT_BINARY; property tag 0x3A010102

Comments

For X.400 environments, PR_ALTERNATE_RECIPIENT corresponds to the MH_T_ALTERNATE_RECIPIENT_NAME attribute as defined by the X.400 message-handling standard.

See Also

PR_ALTERNATE_RECIPIENT_ALLOWED property

PR_ALTERNATE_RECIPIENT_ALLOWED

[New - Windows 95]

Contains TRUE if the messaging system delivers the associated message to an alternate delegated by the original recipient, and FALSE otherwise.

Details

Identifier 0x0002; property type PT_BOOLEAN; property tag 0x0002000B

Comments

For X.400 and EDI environments, PR_ALTERNATE_RECIPIENT_ALLOWED represents the corresponding submission envelope per-message attributes as defined by the X.400 and EDI message-handling standards. For X.400 environments, this MAPI property corresponds to the MH_T_ALTERNATE_RECIP_ALLOWED attribute.

See Also

PR_ALTERNATE_RECIPIENT property

PR_ANR

[New - Windows 95]

Represents a special property that can be placed in a property restriction to enable special behavior in address book containers.

Details

Identifier 0x360c; property type PT_TSTRING; property tag 0x360C001E

Comments

PR_ANR is not a property of any object. It contains an ambiguous name resolution (ANR) restriction that can be tested against an address book contents table to find corresponding message recipients. It is used exclusively by providers.

See Also

PR_ENTRYID property

PR_ASSISTANT

[New - Windows 95]

Contains the name of the messaging user's administrative assistant.

Details

Identifier 0x3A30; property type PT_TSTRING; property tag 0x3A30001E

Comments

Another related property, PR_ASSISTANT_TELEPHONE_NUMBER, contains the assistant's phone number.

See Also

PR_ASSISTANT_TELEPHONE_NUMBER property

PR_ASSISTANT_TELEPHONE_NUMBER

[New - Windows 95]

Contains the phone number of the messaging user's administrative assistant.

Details

Identifier 0x3A2E; property type PT_TSTRING; property tag 0x3A2E001E

Comments

The phone number is for the assistant specified by the property PR_ASSISTANT.

See Also

PR_ENTRYID property

PR_ASSOC_CONTENT_COUNT

[New - Windows 95]

Contains the count of items in the associated contents folder.

Details

Identifier 0x3617; property type PT_LONG; property tag 0x36170003

Comments

The PR_FOLDER_ASSOCIATED_CONTENTS property indicates the associated contents folder object.

See Also

PR_ACCESS property, PR_FOLDER_ASSOCIATED_CONTENTS property

PR_ATTACH_DATA_BIN

[New - Windows 95]

Contains binary attachment data typically accessed through an OLE IStream interface.

Details

Identifier 0x3701; property type PT_BINARY; property tag 0x37010102

Comments

For X.400 environments, PR_ATTACH_DATA_BIN corresponds to the IM_EXTERNAL_DATA or IM_BILATERAL_DATA attributes, depending on the object, as defined by the X.400 message-handling standard.

See Also

PR_ATTACH_DATA_OBJ property, PR_ATTACH_METHOD property

PR_ATTACH_DATA_OBJ

[New - Windows 95]

Contains an object accessible through an OLE interface.

Details

Identifier 0x3701; property type PT_OBJECT; property tag 0x3701000D

Comments

For an embedded dynamic OLE object, PR_ATTACH_DATA_OBJ contains its own rendering information. The PR_ATTACH_RENDERING property is either nonexistent or empty.

If your client application or service provider cannot open an attachment object using PR_ATTACH_DATA_OBJ with PR_ATTACH_METHOD, use PR_ATTACH_DATA_BIN.

For X.400 environments, PR_ATTACH_DATA_OBJ corresponds to the IM_EXTERNAL_DATA attribute as defined by the X.400 message-handling standard.

See Also

PR_ATTACH_DATA_BIN property, PR_ATTACH_METHOD property, PR_ATTACH_RENDERING property

PR_ATTACH_ENCODING

[New - Windows 95]

Contains an ASN.1 object identifier specifying the encoding for an attachment.

Details

Identifier 0x3702; property type PT_BINARY; property tag 0x37020102

Comments

The syntax and sample object identifiers (OIDs) are defined in the header file MAPIOID.H. For example, the header file includes OIDs for TNEF, MAPI 1.0 OLE 2.0 IStorage, and MacBinary.

For complete information on these types, see the ASN.1 documentation.

See Also

PR_ATTACH_DATA_BIN property, PR_ATTACH_METHOD property

PR_ATTACH_EXTENSION

[New - Windows 95]

Contains a filename extension (for example, "TXT" or "DOC") that indicates the document type of the attachment.

Details

Identifier 0x3703; property type PT_TSTRING; property tag 0x3703001E

Comments

The message store maintains the attachment data associated with this property and generates an appropriate property tag. The client sets this property.

Your service provider uses PR_ATTACH_EXTENSION when converting message attachments (in-route conversion) or launching applications based on attachments in received messages. If the provider does not implement this property, MAPI generates its own filename extension for an attachment based on the extension provided by the attachment's PR_ATTACH_FILENAME property.

See Also

PR_ATTACH_FILENAME property

PR_ATTACH_FILENAME

[New - Windows 95]

Provides a message recipient with an attachment's base filename and extension, excluding path.

Details

Identifier 0x3704; property type PT_TSTRING; property tag 0x3704001E

Comments

The filename is restricted to an 8.3 filename: eight characters plus an extension. For long filenames, use the property PR_ATTACH_LONG_FILENAME.

PR_ATTACH_FILENAME pertains to the attachment methods ATTACH_BY_REFERENCE, ATTACH_BY_REF_RESOLVE, and ATTACH_BY_REF_ONLY, as described for the PR_ATTACH_METHOD property.

For X.400 environments, PR_ATTACH_FILENAME corresponds to the IM_EXTERNAL_PARAMETERS attribute as defined by the X.400 message-handling standard.

See Also

PR_ATTACH_LONG_FILENAME property, PR_ATTACH_METHOD property, PR_ATTACH_PATHNAME property

PR_ATTACH_LONG_FILENAME

[New - Windows 95]

Provides the message recipient with an attachment's long filename, excluding path.

Details

Identifier 0x3707; property type PT_TSTRING; property tag 0x3707001E

Comments

Your application should set this property to a suggested long filename to be used if the host machine receiving a message supports long filenames.

Unlike the filename provided by PR_ATTACH_FILENAME, this name is not restricted to the 8.3 naming convention (up to eight characters and up to three character extension).

See Also

PR_ATTACH_FILENAME property

PR_ATTACH_METHOD

[New - Windows 95]

Contains a MAPI-defined constant representing the method used to access the contents of an attachment.

Details

Identifier 0x3705; property type PT_LONG; property tag 0x37050003

Comments

On creation, all attachment objects have an initial PR_ATTACH_METHOD value of 0, NO_ATTACHMENT.

The MAPI spooler presents the attachment to the transport as if it were attached by value, so that the actual attachment data is not copied to the message store or the message. The attachment data itself is present in the property PR_ATTACH_DATA_BIN.

Client applications and service providers are only required to support the ATTACH_BY_VALUE method. The other methods are optional.

UNC names are recommended for fully-resolved pathnames.

The message store does not enforce any consistency between the value of PR_ATTACH_METHOD and the values of the other attachment properties.

The PR_ATTACH_METHOD property can have the following values:

Value	Description
0	Value indicating that 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 property contains a fully-resolved pathname identifying the attachment to recipients with access to a common file server. If ATTACH_BY_REFERENCE is present, PR_ATTACH_DATA_BIN is empty. A gateway can turn an ATTACH_BY_REFERENCE attachment into an attachment using ATTACH_BY_VALUE to move the attachment data to the PR_ATTACH_DATA_BIN property.
ATTACH_BY_REF_RESOLVE	The PR_ATTACH_PATHNAME property contains a fully-resolved pathname identifying the attachment. If ATTACH_BY_REF_RESOLVE is present, PR_ATTACH_DATA_BIN is empty. When the message that contains the

	ATTACH_BY_REF_RESOLVE attachment is sent, the MAPI spooler resolves the attachment into an ATTACH_BY_VALUE attachment. This resolution process places the attachment data in PR_ATTACH_DATA_BIN.
ATTACH_BY_REF_ONLY	The PR_ATTACH_PATHNAME property contains a fully-resolved pathname identifying the attachment. If ATTACH_BY_REF_ONLY is present, PR_ATTACH_DATA_BIN is empty and the messaging system never resolves the attachment reference. Use this value when you want to send the link but not the data.
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 the OLE object is in OLE 2.0 IStorage format, the data is accessible via PR_ATTACH_DATA_OBJ. When the OLE object is in OLE 1.0 OleStream format, the data is accessible via PR_ATTACH_DATA_BIN as an IStream. The type of the OLE encoding can be determined by the PR_ATTACH_TAG value.

See Also

PR_ATTACH_TAG property, PR_STORE_SUPPORT_MASK property

PR_ATTACH_NUM

[New - Windows 95]

Contains a number that uniquely identifies the attachment within its parent message.

Details

Identifier 0x3706; property type PT_LONG; property tag 0x37060003

Comments

The message store object generates and maintains the PR_ATTACH_NUM property. The attachment number is the secondary sort key in the attachment table.

Used when the attachment is opened using 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. MAPI propagates changes to the table using the **IMessage::CreateAttach** and the **IMessage::DeleteAttach** methods and generates a table notification on the open attachment tables so that client applications can resynchronize to those changes.

See Also

PR_ATTACH_METHOD property

PR_ATTACH_PATHNAME

[New - Windows 95]

Contains the fully qualified path and filename for the file that has been attached to a message by reference.

Details

Identifier 0x3708; property type PT_TSTRING; property tag 0x3708001E

Comments

The PR_ATTACH_PATHNAME property indicates that the actual input data resides in the attachment.

Clients should use a UNC file naming convention when the file is shared, and should use an absolute pathname when the file is local. The provider that moves the message from that computer must convert absolute filenames to UNC.

PR_ATTACH_PATHNAME is required when you use any of the PR_ATTACH_METHOD flags that indicate attach by reference: ATTACH_BY_REFERENCE, ATTACH_BY_REF_RESOLVE, or ATTACH_BY_REF_ONLY.

See Also

PR_ATTACH_FILENAME property, PR_ATTACH_METHOD property, **ScLocalPathFromUNC** function, **ScUNCFromLocalPath** function

PR_ATTACH_RENDERING

[New - Windows 95]

Contains rendering information, such as an icon or metafile, that you can use to represent the attachment in a message.

Details

Identifier 0x3709; property type PT_BINARY; property tag 0x37090102

Comments

Your application can use this property when representing an attachment in a message.

For an attached file, the property usually contains the icon for the file.

For an embedded static OLE object, it 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.

See Also

PR_ATTACH_PATHNAME property

PR_ATTACH_SIZE

[New - Windows 95]

Contains the sum in bytes of the sizes of all properties in the attachment object.

Details

Identifier 0x0E20; property type PT_LONG; property tag 0x0E200003

Comments

PR_ATTACH_SIZE contains the sum of the sizes of all the properties in the attachment object, including the size of PR_ATTACH_DATA_BIN or PR_ATTACH_DATA_OBJ. Accordingly, PR_ATTACH_SIZE is usually larger than the contents of the attachment alone. For OLE objects, PR_ATTACH_SIZE provides a useful estimate of the size of the attached object.

This property can be used to check the size of the attachment before performing a remote transfer by modem, and to display progress indicators when saving the attachment to disk.

PR_ATTACH_SIZE provides the same information for the attachment object that PR_MESSAGE_SIZE provides for the message.

See Also

PR_ATTACH_NUM property, PR_MESSAGE_SIZE property

PR_ATTACH_TAG

[New - Windows 95]

Contains further information about the attachment data.

Details

Identifier 0x370A; property type PT_BINARY; property tag 0x370A0102

Comments

For example, when PR_ATTACH_ENCODING indicates the format, PR_ATTACH_TAG contains an ASN.1 attachment object identifier (OID) identifying a document or file type that should be associated with the attachment.

For X.400 environments, PR_ATTACH_TAG corresponds to the IM_EXTERNAL_PARAMETERS attribute as defined by the X.400 message-handling standard.

See Also

PR_ATTACH_ENCODING property

PR_ATTACH_TRANSPORT_NAME

[New - Windows 95]

Contains the name of an attachment file modified so that it can be correlated with TNEF messages.

Details

Identifier 0x370C; property type PT_TSTRING; property tag 0x370C001E

Comments

TNEF and the transport provider use this property. It is normally not available to a client application.

This property is commonly used by TNEF when your underlying messaging system wouldn't otherwise support the supplied filenames. For example, this property would be used when the user supplies multiple files with the same name, such as five files named CONFIG.SYS. The transport must modify the names to ensure uniqueness. The modified names appear in PR_ATTACH_TRANSPORT_NAME.

See Also

PR_ATTACH_FILENAME property, PR_ATTACH_PATHNAME property

PR_ATTACHMENT_X400_PARAMETERS

[New - Windows 95]

Contains an ASN.1 object identifier describing an attachment for handling in transit, for example, between platforms.

Details

Identifier 0x3700; property type PT_BINARY; property tag 0x37000102

Comments

In X.400 environments, PR_ATTACHMENT_X400_PARAMETERS corresponds to the IM_EXTERNAL_PARAMETERS attribute as defined by the X.400 message-handling standard. X.400 applications use this property to make the "Parameters" component of a text (body) part of a message.

See Also

PR_ATTACH_ENCODING property

PR_AUTHORIZING_USERS

[New - Windows 95]

Contains a list of entry identifiers for users who have authorized the sending of a message.

Details

Identifier 0x0003; property type PT_BINARY; property tag 0x00030102

Comments

This MAPI property corresponds to the interpersonal notification IM_AUTHORIZING_USERS attribute as defined by the X.400 message-handling standard. X.400 transport providers use this property for interpersonal messages delivered to client applications.

The message store does not maintain the PR_AUTHORIZING_USERS property.

See Also

PR_ENTRYID property

PR_AUTO_FORWARD_COMMENT

[New - Windows 95]

Contains a comment added by the autoforwarding agent.

Details

Identifier 0x0004; property type PT_TSTRING; property tag 0x0004001E

Comments

This MAPI property corresponds to the interpersonal notification IM_AUTO_FORWARD_COMMENT attribute as defined by the X.400 message-handling standard.

See Also

PR_AUTO_FORWARDED property

PR_AUTO_FORWARDED

[New - Windows 95]

Contains TRUE if an automatic agent has forwarded a message, and FALSE otherwise.

Details

Identifier 0x0005; property type PT_BOOLEAN; property tag 0x0005000B

Comments

This MAPI property corresponds to the IM_AUTO_FORWARDED attribute as defined by the X.400 message-handling standard. X.400 transport providers use this property for interpersonal messages delivered to client applications.

See Also

PR_AUTO_FORWARD_COMMENT property

PR_BEEPER_TELEPHONE_NUMBER

[New - Windows 95]

Contains the telephone number of the messaging user's beeper, or pager.

Details

Identifier 0x32A1; property type PT_TSTRING; property tag 0x32A1001E

Comments

This property is the same as the PR_PAGER_TELEPHONE_NUMBER. property

See Also

PR_PAGER_TELEPHONE_NUMBER property

PR_BODY

[New - Windows 95]

Contains the text (body) of a message.

Details

Identifier 0x1000; property type PT_TSTRING; property tag 0x1000001E

Comments

The value for PR_BODY must be expressed in the code page of the operating system that MAPI is running on.

The PR_BODY property is only used with interpersonal messaging (IPM).

For X.400 environments, PR_BODY corresponds to the IM_BODY or IM_TEXT attribute as defined by the X.400 message-handling standard.

See Also

PR_RENDERING_POSITION property, PR_RTF_COMPRESSED property, PR_RTF_IN_SYNC property

PR_BODY_CRC

[New - Windows 95]

Contains a current circular redundancy check (CRC) value computed by a service provider using any CRC algorithm that generates a PT_LONG value.

Details

Identifier 0x0E1C; property type PT_LONG; property tag 0x0E1C0003

Comments

The application only uses PR_BODY_CRC to compare message text (body) strings contained in PR_BODY properties or their variants.

Using this property, your application can quickly and easily detect when the body of a message has changed. It can realize significant performance gains from using PR_BODY_CRC instead of obtaining PR_BODY from the server and comparing it with a local version.

For RTF body values, the RTF_IN_SYNC property.

See Also

PR_BODY property, PR_RTF_IN_SYNC property

PR_BUSINESS_FAX_NUMBER

[New - Windows 95]

Contains the telephone number of the messaging user's business fax machine.

Details

Identifier 0x3A24; property type PT_TSTRING; property tag 0x3A24001E

Comments

Other messaging user properties include the business telephone number.

See Also

PR_BUSINESS_TELEPHONE_NUMBER property

PR_BUSINESS_TELEPHONE_NUMBER

[New - Windows 95]

Contains the primary telephone number for the messaging user's place of business.

Details

Identifier 0x3A08; property type PT_TSTRING; property tag 0x3A08001E

Comments

Other messaging user business-related properties include the fax number and secondary business phone number.

See Also

PR_BUSINESS_FAX_NUMBER property

PR_BUSINESS2_TELEPHONE_NUMBER

[New - Windows 95]

Contains the secondary telephone number for the messaging user's place of business.

Details

Identifier 0x3A1B; property type PT_TSTRING, Property tag: 3a1b001e

Comments

Other messaging user business-related properties include the fax number and primary business phone number.

See Also

PR_BUSINESS_FAX_NUMBER property

PR_CALLBACK_TELEPHONE_NUMBER

[New - Windows 95]

Contains a telephone number that the message recipient can use to reach the sender.

Details

Identifier 0x3A02; property type PT_TSTRING; property tag 0x3A02001E

Comments

Other messaging user properties include the fax number, primary, and secondary business phone number.

See Also

PR_BUSINESS_FAX_NUMBER property

PR_CAPABILITIES_TABLE

[New - Windows 95]

This property is not supported in MAPI 1.0.

Details

Identifier 0x3903; property type PT_OBJECT; property tag 0x3903000D

Comments

In previous releases, this property contained an embedded table object that provides a summary of the address book capabilities.

PR_CAR_TELEPHONE_NUMBER

[New - Windows 95]

Contains the car telephone number of the messaging user.

Details

Identifier 0x3A1E; property type PT_TSTRING; property tag 0x3A1E001E

Comments

Several properties are available for the messaging user, including properties for the fax telephone number, cellular telephone number, and business telephone numbers.

See Also

PR_BUSINESS_TELEPHONE_NUMBER property

PR_CELLULAR_TELEPHONE_NUMBER

[New - Windows 95]

Contains the cellular telephone number of the messaging user.

Details

Identifier 0x3A1C; property type PT_TSTRING; property tag 0x3A1C001E

Comments

Other messaging user properties include the fax number, primary, and secondary business phone number.

See Also

PR_BUSINESS_TELEPHONE_NUMBER property

PR_CLIENT_SUBMIT_TIME

[New - Windows 95]

Contains the time computed by a transport provider that indicates when the message sender marked a message for submission.

Details

Identifier 0x0039; property type PT_SYSTIME; property tag 0x00390040

Comments

This MAPI property corresponds to the MH_T_SUBMISSION_TIME attribute as defined by the X.400 message-handling standard. This property is added to a message based on the results of a submission.

See Also

PR_PROVIDER_SUBMIT_TIME property

PR_COMMENT

[New - Windows 95]

Contains a messaging user-defined comment about the purpose or content of an object.

Details

Identifier 0x3004; property type PT_TSTRING; property tag 0x3004001E

Comments

The content of the string is defined by the specific needs of the messaging user.

See Also

PR_AUTO_FORWARD_COMMENT property

PR_COMMON_VIEWS_ENTRYID

[New - Windows 95]

Contains the entry identifier of the common views folder.

Details

Identifier 0x35E6; property type PT_BINARY; property tag 0x35E60102

Comments

This folder is not visible in the IPM hierarchy.

See Also

PR_DEFAULT_VIEW_ENTRYID property, PR_VIEWS_ENTRYID property

PR_COMPANY_NAME

[New - Windows 95]

Contains the company name of the messaging user.

Details

Identifier 0x3A16; property type PT_TSTRING; property tag 0x3A16001E

Comments

Other messaging user properties include the fax number, primary, and secondary business phone number.

See Also

PR_BUSINESS_TELEPHONE_NUMBER property

PR_CONTAINER_CLASS

[New - Windows 95]

Contains a text string that identifies the folder, as PR_MESSAGE_CLASS identifies the message class.

Details

Identifier 0x3613; property type PT_TSTRING; property tag 0x3613001E

Comments

Your application should set this property for a folder object that it creates. The application should not attempt further accesses to folders that have a PR_CONTAINER_CLASS value that your application does not recognize.

See Also

PR_CONTAINER_CONTENTS property

PR_CONTAINER_CONTENTS

[New - Windows 95]

Contains an embedded contents table object that describes the characteristics of a container.

Details

Identifier 0x360f; property type PT_OBJECT; property tag 0x360F000D

Comments

PR_CONTAINER_CONTENTS describes the characteristics of a container, similar to the way that PR_STORE_SUPPORT_MASK describes the characteristics of a message store.

To retrieve container or folder contents, your application should call **IMAPIContainer::GetContentsTable**. The application cannot retrieve PR_CONTAINER_CONTENTS data by calling **IMAPIProp::GetPropList** or **IMAPIProp::GetProps**.

See Also

IMAPIContainer::GetContentsTable method, PR_CONTAINER_FLAGS property

PR_CONTAINER_FLAGS

[New - Windows 95]

Contains a bitmask of flags describing the address book container.

Details

Identifier 0x3600; property type PT_LONG; property tag 0x36000003

Comments

The flags refer to the container itself than to the items within the container. For example, AB_MODIFIABLE indicates that the container allows entries to be added or deleted. The flag does not refer to whether the individual objects themselves can be added or deleted.

Either AB_MODIFIABLE or AB_UNMODIFIABLE must be set. When both flags are cleared, an error occurs. Note that it is valid for both flags to be set. When both are set, you must attempt a call and examine the return code to determine the container's capabilities. Most client applications examine only the AB_MODIFIABLE bit. When it is set, the client makes a call that attempts to modify the container, and check the return value.

Note that both the AB_UNMODIFIABLE and AB_MODIFIABLE bits can be set at the same time, to indicate that the container does not know whether it can be modified or not. One or both must be set.

The following flags can be set for the PR_CONTAINER_FLAGS bitmask:

Value	Description
AB_FIND_ON_OPEN	Display UI to request a restriction before it displays any contents of the container.
AB_MODIFIABLE	You can add and remove entries from the container. Note that this setting does not indicate creation or deletion of the objects, but solely whether they can be placed in the container.
AB_RECIPIENTS	It is possible to modify the contents table, not just the hierarchy table. It is possible for the container to contain recipients. Note that this flag does not indicate whether any recipients are present in the container.
AB_SUBCONTAINERS	Indicates the container holds child containers. This flag must be set for the container to support IMAPIContainer::GetHierarchyTable.
AB_UNMODIFIABLE	The container cannot be modified.

See Also

PR_CONTAINER_CONTENTS property

PR_CONTAINER_HIERARCHY

[New - Windows 95]

Contains an embedded hierarchy table object that provides information about the child containers. Details

Identifier 0x360e; property type PT_OBJECT; property tag 0x360E000D

Comments

This property is equivalent to the GetHierarchyTable method.

Your application can use the PR_CONTAINER_HIERARCHY property to exclude an item from a copy operation.

To retrieve the data from this property, your application should call IMAPIContainer::GetHierarchyTable. The application cannot retrieve hierarchy data by calling IMAPIProp::GetPropList or IMAPIProp::GetProps.

Several MAPI properties provide access to table objects:

Property	**Table**
PR_CONTAINER_HIERARCHY	Hierarchy table
PR_FOLDER_ASSOCIATED_CONTENTS	Associated contents table
PR_MESSAGE_ATTACHMENTS	Attachments table
PR_MESSAGE_RECIPIENTS	Recipients table

See Also

IMAPIProp::CopyProps method, **IMAPIProp::CopyTo** method, PR_CONTAINER_CONTENTS property, PR_MESSAGE_ATTACHMENTS property, PR_MESSAGE_RECIPIENTS property

PR_CONTAINER_MODIFY_VERSION

[New - Windows 95]

Contains the current modification version for the container.

Details

Identifier 0x3614; property type PT_I8; property tag 0x36140014

Comments

PT_I8 is an eight-byte, or 64-bit integer. The format is described in the reference documentation for the structure LARGE_INTEGER.

See Also

PR_CONTAINER_CLASS property

PR_CONTENT_CONFIDENTIALITY_ALGORITHM_ID

[New - Windows 95]

Contains an identifier for the algorithm used to confirm message content confidentiality.

Details

Identifier 0x0006; property type PT_BINARY; property tag 0x00060102

Comments

X.400 and EDI transport providers use this property for messages delivered to client applications.

This MAPI property represents the corresponding X.400 and EDI submission envelope per-message attributes as defined by the X.400 and EDI message-handling standards. For X.400 environments, this property corresponds to the MH_T_ALGORITHM_ID or MH_T_CONFIDENTIALITY_ALGORITHM attribute.

See Also

PR_SECURITY property

PR_CONTENT_CORRELATOR

[New - Windows 95]

Contains a copy of any reports generated by an outbound message to assist the message sender in matching a report with the original message.

Details

Identifier 0x0007; property type PT_BINARY; property tag 0x00070102

Comments

This MAPI property represents the corresponding X.400 and EDI submission envelope per-message attributes as defined by the X.400 and EDI message handling standards. For X.400 environments, PR_CONTENT_CORRELATOR corresponds to the MH_T_CONTENT_CORRELATOR attribute.

X.400 environments use this property in probe messages and to represent report per-message attributes. EDI transport providers use it for messages delivered to client applications.

See Also

PR_CONTENT_IDENTIFIER property

PR_CONTENT_COUNT

[New - Windows 95]

Contains the number of entries in the container, as computed by the message store.

Details

Identifier 0x3602; property type PT_LONG; property tag 0x36020003

Comments

The number does not include associated entries.

See Also

PR_CONTENT_UNREAD property

PR_CONTENT_IDENTIFIER

[New - Windows 95]

Contains a key value that enables the recipient of a message to identify its content.

Details

Identifier 0x0008; property type PT_TSTRING; property tag 0x0008001E

Comments

This MAPI property represents the corresponding X.400 and EDI submission envelope per-message attributes as defined by the X.400 and EDI message-handling standards. For X.400 environments, PR_CONTENT_IDENTIFIER corresponds to the MH_T_CONTENT_IDENTIFIER attribute. Both X.400 and EDI transport providers use this property for messages delivered to client applications.X.400 environments also use this property in probe messages.

See Also

PR_CONTENT_LENGTH property

PR_CONTENT_INTEGRITY_CHECK

[New - Windows 95]

Contains an ASN.1 content integrity check value that allows a message sender to protect message content from disclosure to unauthorized recipients.

Details

Identifier 0x0C00; property type PT_BINARY; property tag 0x0C000102

Comments

Both X.400 and EDI transport providers use this property for messages delivered to client applications. For X.400 environments, PR_CONTENT_INTEGRITY_CHECK corresponds to the MH_T_INTEGRITY_CHECK attribute.

See Also

PR_CONTENT_IDENTIFIER property

PR_CONTENT_LENGTH

[New - Windows 95]

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.

Details

Identifier 0x0009; property type PT_LONG; property tag 0x00090003

Comments

This MAPI property corresponds to the probe submission per-message MH_T_CONTENT_LENGTH attribute as defined by the X.400 message-handling standard.

See Also

PR_CONTENT_IDENTIFIER property

PR_CONTENT_RETURN_REQUESTED

[New - Windows 95]

Contains TRUE if a particular message should be returned with a nondelivery report, and FALSE otherwise.

Details

Identifier 0x000A; property type PT_BOOLEAN; property tag 0x000A000B

Comments

This MAPI property represents the corresponding X.400 and EDI submission envelope per-message attributes as defined by the X.400 and EDI message-handling standards. For X.400 environments, PR_CONTENT_RETURN_REQUESTED corresponds to the MH_T_CONTENT_RETURN_REQUESTED attribute.

See Also

PR_CONTENT_LENGTH property

PR_CONTENT_UNREAD

[New - Windows 95]

Contains a count of the number of unread messages in an address book container or folder contents table.

Details

Identifier 0x3603; property type PT_LONG; property tag 0x36030003

Comments

For a folder, contains the count of the number of messages in the contents table for which the message read flag is not set.

See Also

PR_CONTENT_COUNT property, PR_CONTENT_IDENTIFIER property

PR_CONTENTS_SORT_ORDER

[New - Windows 95]

Contains a value specifying the sort order for the columns of the container or folder contents table.

Details

Identifier 0x360D; property type PT_MV_LONG; property tag 0x360D1003

Comments

The PR_CONTENTS_SORT_ORDER property can have the following values:

Value	**Description**
TABLE_SORT_ASCEND	Ascending order
TABLE_SORT_DESCEND	Descending order
TABLE_SORT_COMBINE	Combine to right order.

See Also

PR_CONTENT_IDENTIFIER property, **SSortOrder** structure

PR_CONTROL_FLAGS

[New - Windows 95]

Contains a value that controls the behavior associated with the display table.

Details

Identifier 0x3F00; property type PT_LONG; property tag 0x3F000003

Comments

The following flags can be set for the PR_CONTROL_FLAGS bitmask:

Value	Description
DT_ACCEPT_DBCS	Used with edit controls and combo box controls. Allows multiple-byte character sets.
DT_EDITABLE	Indicates that you can edit the control; 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	Indicates that the edit control can contain multiple lines. This indicates whether you can enter a return character within the control.
DT_PASSWORD_EDIT	Applies to edit controls. Indicates that when you enter a password, the password is not echoed to the control but is displayed using asterisk characters.
DT_REQUIRED	Indicates that if the control allows changes (DT_EDITABLE), the control must have a value before you can call SaveChanges.
DT_SET_IMMEDIATE	Enables immediate setting of a value; as soon as you change a value in the control, MAPI calls the SetProps method for the property associated with that control. When this flag is not set, the values are set when the control loses focus.
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, **IMAPIProp::SaveChanges** method, PR_CONTROL_ID property

PR_CONTROL_ID

[New - Windows 95]

Contains the unique identifier for a control used in a dialog box.

Details

Identifier 0x3F07; property type PT_BINARY; property tag 0x3F070102

Comments

PR_CONTROL_ID contains a unique identifier for the control. It is recommended that this be formed from a GUID followed by a LONG, in binary format. Use the same GUID to identify the provider, and use unique LONG values for each control to ensure that the controls do not collide.

The PR_CONTROL_ID value 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_TYPE property

PR_CONTROL_STRUCTURE

[New - Windows 95]

Contains a pointer to a specific type of dialog box control structure.

Details

Identifier 0x3F01; property type PT_BINARY; property tag 0x3F010102

Comments

This property represents a binary blob that is cast to one of the control structures. The control structures include DTBLBUTTON, DTBLDTBLCHECKBOX, DTBLCOMBOBOX, DTBLDDLBX, DTBLEDIT, DTBLGROUPBOX, DTBLLABEL, DTBLLBX, DTBLMVLBOX, DTMVLISTBOX, DTBLRADIOBUTTON, and DTPAGE.

See Also

DTCTL structure, PR_CONTROL_TYPE property

PR_CONTROL_TYPE

[New - Windows 95]

Contains a value indicating a control type for a given dialog box.

Details

Identifier 0x3F02; property type PT_LONG; property tag 0x3F020003

Comments

The PR_CONTROL_TYPE property can have 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

[New - Windows 95]

Contains an index that indicates the relative position of this message within a conversation thread.

Details

Identifier 0x0071; property type PT_BINARY; property tag 0x00710102

Comments

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. For all messages that have the same value for PR_CONVERSATION_TOPIC, you can then sort by PR_CONVERSATION_INDEX to indicate the hierarchical relationship of the messages.

PR_CONVERSATION_INDEX and PR_CONVERSATION_TOPIC replace the obsolete property PR_CONVERSATION_KEY.

See Also

PR_CONVERSATION_TOPIC property

PR_CONVERSATION_KEY

[New - Windows 95]

Obsolete. Replaced by the properties PR_CONVERSATION_TOPIC and PR_CONVERSATION_INDEX.

Details

Identifier 0x000B; property type PT_BINARY; property tag 0x000B0102

Comments

This property was part of the original design for organizing conversations. It has been replaced by PR_CONVERSATION_TOPIC and PR_CONVERSATION_INDEX.

See Also

PR_CONVERSATION_INDEX property, PR_CONVERSATION_TOPIC property

PR_CONVERSATION_TOPIC

[New - Windows 95]

Represents the topic of the first message in a conversation thread. Used with PR_CONVERSATION_INDEX.

Details

Identifier 0x0070; property type PT_TSTRING; property tag 0x0070001E

Comments

A conversation thread represents a series of messages and replies. The PR_CONVERSATION_TOPIC 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 message in the conversation thread use the same PR_CONVERSATION_TOPIC. The property PR_CONVERSATION_INDEX indicates the relationship between subsequent messages and replies.

PR_CONVERSATION_INDEX and PR_CONVERSATION_TOPIC replace the obsolete property PR_CONVERSATION_KEY.

See Also

PR_CONVERSATION_INDEX property

PR_CONVERSION_EITS

[New - Windows 95]

Contains the encoded information types (EITs) that are applied to a message in transit to describe conversions.

Details

Identifier 0x000C; property type PT_BINARY; property tag 0x000C0102

Comments

This MAPI property corresponds to the interpersonal notification IM_CONVERSION_EITS attribute as defined by the X.400 message-handling standard.

X.400 environments use the PR_CONVERSION_EITS property for both nondelivery reports and delivery reports.

See Also

PR_CONVERSION_PROHIBITED property

PR_CONVERSION_PROHIBITED

[New - Windows 95]

Contains TRUE if message conversions are prohibited by default for the associated messaging user, and FALSE otherwise.

Details

Identifier 0x3A03; property type PT_BOOLEAN; property tag 0x3A03000B

Comments

For X.400 environments, PR_CONVERSION_PROHIBITED corresponds to the MH_T_CONVERSION_PROHIBITED attribute as defined by the X.400 message-handling standard.

See Also

PR_CONVERSION_WITH_LOSS_PROHIBITED property

PR_CONVERSION_WITH_LOSS_PROHIBITED

[New - Windows 95]

Contains TRUE if a message transfer agent (MTA) is prohibited from making message text (body) conversions that lose information, such as "lossy" character set mapping. Contains FALSE otherwise.

Details

Identifier 0x000D; property type PT_BOOLEAN; property tag 0x000D000B

Comments

For X.400 environments, PR_CONVERSION_WITH_LOSS_PROHIBITED corresponds to the MH_T_CONVERSION_LOSS_PROHIBITED attribute.

X.400 and EDI transport providers use this property for messages delivered to client applications. X.400 probe messages also use this property.

See Also

PR_CONVERSION_PROHIBITED property

PR_CONVERTED_EITS

[New - Windows 95]

Contains an identifier for the types of text (body) parts in a message after conversion.

Details

Identifier 0x000E; property type PT_BINARY; property tag 0x000E0102

Comments

For X.400 environments, PR_CONVERTED_EITS corresponds to the MH_T_CONVERTED_EITS attribute.

X.400 and EDI transport providers use this property for messages delivered to client applications.

See Also

PR_CONVERSION_EITS property

PR_CORRELATE

[New - Windows 95]

Contains TRUE if the messaging system correlates sent messages with reports, and FALSE otherwise.

Details

Identifier 0x0E0C; property type PT_BOOLEAN; property tag 0x0E0C000B

Comments

The information in the property is provided by the message tracking and report generation facilities of the messaging system.

When your transport provider encounters a submitted message with PR_CORRELATE set to TRUE, it uses the PR_CORRELATE_MTSID property to store the message transport system (MTS) identifier for the message.

See Also

PR_CORRELATE_MTSID property

PR_CORRELATE_MTSID

[New - Windows 95]

Contains the message transport system (MTS) identifier used with the PR_CORRELATE property in correlating messages with reports.

Details

Identifier 0x0E0D; property type PT_BINARY; property tag 0x0E0D0102

Comments

When your transport provider encounters a submitted message with the PR_CORRELATE property set to TRUE, it uses PR_CORRELATE_MTSID to store the MTS identifier for the message. When information comes in from the message transfer agent (MTA) pertaining to the tracked message, the provider updates PR_CORRELATE_MTSID with the referenced MTS identifier and adds the property to the new report or notification.

Your client application can maintain a search-results folder of all messages having a PR_CORRELATE_MTSID property. When new information comes in about a tracked message, the application can apply restrictions to the search-results folder, find the original version of the tracked message, and correlate the original message information with the new information.

See Also

PR_CORRELATE property

PR_COUNTRY

[New - Windows 95]

Contains the country of the messaging user.

Details

Identifier 0x3A26; property type PT_TSTRING; property tag 0x3A26001E

Comments

Other messaging user properties include the fax number, primary, and secondary business phone number.

See Also

PR_BUSINESS_TELEPHONE_NUMBER property

PR_CREATE_TEMPLATES

[New - Windows 95]

Contains an embedded table object that contains dialog box templates.

Details

Identifier 0x3604; property type PT_OBJECT; property tag 0x3604000D

Comments

To learn what you can create inside a container, call IMAPIProp::OpenProperty on PR_CREATE_TEMPLATES. The resulting object is the template ID table that lists the objects you can create inside a container.

To subsequently create the objects, call the container object's CreateEntry method. To create the template table using an IMAPITable interface, your client application passes PR_CREATE_TEMPLATES to the IMAPIProp::OpenProperty method for the address book container or folder.

See Also

IABContainer::CreateEntry method, PR_DEF_CREATE_DL property, PR_DISPLAY_NAME property, PR_DISPLAY_TYPE property

PR_CREATION_TIME

[New - Windows 95]

Contains the creation time for the object.

Details

Identifier 0x3007; property type PT_SYSTIME; property tag 0x30070040

Comments

A message store sets this property for each message that it creates.

See Also

PR_CREATION_VERSION property

PR_CREATION_VERSION

[New - Windows 95]

Contains the version number of the message store that was current at the time a given message was created.

Details

Identifier 0x0E19; property type PT_I8; property tag 0x0E190014

Comments

This version number is a copy of the message store version number contained in the PR_CURRENT_VERSION property.

Your application or service provider uses PR_CREATION_VERSION to support synchronization between message stores.

This property is being evaluated to determine whether it will be supported in future versions of MAPI 1.0.

See Also

PR_CURRENT_VERSION property, PR_MODIFY_VERSION property

PR_CURRENT_VERSION

[New - Windows 95]

Contains the current version number of a given message store.

Details

Identifier 0x0E00; property type PT_I8; property tag 0x0E000014

Comments

MAPI uses this property to support synchronization between message stores.

See Also

PR_CREATION_VERSION property, PR_MODIFY_VERSION property

PR_DEF_CREATE_DL

[New - Windows 95]

Contains the entry identifier for the template for a default distribution list object. Used by the client to create a distribution list within the container.

Details

Identifier 0x3611; property type PT_BINARY; property tag 0x36110102

Comments

Specifies the entry that can appear in the PR_CREATE_TEMPLATES property for distribution lists. After you obtain the entry, you use it in the call to CreateEntry. The entry represents the template for the default distribution list.

Your provider's container furnishes this property and furnishes the associated PR_CREATE_TEMPLATES property to the address book container.

See Also

IABLogon::CompareEntryIDs method, PR_CREATE_TEMPLATES property, PR_DEF_CREATE_MAILUSER property

PR_DEF_CREATE_MAILUSER

[New - Windows 95]

Contains the entry identifier for the template for a default mail user object. Used by the client to create a mail user within the container.

Details

Identifier 0x3612; property type PT_BINARY; property tag 0x36120102

Comments

Specifies the entry that can appear in the PR_CREATE_TEMPLATES property for mail users. After obtaining the entry, the client uses it in the call to CreateEntry. The entry represents the template for the default mail user.

Your provider's container furnishes this property and the associated PR_CREATE_TEMPLATES property to the address book container.

See Also

IABLogon::CompareEntryIDs method, PR_CREATE_TEMPLATES property, PR_DEF_CREATE_DL property

PR_DEFAULT_PROFILE

[New - Windows 95]

Contains TRUE if a messaging user profile is the MAPI default profile, and FALSE otherwise.

Details

Identifier 0x3D04; property type PT_BOOLEAN; property tag 0x3D04000B

Comments

This property does not appear as a property of any object; it appears as a column in a profiles table.

See Also

PR_DEFAULT_STORE property

PR_DEFAULT_STORE

[New - Windows 95]

Contains TRUE if a message store is the default message store in the message store table, and FALSE otherwise.

Details

Identifier 0x3400; property type PT_BOOLEAN; property tag 0x3400000B

Comments

This property appears as a column in the message stores table. The value is based on PR_RESOURCE_FLAGS.

See Also

PR_RESOURCE_FLAGS property

PR_DEFAULT_VIEW_ENTRYID

[New - Windows 95]

Contains the entryid for the default folder view that should be set as the initial view.

Details

Identifier 0x3616; property type PT_BINARY; property tag 0x36160102

Comments

PR_DEFAULT_VIEW_ENTRYID is the entry identifier of the folder view that should be set as the initial view. The property is not set when the "Normal" view is to be used as the initial view.

Because the client obtains this property at the time it opens the folder, the client can achieve significant performance gains. The PR_DEFAULT_VIEW_ENTRYID property can be used as a shortcut to obtain the default view, instead of opening the associated contents table and submitting a restriction.

A provider implementation of **IMAPIFolder::CopyFolder** can choose to copy this property when it copies folders.

See Also

IMAPIFolder::CopyFolder method, PR_COMMON_VIEWS_ENTRYID property, PR_VIEWS_ENTRYID property

PR_DEFERRED_DELIVERY_TIME

[New - Windows 95]

Contains the time at which a message sender wants a message delivered.

Details

Identifier 0x000F; property type PT_SYSTIME; property tag 0x000F0040

Comments

For X.400 environments, PR_DEFERRED_DELIVERY_TIME corresponds to the MH_T_DEFERRED_DELIVERY_TIME attribute as defined by the X.400 message-handling standard.

Note that MAPI does not perform the deferred delivery; it is an option of the underlying messaging system to handle deferred delivery.

See Also

PR_MESSAGE_DELIVERY_TIME property

PR_DELETE_AFTER_SUBMIT

[New - Windows 95]

Contains TRUE if a client application wants MAPI to delete the associated message after submission, and FALSE otherwise.

Details

Identifier 0x0E01; property type PT_BOOLEAN; property tag 0x0E01000B

Comments

The application uses this 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.

The IPC message is created in the receive folder for the message class.

See Also

PR_SENTMAIL_ENTRYID property

PR_DELIVER_TIME

[New - Windows 95]

Appears in reports; contains the time at which the original message was delivered.

Details

Identifier 0x0010; property type PT_SYSTIME; property tag 0x00100040

Comments

This MAPI property represents the corresponding delivered message and report per-recipient (delivery reports only) attributes as described in the X.400 message-handling standard. PR_DELIVER_TIME is a per recipient property on the report that indicates the time the original message was delivered to that user/recipient.

See Also

IMAPISupport::StatusRecips method

PR_DELTAX

[New - Windows 95]

Contains the width of a control in standard Windows dialog units.

Details

Identifier 0x3F03; property type PT_LONG; property tag 0x3F030003

Comments

The PR_XPOS, PR_YPOS, PR_DELTAX, and PR_DELTAY properties position and size the control.

See Also

PR_CONTROL_TYPE property, PR_DELTAY property, PR_XPOS property, PR_YPOS property

PR_DELTAY

[New - Windows 95]

Contains the height of a control in standard Windows dialog units.

Details

Identifier 0x3F04; property type PT_LONG; property tag 0x3F040003

Comments

The PR_XPOS, PR_YPOS, PR_DELTAX, and PR_DELTAY properties position and size the control.

See Also

PR_DELTAX property, PR_XPOS property, PR_YPOS property

PR_DEPARTMENT_NAME

[New - Windows 95]

Contains a name for the department in which a messaging user works.

Details

Identifier 0x3A18; property type PT_TSTRING; property tag 0x3A18001E

Comments

Several properties are provided for information about messaging users, including several telephone numbers and fields of the user's address.

See Also

PR_LOCATION property

PR_DEPTH

[New - Windows 95]

Represents the relative level of indentation (depth) of an object in a hierarchy table. It also represents the categorization level of a row in a contents table. The depth is zero-based, where zero represents the leftmost category.

Details

Identifier 0x3005; property type PT_LONG; property tag 0x30050003

Comments

PR_DEPTH can also specify the categorization level of a row in a contents table or the hierarchy depth in a hierarchy table. 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 you retrieved the hierarchy table. The depth does not represent an absolute depth from the root container.

See Also

PR_OBJECT_TYPE property

PR_DETAILS_TABLE

[New - Windows 95]

Contains an embedded display table object.

Details

Identifier 0x3605; property type PT_OBJECT; property tag 0x3605000D

Comments

Passing this property to the IMAPIProp::OpenProperty method for the object returns an IMAPITable interface that allows creation of the display table.

See Also

IAddrBook::RecipOptions method, **IMAPISession::MessageOptions** method, PR_CREATE_TEMPLATES property, PR_SEARCH property

PR_DISC_VAL

[New - Windows 95]

Contains TRUE if a message has been delivered to a distribution list of recipients but cannot be delivered to an individual messaging user in the list. The property contains FALSE otherwise.

Details

Identifier 0x004A; property type PT_BOOLEAN; property tag 0x004A000B

Comments

Your application can also use PR_DISC_VAL in normal message submission to identify the recipients for which a particular transport driver has accepted messages.

See Also

PR_DL_EXPANSION_HISTORY property

PR_DISCARD_REASON

[New - Windows 95]

Contains a reason why a message transfer agent (MTA) has discarded a given message.

Details

Identifier 0x0011; property type PT_LONG; property tag 0x00110003

Comments

This reason is used in a nondelivery report (NDR) for the message. This MAPI property corresponds to the interpersonal notification IM_DISCARD_REASON attribute as defined by the X.400 message-handling standard.

See Also

PR_NDR_REASON_CODE property

PR_DISCLOSE_RECIPIENTS

[New - Windows 95]

Contains TRUE if disclosure of message recipients is allowed, and FALSE otherwise. Disclosure is the default for messages sent to messaging users.

Details

Identifier 0x3A04; property type PT_BOOLEAN; property tag 0x3A04000B

Comments

For X.400 environments, PR_DISCLOSE_RECIPIENTS corresponds to the MH_T_DISCLOSURE_ALLOWED attribute as defined by the X.400 message-handling standard.

See Also

PR_DISCLOSURE_OF_RECIPIENTS property

PR_DISCLOSURE_OF_RECIPIENTS

[New - Windows 95]

Contains TRUE if disclosure of recipients is allowed, and FALSE otherwise.

Details

Identifier 0x0012; property type PT_BOOLEAN; property tag 0x0012000B

Comments

For X.400 environments, PR_DISCLOSURE_OF_RECIPIENTS corresponds to the MH_T_DISCLOSURE_ALLOWED attribute.

See Also

PR_DISCLOSE_RECIPIENTS property

PR_DISCRETE_VALUES

[New - Windows 95]

Contains TRUE if the report applies only to discrete members of a group rather than all members of the group. The recipient table contains entries indicating the entry identifier and name values of the recipients that the report applies to.

Details

Identifier 0x0E0E; property type PT_BOOLEAN; property tag 0x0E0E000B

Comments

PR_DISCRETE_VALUES provides the mechanism for generating non-delivery reports for individuals within a group.

This property is not used in MAPI 1.0. This property is being evaluated to determine whether it will be supported in future releases.

For example, consider a message that is sent to a group and one of the users in the group generates a non-delivery report (NDR). The recipient table of the NDR includes the PR_DISCRETE_VALUES property set to TRUE. The value TRUE indicated that the report's values for PR_ORIGINAL_DISPLAY_NAME, PR_ORIGINAL_SEARCH_KEY and PR_ORIGINAL_ENTRYID are significant.

This property is not transmitted so does not exist on the recipient.

See Also

PR_ORIGINAL_SEARCH_KEY property

PR_DISPLAY_BCC

[New - Windows 95]

Contains an ASCII list of the display names of any blind copy (BCC) message recipients, separated by semicolons (;).

Details

Identifier 0x0E02; property type PT_TSTRING; property tag 0x0E02001E

Comments

The message store computes PR_DISPLAY_BCC using the IMessage::ModifyRecipients method. The 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 the call to IMAPIProp::SaveChanges.

Note that semicolons cannot be used within recipient names in MAPI messaging.

For X.400 environments, PR_DISPLAY_BCC corresponds to the IM_BLIND_COPY_RECIPIENTS attribute as defined by the X.400 message-handling standard.

See Also

IMessage::ModifyRecipients method, PR_DISPLAY_NAME property

PR_DISPLAY_CC

[New - Windows 95]

Contains an ASCII list of the names of carbon copy (CC) message recipients, separated by semicolons (;).

Details

Identifier 0x0E03; property type PT_TSTRING; property tag 0x0E03001E

Comments

The message store computes PR_DISPLAY_CC using the IMessage::ModifyRecipients method. The store also maintains the property so that it always reflects the last saved state of a message.

Note that semicolons cannot be used within recipient names in MAPI messaging.

For X.400 environments, PR_DISPLAY_CC corresponds to the IM_COPY_RECIPIENTS attribute as defined by the X.400 message-handling standard.

See Also

IMessage::ModifyRecipients method, PR_DISPLAY_NAME property

PR_DISPLAY_NAME

[New - Windows 95]

Contains the display name for a given MAPI object.

Details

Identifier 0x3001; property type PT_TSTRING; property tag 0x3001001E

Comments

A status object should furnish PR_DISPLAY_NAME. For this object, the property contains the name of the component that can be displayed by the user interface.

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. Note that this restriction does not apply to other containers, such as address books and distribution lists.

Note that semicolons cannot be used within recipient names in MAPI messaging. This is because some properties consist of a list of display names delimited by semicolons.

To distinguish providers of the same type, 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 the providers. Unconfigured providers should use a string that names the provider. Configured providers should have use the same string followed by a distinguishing string in parenthesis. For example, the unconfigured provider sets PR_DISPLAY_NAME to the string, "Personal Information Store", while the configured provider sets PR_DISPLAY_NAME to the string, "Personal Information Store (July '95)".

For recipients in X.400 environments, PR_DISPLAY_NAME corresponds to the IM_FREE_FORM_NAME attribute as defined by the X.400 message-handling standard.

See Also

PR_DISPLAY_BCC property, PR_EMAIL_ADDRESS property,
PR_TRANSMITTABLE_DISPLAY_NAME property

PR_DISPLAY_TO

[New - Windows 95]

Contains an ASCII list of the names of the primary message (To line) recipients, separated by semicolons (;).

Details

Identifier 0x0E04; property type PT_TSTRING; property tag 0x0E04001E

Comments

The message store computes PR_DISPLAY_TO using the IMessage::ModifyRecipients method. The store also maintains the property so that it always reflects the last saved state of a message.

Note that semicolons cannot be used within recipient names in MAPI messaging.

For X.400 environments, PR_DISPLAY_TO corresponds to the IM_PRIMARY_RECIPIENTS attribute as defined by the X.400 message-handling standard.

See Also

PR_DISPLAY_TYPE property

PR_DISPLAY_TYPE

[New - Windows 95]

Contains a value used to associate an icon with that particular row of the table.

Details

Identifier 0x3900; property type PT_LONG; property tag 0x39000003

Comments

Contains a long integer that is used to associate an icon (or other display elements) with that row of the table.

This property is used on address book and hierarchy table rows.It is not extensible in MAPI 1.0.

Hierarchy table settings include DT_FOLDER and DT_FOLDER_LINK. Address book hierarchy tables and template tables include DT_MODIFIABLE, DT_GLOBAL, DT_LOCAL, and DT_WAN. Address book tables and template tables use the other values listed below.

The PR_DISPLAY_TYPE property can have the following values:

Value	**Description**
DT_MAILUSER	A typical messaging user.
DT_AGENT	An automated agent, such as Quote-Of-The-Day or a weather chart display.
DT_DISTLIST	A distribution list.
DT_FOLDER	A folder hierarchy table. Display default folder icon adjacent to folder.
DT_FOLDER_LINK	A folder hierarchy table. Display default folder link icon adjacent to folder rather than the default folder icon.
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_MODIFIABLE	Modifiable; the container should be denoted as modifiable in the UI.
DT_NOTSPECIFIC	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.

See Also

PR_DISPLAY_TO property

PR_DL_EXPANSION_HISTORY

[New - Windows 95]

Contains a history showing how a distribution list has been expanded during message transmission.

Details

Identifier 0x0013; property type PT_BINARY; property tag 0x00130102

Comments

For X.400 environments, PR_DL_EXPANSION_HISTORY corresponds to the MH_T_EXPANSION_HISTORY attribute. X.400 and EDI transport providers use this property for messages delivered to client applications. X.400 probe messages also use this property.

See Also

PR_DL_EXPANSION_PROHIBITED property

PR_DL_EXPANSION_PROHIBITED

[New - Windows 95]

Contains TRUE if a message transfer agent (MTA) is prohibited from expanding distribution lists.

Details

Identifier 0x0014; property type PT_BOOLEAN; property tag 0x0014000B

Comments

For X.400 environments, PR_DL_EXPANSION_PROHIBITED corresponds to the MH_T_EXPANSION_PROHIBITED attribute as defined by the X.400 message-handling standard.

See Also

PR_DL_EXPANSION_HISTORY property

PR_EMAIL_ADDRESS

[New - Windows 95]

Contains a null-terminated string that specifies the e-mail address of the message recipient in the format required by the messaging system.

Details

Identifier 0x3003; property type PT_TSTRING; property tag 0x3003001E

Comments

PR_EMAIL_ADDRESS is a null-terminated string whose format has meaning only for the underlying messaging service.

Mail user objects must furnish PR_EMAIL_ADDRESS. This property is optional for distribution list objects.

PR_EMAIL_ADDRESS is used in conjunction with PR_ADDRTYPE and PR_MESSAGE_CLASS in addressing messages. The string format is qualified by the PR_ADDRTYPE property.

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
```

See Also

PR_ADDRTYPE property, PR_MESSAGE_CLASS property, PR_SEARCH_KEY property

PR_END_DATE

[New - Windows 95]

Contains the ending time of an appointment as managed by a scheduling application.

Details

Identifier 0x0061; property type PT_SYSTIME; property tag 0x00610040

Comments

Several MAPI properties treat appointments: PR_START_DATE, PR_END_DATE, and PR_OWNER_APPT_ID.

See Also

PR_OWNER_APPT_ID property, PR_START_DATE property

PR_ENTRYID

[New - Windows 95]

Contains a MAPI entry identifier used to open and edit properties of a particular MAPI object.

Details

Identifier 0x0FFF; property type PT_BINARY; property tag 0x0FFF0102

Comments

Mail user, message, distribution list, address book container, folder, and message store objects must furnish PR_ENTRYID. It is an optional property for the status object.

This property can contain either a long-term or a short-term ID. To convert from a short term entry id to a long term entry id, call the object's GetProps method and retrieve PR_ENTRYID.

See Also

ENTRYID structure, **NOTIFICATION** structure, PR_STORE_ENTRYID property

PR_EXPIRY_TIME

[New - Windows 95]

Contains the date and time at which the messaging system can invalidate the content of the associated message.

Details

Identifier 0x0015; property type PT_SYSTIME; property tag 0x00150040

Comments

The property is used to direct the messaging system in handling delivered interpersonal messages. For X.400 environments, this MAPI property corresponds to the delivery envelope per-message IM_EXPIRY_TIME attribute as defined by the X.400 message-handling standard.

Transport providers use PR_EXPIRY_TIME for messages delivered to client applications.

See Also

PR_PROVIDER_SUBMIT_TIME property

PR_EXPLICIT_CONVERSION

[New - Windows 95]

Contains an indication that a message sender specifically requests a message content conversion for a particular recipient.

Details

Identifier 0x0C01; property type PT_LONG; property tag 0x0C010003

Comments

This property represents the X.400 probe submission MH_T_EXPLICIT_CONVERSION attribute as defined in the X.400 message-handling standard.

It also represents the corresponding EDI submission envelope per-recipient attribute as defined by the EDI message-handling standard.

See Also

PR_GENERATION property

PR_FILTERING_HOOKS

[New - Windows 95]

Not supported in MAPI version 1.0.

Details

Identifier 0x3D08; property type PT_BINARY; property tag 0x3D080102

Comments

This property, which was present in pre-release versions, is not supported in the MAPI 1.0 SDK.

PR_FINDER_ENTRYID

[New - Windows 95]

Contains the entry identifier for the folder in which search results are typically created.

Details

Identifier 0x35E7; property type PT_BINARY; property tag 0x35E70102

Comments

The entryid has the same format as the MAPI 1.0 ENTRYID structure.

See Also

PR_ENTRYID property

PR_FOLDER_ASSOCIATED_CONTENTS

[New - Windows 95]

Contains the associated contents table object. This object represents a subfolder that does not appear in the hierarchy.

Details

Identifier 0x3610; property type PT_OBJECT; property tag 0x3610000D

Comments

The associated contents table enumerates associated (hidden) messages in a folder.

Your application can use the PR_FOLDER_ASSOCIATED_CONTENTS property to exclude an item from a copy operation.

Several MAPI properties provide access to tables:

Property	Table
PR_CONTAINER_HIERARCHY	Hierarchy table
PR_FOLDER_ASSOCIATED_CONTENTS	Associated contents table
PR_MESSAGE_ATTACHMENTS	Attachments table
PR_MESSAGE_RECIPIENTS	Recipients table

See Also

PR_ASSOC_CONTENT_COUNT property, PR_CONTAINER_HIERARCHY property

PR_FOLDER_TYPE

[New - Windows 95]

Contains a constant that indicates the folder type. All folders must furnish this property.

Details

Identifier 0x3601; property type PT_LONG; property tag 0x36010003

Comments

The PR_FOLDER_TYPE property can have the following values:

Value	Description
FOLDER_GENERIC	A normal 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). This folder contains messages and other folders. MAPI allows only one root folder per message store. Note that PR_PARENT_ENTRYID does not exist on the root folder.
FOLDER_SEARCH	A search-results folder containing the results of a search, in the form of links to messages that meet search criteria. The information in a search-results folder is mainly stored in its contents table, which contains the same columns as a normal contents table, as well as two extra columns: PR_PARENT_DISPLAY (display name of the folder of the parent) and PR_PARENT_ENTRYID (the parent folder entryid of that message). PR_PARENT_DISPLAY is the display name of the parent of the folder, and is a required column in the search results folder contents table.

See Also

PR_PARENT_ENTRYID property

PR_FORM_CATEGORY

[New - Windows 95]

Contains the category of a form.

Details

Identifier 0x3309; property type PT_TSTRING; property tag 0x3309001E

Comments

The category name is defined by a client application in a way that is appropriate to the application.

See Also

PR_FORM_CATEGORY_SUB property

PR_FORM_CATEGORY_SUB

[New - Windows 95]

Contains the subcategory of a form, as defined by a client application.

Details

Identifier 0x3310; property type PT_TSTRING; property tag 0x3310001E

Comments

The PR_FORM_CATEGORY_SUB category is subordinate to the main form category provided in a PR_FORM_CATEGORY property.

See Also

PR_FORM_CATEGORY property

PR_FORM_CLSID

[New - Windows 95]

Contains the 128-bit OLE globally unique identifier (GUID) of a form.

Details

Identifier 0x3307; property type PT_CLSID; property tag 0x33070048

Comments

The MAPIUID structure contains the definition for the UID.

See Also

MAPIUID structure

PR_FORM_CONTACT_NAME

[New - Windows 95]

Contains the name of a contact for information concerning a form.

Details

Identifier 0x3308; property type PT_TSTRING; property tag 0x3308001E

Comments

The contact typically contains the name of a person or an alias that is responsible for the form.

See Also

PR_FORM_DESIGNER_NAME property

PR_FORM_DESIGNER_GUID

[New - Windows 95]

Contains the unique identifier for the object used to design the form.

Details

Identifier 0x3314; property type PT_CLSID; property tag 0x33140048

Comments

Usually contains the GUID of the design program used to create the form. The property may be empty.

The MAPIUID structure contains the definition of the UID.

See Also

PR_FORM_DESIGNER_NAME property

PR_FORM_DESIGNER_NAME

[New - Windows 95]

Contains the display name for the object used to design the form.

Details

Identifier 0x3313; property type PT_TSTRING; property tag 0x3313001E

Comments

The property PR_FORM_DESIGNER_GUID contains the unique identifier for the form designer object.

See Also

PR_FORM_DESIGNER_GUID property

PR_FORM_HIDDEN

[New - Windows 95]

Specifies whether the form is to be displayed by compose menus and dialogs.

Details

Identifier 0x3312; property type PT_BOOLEAN; property tag 0x3312000B

Comments

Form-related properties are read-only.

See Also

PR_FORM_CATEGORY property

PR_FORM_HOST_MAP

[New - Windows 95]

Contains a host map of available forms.

Details

Identifier 0x3311; property type PT_LONG; property tag 0x33110003

Comments

An application should update this property, along with the PR_DISPLAY_NAME property, when changing the underlying structure in the IMAPIFormProp interface.

See Also

PR_DISPLAY_NAME property

PR_FORM_MESSAGE_BEHAVIOR

[New - Windows 95]

Indicates where the message should be composed.

Details

Identifier 0x3315; property type PT_BOOLEAN; property tag 0x3315000B

Comments

A value of TRUE indicates that the form should be composed in the current folder. A value of FALSE indicates that the message should be composed as any other IPM message.

See Also

PR_FORM_CATEGORY property

PR_FORM_VERSION

[New - Windows 95]

Contains the version number of the message store that was current at the time the specified form was created.

Details

Identifier 0x3306; property type PT_TSTRING; property tag 0x3306001E

Comments

The form version number is a copy of the message store version number contained in the PR_CURRENT_VERSION property.

See Also

PR_CURRENT_VERSION property

PR_GENERATION

[New - Windows 95]

Contains a generational abbreviation (for example, Jr., Sr., III) that follows the full name of the messaging user.

Details

Identifier 0x3A05; property type PT_TSTRING; property tag 0x3A05001E

Comments

For X.400 environments, PR_GENERATION corresponds to the MH_T_GENERATION attribute as defined by the X.400 message-handling standard.

Several properties are available for the messaging user, including several telephone numbers and fields of the user's home address.

See Also

PR_GIVEN_NAME property

PR_GIVEN_NAME

[New - Windows 95]

Contains the first name of a messaging user.

Details

Identifier 0x3A06; property type PT_TSTRING; property tag 0x3A06001E

Comments

For X.400 environments, PR_GIVEN_NAME corresponds to the MH_T_GIVEN_NAME attribute as defined by the X.400 message-handling standard.

See Also

PR_SURNAME property

PR_GOVERNMENT_ID_NUMBER

[New - Windows 95]

Contains a government identifier, such as a Social Security number, for a messaging user.

Details

Identifier 0x3A07; property type PT_TSTRING; property tag 0x3A07001E

Comments

Several properties are provided for the messaging user, including telephone numbers and fields of the user's home address.

See Also

PR_ORGANIZATIONAL_ID_NUMBER property

PR_HASATTACH

[New - Windows 95]

Contains TRUE if a given message contains at least one attachment and FALSE otherwise.

Details

Identifier 0x0E1B; property type PT_BOOLEAN; property tag 0x0E1B000B

Comments

The message store computes the PR_HASATTACH property from the PR_MESSAGE_FLAGS property. Your application can then use PR_HASATTACH to sort on message attachments in a message viewer.

The value of PR_HASATTACH is updated with SaveChanges.

See Also

PR_MESSAGE_FLAGS property

PR_HEADER_FOLDER_ENTRYID

[New - Windows 95]

Contains the entry identifier that the service provider furnishes for a folder.

Details

Identifier 0x3E0A; property type PT_BINARY; property tag 0x3E0A0102

Comments

This property is specific to remote transports that support remote operations using a folder. The property appears in the status object or in the status table row.

When you call the OpenEntry method for this entryid, you obtain a folder from which you can obtain a contents table. The client application can use this identifier to view the headers of the messages in the folder.

Another way to obtain the contents table is to open the status object and query for the folder object interface. This property was designed for a transport tightly-coupled to a store; the transport could publish the store's folder entryid in this property.

See Also

PR_ENTRYID property

PR_HOME_FAX_NUMBER

[New - Windows 95]

Contains the phone number for the messaging user's home fax machine.

Details

Identifier 0x3A25; property type PT_TSTRING; property tag 0x3A25001E

Comments

Several properties are available for information about the messaging user, including various phone numbers.

See Also

PR_HOME_TELEPHONE_NUMBER property

PR_HOME_TELEPHONE_NUMBER

[New - Windows 95]

Contains the home telephone number of a messaging user.

Details

Identifier 0x3A09; property type PT_TSTRING; property tag 0x3A09001E

Comments

Several properties are available for information about the messaging user, including various phone numbers.

See Also

PR_HOME_FAX_NUMBER property, PR_HOME2_TELEPHONE_NUMBER property

PR_HOME2_TELEPHONE_NUMBER

[New - Windows 95]

Contains a second home telephone number of a messaging user.

Details

Identifier 0x3A2F; property type PT_TSTRING; property tag 0x3A2F001E

Comments

Should contain information only if PR_HOME_TELEPHONE_NUMBER also contains information.

See Also

PR_HOME_TELEPHONE_NUMBER property

PR_ICON

[New - Windows 95]

Contains a 32 by 32 icon that is used on forms.

Details

Identifier 0x0FFD; property type PT_BINARY; property tag 0x0FFD0102

Comments

The value is the same as the contents of a .ICO file, that is, a bitmap.

See Also

PR_MINI_ICON property

PR_IDENTITY_DISPLAY

[New - Windows 95]

Contains the display name that corresponds to a messaging user's unique identifier (UID).

Details

Identifier 0x3E00; property type PT_TSTRING; property tag 0x3E00001E

Comments

The PR_IDENTITY_* properties define the user's identity as that provider defines it. These optional properties typically refer to the user's account on the server, but can refer to any representation for that user in the messaging system, as defined by the provider.

This property represents one of three maintained for the user. Opening the entryid to find additional information is a relatively expensive operation, so properties representing the commonly-used display name and search key are defined along with the entryid.

Providers that belong to the same messaging service should expose the same values for the PR_IDENTITY_* properties.

See Also

IMAPISession::QueryIdentity method, PR_IDENTITY_ENTRYID property, PR_IDENTITY_SEARCH_KEY property

PR_IDENTITY_ENTRYID

[New - Windows 95]

Contains the entry identifier for the user as that provider defines it.

Details

Identifier 0x3E01; property type PT_BINARY; property tag 0x3E010102

Comments

The PR_IDENTITY_* properties define the user's identity as that provider defines it. These optional properties typically refer to the user's account on the server, but can refer to any representation for that user in the messaging system, as defined by the provider.

This property represents one of three maintained for the user. Opening the entryid to find additional information is a relatively expensive operation, so properties representing the commonly-used display name and search key are defined along with the entryid.

Providers that belong to the same messaging service should expose the same values for the PR_IDENTITY_* properties.

See Also

IMAPISession::QueryIdentity method, PR_IDENTITY_DISPLAY property, PR_IDENTITY_SEARCH_KEY property

PR_IDENTITY_SEARCH_KEY

[New - Windows 95]

Contains the search key for the user.

Details

Identifier 0x3E05; property type PT_BINARY; property tag 0x3E050102

Comments

The PR_IDENTITY_* properties define the user's identity as that provider defines it. These optional properties typically refer to the user's account on the server, but can refer to any representation for that user in the messaging system, as defined by the provider.

This property represents one of three maintained for the user. Opening the entryid to find additional information is a relatively expensive operation, so properties representing the commonly-used display name and search key are defined along with the entryid.

Providers that belong to the same messaging service should expose the same values for the PR_IDENTITY_* properties.

See Also

IMAPISession::QueryIdentity method, PR_IDENTITY_DISPLAY property, PR_IDENTITY_ENTRYID property

PR_IMPLICIT_CONVERSION_PROHIBITED

[New - Windows 95]

Contains TRUE if a message transfer agent (MTA) is prohibited from implicitly converting message content.

Details

Identifier 0x0016; property type PT_BOOLEAN; property tag 0x0016000B

Comments

For X.400 environments, PR_IMPLICIT_CONVERSION_PROHIBITED corresponds to the MH_T_CONVERSION_PROHIBITED attribute.

X.400 and EDI transport providers use this property for messages delivered to client applications. X.400 probe messages also use this property.

See Also

PR_REPORTING_MTA_CERTIFICATE property

PR_IMPORTANCE

[New - Windows 95]

Contains a value indicating the message sender's opinion of the importance of a message.

Details

Identifier 0x0017; property type PT_LONG; property tag 0x00170003

Comments

This MAPI property corresponds to the submission envelope per-message IM_IMPORTANCE attribute as defined by the X.400 message-handling standard. X.400 transport providers use this property for messages delivered to client applications.

PR_IMPORTANCE is distinguished from PR_PRIORITY. 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 different display by the user interface.

X.400 defines the following values for this property:

Value	Description
IMPORTANCE_LOW	The message has low importance.
IMPORTANCE_HIGH	The message has high importance.
IMPORTANCE_NORMAL	The message has normal importance.

See Also

PR_PRIORITY property

PR_INCOMPLETE_COPY

[New - Windows 95]

Contains TRUE if a message is an incomplete copy of another message, and FALSE otherwise.

Details

Identifier 0x0035; property type PT_BOOLEAN; property tag 0x0035000B

Comments

This MAPI property corresponds to the IM_INCOMPLETE_COPY attribute as defined by the X.400 message-handling standard.

PR_INITIAL_DETAILS_PANE

[New - Windows 95]

Indicates which property page in a property sheet to display first.

Details

Identifier 0x3F08; property type PT_LONG; property tag 0x3F080003

Comments

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 to display first. This property appears in an object from which you can get back a dsiplay table.

See Also

IMAPISupport::DoConfigPropSheet method

PR_INITIALS

[New - Windows 95]

Contains the initials for parts of the full name of a messaging user.

Details

Identifier 0x3A0A; property type PT_TSTRING; property tag 0x3A0A001E

Comments

For X.400 environments, PR_INITIALS corresponds to the MH_T_INITIALS attribute as defined by the X.400 message-handling standard.

Several properties are supplied for the messaging user, including several telephone numbers, and parts of the messaging user's address.

See Also

PR_OFFICE_TELEPHONE_NUMBER property

PR_INSTANCE_KEY

[New - Windows 95]

Represents the unique value that represents a row in a table. These are used in contents and categorization tables, and are not used in display tables.

Details

Identifier 0x0FF6; property type PT_BINARY; property tag 0x0FF60102

Comments

When you explode multivalued properties in a table, you obtain different rows and different instance keys for the properties. Rows added to categorized tables that don't correspond to actual data also get their own unique instance key.

PR_INSTANCE_KEY is also used with notifications. The PR_INSTANCE_KEY property is used as the contents of the TABLE_NOTIFICATION fields propIndex and propPrior. The propIndex field indicates the value of the index property for the row that changed. The propPrior field indicates the value of the index for the row before the changed row (PR_NULL indicates a change to the first row).

This value is not copied as part of the display table.

All instance keys can be compared as binary values.

See Also

TABLE_NOTIFICATION structure

PR_IPM_ID

[New - Windows 95]

Contains an interpersonal messaging (IPM) identifier that transport providers use for messages delivered to client applications.

Details

Identifier 0x0018; property type PT_BINARY; property tag 0x00180102

Comments

This MAPI property corresponds to the IM_THIS_IPM attribute as defined by the X.400 message-handling standard.

This property is similar to the ENTRYID used by MAPI.

See Also

ENTRYID structure

PR_IPM_OUTBOX_ENTRYID

[New - Windows 95]

Contains the entry identifier of an interpersonal messaging (IPM) folder in which outbound messages are usually created.

Details

Identifier 0x35E2; property type PT_BINARY; property tag 0x35E20102

Comments

The Outbox folder is the folder in which outbound messages are usually created.

See Also

PR_IPM_OUTBOX_SEARCH_KEY property

PR_IPM_OUTBOX_SEARCH_KEY

[New - Windows 95]

This property is not supported in MAPI 1.0.

Details

Identifier 0x3411; property type PT_BINARY; property tag 0x34110102

Comments

Although present in pre-release versions, this property is not supported in MAPI 1.0.

In previous releases, this property contained the search key for an interpersonal messaging (IPM) outbound messages (outbox) folder.

See Also

PR_IPM_OUTBOX_ENTRYID property

PR_IPM_RETURN_REQUESTED

[New - Windows 95]

Contains TRUE if a message should be returned with a report, and FALSE otherwise.

Details

Identifier 0x0C02; property type PT_BOOLEAN; property tag 0x0C02000B

Comments

This MAPI property corresponds to the submission envelope per-recipient IM_IPM_RETURN_REQUESTED attribute as defined by the X.400 message-handling standard.

See Also

PR_IPM_SENTMAIL_ENTRYID property

PR_IPM_SENTMAIL_ENTRYID

[New - Windows 95]

Contains the entry identifier of an interpersonal messaging (IPM) folder into which the client places sent mail.

Details

Identifier 0x35E4; property type PT_BINARY; property tag 0x35E40102

Comments

The Sent Mail folder is the folder in which sent messages are usually placed.

The client can use this property to set the PR_SENTMAIL_ENTRYID property of a message.

See Also

PR_IPM_SENTMAIL_SEARCH_KEY property, PR_SENTMAIL_ENTRYID property

PR_IPM_SENTMAIL_SEARCH_KEY

[New - Windows 95]

This property is not supported in MAPI 1.0.

Details

Identifier 0x3413; property type PT_BINARY; property tag 0x34130102

Comments

Although present in pre-release versions, this property is not supported in MAPI 1.0.

In previous releases, this message store object property contained the search key for an interpersonal messaging (IPM) sent-mail folder.

See Also

PR_IPM_SENTMAIL_ENTRYID property

PR_IPM_SUBTREE_ENTRYID

[New - Windows 95]

Contains the entry identifier of the base of the interpersonal messaging (IPM) folder subtree in the message store's folder tree.

Details

Identifier 0x35E0; property type PT_BINARY; property tag 0x35E00102

Comments

This property represents the base of the IPM hierarchy. IPM clients should not display any folders that are not children of the folder represented by this property.

See Also

PR_IPM_SUBTREE_SEARCH_KEY property

PR_IPM_SUBTREE_SEARCH_KEY

[New - Windows 95]

This property is not supported in MAPI 1.0.

Details

Identifier 0x3410; property type PT_BINARY; property tag 0x34100102

Comments

Although present in pre-release versions, this property is not supported in MAPI 1.0.

In previous releases, this message store object property contained the search key for the root folder of the interpersonal messaging (IPM) folder subtree in the message store's folder tree.

See Also

PR_IPM_SUBTREE_ENTRYID property

PR_IPM_WASTEBASKET_ENTRYID

[New - Windows 95]

Contains the entry identifier of an interpersonal messaging (IPM) folder into which deleted messages are usually placed.

Details

Identifier 0x35E3; property type PT_BINARY; property tag 0x35E30102

Comments

A third-party application behaving like a MAPI mail client should move IPM messages to the deleted messages folder indicated by this property. If a message is already in this folder, or if the PR_IPM_WASTEBASKET_ENTRYID property is not supported, your application should delete the message.

See Also

PR_ENTRYID property

PR_IPM_WASTEBASKET_SEARCH_KEY

[New - Windows 95]

This property is not supported in MAPI 1.0.

Details

Identifier 0x3412; property type PT_BINARY; property tag 0x34120102

Comments

Although present in pre-release versions, this property is not supported in MAPI 1.0.

In previous releases, this message store object property contained the search key for an interpersonal messaging (IPM) deleted messages (wastebasket) folder.

See Also

PR_IPM_WASTEBASKET_ENTRYID property

PR_ISDN_NUMBER

[New - Windows 95]

Contains the messaging user's ISDN-capable telephone number.

Details

Identifier 0x3A2D; property type PT_STRING; property tag 0x3A2D001E

Comments

Several properties are available for the messaging user, including the office telephone number, fax number, and cellular telephone number.

See Also

PR_OFFICE_TELEPHONE_NUMBER property

PR_KEYWORD

[New - Windows 95]

Contains a keyword for arbitrary use by the system administrator.

Details

Identifier 0x3A0B; property type PT_TSTRING; property tag 0x3A0B001E

Comments

The contents of the string property are defined based on the needs of the specific   site.

Several properties are available for the messaging user, including the office telephone number, fax number, and cellular telephone number.

See Also

PR_OFFICE_TELEPHONE_NUMBER property

PR_LANGUAGE

[New - Windows 95]

Contains a value indicating the language in which the messaging user is writing messages.

Details

Identifier 0x3A0C; property type PT_TSTRING; property tag 0x3A0C001E

Comments

The string contains a single 2-character country code.

See Also

PR_LANGUAGES property

PR_LANGUAGES

[New - Windows 95]

Contains an ASCII list of the languages incorporated in a message. The underlying string is a sequence of 2-character country codes delimited by commas.

Details

Identifier 0x002F; property type PT_TSTRING; property tag 0x002F001E

Comments

For X.400 environments, PR_LANGUAGES corresponds to the IM_LANGUAGES attribute as defined by the X.400 message-handling standard.

See Also

PR_LANGUAGE property

PR_LAST_MODIFICATION_TIME

[New - Windows 95]

Contains the date and time the mail user object or distribution list object was last modified.

Details

Identifier 0x3008; property type PT_SYSTIME; property tag 0x30080040

Comments

PR_LAST_MODIFICATION_TIME is initially set to the same value as PR_CREATION_TIME. The client can set this property until the first call to SaveChanges.

See Also

PR_CREATION_TIME property

PR_LATEST_DELIVERY_TIME

[New - Windows 95]

Contains the latest date and time when a message transfer agent (MTA) should deliver a message.

Details

Identifier 0x0019; property type PT_SYSTIME; property tag 0x00190040

Comments

For X.400 environments, PR_LATEST_DELIVERY_TIME corresponds to the MH_T_LATEST_DELIVERY_TIME attribute.

If an MTA cannot deliver a message by the time this property specifies, it cancels the message without delivery.

See Also

PR_DEFERRED_DELIVERY_TIME property

PR_LOCALITY

[New - Windows 95]

Contains the name of the messaging user's locality, such as the town or city.

Details

Identifier 0x3A27; property type PT_TSTRING; property tag 0x3A27001E

Comments

Several properties are available for the messaging user, including the office telephone number.

See Also

PR_STREET_ADDRESS property

PR_LOCATION

[New - Windows 95]

Contains the location of the messaging user in a format that is useful to your organization.

Details

Identifier 0x3A0D; property type PT_TSTRING; property tag 0x3A0D001E

Comments

The contents of the string are defined by your particular needs. For example, some users might identify messaging users by specifying the building number and office number.

See Also

PR_BUSINESS_TELEPHONE_NUMBER property

PR_MAIL_PERMISSION

[New - Windows 95]

Contains TRUE if the messaging user has permissions for a message, and FALSE otherwise.

Details

Identifier 0x3A0E; property type PT_BOOLEAN; property tag 0x3A0E000B

Comments

Several properties are available for the messaging user, including several that represent telephone numbers and parts of the messaging user's address.

See Also

PR_OFFICE_TELEPHONE_NUMBER property

PR_MAPPING_SIGNATURE

[New - Windows 95]

Contains the mapping signature for named properties of a particular MAPI object. Objects that have the same signature represent the same property.

Details

Identifier 0x0FF8; property type PT_BINARY; property tag 0x0FF80102

Comments

Your application should check the PR_MAPPING_SIGNATURE of both objects when copying named properties from one object to another. Use of this property allows the application to avoid translating between the copied property's name and its identifier.

When two objects have the same PR_MAPPING_SIGNATURE value, you don't have to translate name to identifier and identifier to name. You can simply call SetProps. This is convenient for applications that perform custom copying of named properties, and for providers implementing CopyTo and CopyProps.

See Also

IMAPIProp::SetProps method

PR_MDB_PROVIDER

[New - Windows 95]

Contains a provider-defined identifier that indicates the type of the message store.

Details

Identifier 0x3414; property type PT_BINARY; property tag 0x34140102

Comments

The uid identifies the type of message store. The value is generated by, and is unique to, each provider. It is typically used for browsing through the message stores table to find a store of the desired type, such as public folders.

This property is analogous to the PR_AB_PROVIDER_ID property for address books.

Address book containers and hierarchy tables should provide this property. Your application can use it to find related rows in an address book hierarchy table.

See Also

PR_AB_PROVIDER_ID property

PR_MESSAGE_ATTACHMENTS

[New - Windows 95]

Contains a table of restrictions that can be applied to the folder's contents table to find all messages that contain the attachments specified in the restrictions.

Details

Identifier 0x0E13; property type PT_OBJECT; property tag 0x0E13000D

Comments

You can restrict the contents table of the folder to find all messages where the attachment meets the conditions specified in the restriction. The restriction is true if one row remains in the table.

The restrictions are commonly used to exclude attachments from **IMessage::CopyTo** operations and to include attachments in **IMessage::CopyProps** operations.

This property can also be used in the IMessage (and IMAPIProp) methods GetPropList, GetProps, and OpenProperty.

Note that using restrictions results in an open operation on every message in the folder. Depending on your application and the number of messages to be searched, it can affect application performance.

PR_MESSAGE_RECIPIENTS is used for recipients as PR_MESSAGE_ATTACHMENTS is used for attachments.

Several MAPI properties provide access to table objects:

Property	**Table**
PR_CONTAINER_HIERARCHY	Hierarchy table
PR_FOLDER_ASSOCIATED_CONTENTS	Associated contents table
PR_MESSAGE_ATTACHMENTS	Attachments table
PR_MESSAGE_RECIPIENTS	Recipients table

See Also

IMAPIProp::CopyProps method, **IMAPIProp::CopyTo** method, PR_MESSAGE_RECIPIENTS property

PR_MESSAGE_CC_ME

[New - Windows 95]

Set to TRUE if the messaging user was specifically named as a carbon copy (Cc) recipient and was not part of a group.

Details

Identifier 0x0058; property type PT_BOOLEAN; property tag 0x0058000B

Comments

PR_MESSAGE_CC_ME is 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, 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.

Unsent messages generally do not include this property. If the property 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.

See Also

PR_MESSAGE_RECIP_ME property, PR_MESSAGE_TO_ME property

PR_MESSAGE_CLASS

[New - Windows 95]

Contains a text string that identifies the sender-defined message class, such as "IPM.FAX". The message class specifies the type of the delivered message.

Details

Identifier 0x001A; property type PT_STRING8; property tag 0x001A001E

Comments

The message class describes the kind of message. It provides information about the set of properties defined for the message, such as the kind of information that the message conveys and how to handle the message.

Each message class can be considered to have its own name space. A range of property values (0x7C00 to 0x7FFF) is reserved for properties that are specific to the message-class. When you create a new message class, you can also specify the meaning of the properties in the reserved range. Within this range, properties for your message class are guaranteed not to collide with properties for other message classes.

To date, the message class has been used for three primary purposes.

1. Receive foldering. A column in the receive folders table of the receive store organizes all incoming mail with this message class prefix. This allows you to specify where to store the incoming messages.

2. Launching forms. All form-aware clients can choose the form associated with a message based on the value of the message class property.

3. Rules. The message class can be used to trigger processing rules.

Any application that creates a message must set the PR_MESSAGE_CLASS property. Message store providers should set its value using the prefix "IPM." For example, MAPI uses the message class, "IPM.Note." PR_MESSAGE_CLASS also appears in recipients of delivery reports (DRs) and non-delivery reports (NDRs).

The message class name must be case-insensitive and must consist of the ASCII characters 32 through 127.

For a report, the value of PR_MESSAGE_CLASS is based on the class of the subject message. The format of the message class for the report is REPORT.X.Y, where REPORT is the report name, X is the class of the subject message, and Y is DR for a delivery report, NDR for a nondelivery report, IPNRN for a read notification report, or IPNNRN for a nonread notification report.

See Also

IMAPIForm : IUnknown interface, **IMsgStore::GetReceiveFolderTable** method

PR_MESSAGE_DELIVERY_ID

[New - Windows 95]

Contains a message transport system (MTS) identifier for a message delivered to a client application.

Details

Identifier 0x001B; property type PT_BINARY; property tag 0x001B0102

Comments

For X.400 environments, PR_MESSAGE_DELIVERY_ID corresponds to the MH_T_MTS_IDENTIFIER attribute.

See Also

PR_MESSAGE_SUBMISSION_ID property

PR_MESSAGE_DELIVERY_TIME

[New - Windows 95]

Contains the date and time that the message was delivered at the server.

Details

Identifier 0x0E06; property type PT_SYSTIME; property tag 0x0E060040

Comments

This property describes delivery time at the server rather than the download time that the transport copied the message from the server to the local store.

For X.400 environments, PR_MESSAGE_DELIVERY_TIME corresponds to the MH_T_DELIVERY_TIME attribute as defined by the X.400 message-handling standard.

See Also

PR_MESSAGE_DOWNLOAD_TIME property

PR_MESSAGE_DOWNLOAD_TIME

[New - Windows 95]

Contains the time a message was downloaded from a messaging server to a local message store.

Details

Identifier 0x0E18; property type PT_LONG; property tag 0x0E180003

Comments

This property describes the download time that the transport copied the message from the server to the local store rather than the delivery time at the server.

See Also

PR_MESSAGE_DELIVERY_TIME property

PR_MESSAGE_FLAGS

[New - Windows 95]

Contains a bitmask of flags indicating the current state of a message object. Every message must furnish this property.

Details

Identifier 0x0E07; property type PT_LONG; property tag 0x0E070003

Comments

Your application or provider can read the current state of the message by calling IMAPIProp::GetProps. The application or provider can also use the property in an IMAPIProp::SetProps call to change the current state of the message.

Your client application or message store provider can read any of the flags set in the PR_MESSAGE_FLAGS bitmask. However, the message object will ignore attempts by the application or provider to write to flags with read-only access.

The following flags can be set for the PR_MESSAGE_FLAGS bitmask:

Value	Description
MSGFLAG_ASSOCIATED	Indicates the message is an associated message of a folder. The application or provider has read-only access to this flag.
MSGFLAG_FROMME	Indicates that the messaging user sending was the messaging user receiving the message. The application or provider has read-only access to this flag. This flag is meant to be set by the transport.
MSGFLAG_HASATTACH	Indicates the message has at least one attachment. The application or provider has read-only access to this flag.
MSGFLAG_READ	Indicates the message has been read. The flag is also set when the client creates the message. The application or provider has read-write access to this flag and can call IMessage::SetReadFlag to change it. Some message stores do not honor read-write access to this flag and thus deny access.
MSGFLAG_RESEND	Indicates the message includes a request for a resend operation with a nondelivery report. After the first call to SaveChanges, the application or provider has read-only access to this flag .
MSGFLAG_SUBMIT	Indicates the message is marked for sending as a result of a call to IMessage::SubmitMessage. The

	application or provider has read-only access to this flag.
MSGFLAG_UNMODIFIED	Indicates the message has not been modified since reception. The application or provider has read-only access to this flag.
MSGFLAG_UNSENT	Indicates the message is still being composed. It is saved, but has not been sent. Typically, this flag is cleared after the message is sent.

See Also

IMAPIProp::GetProps method, **IMAPIProp::SaveChanges** method, **IMAPIProp::SetProps** method, **IMessage::SetReadFlag** method, **IMsgStore::AbortSubmit** method

PR_MESSAGE_RECIP_ME

[New - Windows 95]

Set to TRUE if the messaging user was specifically named as the primary (To), carbon copy (CC), or blind carbon copy (BCC) recipient. The property contains FALSE otherwise.

Details

Identifier 0x0059; property type PT_BOOLEAN; property tag 0x0059000B

Comments

PR_MESSAGE_RECIP_ME is 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, PR_MESSAGE_TO_ME, and the Bcc information (which does not otherwise appear in a property).

This property assists automated handling of received messages at the time of receipt. The transport provider is responsible for setting the PR_MESSAGE_RECIP_ME property. 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.

Distribution list expansion does not cause this property to be set. The user must be explicitly named.

Unsent messages generally do not include this property. If the property 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.

See Also

PR_MESSAGE_CC_ME property, PR_MESSAGE_TO_ME property

PR_MESSAGE_RECIPIENTS

[New - Windows 95]

Contains a table of restrictions that can be applied to the folder's contents table to find all messages that contain the recipients specified in the restrictions.

Details

Identifier 0x0E12; property type PT_OBJECT; property tag 0x0E12000D

Comments

You can restrict the contents table of the folder to find all messages where the recipients meet the conditions specified in the restriction. The restriction is true if one row remains in the table.

The restrictions are commonly used to exclude recipients from **IMessage::CopyTo** operations and to include recipients in **IMessage::CopyProps** operations.   This property can also be used in the **IMAPIProp** methods **GetPropList**, **GetProps**, **OpenProps**, and **AddProps**.

Note that using restrictions results in an open operation on every message in the folder. Depending on your application and the number of messages to be searched, it can affect application performance.

PR_MESSAGE_RECIPIENTS is used for recipients as PR_MESSAGE_ATTACHMENTS is used for attachments.

Several MAPI properties provide access to table objects:

Property	**Table**
PR_CONTAINER_HIERARCHY	Hierarchy table
PR_FOLDER_ASSOCIATED_CONTENTS	Associated contents table
PR_MESSAGE_ATTACHMENTS	Attachments table
PR_MESSAGE_RECIPIENTS	Recipients table

See Also

IMAPIProp::CopyProps method, **IMAPIProp::CopyTo** method, PR_MESSAGE_ATTACHMENTS property

PR_MESSAGE_SECURITY_LABEL

[New - Windows 95]

Contains a security label for a message.

Details

Identifier 0x001E; property type PT_BINARY; property tag 0x001E0102

Comments

For X.400 environments, this property corresponds to the MH_T_SECURITY_LABEL attribute.

X.400 and EDI transport providers use this property for messages delivered to client applications. X.400 environments also use this property to represent the corresponding report per-message attribute as defined in the X.400 message-handling standard.

See Also

PR_MESSAGE_SIZE property

PR_MESSAGE_SIZE

[New - Windows 95]

Contains the sum in bytes of the sizes of all properties of the message object.

Details

Identifier 0x0E08; property type PT_LONG; property tag 0x0E080003

Comments

The message size indicates the number of bytes transferred when the message is moved from one message store to another.

Most message store providers compute this property for messages that they handle. However, some message store providers do not support the property.

PR_ATTACH_SIZE provides the same information for the attachment that PR_MESSAGE_SIZE provides for the message.

See Also

PR_ATTACH_SIZE property, PR_MESSAGE_SUBMISSION_ID property

PR_MESSAGE_SUBMISSION_ID

[New - Windows 95]

Contains a message transport system (MTS) identifier for the message transfer agent (MTA).

Details

Identifier 0x0047; property type PT_BINARY; property tag 0x00470102

Comments

For X.400 environments, PR_MESSAGE_SUBMISSION_ID corresponds to the MH_T_MTS_IDENTIFIER attribute.

See Also

PR_MESSAGE_DELIVERY_ID property

PR_MESSAGE_TO_ME

[New - Windows 95]

Set to TRUE by a transport provider as a message is received into the message store of a messaging user listed directly as a primary (To) recipient in the recipient table.

Details

Identifier 0x0057; property type PT_BOOLEAN; property tag 0x0057000B

Comments

PR_MESSAGE_TO_ME is 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, 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.

Unsent messages generally do not include this property. If the property 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.

See Also

PR_MESSAGE_CC_ME property, PR_MESSAGE_RECIP_ME property

PR_MESSAGE_TOKEN

[New - Windows 95]

Contains an ASN.1 token for a message.

Details

Identifier 0x0C03, property type PT_BINARY; property tag 0x0c030102

Comments

For X.400 environments, PR_MESSAGE_TOKEN corresponds to the MH_T_TOKEN attribute.

X.400 and EDI transport providers use this property for messages delivered to client applications.

See Also

PR_PHYSICAL_RENDITION_ATTRIBUTES property

PR_MHS_COMMON_NAME

[New - Windows 95]

Contains the common name of a messaging user for use in a message header.

Details

Identifier 0x3A0F, property type PT_TSTRING; property tag 0x3a0f001e

Comments

For X.400 environments, PR_MHS_COMMON_NAME corresponds to the MH_T_COMMON_NAME attribute as defined by the X.400 message-handling standard.

See Also

PR_DISPLAY_NAME property

PR_MINI_ICON

[New - Windows 95]

Contains a 16 by 16 icon that is used on forms.

Details

Identifier 0x0FFC, property type PT_BINARY; property tag 0x0FFC0102

Comments

The value is the same as the contents of a .ICO file, that is, a bitmap.

See Also

PR_ICON property

PR_MOBILE_TELEPHONE_NUMBER

[New - Windows 95]

Contains the cellular telephone number for the messaging user.

Details

Identifier 0x3A1C, property type PT_TSTRING; property tag 0x3A1C001E

Comments

This is the same property as PR_CELLULAR_TELEPHONE_NUMBER.

See Also

PR_CELLULAR_TELEPHONE_NUMBER property

PR_MODIFY_VERSION

[New - Windows 95]

Contains the version number of the message store that was current at the time the specified message was last modified.

Details

Identifier 0x0E1A, property type PT_I8; property tag 0x0e1a0014

Comments

This version number is a copy of the message store version number contained in the PR_CURRENT_VERSION property.

Your application or service provider can use this property to support synchronization between message stores.

See Also

PR_CURRENT_VERSION property

PR_MSG_STATUS

[New - Windows 95]

Contains a column in a contents table.

Details

Identifier 0x0E17, property type PT_LONG; property tag 0x0e170003

Comments

A message can exist in a contents table and in one or more search contents tables, and each instance of the message can have a different status. PR_MSG_STATUS should not be considered as a property in a message but as a column in a contents table.

Most values for the message status values are user-defined. See Also

IMAPIFolder::GetMessageStatus method, **IMAPIFolder::SetMessageStatus** method, **IMAPITable::QueryRows** method

PR_NDR_DIAG_CODE

[New - Windows 95]

Contains a diagnostic code that forms part of a nondelivery report.

Details

Identifier 0x0C05, property type PT_LONG; property tag 0x0c050003

Comments

This MAPI property corresponds to the MH_T_NON_DELIVERY_DIAGNOSTIC attribute as defined by the X.400 message-handling standard.

The following values are valid for PR_NDR_DIAG_CODE:

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_DOWNGRADING_IMPOSSIBLE
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

See Also

PR_NDR_REASON_CODE property

PR_NDR_REASON_CODE

[New - Windows 95]

Contains a reason code that forms part of a nondelivery report.

Details

Identifier 0x0C04, property type PT_LONG; property tag 0x0c040003

Comments

This MAPI property corresponds to the MH_T_NON_DELIVERY_REASON attribute as defined by the X.400 message-handling standard.

See Also

PR_NDR_DIAG_CODE property

PR_NON_RECEIPT_NOTIFICATION_REQUESTED

[New - Windows 95]

Contains TRUE if a message sender wants notification of nondelivery for a specified recipient, and FALSE otherwise.

Details

Identifier 0x0C06, property type PT_BOOLEAN; property tag 0x0c06000b

Comments

If this property contains FALSE and the PR_READ_RECEIPT_REQUESTED property contains TRUE, the service provider can override the PR_NON_RECEIPT_NOTIFICATION_REQUESTED property and generate a nondelivery report.

This MAPI property corresponds to the IM_NOTIFICATION_REQUEST attribute as defined by the X.400 message-handling standard.

See Also

PR_READ_RECEIPT_REQUESTED property

PR_NON_RECEIPT_REASON

[New - Windows 95]

Contains reasons why a message was not received (for example, the message was discarded) for use in a nondelivery report.

Details

Identifier 0x003E, property type PT_LONG; property tag 0x003e0003

Comments

This MAPI property corresponds to the IM_NON_RECEIPT_REASON attribute as defined by the X.400 message-handling standard.

See Also

PR_NDR_DIAG_CODE property

PR_NORMALIZED_SUBJECT

[New - Windows 95]

Contains the subject with no prefix or with any prefix except reply and forward.

Details

Identifier 0x0E1D, property type PT_TSTRING; property tag 0x0e1d001e

Comments

A message store computes this property from the PR_SUBJECT and PR_SUBJECT_PREFIX properties. You can set the PR_SUBJECT_PREFIX property to a NULL string. The prefix can be a NULL string or can contain one, two, or three alphanumeric characters, followed by the colon character, ':'. The NULL string is always valid.

For X.400 environments, PR_NORMALIZED_SUBJECT corresponds to the IM_SUBJECT attribute as defined by the X.400 message-handling standard.

See Also

PR_SUBJECT property

PR_NULL

[New - Windows 95]

Reserved. Represents the property with identifier 0x0000.

Details

Identifier 0x0000, property type PT_NULL; property tag 0x00000001

Comments

PR_NULL is used to reserve space in PropValue arrays. It is used in a PropTag array to tell the method to reserve space in the returned PropValue array. This allows for computed properties to be filled in an inexpensive way.

See Also

Extended MAPI Property Types

PR_OBJECT_TYPE

[New - Windows 95]

Contains the type for an object. This type corresponds to the primary interface available for the object.

Details

Identifier 0x0FFE, property type PT_LONG; property tag 0x0ffe0003

Comments

The object type is usually obtained by consulting the [out] parameter returned by OpenEntry. When you obtain the interface in other ways, you can call GetProps to obtain the value for PR_OBJECT_TYPE.

All MAPI objects must furnish an object type. Possible types are:

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	Mail user object.
MAPI_MESSAGE	Message object.
MAPI_PROFSECT	Profile section object.
MAPI_SESSION	Session object.
MAPI_STATUS	Status object.
MAPI_STORE	Message store object.

For more information on object types, see *MAPI Programmer's Guide.*

PR_OBSOLETED_IPMS

[New - Windows 95]

Contains the identifiers of messages (PR_SEARCH_KEY) that a specified message supersedes.

Details

Identifier 0x001F, property type PT_BINARY; property tag 0x001f0102

Comments

This MAPI property corresponds to the submission envelope per-message IM_OBSOLETED_IPMS attribute as defined by the X.400 message-handling standard. X.400 transport providers use this property for messages delivered to client applications.

See Also

PR_SEARCH_KEY property

PR_OFFICE_LOCATION

[New - Windows 95]

Contains the office location of the messaging user.

Details

Identifier 0x3A19, property type PT_TSTRING; property tag 0x3a19001e

Comments

Additional messaging user properties include the telephone number, fax number, and cellular telephone number.

See Also

PR_OFFICE_TELEPHONE_NUMBER property

PR_OFFICE_TELEPHONE_NUMBER

[New - Windows 95]

Contains the office telephone number of a messaging user.

Details

Identifier 0x3A08, property type PT_TSTRING; property tag 0x3a08001e

Comments

Your application can specify another telephone number by setting the PR_OFFICE2_TELEPHONE_NUMBER property.

See Also

PR_OFFICE2_TELEPHONE_NUMBER property

PR_OFFICE2_TELEPHONE_NUMBER

[New - Windows 95]

Contains a second office telephone number for a messaging user.

Details

Identifier 0x3A1B, property type PT_TSTRING; property tag 0x3a1b001e

Comments

Your application sets the PR_OFFICE_TELEPHONE_NUMBER property to specify the primary office number.

See Also

PR_OFFICE_TELEPHONE_NUMBER property

PR_ORGANIZATIONAL_ID_NUMBER

[New - Windows 95]

Contains a number associated with that messaging user by the organization, such as an employee number.

Details

Identifier 0x3A10, property type PT_TSTRING; property tag 0x3a10001e

Comments

Additional messaging user properties include the telephone number, fax number, and cellular telephone number.

See Also

PR_OFFICE_TELEPHONE_NUMBER property

PR_ORIG_MESSAGE_CLASS

[New - Windows 95]

Contains the class of the original message for use in a message report.

Details

Identifier 0x004B, property type PT_TSTRING; property tag 0x004b001e

Comments

This property contains a copy of the contents of PR_MESSAGE_CLASS.

See Also

PR_MESSAGE_CLASS property

PR_ORIGINAL_AUTHOR_ADDRTYPE

[New - Windows 95]

Contains the address type of the author of the first version of a message (that is, the message before it is forwarded or replied to).

Details

Identifier 0x0079, property type PT_TSTRING; property tag 0x0079001E

Comments

The PR_ORIGINAL_AUTHOR_* properties allow you to preserve information that is from outside your messaging domain. When you obtain a message from another messaging domain and use it within Microsoft Exchange, such as when you post a message from the Internet to a public folder, these properties provide a way to ensure that information is not lost.

This property represents one of five maintained for the original author. Opening the entry id to find additional information is a relatively expensive operation, so properties representing the commonly-used display name, search key, address type, and email address are also defined with the entryid.

See Also

PR_ADDRTYPE property, PR_ORIGINAL_AUTHOR_EMAIL_ADDRESS property, PR_ORIGINAL_AUTHOR_ENTRYID property, PR_ORIGINAL_AUTHOR_SEARCH_KEY property

PR_ORIGINAL_AUTHOR_EMAIL_ADDRESS

[New - Windows 95]

Contains the email address of the author of the first version of a message (that is, the message before it is forwarded or replied to).

Details

Identifier 0x007A, property type PT_TSTRING; property tag 0x007A001E

Comments

The PR_ORIGINAL_AUTHOR_* properties allow you to preserve information that is from outside your messaging domain. When you obtain a message from another messaging domain and use it within Microsoft Exchange, such as when you post a message from the Internet to a public folder, these properties provide a way to ensure that information is not lost.

This property represents one of five maintained for the original author. Opening the entry id to find additional information is a relatively expensive operation, so properties representing the commonly-used display name, search key, address type, and email address are also defined with the entryid.

See Also

PR_EMAIL_ADDRESS property, PR_ORIGINAL_AUTHOR_ADDRTYPE property, PR_ORIGINAL_AUTHOR_ENTRYID property, PR_ORIGINAL_AUTHOR_NAME property

PR_ORIGINAL_AUTHOR_ENTRYID

[New - Windows 95]

Contains the entry identifier of the author of the first version of a message; that is, the message before it is forwarded or replied to.

Details

Identifier 0x004C, property type PT_BINARY; property tag 0x004C0102

Comments

The PR_ORIGINAL_AUTHOR_* properties allow you to preserve information that is from outside your messaging domain. When you obtain a message from another messaging domain and use it within Microsoft Exchange, such as when you post a message from the Internet to a public folder, these properties provide a way to ensure that information is not lost.

At first submission of a message within the same messaging domain, your application should set this optional property to the value of PR_SENDER_ENTRYID. It is never changed when the message is forwarded.

This property represents one of five maintained for the original author. Opening the entry id to find additional information is a relatively expensive operation, so properties representing the commonly-used display name, search key, address type, and email address are also defined with the entryid.

See Also

PR_ORIGINAL_AUTHOR_ADDRTYPE property, PR_ORIGINAL_AUTHOR_EMAIL_ADDRESS property, PR_ORIGINAL_AUTHOR_NAME property, PR_ORIGINAL_AUTHOR_SEARCH_KEY property, PR_SENDER_ENTRYID property

PR_ORIGINAL_AUTHOR_NAME

[New - Windows 95]

Contains the name of the author of the first version of a message (that is, the message before it is forwarded or replied to).

Details

Identifier 0x004D, property type PT_TSTRING; property tag 0x004D001E

Comments

The PR_ORIGINAL_AUTHOR_* properties allow you to preserve information that is from outside your messaging domain. When you obtain a message from another messaging domain and use it within Microsoft Exchange, such as when you post a message from the Internet to a public folder, these properties provide a way to ensure that information is not lost.

This property represents one of five maintained for the original author. Opening the entry id to find additional information is a relatively expensive operation, so properties representing the commonly-used display name, search key, address type, and email address are also defined with the entryid.

At first submission of a message, your application should set this property to the value of PR_SENDER_NAME. It is never changed when the message is forwarded.

For X.400 environments, PR_ORIGINAL_AUTHOR_NAME corresponds to the MH_T_ORIGINATOR_NAME attribute as defined by the X.400 message-handling standard.

See Also

PR_ORIGINAL_AUTHOR_ADDRTYPE property, PR_ORIGINAL_AUTHOR_EMAIL_ADDRESS property, PR_ORIGINAL_AUTHOR_ENTRYID property, PR_ORIGINAL_AUTHOR_NAME property, PR_SENDER_NAME property

PR_ORIGINAL_AUTHOR_SEARCH_KEY

[New - Windows 95]

Contains a copy of the original PR_SENDER_SEARCH_KEY data in a conversation thread. This value is copied in subsequent reply or forward operations.

Details

Identifier 0x0056, property type PT_BINARY; property tag 0x00560102

Comments

The PR_ORIGINAL_AUTHOR_* properties allow you to preserve information that is from outside your messaging domain. When you obtain a message from another messaging domain and use it within Microsoft Exchange, such as when you post a message from the Internet to a public folder, these properties provide a way to ensure that information is not lost.

See Also

PR_ORIGINAL_AUTHOR_ADDRTYPE property, PR_ORIGINAL_AUTHOR_EMAIL_ADDRESS property, PR_ORIGINAL_AUTHOR_ENTRYID property, PR_ORIGINAL_AUTHOR_NAME property, PR_SENDER_SEARCH_KEY property

PR_ORIGINAL_DELIVERY_TIME

[New - Windows 95]

Contains a copy of the original PR_MESSAGE_DELIVERY_TIME data in a thread.

Details

Identifier 0x0055, property type PT_SYSTIME; property tag 0x00550040

Comments

PR_ORIGINAL_DELIVERY_TIME is copied in subsequent reply or forward operations and used in delivery, read, and nonread reports.

See Also

PR_MESSAGE_DELIVERY_TIME property

PR_ORIGINAL_DISPLAY_BCC

[New - Windows 95]

Contains the original display name of a hidden carbon copy entry.

Details

Identifier 0x0072, property type PT_TSTRING; property tag 0x0072001E

Comments

This property contains the original BCC display name values for a copied entry.

See Also

PR_ORIGINAL_DISPLAY_CC property

PR_ORIGINAL_DISPLAY_CC

[New - Windows 95]

Contains the original display names of the message CC list.

Details

Identifier 0x0073, property type PT_TSTRING; property tag 0x0073001E

Comments

This property contains the original CC display name values for a copied entry.

See Also

PR_ORIGINAL_DISPLAY_BCC property

PR_ORIGINAL_DISPLAY_NAME

[New - Windows 95]

Contains the original display name of an entry copied from an address book to a personal address book or another writable address book.

Details

Identifier 0x3A13, property type PT_TSTRING; property tag 0x3A13001E

Comments

This property and the PR_ORIGINAL_ENTRYID property contain 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 group is preserved for the report.

Your 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

[New - Windows 95]

Contains the original display names of the message To: list.

Details

Identifier 0x0074, property type PT_TSTRING; property tag 0x0074001E

Comments

This property contains the original To: display name values for a copied entry.

See Also

PR_ORIGINAL_DISPLAY_CC property

PR_ORIGINAL_EITS

[New - Windows 95]

Contains a copy of the original embedded information types (EITs) for text (body) parts of a message.

Details

Identifier 0x0021, property type PT_BINARY; property tag 0x00210102

Comments

For X.400 environments, PR_ORIGINAL_EITS corresponds to the MH_T_ORIGINAL_EITS attribute.

X.400 and EDI transport providers use this property for messages delivered to client applications. X.400 environments also use it to represent the corresponding probe submission per-message and report per-message attributes as defined by the X.400 message-handling standard.

See Also

PR_CONVERSION_EITS property, PR_CONVERTED_EITS property

PR_ORIGINAL_ENTRYID

[New - Windows 95]

Contains the original entry identifier of an entry copied from an address book   to a personal address book or another writeable address book.

Details

Identifier 0x3A12, property type PT_BINARY; property tag 0x3A120102

Comments

This property and the PR_ORIGINAL_DISPLAY_NAME property contain the original source of a copied entry.

For a nonread notification 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 group is preserved for the report.

See Also

PR_ORIGINAL_DISPLAY_NAME property

PR_ORIGINAL_SEARCH_KEY

[New - Windows 95]

Contains the original search key associated with an entry copied from an address book to a personal address book or another writeable address book.

Details

Identifier 0x3A14, property type PT_BINARY; property tag 0x3A140102

Comments

The PR_ORIGINAL_* properties contain information about the original source of a copied entry.

See Also

PR_ORIGINAL_ENTRYID property

PR_ORIGINAL_SENDER_ADDRTYPE

[New - Windows 95]

Contains the address type of the sender of the first version of a message (that is, the message before it is forwarded or replied to).

Details

Identifier 0x0066, property type PT_TSTRING; property tag 0x0066001E

Comments

This is set to the same value as PR_SENDER_ADDRTYPE.

This is one of several PR_ORIGINAL_SENDER_* properties that preserve information from the corresponding PR_SENDER_* properties.

See Also

PR_SENDER_ADDRTYPE property

PR_ORIGINAL_SENDER_EMAIL_ADDRESS

[New - Windows 95]

Contains the email address of the sender of the first version of a message (that is, the message before it is forwarded or replied to).

Details

Identifier 0x0067, property type PT_TSTRING; property tag 0x0067001E

Comments

This is initially set to the same value as PR_SENDER_EMAIL_ADDRESS.

This is one of several PR_ORIGINAL_SENDER_* properties that preserve information from the corresponding PR_SENDER_* properties.

See Also

PR_SENDER_EMAIL_ADDRESS property

PR_ORIGINAL_SENDER_ENTRYID

[New - Windows 95]

Contains the entry identifier of the sender of the first version of a message (that is, the message before it is forwarded or replied to).

Details

Identifier 0x005B, property type PT_BINARY; property tag 0x005B0102

Comments

This is initially set to the same value as PR_SENDER_ENTRYID.

The PR_ORIGINAL_AUTHOR_* values are usually set to the same values as the PR_ORIGINAL_SENDER_* values.

This is one of several PR_ORIGINAL_SENDER_* properties that preserve information from the corresponding PR_SENDER_* properties.

See Also

PR_ORIGINAL_AUTHOR_ENTRYID property, PR_SENDER_ENTRYID property

PR_ORIGINAL_SENDER_NAME

[New - Windows 95]

Contains the display name of the sender of the first version of a message (that is, the message before it is forwarded or replied to).

Details

Identifier 0x005A, property type PT_TSTRING; property tag 0x005A001E

Comments

This is initially set to the same value as PR_SENDER_NAME.

The PR_ORIGINAL_AUTHOR_* values are usually set to the same values as the PR_ORIGINAL_SENDER_* values.

For X.400 environments, PR_ORIGINAL_SENDER_NAME corresponds to the MH_T_ORIGINATOR_NAME attribute as defined by the X.400 message-handling standard.

See Also

PR_ORIGINAL_AUTHOR_NAME property, PR_SENDER_NAME property

PR_ORIGINAL_SENDER_SEARCH_KEY

[New - Windows 95]

Contains the search key for the sender of the first version of a message (that is, the message before it is forwarded or replied to).

Details

Identifier 0x005C, property type PT_BINARY; property tag 0x005C0102

Comments

This is set to the same value as PR_SENDER_SEARCH_KEY.

The PR_ORIGINAL_AUTHOR_* values are usually set to the same values as the PR_ORIGINAL_SENDER_* values.

See Also

PR_ORIGINAL_AUTHOR_SEARCH_KEY property, PR_SENDER_SEARCH_KEY property

PR_ORIGINAL_SENT_REPRESENTING_ADDRTYPE

[New - Windows 95]

Contains the address type of the messaging user on whose behalf the original message is sent for a conversation thread.

Details

Identifier 0x0068, property type PT_TSTRING; property tag 0x0068001E

Comments

This property applies when the sender is sending the message as the delegated agent of the messaging user.

See Also

PR_ORIGINAL_SENT_REPRESENTING_ENTRYID property

PR_ORIGINAL_SENT_REPRESENTING_EMAIL_ADDRESS

[New - Windows 95]

Contains the email address of the messaging user on whose behalf the original message is sent for a conversation thread.

Details

Identifier 0x0069, property type PT_TSTRING; property tag 0x0069001E

Comments

This property applies when the sender is sending the message as the delegated agent of the messaging user.

See Also

PR_ORIGINAL_SENT_REPRESENTING_ENTRYID property

PR_ORIGINAL_SENT_REPRESENTING_ENTRYID

[New - Windows 95]

Contains the entry identifier of the messaging user on whose behalf the original message is sent for a conversation thread (PR_SENT_REPRESENTING_ENTRYID).

Details

Identifier 0x005E, property type PT_BINARY; property tag 0x005E0102

Comments

This property applies when the sender is sending the message as the delegated agent of the messaging user.

See Also

PR_SENT_REPRESENTING_ENTRYID property

PR_ORIGINAL_SENT_REPRESENTING_NAME

[New - Windows 95]

Contains the display name of the messaging user on whose behalf the original message is sent for a conversation thread.

Details

Identifier 0x005D, property type PT_TSTRING; property tag 0x005D001E

Comments

Like PR_SENT_REPRESENTING_NAME, this property applies when the sender is sending the message as the delegated agent of the messaging user.

See Also

PR_SENT_REPRESENTING_NAME property

PR_ORIGINAL_SENT_REPRESENTING_SEARCH_KEY

[New - Windows 95]

Contains the search key of the messaging user on whose behalf the original message is sent for a conversation thread.

Details

Identifier 0x005F, property type PT_BINARY; property tag 0x005F0102

Comments

Like PR_SENT_REPRESENTING_SEARCH_KEY, this property applies when the sender is sending the message as the delegated agent of the messaging user.

See Also

PR_SENT_REPRESENTING_SEARCH_KEY property

PR_ORIGINAL_SUBJECT

[New - Windows 95]

Contains the subject of an original message for use in a report about the message.

Details

Identifier 0x0049, property type PT_TSTRING; property tag 0x0049001E

Comments

Set to the same value as PR_SUBJECT.

For X.400 environments, PR_ORIGINAL_SUBJECT corresponds to the IM_SUBJECT attribute as defined by the X.400 message-handling standard.

See Also

PR_SUBJECT property

PR_ORIGINAL_SUBMIT_TIME

[New - Windows 95]

Contains the original submission time of the message in the report.

Details

Identifier 0x004E, property type PT_SYSTIME; property tag 0x004E0040

Comments

At first submission of a message, your application should set this 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.

See Also

PR_CLIENT_SUBMIT_TIME property

PR_ORIGINALLY_INTENDED_RECIP_ADDRTYPE

[New - Windows 95]

Contains the address type of the originally intended recipient of an autoforwarded message.

Details

Identifier 0x007B, property type PT_TSTRING; property tag 0x007B001E

Comments

This property represents one of several maintained for the intended recipient, including the name, entryid, and email address.

X.400 and EDI transport providers use this property for messages delivered to client applications. X.400 environments also use it to represent the IM_IPM_INTENDED_RECIPIENT attribute as defined by the X.400 message-handling standard.

See Also

PR_ADDRTYPE property, PR_ORIGINALLY_INTENDED_RECIP_EMAIL_ADDRESS property, PR_ORIGINALLY_INTENDED_RECIPIENT_NAME property

PR_ORIGINALLY_INTENDED_RECIP_EMAIL_ADDRESS

[New - Windows 95]

Contains the email address of the originally intended recipient of an autoforwarded message.

Details

Identifier 0x007C, property type PT_TSTRING; property tag 0x007C001E

Comments

This property represents one of several maintained for the intended recipient, including the name, entryid, and address type.

X.400 and EDI transport providers use this property for messages delivered to client applications. X.400 environments also use it to represent the corresponding report per-recipient attribute as defined by the X.400 message-handling standard.

See Also

PR_EMAIL_ADDRESS property, PR_ORIGINALLY_INTENDED_RECIP_ADDRTYPE property, PR_ORIGINALLY_INTENDED_RECIPIENT_NAME property

PR_ORIGINALLY_INTENDED_RECIP_ENTRYID

[New - Windows 95]

Contains the entryid of the originally intended recipient of an autoforwarded message.

Details

Identifier 0x1012, property type PT_BINARY; property tag 0x10120102

Comments

This property represents one of several maintained for the intended recipient, including the name, address type, and email address.

X.400 and EDI transport providers use this property for messages delivered to client applications. X.400 environments also use it to represent the corresponding report per-recipient attribute as defined by the X.400 message-handling standard.

See Also

PR_ENTRYID property, PR_ORIGINALLY_INTENDED_RECIP_ADDRTYPE property, PR_ORIGINALLY_INTENDED_RECIPIENT_NAME property

PR_ORIGINALLY_INTENDED_RECIPIENT_NAME

[New - Windows 95]

Contains the entry identifier of the originally intended recipient of an autoforwarded message.

Details

Identifier 0x0020, property type PT_BINARY; property tag 0x00200102

Comments

For X.400 environments, PR_ORIGINALLY_INTENDED_RECIPIENT_NAME corresponds to the MH_T_ORIGINALLY_INTENDED_RECIP attribute.

X.400 and EDI transport providers use this property for messages delivered to client applications. X.400 environments also use it to represent the corresponding report per-recipient attribute as defined by the X.400 message-handling standard.

See Also

PR_ORIGINALLY_INTENDED_RECIP_ENTRYID property

PR_ORIGINATING_MTA_CERTIFICATE

[New - Windows 95]

Contains an identifier for the message transfer agent (MTA) that originated the message. Details

Identifier 0x0E25, property type PT_BINARY; property tag 0x0E250102

Comments

This MAPI property represents the corresponding X.400 report per-message attribute as defined by the X.400 message-handling standard.

See Also

PR_REPORTING_MTA_CERTIFICATE property

PR_ORIGINATOR_AND_DL_EXPANSION_HISTORY

[New - Windows 95]

Contains information about a message originator and a distribution list expansion history for use in report generation.

Details

Identifier 0x1002, property type PT_BINARY; property tag 0x10020102

Comments

This MAPI property corresponds to the report per-message MH_T_ORIG_AND_EXPANSION_HISTORY attribute as defined by the X.400 message-handling standard.

See Also

PR_ORIGINATOR_CERTIFICATE property

PR_ORIGINATOR_CERTIFICATE

[New - Windows 95]

Contains an authentication certificate for a message originator. An authentication certificate is similar to a digital signature.

Details

Identifier 0x0022, property type PT_BINARY; property tag 0x00220102

Comments

For X.400 environments, PR_ORIGINATOR_CERTIFICATE corresponds to the MH_T_ORIGINATOR_CERTIFICATE attribute.

X.400 and EDI transport providers use this property for messages delivered to client applications. X.400 environments also use it to represent the corresponding probe submission per-message attribute as defined by the X.400 message-handling standard.

See Also

PR_ORIGINATOR_AND_DL_EXPANSION_HISTORY property

PR_ORIGINATOR_DELIVERY_REPORT_REQUESTED

[New - Windows 95]

Contains TRUE if a message sender requests a delivery report from the messaging system before the message is placed in the message store for a particular recipient. The property contains FALSE otherwise.

Details

Identifier 0x0023, property type PT_BOOLEAN; property tag 0x0023000B

Comments

This property is used to direct the messaging system in handling delivered messages.

For X.400 and EDI environments, the PR_ORIGINATOR_DELIVERY_REPORT_REQUESTED property represents the corresponding submission envelope per-recipient attributes as defined by the X.400 and EDI message-handling standards. For X.400 environments, PR_ORIGINATOR_DELIVERY_REPORT_REQUESTED set to TRUE corresponds to the MH_T_ORIGINATOR_REPORT_REQUEST attribute. In this case, the message must also furnish PR_ORIGINATOR_NON_DELIVERY_REPORT_REQUESTED set to FALSE.

X.400 environments also use PR_ORIGINATOR_DELIVERY_REPORT_REQUESTED to represent the corresponding probe submission per-recipient attribute as defined in the X.400 message-handling standard.

See Also

PR_ORIGINATOR_NON_DELIVERY_REPORT_REQUESTED property

PR_ORIGINATOR_NON_DELIVERY_REPORT_REQUESTED

[New - Windows 95]

Contains TRUE if a message sender requests a nondelivery report for a particular recipient, and FALSE otherwise.

Details

Identifier 0x0C08, property type PT_BOOLEAN; property tag 0x0C08000B

Comments

For X.400 environments, this property set to TRUE corresponds to the MH_T_ORIGINATOR_REPORT_REQUEST attribute. In this case, the message must also furnish PR_ORIGINATOR_DELIVERY_REPORT_REQUESTED set to FALSE.

See Also

PR_ORIGINATOR_DELIVERY_REPORT_REQUESTED property

PR_ORIGINATOR_REQUESTED_ALTERNATE_RECIPIENT

[New - Windows 95]

Contains the entry identifier of an alternate recipient chosen by a message sender to receive a message if it cannot be delivered to the original recipient.

Details

Identifier 0x0C09, property type PT_BINARY; property tag 0x0C090102

Comments

For X.400 environments, this property corresponds to the MH_T_ALTERNATE_RECIPIENT_NAME attribute. X.400 environments also use this property to represent the corresponding probe submission per-recipient attribute as defined by the X.400 message-handling standard.

See Also

PR_ORIGINATOR_DELIVERY_REPORT_REQUESTED property

PR_ORIGINATOR_RETURN_ADDRESS

[New - Windows 95]

Contains the return address of the message originator.

Details

Identifier 0x0024, property type PT_BINARY; property tag 0x00240102

Comments

Represents submission envelope per-recipient attributes as defined by the X.400 and EDI message-handling standards. For X.400 environments, PR_ORIGINATOR_RETURN_ADDRESS corresponds to the MH_T_ORIGINATOR_RETURN_ADDRESS attribute. X.400 and EDI transport providers use this property for messages delivered to client applications.

See Also

PR_ORIGINATOR_CERTIFICATE property

PR_ORIGIN_CHECK

[New - Windows 95]

Contains the Extended MAPI property corresponding to an X.400 attribute used for authentication checks.

Details

Identifier 0x0027, property type PT_BINARY; property tag 0x00270102

Comments

This property replaces several properties supplied in pre-release versions of Extended MAPI, including PR_MESSAGE_ORIGIN_AUTHENTICATION_CHECK, PR_PROBE_SUBMISSION_AUTHENTICATION_CHECK, and PR_REPORT_ORIGIN_AUTHENTICATION_CHECK.

See Also

PR_ORIGINATOR_CERTIFICATE property

PR_OTHER_TELEPHONE_NUMBER

[New - Windows 95]

Contains an alternate telephone number (that is, a number other than the home or office number) for a messaging user.

Details

Identifier 0x3A1F, property type PT_TSTRING; property tag 0x3A1F001E

Comments

Additional messaging user properties include the telephone number, fax number, and cellular telephone number.

See Also

PR_OFFICE_TELEPHONE_NUMBER property

PR_OWN_STORE_ENTRYID

[New - Windows 95]

Contains the entry identifier of the transport's tightly-coupled message store.

Details

Identifier 0x3E06, property type PT_BINARY; property tag 0x3E060102

Comments

Specifies the entry identifier for the tightly-coupled store, if one exists. Appears on the transport status rows or status objects. For example, a transport might specify the private folder store entryid so that the spooler knows how to connect the transport to the store.

See Also

PR_STORE_ENTRYID property

PR_OWNER_APPT_ID

[New - Windows 95]

Contains an identifier that represents the appointment in the owner's schedule. Used in a meeting request.

Details

Identifier 0x0062, property type PT_LONG; property tag 0x00620003

Comments

Note that this property does not represent an entry identifier, but a long integer that uniquely identifies the appointment within the owner's schedule.

See Also

PR_END_DATE property, PR_ORIGINAL_AUTHOR_SEARCH_KEY property, PR_START_DATE property

PR_PAGER_TELEPHONE_NUMBER

[New - Windows 95]

Contains the messaging user's pager telephone number.

Details

Identifier 0x3A21, property type PT_TSTRING; property tag 0x3A21001E

Comments

Several properties are available for the messaging user, including the home, office, mobile, and fax telephone numbers.

See Also

PR_OFFICE_TELEPHONE_NUMBER property

PR_PARENT_DISPLAY

[New - Windows 95]

Contains a copy of the parent folder's display name.

Details

Identifier 0x0E05, property type PT_TSTRING; property tag 0x0E05001E

Comments

Represents a column in a search folder contents table. Note that the search folder contents table differs from the folder contents table.

Message stores compute the PR_PARENT_DISPLAY property for all messages.

All search-results folders must make this property available.

See Also

PR_PARENT_ENTRYID property

PR_PARENT_ENTRYID

[New - Windows 95]

Contains the entry identifier of the folder in which a message was found during a search. All search-results folders must furnish this property in their contents tables.

Details

Identifier 0x0E09, property type PT_BINARY; property tag 0x0E090102

Comments

Message stores compute the PR_PARENT_ENTRYID property for all messages.

All search-results folders must make this property available. Folder and interpersonal message objects must furnish this property.

For a message store root folder, PR_PARENT_ENTRYID contains the folder's own entry identifier.

See Also

PR_PARENT_DISPLAY property

PR_PARENT_KEY

[New - Windows 95]

Represents the X.400 property IM_REPLIED_TO_IPM.

Details

Identifier 0x0025, property type PT_BINARY; property tag 0x00250102

Comments

For X.400 environments, PR_PARENT_KEY corresponds to the IM_REPLIED_TO_IPM attribute as defined by the X.400 message-handling standard.

See Also

PR_PARENT_ENTRYID property

PR_PHYSICAL_DELIVERY_BUREAU_FAX_DELIVERY

[New - Windows 95]

Contains TRUE if the messaging system should use a fax bureau for physical delivery of a specified message.

Details

Identifier 0x0C0A, property type PT_BOOLEAN; property tag 0x0C0A000B

Comments

For X.400 environments, this MAPI property corresponds to the MH_T_BUREAU_FAX_DELIVERY attribute.

X.400 and EDI transport providers use this property for messages delivered to client applications.

See Also

PR_PHYSICAL_DELIVERY_MODE property

PR_PHYSICAL_DELIVERY_MODE

[New - Windows 95]

Contains a bitmask of flags defining the physical delivery mode (for example, special delivery) for a message designated for a specific recipient.

Details

Identifier 0x0C0B, property type PT_LONG; property tag 0x0C0B0003

Comments

For X.400 environments, this MAPI property corresponds to the MH_T_POSTAL_MODE attribute.

X.400 and EDI transport providers use this property for messages delivered to client applications.

See Also

PR_PHYSICAL_DELIVERY_BUREAU_FAX_DELIVERY property

PR_PHYSICAL_DELIVERY_REPORT_REQUEST

[New - Windows 95]

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 (MHS).

Details

Identifier 0x0C0C, property type PT_LONG; property tag 0x0C0C0003

Comments

For X.400 environments, this MAPI property corresponds to the MH_T_POSTAL_REPORT attribute as defined by the X.400 message-handling standard. X.400 and EDI transport providers use this property for messages delivered to client applications.

See Also

PR_PHYSICAL_DELIVERY_MODE property

PR_PHYSICAL_FORWARDING_ADDRESS

[New - Windows 95]

Contains the physical forwarding address of a message recipient and is used only with message reports.

Details

Identifier 0x0C0D, property type PT_BINARY; property tag 0x0C0D0102

Comments

This property corresponds to the report per-recipient MH_T_FORWARDING_ADDRESS attribute as defined by the X.400 message-handling standard.

See Also

PR_PHYSICAL_FORWARDING_ADDRESS_REQUESTED property

PR_PHYSICAL_FORWARDING_ADDRESS_REQUESTED

[New - Windows 95]

Contains TRUE if a message sender requests the message transfer agent (MTA) to attach a physical forwarding address for a message recipient, and FALSE otherwise.

Details

Identifier 0x0C0E, property type PT_BOOLEAN; property tag 0x0C0E000B

Comments

This MAPI property represents the corresponding X.400 and EDI submission envelope per-recipient attributes as defined by the X.400 and EDI message-handling standards.For X.400 environments, this MAPI property corresponds to the MH_T_FORWARDING_ADR_REQUESTED attribute.

X.400 and EDI transport providers use this property for messages delivered to client applications.

See Also

PR_PHYSICAL_FORWARDING_ADDRESS property

PR_PHYSICAL_FORWARDING_PROHIBITED

[New - Windows 95]

Contains TRUE if a message sender prohibits physical message forwarding for a specific recipient, and FALSE otherwise.

Details

Identifier 0x0C0F, property type PT_BOOLEAN; property tag 0x0C0F000B

Comments

For X.400 environments, this property corresponds to the MH_T_FORWARDING_PROHIBITED attribute.

X.400 and EDI transport providers use this property for messages delivered to client applications.

See Also

PR_PHYSICAL_FORWARDING_ADDRESS_REQUESTED property

PR_PHYSICAL_RENDITION_ATTRIBUTES

[New - Windows 95]

Contains an ASN.1 object identifier (OID) used for rendering message attachments.

Details

Identifier 0x0C10, property type PT_BINARY; property tag 0x0C100102

Comments

For X.400 environments, this property corresponds to MH_T_RENDITION_ATTRIBUTES.   X.400 and EDI transport providers use this property for messages delivered to client applications. X.400 environments also use the property for probe messages.

See Also

PR_ATTACH_ENCODING property

PR_POST_OFFICE_BOX

[New - Windows 95]

Contains the messaging user's post office box.

Details

Identifier 0x3A2B, property type PT_TSTRING; property tag 0x3A2B001E

Comments

Several properties are available for the messaging user, including several telephone numbers and fields of the user's home or business mailing address.

See Also

PR_BUSINESS_TELEPHONE_NUMBER property

PR_POSTAL_ADDRESS

[New - Windows 95]

Contains the postal address of a messaging user.

Details

Identifier 0x3A15, property type PT_TSTRING; property tag 0x3A15001E

Comments

Several properties are available for the messaging user, including several telephone numbers and fields of the user's home or business mailing address.

See Also

PR_BUSINESS_TELEPHONE_NUMBER property

PR_POSTAL_CODE

[New - Windows 95]

Contains the postal code for the messaging user's address.

Details

Identifier 0x3A2A, property type PT_TSTRING; property tag 0x3A2A001E

Comments

The postal code is specific to the messaging user's country. In the United States of America, this property contains the ZIP code.

See Also

PR_BUSINESS_TELEPHONE_NUMBER property

PR_PREPROCESS

[New - Windows 95]

Reserved for MAPI.

Details

Identifier 0x0E22, property type PT_BOOLEAN; property tag 0x0E22000B

Comments

This property is for use by MAPI only.   Do not use.

See Also

PR_SUBMIT_FLAGS property

PR_PRIMARY_CAPABILITY

[New - Windows 95]

Reserved for MAPI.

Details

Identifier 0x3904, property type PT_BINARY; property tag 0x39040102

Comments

This property is reserved for use by MAPI.   Do not use.

See Also

PR_AB_PROVIDER_ID property

PR_PRIMARY_FAX_NUMBER

[New - Windows 95]

Contains the telephone number for the messaging user's primary fax machine.

Details

Identifier 0x3A23, property type PT_TSTRING; property tag 0x3A23001E

Comments

Several properties are available for the messaging user, including several telephone numbers and fields of the messaging user home mailing address.

See Also

PR_OFFICE_TELEPHONE_NUMBER property

PR_PRIMARY_TELEPHONE_NUMBER

[New - Windows 95]

Contains the primary telephone number of a messaging user.

Details

Identifier 0x3A1A, property type PT_TSTRING; property tag 0x3A1A001E

Comments

Several properties are available for the messaging user, including several telephone numbers and fields of the messaging user home mailing address.

For X.400 environments, this property corresponds to IM_TELEPHONE_NUMBER as defined by the X.400 messaging standard.

See Also

PR_OFFICE_TELEPHONE_NUMBER property

PR_PRIORITY

[New - Windows 95]

Contains the relative priority of a message.

Details

Identifier 0x0026, property type PT_LONG; property tag 0x00260003

Comments

PR_PRIORITY is distinguished from PR_IMPORTANCE . 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 different display by the user interface.

For X.400 environments, this MAPI property corresponds to MH_T_PRIORITY. X.400 and EDI transport providers use this property for messages delivered to client applications. The PR_PRIORITY property can have the values listed below. Note that the priority of a report message should be the same as the priority of the original message being reported.

Value	**Description**
PRIO_NONURGENT	The message is not urgent.
PRIO_NORMAL	The message has normal priority.
PRIO_URGENT	The message is urgent.

See Also

PR_IMPORTANCE property

PR_PROFILE_NAME

[New - Windows 95]

Contains the profile name.

Details

Identifier 0x3D12, property type PT_TSTRING, property tag 0x3D12001E

Comments

PR_PROFILE_NAME is a computed property and is always present on the profile section. A provider's **ServiceEntry** code can use this property to discover the profile name.

Applications can use this property as a convenient alternative to the other method used to obtain the profile name, which involves examining the PR_DISPLAY_NAME property on the MAPI subsystem's status table row.

See Also

PR_DISPLAY_NAME property

PR_PROOF_OF_DELIVERY

[New - Windows 95]

Contains an ASN.1 proof of delivery value.

Details

Identifier 0x0C11, property type PT_BINARY; property tag 0x0C110102

Comments

This MAPI property represents the report per-recipient MH_T_PROOF_OF_DELIVERY attribute as defined by the X.400 message-handling standard.

See Also

PR_PROOF_OF_DELIVERY_REQUESTED property

PR_PROOF_OF_DELIVERY_REQUESTED

[New - Windows 95]

Contains TRUE if a message sender requests proof of delivery for a particular recipient, and FALSE otherwise.

Details

Identifier 0x0C12, property type PT_BOOLEAN; property tag 0x0C12000B

Comments

For X.400 environments, this property corresponds to the MH_T_PROOF_OF_DELIV_REQUESTED attribute. X.400 and EDI transport providers use this property for messages delivered to client applications.

See Also

PR_PROOF_OF_DELIVERY property

PR_PROOF_OF_SUBMISSION

[New - Windows 95]

Contains an ASN.1 proof of submission value.

Details

Identifier 0x0E26, property type PT_BINARY; property tag 0x0E260102

Comments

This MAPI property represents the corresponding X.400 and EDI submission envelope per-message attributes as defined by the X.400 and EDI message-handling standards. See Also

PR_PROOF_OF_SUBMISSION_REQUESTED property

PR_PROOF_OF_SUBMISSION_REQUESTED

[New - Windows 95]

Contains TRUE if a message sender requests proof that the message transport system (MTS) has submitted a message for delivery to the originally intended recipient, and FALSE otherwise.

Details

Identifier 0x0028, property type PT_BOOLEAN; property tag 0x0028000B

Comments

This MAPI property represents the corresponding X.400 and EDI submission envelope per-message attributes as defined by the X.400 and EDI message-handling standards. For X.400 environments, this property corresponds to the MH_T_PROOF_OF_SUBMSN_REQUESTED attribute.

See Also

PR_PROOF_OF_SUBMISSION property

PR_PROVIDER_DISPLAY

[New - Windows 95]

Contains the vendor-defined display name for a service provider.

Details

Identifier 0x3006, property type PT_TSTRING; property tag 0x3006001E

Comments

PR_PROVIDER_DISPLAY and PR_PROVIDER_DLL_NAME are defined only on profile sections belonging to service providers. They must be present in MAPISVC.INF.

See Also

PR_PROVIDER_DLL_NAME property

PR_PROVIDER_DLL_NAME

[New - Windows 95]

Contains the base filename of the MAPI service provider DLL.

Details

Identifier 0x300A, property type PT_TSTRING; property tag 0x300A001E

Comments

MAPI uses a DLL file naming convention. The base name 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_PROVIDER_DISPLAY property, PR_SERVICE_DLL_NAME property

PR_PROVIDER_ORDINAL

[New - Windows 95]

Contains the zero-based index of a service provider's position in a provider table created by IMsgServiceAdmin::GetProviderTable.

Details

Identifier 0x300D, property type PT_LONG; property tag 0x300D0003

Comments

Represents a computed property that appears only in the provider table. Sort the provider table by PR_PROVIDER_ORDINAL to display the transport order.

See Also

IMsgServiceAdmin::GetProviderTable method

PR_PROVIDER_SUBMIT_TIME

[New - Windows 95]

Contains the date and time when a message sender marked a message for submission by a transport provider to its underlying messaging system.

Details

Identifier 0x0048, property type PT_SYSTIME; property tag 0x00480040

Comments

This property is computed by transport providers, which add it to a message based on the results of message submission.

This MAPI property represents the corresponding X.400 and EDI submission envelope per-message attributes as defined by the X.400 and EDI message-handling standards.See Also

PR_CLIENT_SUBMIT_TIME property

PR_PROVIDER_UID

[New - Windows 95]

Contains the unique identifier (UID) of the service provider that is handling a message.

Details

Identifier 0x300C, property type PT_BINARY; property tag 0x300C0102

Comments

PR_PROVIDER_UID is a unique MAPIUID associated with (and usually hard-coded by) the provider. It is typically used by a client that is interested in only the address book containers supplied by a particular provider.

Appears only in the provider table.

See Also

PR_PROVIDER_DISPLAY property

PR_RADIO_TELEPHONE_NUMBER

[New - Windows 95]

Contains the radio telephone number of a messaging user.

Details

Identifier 0x3A1D, property type PT_TSTRING; property tag 0x3A1D001E

Comments

Several properties are available for the messaging user, including several telephone numbers and fields of the messaging user home mailing address.

See Also

PR_OFFICE_TELEPHONE_NUMBER property

PR_RCVD_REPRESENTING_ADDRTYPE

[New - Windows 95]

Contains the address type for the messaging user that is represented by another messaging user.

Details

Identifier 0x0077, property type PT_TSTRING; property tag 0x0077001E

Comments

When user B receives mail for user A, the PR_RECEIVED_BY_* properties are set to user B and the PR_RCVD_REPRESENTING_* properties are set to user A.

The PR_RCVD_REPRESENTING_* properties on a message indicate that one user receives the message on behalf of another messaging user. This property is often set by the transport provider, which is also responsible for authorization or verification of the representative. Note that the PR_RCVD_REPRESENTING_* properties are not required and may not be present for all messages.

Client applications that use this property to categorize messages should set default values for the PR_RCVD_REPRESENTING_* properties when they are not present, such as setting them to the same values as the PR_RECEIVED_BY_* properties.

This property represents one of five maintained for the represented messaging user. Opening the entry id to find additional information is a relatively expensive operation, so properties representing the commonly-used display name, search key, address type, and email address are also defined with the entryid.

See Also

PR_ADDRTYPE property, PR_RCVD_REPRESENTING_ADDRTYPE property, PR_RCVD_REPRESENTING_EMAIL_ADDRESS property, PR_RCVD_REPRESENTING_ENTRYID property, PR_RCVD_REPRESENTING_SEARCH_KEY property

PR_RCVD_REPRESENTING_EMAIL_ADDRESS

[New - Windows 95]

Contains the email address for the messaging user that is represented by another messaging user.

Details

Identifier 0x0078, property type PT_TSTRING; property tag 0x0078001E

Comments

When user B receives mail for user A, the PR_RECEIVED_BY_* properties are set to user B and the PR_RCVD_REPRESENTING_* properties are set to user A.

The PR_RCVD_REPRESENTING_* properties on a message indicate that one user receives the message on behalf of another messaging user. This property is often set by the transport provider, which is also responsible for authorization or verification of the delegate. Note that the PR_RCVD_REPRESENTING_* properties are not required and may not be present for all messages.

Client applications that use this property to categorize messages should set default values for the PR_RCVD_REPRESENTING_* properties when they are not present, such as setting them to the same values as the PR_RECEIVED_BY_* properties.

This property represents one of five maintained for the represented messaging user. Opening the entry id to find additional information is a relatively expensive operation, so properties representing the commonly-used display name, search key, address type, and email address are also defined with the entryid.

See Also

PR_RCVD_REPRESENTING_ADDRTYPE property, PR_RCVD_REPRESENTING_ENTRYID property, PR_RCVD_REPRESENTING_SEARCH_KEY property, PR_RECEIVED_BY_EMAIL_ADDRESS property, PR_RECEIVED_BY_NAME property

PR_RCVD_REPRESENTING_ENTRYID

[New - Windows 95]

Contains the entryid for the messaging user that is represented by another messaging user.

Details

Identifier 0x0043, property type PT_BINARY; property tag 0x00430102

Comments

When user B receives mail for user A, the PR_RECEIVED_BY_* properties are set to user B and the PR_RCVD_REPRESENTING_* properties are set to user A.

The PR_RCVD_REPRESENTING_* properties on a message indicate that one user receives the message on behalf of another messaging user. This property is often set by the transport provider, which is also responsible for authorization or verification of the delegate. Note that the PR_RCVD_REPRESENTING_* properties are not required and may not be present for all messages.

Client applications that use this property to categorize messages should set default values for the PR_RCVD_REPRESENTING_* properties when they are not present, such as setting them to the same values as the PR_RECEIVED_BY_* properties.

This property represents one of five maintained for the represented messaging user. Opening the entry id to find additional information is a relatively expensive operation, so properties representing the commonly-used display name, search key, address type, and email address are also defined with the entryid.

See Also

PR_RCVD_REPRESENTING_ADDRTYPE property, PR_RCVD_REPRESENTING_EMAIL_ADDRESS property, PR_RCVD_REPRESENTING_NAME property, PR_RCVD_REPRESENTING_SEARCH_KEY property, PR_RECEIVED_BY_ENTRYID property

PR_RCVD_REPRESENTING_NAME

[New - Windows 95]

Contains the entryid for the messaging user that is represented by another messaging user.

Details

Identifier 0x0044, property type PT_TSTRING; property tag 0x0044001E

Comments

When user B receives mail for user A, the PR_RECEIVED_BY_* properties are set to user B and the PR_RCVD_REPRESENTING_* properties are set to user A.

The PR_RCVD_REPRESENTING_* properties on a message indicate that one user receives the message on behalf of another messaging user. This property is often set by the transport provider, which is also responsible for authorization or verification of the delegate. Note that the PR_RCVD_REPRESENTING_* properties are not required and may not be present for all messages.

Client applications that use this property to categorize messages should set default values for the PR_RCVD_REPRESENTING_* properties when they are not present, such as setting them to the same values as the PR_RECEIVED_BY_* properties.

This property represents one of five maintained for the represented messaging user. Opening the entry id to find additional information is a relatively expensive operation, so properties representing the commonly-used display name, search key, address type, and email address are also defined with the entryid.

See Also

PR_RCVD_REPRESENTING_ADDRTYPE property, PR_RCVD_REPRESENTING_EMAIL_ADDRESS property, PR_RCVD_REPRESENTING_ENTRYID property, PR_RCVD_REPRESENTING_SEARCH_KEY property, PR_RECEIVED_BY_NAME property

PR_RCVD_REPRESENTING_SEARCH_KEY

[New - Windows 95]

Contains the entry identifier for the messaging user that is represented by another messaging user.

Details

Identifier 0x0052, property type PT_BINARY; property tag 0x00520102

Comments

When user B receives mail for user A, the PR_RECEIVED_BY_* properties are set to user B and the PR_RCVD_REPRESENTING_* properties are set to user A.

The PR_RCVD_REPRESENTING_* properties on a message indicate that one user receives the message on behalf of another messaging user. This property is often set by the transport provider, which is also responsible for authorization or verification of the delegate. Note that the PR_RCVD_REPRESENTING_* properties are not required and may not be present for all messages.

Client applications that use this property to categorize messages should set default values for the PR_RCVD_REPRESENTING_* properties when they are not present, such as setting them to the same values as the PR_RECEIVED_BY_* properties.

This property represents one of five maintained for the represented messaging user. Opening the entry id to find additional information is a relatively expensive operation, so properties representing the commonly-used display name, search key, address type, and email address are also defined with the entryid.

See Also

PR_RCVD_REPRESENTING_ADDRTYPE property, PR_RCVD_REPRESENTING_EMAIL_ADDRESS property, PR_RCVD_REPRESENTING_ENTRYID property, PR_RCVD_REPRESENTING_NAME property, PR_RECEIVED_BY_SEARCH_KEY property

PR_READ_RECEIPT_ENTRYID

[New - Windows 95]

Contains an entry identifier computed by a transport provider to indicate the messaging user to whom the messaging system should direct a read report for a message.

Details

Identifier 0x0046, property type PT_BINARY; property tag 0x00460102

Comments

This property is ignored unless PR_READ_RECEIPT_REQUESTED is set to TRUE.

If your client application or service provider wants to receive read receipts for messages, it should set PR_READ_RECEIPT_ENTRYID at message submission time. The transport should set this to the value of the PR_SENDER_ENTRYID property.

This property represents one of three maintained for the read receipt request. Opening the entry id to find additional information is a relatively expensive operation, so properties representing the commonly-used request flag and search key are also defined with the entryid.

See Also

PR_READ_RECEIPT_REQUESTED property, PR_READ_RECEIPT_SEARCH_KEY property, PR_SENDER_ENTRYID property

PR_READ_RECEIPT_REQUESTED

[New - Windows 95]

Contains TRUE if a message sender wants the messaging system to generate a read report when the recipient has read a message. It contains FALSE otherwise.

Details

Identifier 0x0029, property type PT_BOOLEAN; property tag 0x0029000B

Comments

If a message with this property set is deleted or expires before the messaging system can generate a read report, a nonread report is generated.

For interpersonal messaging in X.400 environments, this MAPI property corresponds to the submission envelope per-recipient IM_NOTIFICATION_REQUEST attribute.

PR_READ_RECEIPT_REQUESTED must be set to TRUE to validate the values in the properties PR_READ_RECEIPT_ENTRYID and PR_READ_RECEIPT_SEARCH_KEY.

This property represents one of three maintained for the read receipt request. Opening the entry id to find additional information is a relatively expensive operation, so properties representing the commonly-used request flag and search key are also defined with the entryid.

See Also

PR_READ_RECEIPT_ENTRYID property, PR_READ_RECEIPT_SEARCH_KEY property

PR_READ_RECEIPT_SEARCH_KEY

[New - Windows 95]

Contains the search key for a read report.

Details

Identifier 0x0053, property type PT_BINARY; property tag 0x00530102

Comments

This property represents one of three maintained for the read receipt request. Opening the entry id to find additional information is a relatively expensive operation, so properties representing the commonly-used request flag and search key are also defined with the entryid.

See Also

PR_READ_RECEIPT_ENTRYID property, PR_READ_RECEIPT_REQUESTED property

PR_RECEIPT_TIME

[New - Windows 95]

Contains the date and time of a delivery report.

Details

Identifier 0x002A, property type PT_SYSTIME; property tag 0x002A0040

Comments

This MAPI property corresponds to the interpersonal notification IM_RECEIPT_TIME attribute as defined by the X.400 message-handling standard.

See Also

PR_REPORT_ENTRYID property

PR_RECEIVED_BY_ADDRTYPE

[New - Windows 95]

Contains the actual recipient's e-mail address type, such as "SMTP".

Details

Identifier 0x3002, property type PT_TSTRING; property tag 0x3002001E

Comments

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.

Five properties are maintained for the message recipient: the address type, email address, entry id, search key, and display name.

See Also

PR_ADDRTYPE property, PR_RECEIVED_BY_EMAIL_ADDRESS property, PR_RECEIVED_BY_ENTRYID property, PR_RECEIVED_BY_NAME property, PR_RECEIVED_BY_SEARCH_KEY property

PR_RECEIVE_FOLDER_SETTINGS

[New - Windows 95]

Contains the receive folder settings.

Details

Identifier 0x3415, property type PT_OBJECT; property tag 0x3415000D

Comments

PR_RECEIVE_FOLDER_SETTINGS is a message story property.

See Also

PR_FOLDER_TYPE property

PR_RECEIVED_BY_EMAIL_ADDRESS

[New - Windows 95]

Contains the actual recipient's e-mail address.

Details

Identifier 0x3003, property type PT_TSTRING; property tag 0x3003001E

Comments

Five properties are maintained for the actual message recipient: the address type, email address, entry id, search key, and display name.

When the messaging user that receives the message is the same as the intended recipient, the PR_RECEIVED_BY_EMAIL_ADDRESS and PR_EMAIL_ADDRESS properties have the same value.

See Also

PR_EMAIL_ADDRESS property, PR_RECEIVED_BY_ADDRTYPE property, PR_RECEIVED_BY_ENTRYID property, PR_RECEIVED_BY_NAME property, PR_RECEIVED_BY_SEARCH_KEY property

PR_RECEIVED_BY_ENTRYID

[New - Windows 95]

Contains a message recipient's entry identifier set by the transport provider handling a message.

Details

Identifier 0x003F, property type PT_BINARY; property tag 0x003F0102

Comments

This property, along with the PR_RECEIVED_BY_NAME property, is required for all incoming messages.

This property represents one of five that are maintained for the message recipient. Opening the entry id to find additional information is a relatively expensive operation, so properties representing the commonly-used display name, search key, address type, and email address are also defined. The transport should set all five related PR_RECEIVED_BY_* properties.

For X.400 environments, this MAPI property corresponds to the MH_T_ACTUAL_RECIPIENT_NAME attribute as defined by the X.400 message-handling standard.

See Also

PR_RECEIVED_BY_ADDRTYPE property, PR_RECEIVED_BY_EMAIL_ADDRESS property, PR_RECEIVED_BY_NAME property, PR_RECEIVED_BY_SEARCH_KEY property, PR_SENDER_ENTRYID property

PR_RECEIVED_BY_NAME

[New - Windows 95]

Contains a message recipient's display name set by the transport provider handling a message.

Details

Identifier 0x0040, property type PT_TSTRING; property tag 0x0040001E

Comments

This property, along with the PR_RECEIVED_BY_ENTRYID property, is required for all incoming messages.

This property represents one of five that are maintained for the message recipient. Opening the entry id to find additional information is a relatively expensive operation, so properties representing the commonly-used display name, search key, address type, and email address are also defined. The transport should set all five related PR_RECEIVED_BY_* properties.

In X.400 and EDI environments, this MAPI property represents the corresponding delivery envelope per-message attributes as defined by the X.400 and EDI message-handling standards. X.400 and EDI transport providers use this property for messages delivered to client applications.

See Also

PR_RECEIVED_BY_ADDRTYPE property, PR_RECEIVED_BY_EMAIL_ADDRESS property, PR_RECEIVED_BY_ENTRYID property, PR_RECEIVED_BY_SEARCH_KEY property, PR_SENDER_NAME property

PR_RECEIVED_BY_SEARCH_KEY

[New - Windows 95]

Contains the message recipient's search key set by the transport provider handling the message.

Details

Identifier 0x0051, property type PT_BINARY; property tag 0x00510102

Comments

This property represents one of five that are maintained for the message recipient. Opening the entry id to find additional information is a relatively expensive operation, so properties representing the commonly-used display name, search key, address type, and email address are also defined. The transport should set all five related PR_RECEIVED_BY_* properties.

See Also

PR_RECEIVED_BY_ADDRTYPE property, PR_RECEIVED_BY_EMAIL_ADDRESS property, PR_RECEIVED_BY_ENTRYID property, PR_RECEIVED_BY_NAME property, PR_SENDER_NAME property

PR_RECIPIENT_CERTIFICATE

[New - Windows 95]

Contains a message recipient's ASN.1 certificate for use in a report.

Details

Identifier 0x0C13, property type PT_BINARY; property tag 0x0C130102

Comments

This MAPI property corresponds to the report per-recipient MH_T_RECEIPT_CERTIFICATE attribute as defined by the X.400 message-handling standard.

See Also

PR_RECIPIENT_NUMBER_FOR_ADVICE property

PR_RECIPIENT_NUMBER_FOR_ADVICE

[New - Windows 95]

Contains a message recipient's telephone number to call to advise of the physical delivery of a message.

Details

Identifier 0x0C14, property type PT_TSTRING; property tag 0x0C14001E

Comments

For X.400 environments, this MAPI property corresponds to the MH_T_RECIP_NUMBER_FOR_ADVICE attribute. X.400 and EDI transport providers use this property for messages delivered to client applications.

See Also

PR_RECIPIENT_REASSIGNMENT_PROHIBITED property

PR_RECIPIENT_REASSIGNMENT_PROHIBITED

[New - Windows 95]

Contains TRUE if recipient reassignment is prohibited, and FALSE otherwise.

Details

Identifier 0x002B, property type PT_BOOLEAN; property tag 0x002B000B

Comments

For X.400 environments, this MAPI property corresponds to MH_T_REASSIGNMENT_PROHIBITED. X.400 environments use this property for probe messages.

See Also

PR_RECIPIENT_STATUS property

PR_RECIPIENT_STATUS

[New - Windows 95]

Reserved for use by MAPI only.

Details

Identifier 0x0E15, property type PT_LONG; property tag 0x0E150003

Comments

Do not use this property. PR_RECIPIENT_STATUS is reserved for use by MAPI.

See Also

PR_RECIPIENT_TYPE property

PR_RECIPIENT_TYPE

[New - Windows 95]

Contains the recipient type for the message.

Details

Identifier 0x0C15, property type PT_LONG; property tag 0x0C150003

Comments

The recipient type must take one of the three values. None, either, or both of the other flags can also be set.

The PR_RECIPIENT_TYPE property can have 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	The recipient is a P1 resend recipient.
MAPI_SUBMITTED	The recipient is already processed.

See Also

PR_RECIPIENT_STATUS property

PR_RECORD_KEY

[New - Windows 95]

Contains a unique binary-comparable identifier for an object.

Details

Identifier 0x0FF9, property type PT_BINARY; property tag 0x0FF90102

Comments

This property uniquely identifies the object within a message. This identifier is the only attachment characteristic guaranteed to stay the same after the message is closed and reopened. (It must remain unique across sessions so that it will be guaranteed to stay the same after the message is closed and reopened.)

Mail user, distribution list, address book container, folder, message store (both public and private), attachment, and all message objects must furnish this property.

You can compare PR_RECORD_KEY values using memcmp. This is not possible for entryid values.

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 PR_STORE_RECORD_KEY.

The distinction between PR_RECORD_KEY and PR_SEARCH_KEY is that PR_RECORD_KEY is specific to the object, while PR_SEARCH_KEY can be copied to other objects. For example, two copies of the object could have the same search key, but they must have different record keys.

See Also

PR_ENTRYID property, PR_SEARCH_KEY property, PR_STORE_RECORD_KEY property

PR_REDIRECTION_HISTORY

[New - Windows 95]

Contains information about the route covered by a delivered message.

Details

Identifier 0x002C, property type PT_BINARY; property tag 0x002C0102

Comments

X.400 and EDI transport providers use this property for messages delivered to client applications.

For X.400 environments, this property corresponds to the MH_T_REDIRECTION_HISTORY attribute.

See Also

PR_AUTO_FORWARDED property

PR_REGISTERED_MAIL_TYPE

[New - Windows 95]

Contains the type of registration used for physical delivery of a message.

Details

Identifier 0x0C16, property type PT_LONG; property tag 0x0C160003

Comments

For X.400 environments, this MAPI property corresponds to the MH_T_REGISTRATION attribute. X.400 and EDI transport providers use this property for messages delivered to client applications.

See Also

PR_PHYSICAL_FORWARDING_ADDRESS property, PR_X400_CONTENT_TYPE property

PR_RELATED_IPMS

[New - Windows 95]

Contains a list of message identifiers (as in the PR_SEARCH_KEY property) to which a message is related.

Details

Identifier 0x002D, property type PT_BINARY; property tag 0x002D0102

Comments

This MAPI property corresponds to the submission envelope per-message IM_RELATED_IPMS attribute as defined by the X.400 message-handling standard. X.400 transport providers use this property for messages delivered to client applications.

See Also

PR_SEARCH_KEY property

PR_REMOTE_PROGRESS

[New - Windows 95]

Contains a numeric code indicating the status of a remote transfer.

Details

Identifier 0x3E0B, property type PT_LONG; property tag 0x3E0B0003

Comments

The text associated with the numeric status code appears in the PR_REMOTE_PROGRESS_TEXT property.

The following flags can be set for the PR_REMOTE_PROGRESS bitmask:

Value	Description
MSGSTATUS_REMOTE_DELETE	Indicates the message transfer is deleted.
MSGSTATUS_REMOTE_DOWNLOAD	Indicates the message transfer is in progress.

See Also

PR_REMOTE_PROGRESS_TEXT property

PR_REMOTE_PROGRESS_TEXT

[New - Windows 95]

Contains an ASCII string indicating the status of a remote transfer.

Details

Identifier 0x3E0C, property type PT_TSTRING; property tag 0x3E0C001E

Comments

A numeric code associated with this text is passed in the PR_REMOTE_PROGRESS property.

See Also

PR_REMOTE_PROGRESS property

PR_REMOTE_VALIDATE_OK

[New - Windows 95]

When TRUE, indicates that the remote viewer can make repeated **IMAPIStatus::ValidateState** calls to the transport. When FALSE, indicates that the remote viewer can not make the calls.

Details

Identifier 0x3E0D, property type PT_BOOLEAN; property tag 0x3E0D000B

Comments

This 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** repeatedly. A value of FALSE indicates that the remote viewer can not make these calls.

The transport usually sets this property dynamically, setting the value to FALSE to disable additional calls when the transport has a sufficient amount of processing to perform. When the transport is done, it then sets the value to TRUE to allow the client to make further **IMAPIStatus::ValidateState** calls.

See Also

IMAPIStatus::ValidateState method, PR_REMOTE_PROGRESS property

PR_RENDERING_POSITION

[New - Windows 95]

Contains an ordinal offset (in characters) to use in rendering an attachment within the main message text.

Details

Identifier 0x370B, property type PT_LONG; property tag 0x370B0003

Comments

When the supplied offset is -1 (0xFFFFFFFF), the attachment is not rendered using PR_RENDERING_POSITION. All values other than -1 indicate the position within PR_BODY at which the attachment is to be rendered.

Identifies where in the body of the message MAPI should render the attachment. This is not used for RTF messages.

Do not use this property with RTF text. A special RTF tag is defined to indicate the rendering position. All other attachment objects must furnish this property.

See Also

PR_BODY property

PR_REPLY_RECIPIENT_ENTRIES

[New - Windows 95]

Contains a sized array of entry identifiers for recipients that are to get the reply.

Details

Identifier 0x004F, property type PT_BINARY; property tag 0x004F0102

Comments

When this optional property is not present, the reply message is sent only to PR_SENDER. 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 normal reply logic.

Note that PR_REPLY_RECIPIENT_ENTRIES contains a FLATENTRYLIST structure and is not a multivalued property.

For X.400 environments, this MAPI property corresponds to the delivery envelope per-message IM_REPLY_RECIPIENTS attribute as defined by the X.400 message-handling standard. X.400 transport providers use this property for messages delivered to client applications.

See Also

FLATENTRYLIST structure, PR_SENDER_ENTRYID property

PR_REPLY_RECIPIENT_NAMES

[New - Windows 95]

Contains a list of display names of recipients that are to get the reply.

Details

Identifier 0x0050, property type PT_TSTRING; property tag 0x0050001E

Comments

Corresponds to the entryids that appear in PR_REPLY_RECIPIENT_ENTRIES. Contains one string, with the display names separated by semicolons.

See Also

FLATENTRYLIST structure, PR_SENDER_NAME property

PR_REPLY_REQUESTED

[New - Windows 95]

Contains TRUE if a message sender requests a reply from a recipient, and FALSE otherwise.

Details

Identifier 0x0C17, property type PT_BOOLEAN; property tag 0x0C17000B

Comments

For X.400 environments, this MAPI property corresponds to the submission envelope per-recipient IM_REPLY_REQUESTED attribute as defined by the X.400 message-handling standard.

See Also

PR_REPLY_TIME property

PR_REPLY_TIME

[New - Windows 95]

Contains the date and time by which a reply is expected for a message.

Details

Identifier 0x0030, property type PT_SYSTIME; property tag 0x00300040

Comments

For X.400 environments, this MAPI property corresponds to the submission envelope per-message IM_REPLY_TIME attribute as defined by the X.400 message-handling standard. X.400 transport providers use this property for messages delivered to client applications.

See Also

PR_REPLY_REQUESTED property

PR_REPORT_ENTRYID

[New - Windows 95]

Contains the entry identifier for the recipient that should get reports for the message.

Details

Identifier 0x0045, property type PT_BINARY; property tag 0x00450102

Comments

Specifies the recipient for delivery reports (DR) and non-delivery reports (NDR) related to the message.

This property represents one of three maintained for the report recipient. Opening the entry id to find additional information is a relatively expensive operation, so properties representing the commonly-used display name and search key are defined along with the entryid.

Your application should set this property at message submission time. If it is not set, the reports are sent to the recipient specified by PR_SENDER_ENTRYID.

See Also

PR_REPORT_NAME property, PR_REPORT_SEARCH_KEY property, PR_SENDER_ENTRYID property

PR_REPORT_NAME

[New - Windows 95]

Contains the display name of the recipient that should get reports for the message.

Details

Identifier 0x003A, property type PT_TSTRING; property tag 0x003A001E

Comments

Specifies the display name of the recipient that should get delivery reports (DR) and non-delivery reports (NDR) related to the message. For delivery reports, this is a per-recipient property. On read receipts and non-read receipts, this is a message property.

This property represents one of three maintained for the report recipient. Opening the entry id to find additional information is a relatively expensive operation, so properties representing the commonly-used display name and search key are defined along with the entryid.

Your application should set this property at message submission time. If it is not set, the reports are sent to the recipient specified by the PR_SENDER_ENTRYID and other related PR_SENDER_* properties.

See Also

PR_REPORT_ENTRYID property, PR_SENDER_NAME property

PR_REPORT_SEARCH_KEY

[New - Windows 95]

Contains the search key for the recipient that should get the report.

Details

Identifier 0x0054, property type PT_BINARY; property tag 0x00540102

Comments

Specifies the search key for the recipient that should get delivery reports (DR) and non-delivery reports (NDR) related to the message.

This property represents one of three maintained for the report recipient. Opening the entry id to find additional information is a relatively expensive operation, so properties representing the commonly-used display name and search key are defined along with the entryid.

See Also

PR_REPORT_ENTRYID property, PR_REPORT_NAME property, PR_SENDER_SEARCH_KEY property

PR_REPORT_TAG

[New - Windows 95]

Contains a binary tag value that the messaging system should copy to any report generated for the message.

Details

Identifier 0x0031, property type PT_BINARY; property tag 0x00310102

Comments

The PR_REPORT_TAG property is used for correlation between a message and its corresponding report.

See Also

PR_REPORT_NAME property, PR_REPORT_TIME property

PR_REPORT_TEXT

[New - Windows 95]

Contains optional text for a read report or a nonread report generated by the messaging system.

Details

Identifier 0x1001, property type PT_TSTRING; property tag 0x1001001E

Comments

For X.400 environments, PR_REPORT_TEXT corresponds to the IM_SUPPLEMENTARY_RECEIPT_INFO attribute as defined by the X.400 message-handling standard.

See Also

PR_REPORT_NAME property

PR_REPORT_TIME

[New - Windows 95]

Contains the date and time when the messaging system generated a read report or a nonread report.

Details

Identifier 0x0032, property type PT_SYSTIME; property tag 0x00320040

Comments

PR_REPORT_TIME represents a per-recipient property in the delivery report and represents a message property on read and non-read reports.

See Also

PR_REPORT_NAME property, PR_REPORT_TAG property

PR_REPORTING_DL_NAME

[New - Windows 95]

Contains the display name of a distribution list for which the messaging system is delivering a report.

Details

Identifier 0x1003, property type PT_BINARY; property tag 0x10030102

Comments

For X.400 environments, this MAPI property corresponds to the report per-message MH_T_REPORTING_DL_NAME attribute as defined by the X.400 message-handling standard.

See Also

PR_REPORTING_MTA_CERTIFICATE property

PR_REPORTING_MTA_CERTIFICATE

[New - Windows 95]

Contains an identifier for the message transfer agent (MTA) that generated a report.

Details

Identifier 0x1004, property type PT_BINARY; property tag 0x10040102

Comments

This MAPI property represents the corresponding X.400 report per-message attribute as defined by the X.400 message-handling standard.

See Also

PR_REPORTING_DL_NAME property

PR_REQUESTED_DELIVERY_METHOD

[New - Windows 95]

Contains a binary array of delivery methods (service providers), in order of a message sender's preference.

Details

Identifier 0x0C18, property type PT_BINARY; property tag 0x0C180102

Comments

For X.400 environments, this property corresponds to the MH_T_PREFERRED_DELIVERY_MODE attribute. X.400 and EDI transport providers use this property for messages delivered to client applications. X.400 environments also use this property for probe messages.

See Also

PR_PROOF_OF_DELIVERY_REQUESTED property

PR_RESOURCE_FLAGS

[New - Windows 95]

Appears on message services and providers. This property also represents a column in several tables.

Details

Identifier 0x3009, property type PT_LONG; property tag 0x30090003

Comments

All static flags are expected to be set in MAPISVC.INF and never modified. All flags that are not static can be set by an Extended MAPI method.

The following flags can be supplied in PR_RESOURCE_FLAGS for a messaging service, the messaging services table, and a profile.

Value	**Description**
SERVICE_DEFAULT_STORE	Indicates the messaging 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. This flag applies only to messaging services.
SERVICE_NO_PRIMARY_IDENTITY	Indicates that service level flag that should be set to indicate that none of the providers in the service can be used to supply an identity. Either this flag or SERVICE_PRIMARY_IDENTITY should be set, but not both. This flag represents a static value defined in MAPISVC.INF that is not changed by any MAPI method.
SERVICE_PRIMARY_IDENTITY	Indicates that the corresponding service contains the provider used for the primary identity for this session. Use IMAPIMsgServiceAdmin::SetPrimaryIdentity to set this flag. Either this flag or SERVICE_NO_PRIMARY_IDENTITY should be set, but not both.
SERVICE_SINGLE_COPY	Indicates that any attempt to create or copy this service into a profile where the service already exists should fail. To create a single copy service add the PR_RESOURCE_FLAGS property to the service's section in MAPISVC.INF and set this flag. This flag represents a static value defined in MAPISVC.INF that is not changed by any MAPI method.

See Also

IMsgServiceAdmin::MsgServiceTransportOrder method, PR_IDENTITY_ENTRYID property, PR_OWN_STORE_ENTRYID property, PR_SENT_REPRESENTING_NAME property

PR_RESOURCE_METHODS

[New - Windows 95]

Contains a bitmask of flags indicating the status object methods that are supported.

Details

Identifier 0x3E02, property type PT_LONG; property tag 0x3E020003

Comments

All system resources contain a status table for the IMAPIStatus interface and set flags in this 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.

The following flags can be set for the PR_RESOURCE_METHODS bitmask:

Value	Description
STATUS_CHANGE_PASSWORD	Indicates the ChangePassword method is present.
STATUS_FLUSH_QUEUES	Indicates the FlushQueues method is present.
STATUS_SETTINGS_DIALOG	Indicates the SettingsDialog method is present.
STATUS_VALIDATE_STATE	Indicates the ValidateState method is present.

See Also

IMAPIStatus::ChangePassword method, **IMAPIStatus::FlushQueues** method, **IMAPIStatus::SettingsDialog** method, **IMAPIStatus::ValidateState** method

PR_RESOURCE_PATH

[New - Windows 95]

Contains a path to the service provider's server.

Details

Identifier 0x3E07, property type PT_TSTRING; property tag 0x3E07001E

Comments

This property appears on a status object and or a status table row.

For example, this represents the suggested path where the user would find resources. The definition of the proeprty is provider specific. For example, a scheduling application would use this property to specify the suggested location for its scheduling application files.

The messaging user profile furnishes the property as a convenience so that a client application does not have to prompt the messaging user for a network path or network drive letter.

See Also

PR_RESOURCE_TYPE property

PR_RESOURCE_TYPE

[New - Windows 95]

Contains a constant indicating the service provider type.

Details

Identifier 0x3E03, property type PT_LONG; property tag 0x3E030003

Comments

The PR_RESOURCE_TYPE property can have 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	MAPI subsystem
MAPI_TRANSPORT_PROVIDER	Transport provider

See Also

PR_RESOURCE_PATH property

PR_RESPONSE_REQUESTED

[New - Windows 95]

Contains TRUE if a message sender requests a response, and FALSE otherwise.

Details

Identifier 0x0063, property type PT_BOOLEAN; property tag 0x0063000B

Comments

PR_RESPONSE_REQUESTED is a message envelope property.

See Also

PR_OWNER_APPT_ID property

PR_RESPONSIBILITY

[New - Windows 95]

Contains TRUE if a transport provider accepts responsibility for sending the message to the specified recipient, and FALSE otherwise.

Details

Identifier 0x0E0F, property type PT_BOOLEAN; property tag 0x0E0F000B

Comments

Indicates that the transport provider is responsible for sending this message to this recipient. The transport sets the value to true when it accepts responsibility and sends the message.

This property exists only on the recipient table.

For X.400 environments, PR_RESPONSIBILITY corresponds to the MH_T_MTA_RESPONSIBILITY attribute as defined by the X.400 message-handling standard.

See Also

PR_DELETE_AFTER_SUBMIT property

PR_RETURNED_IPM

[New - Windows 95]

Contains TRUE if the original interpersonal message is returned with a nondelivery report, and FALSE otherwise.

Details

Identifier 0x0033, property type PT_BOOLEAN; property tag 0x0033000B

Comments

This property represents the interpersonal notification IM_RETURNED_IPM attribute as defined by the X.400 message-handling standard.

See Also

PR_IPM_ID property, PR_RELATED_IPMS property

PR_ROW_TYPE

[New - Windows 95]

Contains a value that indicates the type of a row in a table.

Details

Identifier 0x0FF5, property type PT_LONG; property tag 0x0FF50003

Comments

PR_ROW_TYPE appears only on contents tables. A category only exists when it has items.

The PR_ROW_TYPE property can have the following values:

Value	Description
TBL_LEAF_ROW	The row represents actual data, rather than a category row.
TBL_EMPTY_CATEGORY	
TBL_EXPANDED_CATEGORY	Indicates that the category is expanded; UI usually displays these with the plus sign '+' next to it.
TBL_COLLAPSED_CATEGORY	Indicates that the category is collapsed; UI usually displays these with the minus sign '-' next to it.

See Also

PR_ROWID property

PR_ROWID

[New - Windows 95]

Contains a unique identifier for a recipient in a recipient table or status table.

Details

Identifier 0x3000, property type PT_LONG; property tag 0x30000003

Comments

PR_ROWID is an ephemeral value that is valid 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

[New - Windows 95]

Contains the compressed rich text version of the body of the message.

Details

Identifier 0x1009, property type PT_BINARY; property tag 0x10090102

Comments

Contains the same content as PR_BODY, in compressed RTF format. To obtain the contents of the property, call OpenProperty, then call WrapCompressedRTFStream with the setting MAPI_READ.

Note that when opening PR_COMPRESSED_RTF for writing, it is always opened with the MAPI_MODIFY and MAPI_CREATE flags. This ensures that the new data completely replaces any old data and that the stream writes are performed using the minimum number of store updates.

See Also

WrapCompressedRTFStream function

PR_RTF_IN_SYNC

[New - Windows 95]

Indicates whether PR_RTF_COMPRESSED contains the same content as the message body PR_BODY.

Details

Identifier 0x0E1F, property type PT_BOOLEAN; property tag 0x0E1F000B

Comments

Set the value to FALSE to force synchronization. RTF-aware message stores should perform the synchronization during a SaveChanges call.

See Also

PR_RTF_COMPRESSED property

PR_RTF_SYNC_BODY_COUNT

[New - Windows 95]

Contains a count of the significant characters of the message body.

Details

Identifier 0x1007, property type PT_LONG; property tag 0x10070003

Comments

The RTFSync routine computes the count of characters in the body 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.

This property is an RTF auxiliary property. The RTF auxiliary properties are used by the RTFSync function and are not intended to be used directly by client applications.

See Also

PR_RTF_SYNC_BODY_CRC property

PR_RTF_SYNC_BODY_CRC

[New - Windows 95]

Contains the cyclical redundancy check (CRC) computed for the body text.

Details

Identifier 0x1006, property type PT_LONG; property tag 0x10060003

Comments

The RTFSync routine computes the CRC using only the characters that it considers to be significant to the message. For example, some white space and other ignorable unimportant characters are omitted from the CRC.

This property is an RTF auxiliary property. The RTF auxiliary properties are used by the RTFSync function and are not intended to be used directly by client applications.

See Also

PR_RTF_COMPRESSED property

PR_RTF_SYNC_BODY_TAG

[New - Windows 95]

Contains significant characters that appear at the beginning of the message body.

Details

Identifier 0x1008, property type PT_TSTRING; property tag 0x1008001E

Comments

The RTFSync routine uses the body tag to indicate the beginning of the body of the message. When the body is modified, the tag is used to find the beginning of the previous body.

This property is an RTF auxiliary property. The RTF auxiliary properties are used by the RTFSync function and are not intended to be used directly by client applications.

See Also

PR_SEND_RICH_INFO property

PR_RTF_SYNC_PREFIX_COUNT

[New - Windows 95]

Contains the count of the number of ignorable characters that appear before the significant characters.

Details

Identifier 0x1010, property type PT_LONG; property tag 0x10100003

Comments

The count of prefix characters does not include white space.

This property is an RTF auxiliary property. The RTF auxiliary properties are used by the RTFSync function and are not intended to be used directly by client applications.

See Also

PR_SEND_RICH_INFO property

PR_RTF_SYNC_TRAILING_COUNT

[New - Windows 95]

Contains a count of the ignorable characters that appear after the significant characters of the message.

Details

Identifier 0x1011, property type PT_LONG; property tag 0x10110003

Comments

This property is an RTF auxiliary property. The RTF auxiliary properties are used by the RTFSync function and are not intended to be used directly by client applications.

See Also

PR_SEND_RICH_INFO property

PR_SEARCH

[New - Windows 95]

Contains an **IMAPIContainer** object that is used to conduct advanced searches.

Details

Identifier 0x3607, property type PT_OBJECT; property tag 0x3607000D

Comments

If your container does not support advanced search capabilities, you do not have to supply this property.

See Also

IMAPIContainer : IMAPIProp interface

PR_SEARCH_KEY

[New - Windows 95]

Contains a binary-comparable key that uniquely identifies an object for searches.

Details

Identifier 0x300B, property type PT_BINARY; property tag 0x300B0102

Comments

Message and distribution list objects must furnish this property.

MAPI uses specific rules for constructing search keys for message recipients. The search key is formed by combining the address type (in upper case characters), the colon character ':', the email address in canonical form, and the terminating null character. The canonical form means that case-sensitive addresses appear in the correct case; addresses that are not case-sensitive are converted to uppercase.

For X.400 and EDI environments, this property represents the corresponding submission envelope per-message attributes as defined by the X.400 and EDI message-handling standards. For X.400 environments, this property corresponds to the IM_THIS_IPMS attribute. EDI environments use PR_SEARCH_KEY for messages delivered to client applications.

See Also

PR_RESPONSIBILITY property, PR_STORE_RECORD_KEY property

PR_SECURITY

[New - Windows 95]

Contains a value that indicates the security level of a message. The underlying messaging system chooses whether to use this value.

Details

Identifier 0x0034, property type PT_LONG; property tag 0x00340003

Comments

The message class can determine whether the underlying messaging system honors the security settings.

The following flags can be set:

Value	Description
SECURITY_SIGNED	
SECURITY_ENCRYPTED	

See Also

PR_MESSAGE_CLASS property

PR_SELECTABLE

[New - Windows 95]

Contains TRUE if the entry in the template table or one-off table can be selected, and contains FALSE otherwise.

Details

Identifier 0x3609, property type PT_BOOLEAN; property tag 0x3609000B

Comments

Applies only to the template table or the one-off table. This does not apply to the address book hierarchy table.

This property is used primarily for formatting. For example, you might want to group X.400 templates or all gateway templates in the table by creating an entry that indicates the heading for the group. Setting the PR_SELECTABLE property to FALSE for the heading ensures that the user cannot select this heading entry, and that the user can select only the actual templates.

See Also

PR_FOLDER_TYPE property

PR_SEND_RICH_INFO

[New - Windows 95]

Indicates whether the recipient can receive all message content, including rich text and OLE objects.

Details

Identifier 0x3A40, property type PT_BOOLEAN; property tag 0x3A40000B

Comments

This property indicates whether the recipient is MAPI-enabled.

When TRUE, the transport or gateway can transmit the full content of the message, including rich text and OLE objects. The transport or gateway should use TNEF.

When FALSE, the transport or gateway is free to discard message content that the native clients of that transport cannot use. For example, when the clients cannot use rich text, the transport may choose to send only plain text.

When the property is not defined, default behavior is determined by the implementation of the transport, MTA, or gateway. Address book providers are not required to support this property. For example, a tighly coupled address book and transport can choose to send TNEF and never use rich text.

By default, MAPI sets the value of this property to TRUE. To set PR_SEND_RICH_INFO to FALSE at the time you call **IAddrBook::CreateOneOff** or **IMAPISupport::CreateOneOff**, set the MAPI_SEND_NO_RICH_INFO bit in the *ulFlags* parameter. One-offs created by UI use the value specified by the creating template.

On calls to **IAddrBook::ResolveName** when the name is unresolved and can be interpreted as an Internet address, PR_SEND_RICH_INFO is set to FALSE. The display name of the unresolved entry must be in the format *x*@*y*.*z*, such as "pete@pinecone.com".

See Also

IAddrBook::CreateOneOff method, **IMAPISupport::CreateOneOff** method, PR_ATTACH_DATA_OBJ property, PR_RTF_COMPRESSED property

PR_SENDER_ADDRTYPE

[New - Windows 95]

Contains a message sender's address type set by the transport provider handling a message.

Details

Identifier 0x0C1E, property type PT_TSTRING; property tag 0x0C1E001E

Comments

This property represents one of several maintained for the sender. Opening the entryid to find additional information is a relatively expensive operation, so properties representing the commonly-used display name, search key, address type, and email address are defined along with the entryid.

When handling a message with sender delegate access, the transport provider must use both the PR_SENDER_* and PR_SENT_REPRESENTING_* properties.

Clients should not set these properties in messages that they send. The transport provider can choose to make sure that the properties exist and can choose to set these properties to the values for the current messaging user. If these properties are not set, MAPI may determine that it cannot send the message.

See Also

PR_ADDRTYPE property, PR_SENT_REPRESENTING_ADDRTYPE property

PR_SENDER_EMAIL_ADDRESS

[New - Windows 95]

Contains a message sender's email address set by the transport provider handling a message.

Details

Identifier 0x0C1F, property type PT_TSTRING; property tag 0x0C1F001E

Comments

This property represents one of several maintained for the sender. Opening the entryid to find additional information is a relatively expensive operation, so properties representing the commonly-used display name, search key, address type, and email address are defined along with the entryid.

When handling a message with sender delegate access, the transport provider must use both the PR_SENDER_* and PR_SENT_REPRESENTING_* properties.

Clients should not set these properties in messages that they send. The transport provider can choose to make sure that the properties exist and can choose to set these properties to the values for the current messaging user. If these properties are not set, MAPI may determine that it cannot send the message.

See Also

PR_EMAIL_ADDRESS property, PR_SENT_REPRESENTING_EMAIL_ADDRESS property

PR_SENDER_ENTRYID

[New - Windows 95]

Contains a message sender's entry identifier set by the transport provider handling a message.

Details

Identifier 0x0C19, property type PT_BINARY; property tag 0x0C190102

Comments

This property represents one of several maintained for the sender. Opening the entryid to find additional information is a relatively expensive operation, so properties representing the commonly-used display name, search key, address type, and email address are defined along with the entryid.

When handling a message with sender delegate access, the transport provider must use both the PR_SENDER_* and PR_SENT_REPRESENTING_* properties.

Clients can read these properties in a receive message, but should not set these properties in messages that they send. The transport provider can choose to make sure that the properties exist and can choose to set these properties to the values for the current messaging user. If these properties are not set, MAPI may determine that it cannot send the message.

Transports should ignore any present settings and set the current one. MAPI should use the current values. If they do not exist, MAPI should use default values.

For X.400 environments, PR_SENDER_ENTRYID corresponds to the IM_ORIGINATOR attribute, or is used to generate the IM_FREE_FORM_NAME attribute, depending on the object, as defined by the X.400 message-handling standard.

See Also

PR_SENDER_NAME property, PR_SENDER_SEARCH_KEY property, PR_SENT_REPRESENTING_ENTRYID property, PR_SENT_REPRESENTING_NAME property

PR_SENDER_NAME

[New - Windows 95]

Contains a message sender's display name set by the transport provider handling a message.

Details

Identifier 0x0C1A, property type PT_TSTRING; property tag 0x0C1A001E

Comments

This property represents one of several maintained for the sender. Opening the entryid to find additional information is a relatively expensive operation, so properties representing the commonly-used display name, search key, address type, and email address are defined along with the entryid.

Clients should not set these properties in messages that they send. The transport provider can choose to make sure that the properties exist and can choose to set these properties to the values for the current messaging user. If these properties are not set, MAPI may determine that it cannot send the message.

When handling a message with sender delegate access, the transport provider must use both the PR_SENDER_* and PR_SENT_REPRESENTING_* properties.

For X.400 environments, PR_SENDER_NAME corresponds to the IM_FREE_FORM_NAME and MH_T_ORIGINATOR_NAME attributes as defined by the X.400 message-handling standard.

See Also

PR_SENDER_ENTRYID property, PR_SENDER_SEARCH_KEY property, PR_SENT_REPRESENTING_ENTRYID property, PR_SENT_REPRESENTING_NAME property

PR_SENDER_SEARCH_KEY

[New - Windows 95]

Contains the message sender's search key set by the transport provider handling a message.

Details

Identifier 0x0C1D, property type PT_BINARY; property tag 0x0C1D0102

Comments

This property represents one of several maintained for the sender. Opening the entryid to find additional information is a relatively expensive operation, so properties representing the commonly-used display name, search key, address type, and email address are defined along with the entryid.

Clients should not set these properties in messages that they send. The transport provider can choose to make sure that the properties exist and can choose to set these properties to the values for the current messaging user. If these properties are not set, MAPI may determine that it cannot send the message.

When handling a message with sender delegate access, the transport provider must use both the PR_SENDER_* and PR_SENT_REPRESENTING_* properties.

See Also

PR_SENDER_ENTRYID property, PR_SENDER_NAME property,
PR_SENT_REPRESENTING_SEARCH_KEY property

PR_SENSITIVITY

[New - Windows 95]

Contains a value indicating a message sender's opinion about the sensitivity of a message.

Details

Identifier 0x0036, property type PT_LONG; property tag 0x00360003

Comments

This MAPI property represents the submission envelope per-message IM_SENSITIVITY attribute as defined by the X.400 message-handling standard.

X.400 transport providers use this property for messages delivered to client applications.

The PR_SENSITIVITY property can have the following values:

Value	**Description**
SENSITIVITY_NONE	The message has no sensitivity.
SENSITIVITY_PERSONAL	The message is personal.
SENSITIVITY_PRIVATE	The message is private.
SENSITIVITY_COMPANY_CONFIDENTIAL	The message is designated Company Confidential.

See Also

PR_IMPORTANCE property

PR_SENT_REPRESENTING_ADDRTYPE

[New - Windows 95]

Contains the address type for the messaging user represented by the sender.

Details

Identifier 0x0068, property type PT_TSTRING; property tag 0x0068001E

Comments

When user B sends mail for user A, the PR_SENDER_* properties are set to user B and the PR_SENT_REPRESENTING_* properties are set to user A.

This property represents one of several maintained for the messaging user that delegates the send operation to another user. Opening the entry id to find additional information is a relatively expensive operation, so properties representing the commonly-used display name, search key, address type, and email address are defined along with the entryid.

When this property is not set, PR_SENDER_ADDRTYPE is used.

See Also

PR_RCVD_REPRESENTING_ADDRTYPE property, PR_SENDER_ADDRTYPE property

PR_SENT_REPRESENTING_EMAIL_ADDRESS

[New - Windows 95]

Contains the email address for the messaging user represented by the sender.

Details

Identifier 0x0069, property type PT_TSTRING; property tag 0x0069001E

Comments

When user B sends mail for user A, the PR_SENDER_* properties are set to user B and the PR_SENT_REPRESENTING_* properties are set to user A.

This property represents one of several maintained for the messaging user that delegates the send operation to another user. Opening the entry id to find additional information is a relatively expensive operation, so properties representing the commonly-used display name, search key, address type, and email address are defined along with the entryid.

When this property is not set, PR_SENDER_EMAIL_ADDRESS is used.

See Also

PR_RCVD_REPRESENTING_EMAIL_ADDRESS property, PR_SENDER_EMAIL_ADDRESS property

PR_SENT_REPRESENTING_ENTRYID

[New - Windows 95]

Contains the entry identifier for the messaging user represented by the sender.

Details

Identifier 0x0041, property type PT_BINARY; property tag 0x00410102

Comments

When user B sends mail for user A, the PR_SENDER_* properties are set to user B and the PR_SENT_REPRESENTING_* properties are set to user A.

This property represents one of several maintained for the messaging user that delegates the send operation to another user. Opening the entry id to find additional information is a relatively expensive operation, so properties representing the commonly-used display name, search key, address type, and email address are defined along with the entryid.

When this property is not set, the PR_SENDER_ENTRYID property is used.

To generate a reply message, a client application should copy PR_RCVD_REPRESENTING_ENTRYID from the received message into the PR_SENT_REPRESENTING_ENTRYID property of the reply.

See Also

PR_RCVD_REPRESENTING_ENTRYID property, PR_SENDER_ENTRYID property

PR_SENT_REPRESENTING_NAME

[New - Windows 95]

Contains the display name for the messaging user represented by the sender.

Details

Identifier 0x0042, property type PT_TSTRING; property tag 0x0042001E

Comments

When user B sends mail for user A, the PR_SENDER_* properties are set to user B and the PR_SENT_REPRESENTING_* properties are set to user A.

This property represents one of several maintained for the messaging user that delegates the send operation to another user. Opening the entry id to find additional information is a relatively expensive operation, so properties representing the commonly-used display name, search key, address type, and email address are defined along with the entryid.

When this property is not set, PR_SENDER_NAME is used.

To generate a reply message, a client application should copy PR_RCVD_REPRESENTING_NAME from the received message into the PR_SENT_REPRESENTING_NAME property of the reply

See Also

PR_RCVD_REPRESENTING_NAME property, PR_SENDER_NAME property, PR_SENT_REPRESENTING_ENTRYID property

PR_SENT_REPRESENTING_SEARCH_KEY

[New - Windows 95]

Contains the search key for the messaging user represented by the sender.

Details

Identifier 0x003B, property type PT_BINARY; property tag 0x003B0102

Comments

When user B sends mail for user A, the PR_SENDER_* properties are set to user B and the PR_SENT_REPRESENTING_* properties are set to user A.

This property represents one of several maintained for the messaging user that delegates the send operation to another user. Opening the entry id to find additional information is a relatively expensive operation, so properties representing the commonly-used display name, search key, address type, and email address are defined along with the entryid.

When this property is not set, the value of PR_SENDER_SEARCH_KEY is used.

See Also

PR_SENDER_SEARCH_KEY property, PR_SENT_REPRESENTING_ENTRYID property, PR_SENT_REPRESENTING_NAME property

PR_SENTMAIL_ENTRYID

[New - Windows 95]

Contains the entry identifier of the sent-mail folder where the message should be moved after submission.

Details

Identifier 0x0E0A, property type PT_BINARY; property tag 0x0E0A0102

Comments

Entryid values are also maintained for other important folders, including the wastebasket and out basket.

See Also

PR_IPM_SENTMAIL_ENTRYID property, PR_IPM_WASTEBASKET_ENTRYID property

PR_SERVICE_DELETE_FILES

[New - Windows 95]

Contains a list of names of files that are to be deleted when the messaging service is uninstalled.

Details

Identifier 0x3D10, property type PT_MV_TSTRING; property tag 0x3D10101E

Comments

Note that these files in the list are deleted from the workstation when using the control panel to uninstall the service. Take care not to include in the list any DLL that supports multiple services, or you may also remove other services.

See Also

PR_SERVICE_SUPPORT_FILES property

PR_SERVICE_DLL_NAME

[New - Windows 95]

Contains the filename of the DLL containing the messaging service provider entry point function to call for configuration.

Details

Identifier 0x3D0A, property type PT_TSTRING; property tag 0x3D0A001E

Comments

When the entry point function name appears in the property PR_SERVICE_ENTRY_NAME, it indicates that the entry point exists.

MAPI uses a DLL file naming convention, where the base file name 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.

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 property

PR_SERVICE_ENTRY_NAME

[New - Windows 95]

Contains the name of the entry point function for configuration of a messaging service.

Details

Identifier 0x3D0B, property type PT_STRING8; property tag 0x3D0B001E

Comments

Service implementors are strongly recommended to provide a service entry point, but the entry point is not required. You should only supply the entry point 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 property PR_SERVICE_DLL_NAME.

See Also

PR_SERVICE_DLL_NAME property

PR_SERVICE_EXTRA_UIDS

[New - Windows 95]

Contains a list of MAPI unique identifiers (UIDs) that identify additional profile sections for the messaging service.

Details

Identifier 0x3D0D, property type PT_BINARY; property tag 0x3D0D0102

Comments

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 provider that uses additional profile sections can store the UIDs of those profile sections in PR_SERVICE_EXTRA_UIDS, allowing MAPI to copy the additional service information.

See Also

PR_SERVICE_NAME property

PR_SERVICE_NAME

[New - Windows 95]

Contains the name of a messaging service as set by the user in the MAPISVC.INF file.

Details

Identifier 0x3D09, property type PT_TSTRING; property tag 0x3D09001E

Comments

The name is specific to the messaging service. PR_SERVICE_NAME appears as a column in the service table and can be used to filter services. Because it is used for identify and filter services, the value should not be localized.

See Also

PR_SERVICE_SUPPORT_FILES property

PR_SERVICE_SUPPORT_FILES

[New - Windows 95]

Contains a list of the files that belong to the messaging service.

Details

Identifier 0x3D0F, property type PT_MV_TSTRING; property tag 0x3D0F101E

Comments

Using a dialog box in the control panel applet, the user can obtain the list of files that belong to the messaging 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 this property to create a support file list in a dialog box for messaging user selection.

See Also

PR_SERVICE_UID property

PR_SERVICE_UID

[New - Windows 95]

Contains the MAPI unique identifier (UID) for the messaging service.

Details

Identifier 0x3D0C, property type PT_BINARY; property tag 0x3D0C0102

Comments

PR_SERVICE_UID appears as a column in the provider table, where it is commonly used to group all the providers that belong to the same service. It is supplied as a parameter to most of the **IMsgServiceAdmin** methods.

See Also

IMsgServiceAdmin::IUnknown method

PR_SERVICES

[New - Windows 95]

Reserved for use by MAPI.

Details

Identifier 0x3D0E, property type PT_BINARY; property tag 0x3D0E0102

Comments

This property should not be used. Reserved for use by MAPI. See Also

MAPIUID structure

PR_SPOOLER_STATUS

[New - Windows 95]

Contains the status of the message based on information available to the spooler.

Details

Identifier 0x0E10, property type PT_LONG; property tag 0x0E100003

Comments

The 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 hook potentially deleted the message while rerouting it.

You can call IMAPIProp::GetProps on PR_SPOOLER_STATUS to determine the message status. The value S_OK indicates that the message was successfully delivered to the 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 this property on messages, recipient tables, and the outqoing queue table. You should be able to set columns on the outgoing queue table and restrict based on this property.

See Also

PR_TRANSPORT_STATUS property

PR_START_DATE

[New - Windows 95]

Contains the starting date and time of the appointment.

Details

Identifier 0x0060, property type PT_SYSTIME; property tag 0x00600040

Comments

PR_START_DATE and PR_END_DATE are usually used by scheduling applications.

See Also

PR_END_DATE property

PR_STATE_OR_PROVINCE

[New - Windows 95]

Contains the messaging user's state or province.

Details

Identifier 0x3A28, property type PT_TSTRING; property tag 0x3A28001E

Comments

Several properties supply information about the messaging user's telephone numbers and fields of the address.

See Also

PR_COUNTRY property

PR_STATUS

[New - Windows 95]

Contains a 32-bit bitmask of flags defining folder status.

Details

Identifier 0x360b, property type PT_LONG; property tag 0x360B0003

Comments

PR_STATUS provides status about the folder, just as PR_MSG_STATUS provides status information about a message. These flags are provided for the client's information only and do not affect the store. Other clients can choose to use or ignore these settings. The client can also choose to define its own values for the client-definable bits of the PR_STATUS property. PR_STATUS appears in both the folder and the hierarchy table row.

See Also

PR_MSG_STATUS property

PR_STATUS_CODE

[New - Windows 95]

Contains a bitmask of flags indicating the current status of a service provider.

Details

Identifier 0x3E04, property type PT_LONG; property tag 0x3E040003

Comments

The status code is a required property that must appear in the status table, the profile, and in the MAPISVC.INF file for all providers.

The following enumerated settings are defined for the PR_STATUS_CODE bitmask:

Value	Description
STATUS_AVAILABLE	Indicates the message service is available for use by MAPI and client applications.
STATUS_FAILURE	Indicates the service provider is experiencing severe, unexpected problems and that the provider session may soon end if the problems do not clear up.
STATUS_INBOUND_ACTIVE	Indicates the transport provider is receiving an inbound message.
STATUS_INBOUND_ENABLED	Indicates the transport provider is enabled to receive inbound messages.
STATUS_INBOUND_FLUSH	Indicates the transport provider is flushing its inbound message queue.
STATUS_OFFLINE	Indicates the services available from the service provider are limited to those that use locally available data.
STATUS_OUTBOUND_ACTIVE	Indicates the transport provider is receiving an outbound message.
STATUS_OUTBOUND_ENABLED	Indicates the transport provider is enabled to handle outbound messages.
STATUS_OUTBOUND_FLUSH	Indicates the transport provider is flushing its outbound message queue.
STATUS_REMOTE_ACCESS	Indicates whether the transport provider supports remotely access.

See Also

PR_STATUS_STRING property

PR_STATUS_STRING

[New - Windows 95]

Contains an ASCII message indicating the current status of a service provider, for example, whether or not it is processing a message.

Details

Identifier 0x3E08, property type PT_TSTRING; property tag 0x3E08001E

Comments

This property gives providers the opportunity to supply specific information about the provider status. PR_STATUS_STRING is optional, based on the value of PR_STATUS_CODE. When the transport 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 spooler; it travels through shared memory rather than being marshalled across a process boundary.

See Also

PR_STATUS_CODE property

PR_STORE_ENTRYID

[New - Windows 95]

Contains the unique entry identifier of the message store in which an object resides.

Details

Identifier 0x0FFB, property type PT_BINARY; property tag 0x0FFB0102

Comments

PR_STORE_ENTRYID is used to open a message store with OpenMessageStore. It is also used to open any object owned by the store.

Message store, folder, and message objects must furnish this property.

For a message store, this property is identical to the store's own PR_ENTRYID property. Your application can compare the two properties using IMAPISession::CompareEntryIDs.

See Also

PR_ENTRYID property

PR_STORE_PROVIDERS

[New - Windows 95]

Reserved for use by MAPI.

Details

Identifier 0x3D00, property type PT_BINARY; property tag 0x3D000102

Comments

This property should not be used.See Also

MAPIUID structure

PR_STORE_RECORD_KEY

[New - Windows 95]

Contains the unique binary-comparable identifier (record key) of the message store in which an object resides.

Details

Identifier 0x0FFA, property type PT_BINARY; property tag 0x0FFA0102

Comments

Message store, folder, and message objects must furnish this property.

For a message store, this 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.

See Also

PR_RECORD_KEY property, PR_STORE_ENTRYID property

PR_STORE_STATE

[New - Windows 95]

Contains a flag that describes the state of the store.

Details

Identifier 0x340E, property type PT_LONG; property tag 0x340E0003

Comments

Note that PR_STORE_STATE is dynamic and can change based on user actions, unlike PR_STORE_SUPPORT_MASK.

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 property

PR_STORE_SUPPORT_MASK

[New - Windows 95]

Contains a bitmask of flags that client applications should query to determine the characteristics of a message store.

Details

Identifier 0x3A0D, property type PT_LONG, Property tag:0x3A0D0003

Comments

The following flags can be set for the PR_STORE_SUPPORT_MASK bitmask:

Value	Description
STORE_ATTACH_OK	Indicates the message store supports attachments (OLE or non-OLE) to messages.
STORE_CATEGORIZE_OK	Indicates the message store supports categorized views of tables.
STORE_CREATE_OK	Indicates the message store supports creation of new messages.
STORE_ENTRYID_UNIQUE	Indicates entry identifiers for the objects in the message store are unique (that is, never reused during the life of the store).
STORE_MODIFY_OK	Indicates the message store supports modification of its existing messages.
STORE_MV_PROPS_OK	Indicates 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	Indicates the message store supports notifications.
STORE_OLE_OK	Indicates the message store supports OLE attachments. The OLE data is accessible through an IStorage interface, such as that available via the PR_ATTACH_DATA_OBJ property.
STORE_READONLY	Indicates all interfaces for the message store have a read-only access level.
STORE_RESTRICTION_OK	Indicates the message store supports restrictions.
STORE_RTF_OK	Indicates the message store supports rich text format (RTF) messages.
STORE_SEARCH_OK	Indicates the message store supports search-results folders, and FALSE otherwise.

STORE_SORT_OK	Indicates the message store supports sorting views of tables.
STORE_SUBMIT_OK	Indicates the message store supports marking a message for submission.

See Also

PR_ATTACH_DATA_OBJ property, PR_RTF_COMPRESSED property

PR_STREET_ADDRESS

[New - Windows 95]

Contains the street address for the messaging user.

Details

Identifier 0x3A29, property type PT_TSTRING; property tag 0x3A29001E

Comments

Several properties are supplied for the messaging user, including several telephone numbers and fields of the messaging user's home and business mailing addresses.

See Also

PR_LOCALITY property

PR_SUBFOLDERS

[New - Windows 95]

Contains TRUE if a folder contains subfolders, and FALSE otherwise.

Details

Identifier 0x360a, property type PT_BOOLEAN; property tag 0x360A000B

Comments

Message stores compute this property for all folders.

See Also

PR_FOLDER_TYPE property

PR_SUBJECT

[New - Windows 95]

Contains the subject of a message.

Details

Identifier 0x0037, property type PT_TSTRING; property tag 0x0037001E

Comments

For a report, this property contains the original message's subject preceded by a string indicating what has happened to the message.

A message store uses this property to compute the PR_NORMALIZED_SUBJECT property.

For X.400 environments, PR_SUBJECT corresponds to the IM_SUBJECT property as defined by the X.400 message-handling standard.

See Also

PR_NORMALIZED_SUBJECT property

PR_SUBJECT_IPM

[New - Windows 95]

Contains message transport system (MTS) identifiers for messages with which a delivery report or nondelivery report is concerned.

Details

Identifier 0x0038, property type PT_BINARY; property tag 0x00380102

Comments

This MAPI property represents the interpersonal notification IM_SUBJECT_IPM attribute as defined by the X.400 message-handling standard.

See Also

PR_SUBJECT property

PR_SUBJECT_PREFIX

[New - Windows 95]

Contains a subject prefix that indicates a reply or forwarded message, such as "RE:" or "FW:".

Details

Identifier 0x003d, property type PT_TSTRING; property tag 0x003D001E

Comments

The subject prefix consists of one, two, or three alphanumeric characters, and the colon character ':'. The property should not contain leading spaces. Client applications prepend the subject prefix to the subject.

See Also

PR_SUBJECT property

PR_SUBMIT_FLAGS

[New - Windows 95]

Contains a bitmask of flags indicating details about a message submission.

Details

Identifier 0x0E14, property type PT_LONG; property tag 0x0E140003

Comments

This property is part of the outgoing message queue table.

The following flags can be set for the PR_SUBMIT_FLAGS bitmask:

Value	Description
SUBMITFLAG_LOCKED	Indicates the spooler currently has the message locked.
SUBMITFLAG_PREPROCESS	Indicates the message needs preprocessing. When the 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

IMessage::SubmitMessage method, **IMsgStore::SetLockState** method

PR_SUPPLEMENTARY_INFO

[New - Windows 95]

Contains arbitrary additional information for use in a delivery report.

Details

Identifier 0x0C1B, property type PT_TSTRING; property tag 0x0C1B001E

Comments

For X.400 environments, this property corresponds to the IM_SUPPLEMENTARY_RECEIPT_INFO attribute.

See Also

PR_REPORT_NAME property

PR_SURNAME

[New - Windows 95]

Contains the surname of a messaging user.

Details

Identifier 0x3A11, property type PT_TSTRING; property tag 0x3A11001E

Comments

Several properties are supplied for the messaging user's name, phone numbers, and addresses.

See Also

PR_GIVEN_NAME property

PR_TELEX_NUMBER

[New - Windows 95]

Contains the messaging user's telex number.

Details

Identifier 0x3A2C, property type PT_TSTRING; property tag 0x3A2C001E

Comments

Several properties are supplied for the messaging user's name, phone numbers, and addresses.

See Also

PR_OFFICE_TELEPHONE_NUMBER property

PR_TEMPLATEID

[New - Windows 95]

Contains an entry identifier that can find the code associated with the provider.

Details

Identifier 0x3902, property type PT_BINARY; property tag 0x39020102

Comments

The entryid can bind the code to the data.

See Also

IABLogon::OpenTemplateID method, **IMAPISupport::OpenTemplateID** method, PR_PRIMARY_CAPABILITY property

PR_TITLE

[New - Windows 95]

Contains a messaging user's formal title, such as M.D. or Ph.D.

Details

Identifier 0x3A17, property type PT_TSTRING; property tag 0x3A17001E

Comments

Several properties are supplied for the messaging user's name, phone numbers, and addresses.

See Also

PR_SURNAME property

PR_TRANSMITTABLE_DISPLAY_NAME

[New - Windows 95]

Contains the messaging user's display name in a secure form that can not be changed.

Details

Identifier 0x3a20, property type PT_TSTRING; property tag 0x3A20001E

Comments

PR_TRANSMITTABLE_DISPLAY_NAME should be provided by all address book providers. The property contains the version of the recipient's display name that is transmitted with the message. This property is used to prevent spoofing. For most providers this property has the same value as PR_DISPLAY_NAME. Providers that do not have a secure display name return PT_ERROR and MAPI changes the display name by adding quote marks around the name.

In previous releases of the MAPI 1.0 SDK, this property was defined using the spelling, "PR_TRANSMITABLE_DISPLAY_NAME." MAPI 1.0 supports both spellings of this property. Both versions use the same property identifier values and property types.

See Also

PR_DISPLAY_NAME property

PR_TRANSPORT_KEY

[New - Windows 95]

Reserved for MAPI.

Details

Identifier 0x0E16, property type PT_LONG; property tag 0x0E160003

Comments

This property is reserved for MAPI.   Do not use.

See Also

PR_TRANSPORT_PROVIDERS property

PR_TRANSPORT_MESSAGE_HEADERS

[New - Windows 95]

Contains transport-specific message envelope information.

Details

Identifier 0x007D, property type PT_TSTRING; property tag 0x007D001E

Comments

The transport provider can generate the message header information for inbound messages.

This property offers an alternative to either discarding the transport message header information, or prepending the transport message header information to the message body text. The client can choose whether or not to display the information.

See Also

PR_BODY property

PR_TRANSPORT_PROVIDERS

[New - Windows 95]

Reserved for use by MAPI.

Details

Identifier 0x3D02, property type PT_BINARY; property tag 0x3D020102

Comments

This property should not be used. It contains a list of MAPI unique identifiers (UIDs), each identifying a transport provider.

See Also

MAPIUID structure

PR_TRANSPORT_STATUS

[New - Windows 95]

Reserved for MAPI. Do not use.

Details

Identifier 0x0E11, property type PT_LONG; property tag 0x0E110003

Comments

Do not use this property. It is reserved for use by MAPI.

See Also

PR_SPOOLER_STATUS property

PR_TYPE_OF_MTS_USER

[New - Windows 95]

Contains the type of a message recipient, for use in a report.

Details

Identifier 0x0C1C, property type PT_LONG; property tag 0x0C1C0003

Comments

This MAPI property corresponds to the report per-recipient MH_T_DELIVERY_POINT attribute as defined by the X.400 message-handling standard.

See Also

PR_CORRELATE_MTSID property

PR_UPDATE_PAB

[New - Windows 95]

This property is not supported in MAPI version 1.0.

Comments

The PR_UPDATE_PAB property, which was present in pre-release versions, is not supported in the MAPI 1.0 SDK.

See Also

PR_AB_DEFAULT_PAB property

PR_USER_CERTIFICATE

[New - Windows 95]

Contains an authentication certificate for the user. An authentication certificate is similar to a digital signature.

Details

Identifier 0x3A22, property type PT_BINARY; property tag 0x3A220102

Comments

Several MAPI properties supply ASN.1 certificates.

See Also

PR_ORIGINATOR_CERTIFICATE property, PR_RECIPIENT_CERTIFICATE property

PR_VALID_FOLDER_MASK

[New - Windows 95]

Contains a bitmask of flags set by a message store to indicate the validity of the entry identifiers of the folders in the store.

Details

Identifier 0x35DF, property type PT_LONG; property tag 0x35DF0003

Comments

The following flags can be set for the PR_VALID_FOLDER_MASK bitmask:

Value	Description
FOLDER_COMMON_VIEWS_VALID	Indicates the common views folder has a valid entry identifier (see PR_COMMON_VIEWS_ENTRYID).
FOLDER_FINDER_VALID	Indicates the finder folder has a valid entry identifier (see PR_FINDER_ENTRYID).
FOLDER_INBOX_VALID	Indicates the IPM receive folder has a valid entry identifier. (IPM_INBOX_VALID)
FOLDER_IPM_OUTBOX_VALID	Indicates the IPM outbox folder has a valid entry identifier (see PR_IPM_OUTBOX_ENTRYID).
FOLDER_IPM_SENTMAIL_VALID	Indicates the IPM sent-mail folder has a valid entry identifier (see PR_IPM_SENTMAIL_ENTRYID).
FOLDER_IPM_SUBTREE_VALID	Indicates the interpersonal messaging (IPM) folder subtree has a valid entry identifier (see PR_IPM_SUBTREE_ENTRYID).
FOLDER_IPM_WASTEBASKET_VALID	Indicates the IPM wastebasket folder has a valid entry identifier (see PR_IPM_WASTEBASKET_ENTRYID).
FOLDER_VIEWS_ENTRYID	Indicates the views folder has a valid entry identifier.

See Also

PR_FOLDER_TYPE property

PR_VIEWS_ENTRYID

[New - Windows 95]

Contains the entry identifier of the user-defined views folder.

Details

Identifier 0x35E5, property type PT_BINARY; property tag 0x35E50102

Comments

The Common Views folder contains a predefined set of views, while the Views folder is defined by the messaging user.

See Also

PR_COMMON_VIEWS_ENTRYID property

PR_X400_CONTENT_TYPE

[New - Windows 95]

Contains the content type for a submitted message.

Details

Identifier 0x003C, property type PT_BINARY; property tag 0x003C0102

Comments

For X.400 environments, this MAPI property corresponds to the MH_T_CONTENT_TYPE attribute.

See Also

PR_X400_DEFERRED_DELIVERY_CANCEL property

PR_X400_DEFERRED_DELIVERY_CANCEL

[New - Windows 95]

Contains TRUE if the message transport system (MTS) allows X.400 deferred delivery cancellation, and FALSE otherwise.

Details

Identifier 0x3E09, property type PT_BOOLEAN; property tag 0x3E09000B

Comments

A comprehensive list of MAPI properties and their mappings to X.400 appears in the *MAPI Programmer's Guide*.

See Also

PR_X400_CONTENT_TYPE property

PR_XPOS

[New - Windows 95]

Contains the X coordinate of the starting position (the upper left corner) of the control, in standard Windows dialog units.

Details

Identifier 0x3F05, property type PT_LONG; property tag 0x3F050003

Comments

The PR_XPOS, PR_YPOS, PR_DELTAX, and PR_DELTAY properties position and size the control.

See Also

PR_DELTAX property, PR_DELTAY property, PR_YPOS property

PR_YPOS

[New - Windows 95]

Contains the location of the Y coordinate of the starting position (the upper left corner) of a control, in standard Windows dialog units.

Details

Identifier 0x3F06, property type PT_LONG; property tag 0x3F060003

Comments

The PR_XPOS, PR_YPOS, PR_DELTAX, and PR_DELTAY properties position and size the control.

See Also

PR_DELTAX property, PR_DELTAY property, PR_XPOS property

Extended MAPI Structures and Simple Data Types

This chapter contains a reference entry for each structure and data type.

Structures

This section contains a reference entry for each structure.

ADRENTRY

An **ADRENTRY** structure holds all properties related to one recipient of a message,   for example, the recipient's display name.

Syntax

```
typedef struct _ADRENTRY
{
     ULONG ulReserved1;
     ULONG cValues;
     LPSPropValue rgPropVals;
} ADRENTRY, FAR *LPADRENTRY;
```

Members

ulReserved1
Reserved; must be zero.

cValues
Number of elements in the array pointed to by the **rgPropVals** member.

rgPropVals
Pointer to an array of **SPropValue** structures containing values for the properties of the message recipient.

Comments

Each message recipient has a collection of properties contained in an **ADRENTRY** structure. Among other properties, this structure contains:

- The display name
- The messaging system type
- The messaging system address
- The entry identifier

ADRENTRY structures typically exist as components of **ADRLIST** structures. These structures are used to represent message recipients and recipient lists, respectively.

Each instance of an **ADRENTRY rgPropVals** member in an **ADRLIST** structure must be allocated separately, using **MAPIAllocateBuffer**, 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** members must be deallocated prior to deallocation of the containing **ADRLIST** structure so that pointers to allocated **SPropValue** structures are not lost.

The structure member types and allocation rules for the **ADRENTRY** structures and **SRow** structures are identical.

For more information on **ADRENTRY** allocation and deallocation issues, see the entries for **IMessage::ModifyRecipients** and **IAddrBook::Address.**

See Also

IAddrBook::Address method, **IMessage::ModifyRecipients** method, **MAPIAllocateBuffer** function, **SRow** structure

ADRLIST

An **ADRLIST** structure holds one **ADRENTRY** structure for each recipient of a particular message. Therefore, an **ADRLIST** structure contains all the property information for an address list.

Syntax

```
typedef struct _ADRLIST
{
     ULONG cEntries;
     ADRENTRY aEntries[MAPI_DIM];
} ADRLIST, FAR *LPADRLIST;
```

Members

cEntries
 Number of entries in the array specified by the **aEntries** member.

aEntries
 Array of **ADRENTRY** structures, each typically describing a message recipient.

Comments

An **ADRLIST** structure has the same form as an **SRowSet** structure. It can contain both unresolved and resolved entries; an unresolved entry is one that consists only of a message recipient name, whereas a resolved entry contains additional information about the recipient, including an e-mail address type and an entry identifier. Commonly, client applications that enable users to type recipient names create unresolved entries.

If your application deals with a message recipient list that is too large to fit in memory, you can use an **ADRLIST** structure along with the **IMessage::ModifyRecipients** method to work with a subset of the list. Your application should not use the address-book common dialog boxes for the user interface in such a situation.

For information about allocation rules for **ADRLIST** structures, see the entry for **ADRENTRY** earlier in this chapter.

Use the **CbADRLIST** macro to return 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 parameter *_centries* specifies the number of recipients. This macro returns the number of bytes of memory that an **ADRLIST** structure with *_centries* recipients 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 parameter *_centries* specifies the number of **ADRENTRY** structures. The structure type is defined with the tag _ADRLIST_ *_name* and type name *_name*.

SizedADRLIST provides a means of defining 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

An **ADRPARM** structure holds data used for controlling the display and behavior of an address-book common dialog box. This structure is used with the **IAddrBook::Address** and **IMAPISupport::Address** methods.

Syntax

```
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
The list of entries that can be created into the recipient wells.

lpABContEntryID
Pointer to the ENTRYID of a container that will supply the list of one-offs that can be added to the recipient wells of this dialog box.

ulFlags
Bitmask of flags associated with various address-book dialog box options.The following flags can be set:

AB_RESOLVE
Causes all names to be resolved after the address book dialog box is closed. The Resolve Name dialog box will be displayed if there are ambiguous entries in the recipient list.

AB_SELECTONLY
Forces the user to select recipients from the supplied list of resolved names. This flag disables the creation of custom recipient 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 of the dialog box is allowed to pick exactly one message recipient, instead of a number of recipients from a recipient list. This flag is valid only when *cDestFields* is 0. This flag is used only if the dialog box is modal.

DIALOG_MODAL
Causes a modal dialog box to be displayed. Your application 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 allowed 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. Your application must set either this flag or DIALOG_MODAL, but not both.

lpReserved

Reserved, must be zero.

ulHelpContext

Input parameter specifying the context within the Help that will first be shown when the Help button is pressed in the address dialog box.

lpszHelpFileName

Input parameter pointing to the name of a Help file that will be associated with the address dialog box that is a result of this call. Used in conjunction with *ulHelpContext* to call the Windows function **WinHelp**.

lpfnABSDI

Output parameter that points to a MAPI function based on the **ACCELERATEABSDI** prototype that the client application must call in its Windows message loop for a modeless dialog box. This allows for the dialog box to have custom accelerators.

lpfnDismiss

Pointer to a client application dismiss function, a function based on the **DISMISSMODELESS** prototype. MAPI calls a client application dismiss function when it has dismissed a modeless address-book dialog box. 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 constant is set in **ulFlags**.

lpvDismissContext

Pointer to context information passed in with the **IAddrBook::Address** call and passed back unchanged. This member contains values that are necessary for the operation of the client application dismiss function. The **lpvDismissContext** member is used only if the DIALOG_SDI flag is set in **ulFlags**.

lpszCaption

Pointer to text to be used as a caption for the address-book dialog box.

lpszNewEntryTitle

Pointer to text to be used as a new-entry prompt for an edit box in an address book dialog box.

lpszDestWellsTitle

Pointer to text to be used as a title for for the set of recipient-name edit boxes that appear in the dialog box. This member is used only if the address-book dialog box is modal.

cDestFields

Number of recipient-name edit boxes (that is, destination fields) in the address-book dialog box. A number from 0 through 3 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.

nDestFieldFocus

Field in the address-book 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

Pointer to an array of text titles to be displayed in the recipient-name edit boxes of the address-book dialog box. The size of the array is the value of **cDestFields**. If the **lppszDestTitles** member is NULL, **IAddrBook::Address** chooses default titles.

lpulDestComps

Pointer to an array of recipient types, such as MAPI_TO, MAPI_CC, MAPI_BCC associated with each recipient-name edit box. The size of the array is the value of **cDestFields**. If the **lpulDestComps** member is NULL, the **IAddrBook::Address** interface chooses default recipient types.

lpContRestriction

Pointer to an **SRestriction** structure of restrictions that the client application sets on any address book container to be viewed. the **IAddrBook::Address** interface combines using the logical AND operator with any restrictions the user specifies while using the address-book dialog box.

lpHierRestriction

Pointer to an **SRestriction** structure of restrictions on the hierarchy table used in the address-book dialog box.

Comments

Passing in 0 for the **cbABContEntryID** member uses the default list of one-off templates of Recipients that can be created for a message. When we create ENTRYIDs into the recipient well MAPI calls **CreateEntry**.

If the address book is open for browsing only, **IAddrBook::Address** ignores its *lppAdrList* parameter. If **cDestFields** has the value 0xFFFFFFFF, the **Address** method displays the address-book dialog box in its default configuration, ignoring the **lppszDestTitles** and **lpulDestComps** members of **ADRPARM**.

The **lpfnABSDI** member must be NULL in order to display the modeless dialog box in **IAddrBook::Address** and **IMAPISupport::Address** calls. Also, the **lpfnABSDI** member is used only if the DIALOG_SDI flag is set in the **ulFlags** member. **IAddrBook::Address** fills in the **lpfnABSDI** member before returning.

Which address book container **lpABContEntryID** points to determines what is listed in the edit box within the dialog box that holds possible recipient names. Usually, **lpABContEntryID** is NULL, indicating the use of a custom recipient provider.

The **ADRPARM** structure is defined in MAPIDEFS.H.

See Also

ACCELERATEABSDI function prototype, **DISMISSMODELESS** function prototype, **ENTRYID** structure, **IAddrBook::Address** method, **SRestriction** structure

ADRPARM, ACCELERATEABSDI

The **ACCELERATEABSDI** prototype function defines a client application callback function that MAPI must call in its Windows message loop during execution of a modeless address book dialog box for **IAddrBook::Address**. A function with this prototype accelerates the addition of names to the address book.

Syntax

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 or zero. In Microsoft Windows applications, *ulUIParam* is the parent window handle for a dialog and is of type HWND, cast to a ULONG. A value of zero indicates there is no parent window.

lpvmsg
: Input parameter specifying a pointer to a Windows message.

Return Values

A client application function with the **ACCELERATEABSDI** prototype returns TRUE if it handles the message.

Comments

A client application's **ACCELERATEABSDI** prototype-based function is used only if the application has set the DIALOG_SDI flag in the **ulFlags** member of the **ADRPARM** structure.

The **ACCELERATEABSDI** prototype is defined in MAPIDEFS.H.

See Also

ADRPARM structure, **IAddrBook::Address** method

ADRPARM, DISMISSMODELESS

The **DISMISSMODELESS** prototype function represents a client application callback function that MAPI calls when it has dismissed a modeless address book dialog box. This call is necessary so that MAPI will not keep calling the application's accelerator function, **ACCELERATEABSDI** prototype, when the modeless dialog box is not active.

Syntax

void (STDMETHODCALLTYPE DISMISSMODELESS)(ULONG *ulUIParam*, **LPVOID** *lpvContext***)**

Parameters

ulUIParam
: Input parameter specifying an implementation-specific 32-bit value normally 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 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. Typically, for C++ code, *lpvContext* represents a pointer to the address of a C++ object.

Comments

The **DISMISSMODELESS** prototype is defined in MAPIDEFS.H.

See Also

ACCELERATEABSDI function prototype, **ADRPARM** structure

DTBLBUTTON

A **DTBLBUTTON** structure holds information about a button control for a display table dialog box.

Syntax

```
typedef struct _DTBLBUTTON
{
     ULONG ulbLpszLabel;
     ULONG ulFlags;
     ULONG ulPRControl;
} DTBLBUTTON, FAR *LPDTBLBUTTON;
```

Members

ulbLpszLabel
Offset from the beginning of the structure, in bytes, to text to be displayed on the button.

ulFlags
Bitmask of flags designating 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 8-bit format.

ulPRControl
A property of type PT_OBJECT on which you can open an **IMAPIControl** interface using an **OpenProperty** call.

Comments

A **DTBLBUTTON** 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 **SizedDtblButton** macro to create a structure similar to **DTBLBUTTON** but contains a label of specified character length. The syntax is:

SizedDtblButton (**int** *n*, *u*)

The parameter *n* specifies the character length of the button's label. The parameter *u* 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 of it.

The **DTBLBUTTON** structure is defined in MAPIDEFS.H.

See Also

DTCTL structure, PR_CONTROL_TYPE property

DTBLCHECKBOX

A **DTBLCHECKBOX** structure holds information about a check box to be used in a display table dialog box.

Syntax

```
typedef struct _DTBLCHECKBOX
{
     ULONG ulbLpszLabel;
     ULONG ulFlags;
     ULONG ulPRPropertyName;
} DTBLCHECKBOX, FAR *LPDTBLCHECKBOX;
```

Members

ulbLpszLabel
Offset from the beginning of the structure, in bytes, to be the label of the check box.

ulFlags
Bitmask of flags designating 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 8-bit format.

ulPRPropertyName
A property of type PT_BOOLEAN whose value is maipulated by the user via a check box. **GetProps** is used to initalize the box. **SetProps** is used to manipulate this properties value as the user makes changes to the check box.

Comments

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 contains 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 means of defining 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 of 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.

See Also

DTCTL structure, PR_CONTROL_TYPE property

DTBLCOMBOBOX

A **DTBLCOMBOBOX** structure holds information about a combo box that is to be part of a dialog box.

Syntax

```
typedef struct _DTBLCOMBOBOX
{
     ULONG ulbLpszCharsAllowed;
     ULONG ulFlags;
     ULONG ulNumCharsAllowed;
     ULONG ulPRPropertyName;
     ULONG ulPRTableName;
} DTBLCOMBOBOX, FAR *LPDTBLCOMBOBOX;
```

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 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 designating 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 8-bit format.

ulNumCharsAllowed
Maximum number of characters that the user can type into the combo box's edit box.

ulPRTableName
A property 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.

ulPRPropertyName
A property name 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. Initially, the **ulPRPropertyName** member holds the value of a default selection. If the property name is PR_NULL, the list box is not a single selection type.

Comments

A **DTBLCOMBOBOX** structure does not specify the character length of the allowed character string, it only contains an offset to the beginning of the it in memory allocated for the structure.

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 means of defining a combo box when the length of the allowed character string is known. The syntax is:

SizedDtblComboBox (**int** *n*, *u*)

The parameter *n* specifies the length of the allowed characters string. 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)

u

The structure type is defined with the tag _DTBLCOMBOBOX_ *u* and type name *u*.

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.

See Also

DTCTL structure, PR_CONTROL_TYPE property

DTBLDDLBX

A **DTBLDDLBX** structure holds information about a drop-down list box to be part of a dialog box.

Syntax

```
typedef struct _DTBLDDLBX
{
     ULONG ulFlags;
     ULONG ulPRDisplayProperty;
     ULONG ulPRSetProperty;
     ULONG ulPRTableName;
} DTBLDDLBX, FAR *LPDTBLDDLBX;
```

Members

ulFlags
Reserved; must be zero.

ulPRTableName
A property of type PT_OBJECT on which an **IMAPITable** interface can be opened using an **OpenProperty** call. The rows of the table are to correspond to items in the list box.

ulPRDisplayProperty
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
A property name whose values uniquely identify rows in the table (e.g. PR_ENRTYID). The value of this property identifies the row selected via a choice in the list box. Initially, the **ulPRPropertyName** member holds the value of a default selection. If the property name is PR_NULL, the list box is not a single selection type.

Comments

The DTBLDDLBX structure is defined in MAPIDEFS.H.

See Also

DTCTL structure

DTBLEDIT

A **DTBLEDIT** structure holds information about a edit box to be part of a dialog box.

Syntax

```
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 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 designating 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 8-bit format.

ulNumCharsAllowed
Maximum number of characters that the user can type into the edit box.

ulPropTag
A property name of type PT_TSTRING. The text in this property is displayed and edited in the edit box.

Comments

A **DTBLEDIT** structure does not specify the character length of the allowed characters 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 parameter *n* specifies the length of the allowed characters string. The allowed characters 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)

The structure type is defined with the tag _DTBLEDIT_ *u* and type name *u*.

SizedDtblEdit provides a means of defining 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 of the string.

To use a sized display table edit pointer *lpSizedDtblEdit* 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.

See Also

DTCTL structure, **IMAPIProp::GetProps** method, PR_CONTROL_TYPE property,

DTBLGROUPBOX

A **DTBLGROUPBOX** structure holds information about a group box to be part of a dialog box.

Syntax

```
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 designating 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 8-bit format.

Comments

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** except that it contains a label of specified character length. The syntax is:

SizedDtblGroupBox (**int** *n*, *u*)

The parameter *n* 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 means of defining 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 of 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.
```

The **DTBLGROUPBOX** structure is defined in MAPIDEFS.H.

See Also

DTCTL structure

DTBLINKEDIT

A **DTBLINKEDIT** structure holds information about an ink-aware edit box to be part of a dialog box used in pen computing.

Syntax

```
typedef struct _DTBLINKEDIT
{
     ULONG ulbLpszCharsAllowed;
     ULONG ulFlags;
     ULONG ulNumCharsAllowed;
     ULONG ulTextPropTag;
     ULONG ulInkDataPropTag;
} DTBLINKEDIT, FAR *LPDTBLINKEDIT;
```

Members

ulbLpszCharsAllowed

Offset from the beginning of the structure, in bytes, to a string that lists the characters allowed in the ink aware edit box.

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 designating 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 8-bit format.

ulNumCharsAllowed

Maximum number of characters that the user can type into the ink-aware edit box.

ulTextPropTag

A property name of type PT_TSTRING. The text in this property is displayed and edited in the edit box.

ulInkDataPropTag

A property name of type PT_BINARY. The property is used to store binary data (that is, graphics input) entered in the ink-aware edit box.

A **DTBLINKEDIT** structure does not specify the character length of the allowed characters string, it only contains an offset to it in memory allocated for the structure.

Use the **SizedDtblInkEdit** macro to define a structure similar to **DTBLINKEDIT** but explicitly specifies the number of character contained in the allowed characters member. The syntax is:

SizedDtblInkEdit (**int** *n*, *u*)

The parameter *n* is the length of the allowed character string. The allowed character string follows the following 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)

The structure type is defined with the tag _DTBLINKEDIT_ *u* and type name *u*.

SizedDtblEdit provides a means of defining a ink aware text edit box when the length of the allowed characters string is known. A **DTBLINKEDIT** structure does not explicitly give the length of the *lpszCharsAllowed* string, it only provides an offset to the first character of the string.

To use a sized ink aware edit box pointer such as *lpSizedDtblInkEdit* in any function call or structure that expects a **LPDTBLINKEDIT** pointer, perform the following cast:

```
lpDtblInkEdit = (LPDTBLINKEDIT) lpSizedDtblInkEdit.
```

The **DTBLLINKEDIT** structure is defined in MAPIDEFS.H.

See Also

DTCTL structure

DTBLLABEL

A **DTBLLABEL** structure holds information about a label to be part of a dialog box.

Syntax

```
typedef struct _DTBLLABEL
{
     ULONG ulbLpszLabelName;
     ULONG ulFlags;
} DTBLLABEL, FAR *LPDTBLLABEL;
```

Members

ulbLpszLabelName
Offset from the beginning of the structure, in bytes, to be the label text string.

ulFlags
Bitmask of flags designating 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 8-bit format.

Comments

Use the **SizedDtblLabel** macro to create a structure definition similar to **DTBLLABEL** but contains a specified character length for the label. The syntax is:

SizedDtblLabel (**int** *n*, *u*)

The parameter *n* specifies the character length of the label. The structure type is defined with the tag _DTBLLABEL_ *u* and type name *u*.

SizedDtblLabel provides a means of defining 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 sized display table label pointer such as *lpSizedDtblLabel* in any function call or structure that expects a **LPDTBLLABEL** pointer, perform the following cast:

```
lpDtblLabel = (LPDtblLabel) lpSizedDtblLabel.
```

The **DTBLLABEL** structure is defined in MAPIDEFS.H.

See Also

DTCTL structure

DTBLLBX

A **DTBLLBX** structure holds information about a list box to be part of a dialog box.

Syntax

```
typedef struct _DTBLLBX
{
     ULONG ulFlags;
     ULONG ulPRSetProperty;
     ULONG ulPRTableName;
} DTBLLBX, FAR *LPDTBLLBX
```

Members

ulFlags
Reserved; must be zero.

ulPRSetProperty
Unique property (such as an entry identifier) that holds a value indicating which entry in the list box was selected by the user. The property name indicates the column in the table whose values fill the list box that is connected with the selected list-box entry. Before the user selects a value in the list box, the **ulPRSetProperty** member holds the default selection.

ulPRTableName
A property of type PT_OBJECT on which an **IMAPITable** interface can be opened using an **OpenProperty** call. The rows of the table are to correspond to items in the list box.

Comments

DTBLLX is defined in MAPIDEFS.H.

See Also

DTCTL structure

DTBLMVDDLBOX

The **DTBLMVDDLBOX** structure specifies a multivalued property to be used in a display table drop down list box.

Syntax

```
typedef struct _DTBLMVDDLBX
{
     ULONG ulFlags;
     ULONG ulMVPropTag;
} DTBLMVDDLBX, FAR * LPDTBLMVDDLBX;
```

Members

ulFlags
Reserved; must be zero.

ulMVPropTag
A property name that is a multivalued type (e.g. PT_MV_TSTRING). The different values of this property are displayed as distinct entries in the drop-down list box.

Comments

The list box defined by this structure is browse-only, i.e. no selection can be made by the user.

DTBLMVDDLBOX is defined in MAPIDEFS.H.

See Also

DTCTL structure

DTBLMVLISTBOX

The **DTBLMVLISTBOX** structure specifies a multivalued property to be displayed in a list box.

Syntax

```
typedef struct _DTBLMVLISTBOX
{
     ULONG ulFlags;
     ULONG ulMVPropTag;
} DTBLMVLISTBOX, FAR * LPDTBLMVLISTBOX;
```

Members

ulFlags
Reserved; must be zero.

ulMVPropTag
A property name that is a multivalued type (e.g. PT_MV_TSTRING). The different values of this property are displayed as distinct entries in the drop-down list box.

Comments

The list box defined by this structure is browse-only, i.e., no selection can be made by the user.

DTBLMVLISTBOX is defined in MAPIDEFS.H.

See Also

DTCTL structure

DTBLPAGE

A **DTBLPAGE** structure holds information about a tabbed page to be part of a dialog box.

Syntax

```
typedef struct _DTBLPAGE
{
     ULONG ulbLpszLabel;
     ULONG ulFlags;
     ULONG ulbLpszComponent;
     ULONG ulContext;
} DTBLPAGE, FAR *LPDTBLPAGE;
```

Members

ulbLpszLabel
Offset from the beginning of the structure, in bytes, to the label text for the page tab.

ulFlags
Bitmask of flags designating 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 8-bit format.

ulbLpszComponent
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
Unique identifier for the tabbed page within the string defined by the **ulbLpszComponent** member. If this identifier is 0 and the component string is NULL, there is no Help associated with the page.

Comments

A **DTBLPAGE** structure does not specify the character length of the label and Help component strings, it only contains offsets to the beginning of these strings in memory allocated for the structure.

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 parameter *n* speciifes the character length of the page's label. The parameter *n1* 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 a sized display table page pointer *lpSizedDtblPage* 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

A **DTBLRADIOBUTTON** structure holds information about one radio button which is to be a part of radio group in a display table dialog.

Syntax

```
typedef struct _DTBLRADIOBUTTON
{
     ULONG ulbLpszLabel;
     ULONG ulFlags;
     ULONG ulcButtons;
     ULONG ulPropTag;
     long lReturnValue;
} DTBLRADIOBUTTON, FAR *LPDTBLRADIOBUTTON;
```

Members

ulbLpszLabel
Offset from the beginning of the structure, in number of bytes, to the label text for the radio button.

ulFlags
Bitmask of flags designating 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 8-bit format.

ulcButtons
Number of buttons in the radio button group. The actual button structures must appear in successive rows of the display table. Each of these rows should specify the same value for the **ulcButtons** member.

ulPropTag
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
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 writes the value of **lReturnValue** to **ulPropTag** when the dialog is closed. If this flag is set, the client application writes **lReturnValue** to **ulPropTag** at the time the radio button is selected.

Comments

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** is defined in MAPIDEFS.H.

See Also

BuildDisplayTable function, **DTCTL** structure,

DTCTL

A **DTCTL** structure holds 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.

Syntax

```
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

Value indicating the control represented by the **DTCTL** structure. The possible values are:

Value	Constant	Meaning
0	DTCT_LABEL	A dialog label.
1	DTCT_EDIT	A dialog edit text box.
2	DTCT_LBX	A dialog list box.
3	DTCT_COMBOBOX	A dialog combo box.
4	DTCT_DDLBX	A dialog drop-down list box.
5	DTCT_CHECKBOX	A dialog check box.
6	DTCT_GROUPBOX	A dialog group box.
7	DTCT_BUTTON	A dialog button control.
8	DTCT_PAGE	A dialog tabbed page.
9	DTCT_RADIOBUTTON	A dialog radio button.
A	DTCT_INKEDIT	A dialog ink-aware edit box.
B	DTCT_MVLISTBOX	A multi-valued list box.
C	DTCT_MVDDLBX	A multi-valued 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 the ability to edit the dialog box control is protected by a password.

DT_REQUIRED
: Indicates the dialog box control is required.

DT_SET_IMMEDIATE
: Enables immediate output of a value upon selection of a radio button.

lpbNotif
Pointer to notification data.

cbNotif
Size, in bytes, of the notification data.

lpszFilter
Pointer to a character filter for an edit box, (standalone or part of a combo box).

ulItemID
Item identifier validating parallel dialog-box template entries.

lpv
Pointer to a value that the client application should initialize to avoid warnings when calling **BuildDisplayTable**.

lplabel
Pointer to a **DTBLLABEL** structure.

lpedit
Pointer to a **DTBLEDIT** structure.

lplbx
Pointer to a **DTBLLBX** structure.

pcombobox
Pointer to a **DTBLCOMBOBOX** structure.

lpddlbx
Pointer to a **DTBLDDLBX** structure.

lpcheckbox
Pointer to a **DTBLCHECKBOX** structure.

lpgroupbox
Pointer to a **DTBLGROUPBOX** structure.

lpbutton
Pointer to a **DTBLBUTTON** structure.

lpradiobutton
Pointer to a **DTBLRADIOBUTTON** structure.

lpmvlbx
Pointer to a **DTBLMVLISTBOX** structure.

lpmvddlbx
Pointer to a **DTBLMVDDLBOX** structure.

lppage
Pointer to a **DTBLPAGE** structure.

Comments

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.

DTCTL 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

A **DTPAGE** structure holds information about a tabbed dialog-box page for inclusion in the associated display table. The **BuildDisplayTable** function uses this structure for building the display table from control resources.

Syntax

```
typedef struct {
     ULONG cctl;
     LPTSTR      lpszResourceName;
     LPDTCTL      lpctl;
} DTPAGE, FAR *LPDTPAGE;
```

Members

cctl
  Contains the count of controls in the **lpctl** array member of the same **DTPAGE** structure.

lpszResourceName
  Pointer to the name of the tabbed-page resource or to an integer identifier for the resource.

lpctl
  Pointer to a **DTCTL** structure defining the page.

Comments

DTPAGE is defined in MAPIUTIL.H.

See Also

BuildDisplayTable function, **DTBLPAGE** structure, **DTCTL** structure

ENTRYID

An **ENTRYID** structure holds an entry identifier for a MAPI object. 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.

Syntax

```
typedef struct {
     BYTE   abFlags[4];
     BYTE     ab[MAPI_DIM];
} ENTRYID, FAR *LPENTRYID;
```

Members

abflags
Array of flags that provide information describing the object. The following flags can be set:

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 be clear for permanent entry identifiers and set for short-term entry identifiers, as described in "Comments."

ab
Array of binary data used by service providers. The client application cannot use this array.

Comments

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.

Directly comparing the binary data in two entry identifiers does not provide much information to your application, as MAPI may use different binary values in **ENTRYID** structures to refer to the same object. To determine if two entry identifiers refer to the same object, your application can use the **IMAPISupport::CompareEntryIDs** method for the appropriate object. To determine the identity of an object that does not have an entry identifier, your application can compare the two objects' PR_RECORD_KEY properties.

The following rules apply for using the **abFlags[0]** byte in entry identifiers.

- Each object, if queried for its entry identifier, 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, you can use a short-term entry identifier to open an object only in the current MAPI session. You can 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 may be just as good as a permanent entry identifier. In such cases, clear the bits in the **abFlags** array as described in the first rule preceding.

The **CbNewENTRYID** macro determines the memory allocation requirements of an **ENTRYID** structure with an entry identifier of a specifed 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** that contains a _*cb* byte entry identifier.

The **SizedENTRYID** macro creates a structure type definition identical to that of **ENTRYID** but with 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 parameter _*cb* 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 means of defining 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 a **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, **SizedENTRYID** macro

ENTRYLIST

An **ENTRYLIST** structure is an **SBinaryArray** structure that contains a list of MAPI entry identifier structures (that is, **ENTRYID** structures).

Syntax

```
typedef SBinaryArray ENTRYLIST, FAR *LPENTRYLIST;
```

Comments

The ENTRYLIST structure is defined in MAPIDEFS.H.

See Also

ENTRYID structure, **SBinaryArray** structure

ERROR_NOTIFICATION

An **ERROR_NOTIFICATION** structure defines an error notification event that indicates a critical error for an object. MAPI only uses this structure as a member of the **NOTIFICATION** structure for the advise sink.

Syntax

```
typedef struct _ERROR_NOTIFICATION
{
     ULONG cbEntryID;
     LPENTRYID lpEntryID;
     SCODE scode;
     ULONG ulFlags;
     LPMAPIERROR lpMAPIError;
} ERROR_NOTIFICATION;
```

Members

cbEntryID
Size, in bytes, of the entry identifier of the object causing the error.

lpEntryID
Pointer to the entry identifier of the object causing the error.

scode
Error code for the critical error.

ulFlags
Bitmask of flags designating the format of the text pointed to by the **lpszError** 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 8-bit format.

lpszError
Pointer to a text string describing the error.

Comments

The **ERROR_NOTIFICATION** is defined in MAPIDEFS.H.

See Also

NOTIFICATION structure

EXTENDED_NOTIFICATION

An **EXTENDED_NOTIFICATION** structure holds a notification for a specialized, provider-specific, extended event. MAPI only uses this structure as a member of the **NOTIFICATION** structure for the advise sink.

Syntax

```
typedef struct _EXTENDED_NOTIFICATION
{
     ULONG ulEvent;
     ULONG cb;
     LPBYTE pbEventParameters;
} EXTENDED_NOTIFICATION;
```

Members

ulEvent
Extended event code.

cb
Size, in bytes, of the event-specific parameters defined in the **pbEventParameters** member.

pbEventParameters
Pointer 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.

Comments

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

A **FILETIME** structure, implemented for 32-bit Windows, holds a 64-bit time value for a file. This value represents the number of 100-nanosecond intervals since January 1, 1601.

Syntax

```
typedef struct _FILETIME {
     DWORD dwLowDateTime;
     DWORD dwHighDateTime;
} FILETIME, FAR *LPFILETIME;
```

Members

dwLowDateTime
Low-order 32 bits of the file time value.

dwHighDateTime
High-order 32 bits of the file time value.

Comments

Each MAPI property includes a **FILETIME** structure in its definition in an **SPropValue** structure.

The **FILETIME** structure is defined in MAPIDEFS.H.

See Also

SPropValue structure

FLAGLIST

Contains a list of flags.

Syntax

```
typedef struct {
     ULONG cFlags;
     ULONG ulFlag[MAPI_DIM];
} FlagList, FAR * LPFlagList;
```

Members

cFlags
Number of MAPI-defined flags in the list.

ulFlags
First flag in the list.

Comments

See **IABContainer::ResolveNames** and **IDistList::ResolveNames** for descriptions of the flags that can be part of this list.

A **FLAGLIST** structure does not specify the length of the flag list, it only names the first flag in memory allocated for the list.

The **FLAGLIST** structure is defined in MAPIDEFS.H.

See Also

IABContainer::ResolveNames method

FLATENTRY

A **FLATENTRY** structure holds a "flattened" entry identifier. That is, it holds the actual data of an entry identifier, in contrast to an **ENTRYID** structure, which contains only a pointer to that data..

Syntax

```
typedef struct {
     ULONG cb;
     BYTE abEntry[MAPI_DIM];
} FLATENTRY, FAR *LPFLATENTRY;
```

Members

cb
Size, in bytes, of the data in the **abEntry** member.

abEntry
Byte array that contains the data of an entry identifier.

Comments

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** 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

A **FLATENTRYLIST** structure contains an array of **FLATENTRY** structures. A **FLATENTRY** structure holds a "flattened" entry identifier (that is, the actual data of an entry identifier instead of a pointer to that data).

Syntax

```
typedef struct {
     ULONG cEntries;
     ULONG cbEntries;
     BYTE        abEntries[MAPI_DIM];
} FLATENTRYLIST, FAR *LPFLATENTRYLIST;
```

Members

cEntries
Number of **FLATENTRY** structures in the array described by the **abEntries** member.

cbEntries
Number of bytes occupied by the entire array described by **abEntries**.

abEntries
Byte array containing the number of **FLATENTRY** structures designated by the **cEntries** member, arranged end to end.

Comments

In the **abEntries** array, each **FLATENTRY** structure is aligned on a 4-byte boundary. Extra bytes are included as padding to ensure 4-byte alignment between any two **FLATENTRY** structures. For example, the second **FLATENTRY** starts at an offset made up of the first **FLATENTRY**, plus extra bytes as padding to round the space taken up to the next multiple of 4 bytes, plus the offset of the second **FLATENTRY**.

PR_REPLY_RECIPIENT_ENTRIES is a binary property that contains a **FLATENTRYLIST**.

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**s is of a specified byte size. The syntax is:

int CbNewFLATENTRY (**int** _*cb*)

The parameter _*cb* specifies the byte size of the list of **FLATENTRYs.** 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

A **FLATMTSIDLIST** structure contains an array of **MTSID** structures, each of which holds an X.400 message transport system (MTS) entry identifier. .

Syntax

```
typedef struct {
     ULONG cMTSIDs;
     ULONG cbMTSIDs;
     BYTE        abMTSIDs[MAPI_DIM];
} FLATMTSIDLIST, FAR *LPFLATMTSIDLIST;
```

Members

cMTSIDs
Number of **MTSID** structures in the array described by the **abMTSIDs** member.

cbMTSIDs
Number of bytes in the array described by **abMTSIDs**.

abMTSIDs
Byte array of **MTSID** structures.

Comments

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.

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**s is of a specified byte size. The syntax is:

int CbNewMTSID (**int** *_cb*)

The parameter *_cb* 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

Describes the print setup information for the form object.

Syntax

```
typedef struct {
ULONGulFlags;
HDEVMODE         hDevMode;
HDEVNAMES  hDevNames;
ULONGulFirstPageNumber;
ULONGulFPrintAttachments;
} FORMPRINTSETUP, FAR * LPFORMPRINTSETUP;
```

Parameters

ulFlags
Input parameter used to control 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 8-bit format.

hDevmode
Input parameter specifying the mode of the printer.

hDevnames
Input parameter specifying the pathname of the printer.

ulFirstPageNumber
Input parameter containing the page number of the first page to be printed.

ulFPrintAttachments

Input parameter indicating whether there are attachments to be printed. Comments

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 *hDevmode* and *hDevnames* should be in Unicode format. Otherwise, the strings should be ANSI format.

The *hDevMode* and *hDevNames* parameters must be allocated using the Win32 function **GlobalAlloc** and must be freed using Win32 **GlobalFree.** The **FORMPRINTSETUP** structure must be freed by the calling form object using the **MAPIFreeBuffer** function.

The **FORMPRINTSETUP** structure is defined in MAPIFORM.H.

See Also

IMAPIViewContext::GetPrintSetup **MAPIFreeBuffer**

FORMPROPSPECIALTYPE

A **FORMPROPSPECIALTYPE** enumeration lists possible special type tags for the **nSpecialType** member of the **SMAPIFormProp** structure. **SMAPIFormProp** holds information about a form property used as part of the definitions of the **IMAPIFormInfo** interface; **nSpecialType** holds a tag that applies to the **u** union that is part of **SMAPIFormProp**.

Syntax

```
typedef enum FORMPROPSPECIALTYPE
{
     FPST_VANILLA = 0,
     FPST_ENUM_PROP = 1
} FORMPROPSPECIALTYPE;
```

Members

FPST_VANILLA
Indicates no enumeration is used.

FPST_ENUM_PROP
Indicates the enumeration contains a MAPI property.

Comments

The **FORMPROPSPECIALTYPE** is defined in MAPIFORM.H.

See Also

SMAPIFormProp structure

GUID

A **GUID** structure holds a globally unique identifier (GUID), which identifies a particular object class and interface. This identifier is a 128-bit value created with a **DEFINE_GUID** OLE macro. For more information about GUIDs, see the Remote Procedure Call (RPC) documentation, *OLE Programmer's Reference, Volume One,* and *Inside OLE, Second Edition.*

Syntax

typedef struct _GUID

```
{
    unsigned long  Data1;
    unsigned short     Data2;
    unsigned short      Data3;
    unsigned char      Data4[8];
} GUID;
```

Members

Data1
Unsigned long integer data value.

Data2
Unsigned short integer data value.

Data3
Unsigned short integer data value.

Data4
Array of unsigned characters.

Comments

The **GUID** structure is defined in MAPIGUID.H.

IID

An **IID** is a specialized **GUID** structure containing an interface identifier (IID). For more information about IIDs, see the Remote Procedure Call (RPC) documentation, *OLE Programmer's Reference, Volume One,* and *Inside OLE, Second Edition.*

Comments

The **IID** structure is defined in MAPIGUID.H.

See Also

GUID structure

LARGE_INTEGER

A **LARGE_INTEGER** structure, implemented for 32-bit Windows, holds a 64-bit signed integer value.

Syntax

```
typedef struct _LARGE_INTEGER {
     DWORD LowPart;
     DWORD HighPart;
} LARGE_INTEGER, FAR *LARGE_INTEGER;
```

Members

LowPart
  Low-order 32 bits of the value.

HighPart
  High-order 32 bits of the value.

Comments

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

Stores an error code typically generated by the operating system and appropriate help topic information retrieved by **GetLastError** methods.

Syntax

```
typedef struct _MAPIERROR
{
     ULONG ulVersion;
     LPTSTR      lpszError;
     LPTSTR      lpszComponent;
     ULONG ulLowLevelError;
     ULONG ulContext;

} MAPIERROR, FAR * LPMAPIERROR;
```

Members

ulVersion
Input parameter containing the version number of the component.

lpszError
Input parameter pointing to an error message string.

lpszComponent
Input parameter pointing to a component string that appears in a help file mapping entry in MAPISVC.INF. . Each entry consists of a component string on the lefthand side and a Help file path on the right. It is recommended that component strings be no longer that 30 characters. For example:

```
[Help File Mappings]
Microsoft Mail for PC LANs=sfserr.hlp
```

ulLowLevelError
Input parameter specifying the low-level error code used only when the error to be returned is low-level.

ulContext
Input parameter that uniquely identifies the error within **lpszComponent**. If there is a Helpfile for this component, **ulContext** is a Helpfile context identifier; otherwise, it specifies a unique numeric identifier.

Comments

The **MAPIERROR** structure is used to provide error information to calling processes. MAPI methods and functions that return error information typcally include an *lppMAPIError* parameter in which the MAPIERROR structure is placed. To get information on the error, call **IMAPIProp::GetLastError.**

The **MAPIERROR** structure is defined in MAPIDEFS.H.

See Also

IMAPIProp::GetLastError

MapiFile

The **MapiFile** type contains file attachment information.

Syntax

```
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 for future use; 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 a static OLE object file attachment.

Position
An integer describing where the attachment should be placed in the message body. Attachments replace the character found at a certain position in the message body; 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 body. **PathName**
The full path name of the attached file. The file should be closed before this call is made.

FileName
The filename seen by the recipient. This name can differ from the filename in **PathName** if temporary files are being used. If **FileName** 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 tells 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 "" for this parameter.

Comments

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 recreate an OLE object from the file by calling the **OleLoadFromStream** function 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.

With Microsoft Mail version 3.0b, if you send an attachment with *Position* equal to -1, the attachment is placed at the beginning of the message, with a space character on either side of the attachment.

MapiFileDesc

A **MapiFileDesc** structure holds information about a message attachment, which is either an OLE object file or a simple data file.

Syntax

```
typedef struct {
     ULONG ulReserved;
     ULONG flFlags;
     ULONG nPosition;
     LPTSTR lpszPathName;
     LPTSTR lpszFileName;
     LPVOID lpFileType;
} MapiFileDesc, FAR *lpMapiFileDesc;
```

Members

ulReserved
Reserved; must be zero.

flFlags
Bitmask of attachment 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
Set to iIndicates a static OLE object file attachment.

nPosition
Integer used in the **MapiMessage** structure member **lpszNoteText** to indicate at which character the attachment should be rendered in the message body. The application renders an attachment within the message text at the character and offset designated by **lpszNoteText[nPosition]** in **MapiMessage**. A value of - 1 (0xFFFFFFFF) means attachment position is not indicated in this way.

lpszPathName
Pointer to the fully qualified path of the attachment file. This path should include the disk drive letter and directory name.

lpszFileName
Optional pointer 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
Optional pointer to a descriptor that can be used to indicate to message recipients the type of the attached file; an example is a filename extension such as .DOC. A value of NULL indicates an unknown or operating system-determined file type.

Comments

The flag set in the **flFlags** member determines what kind a particular attachment is.

OLE object files are file representations of OLE object streams. When a file attachment is being read from an inbound message, your application can direct the OLE object file attachment stream into an object straight from the attachment file. When a user includes a file attachment in an outbound message, your application should write the OLE object stream directly to the attachment file. For more details about OLE object streams, see *OLE Programmer's Reference, Volume One,* and *Inside OLE,* *Second Edition.*

Extended MAPI determines what kind of file an attachment contains in a different fashion. In Extended MAPI, the **lpFileType** member can be a FAR pointer to an extension structure (that is, the **MapiFileTagExt** structure) that contains full attachment tag information, as defined in Extended MAPI. For most applications running with Microsoft Windows, the attachment filename extension can be reliably used to identify the file type. For example, the filename extension .DOC means that the file is a Microsoft Word for Windows document.

MapiFileDesc is defined in the MAPI header files as **MapiFileDescA** for ANSI or DBCS platforms and **MapiFileDescW** for Unicode platforms. Your application, however, can simply refer to the name **MapiFileDesc**. Be sure to define the symbolic constant UNICODE and limit the platform where necessary. MAPI will interpret the platform information and internally translate **MapiFileDesc** to the appropriate alternative.

When using the **MapiFileDesc** member **nPosition**, your application should not make two attachments in the same location. Applications may not display file attachments beyond the length of the note text.

The **MapiFileDesc** structure is defined in MAPI.H.

See Also

MapiFileTagExt structure

MapiFileTagExt

A **MapiFileTagExt** structure specifies a message attachment's type at its creation and its current form of encoding so it can be restored to its original type at its destination.

Syntax

```
typedef struct {
     ULONG ulReserved;
     ULONG cbTag;
     LPBYTE lpTag;
     ULONG cbEncoding;
     LPBYTE lpEncoding
} MapiFileTagExt, FAR *lpMapiFileTagExt;
```

Members

ulReserved
Reserved; must be zero.

cbTag
Size, in bytes, of the value defined by the **lpTag** member.

lpTag
Pointer to an X.400 object identifier indicating the type of the attachment in its original form, for example "Microsoft Excel worksheet."

cbEncoding
Size, in bytes, of the value defined by the **lpEncoding** member.

lpEncoding
Pointer to an X.400 object identifier indicating the form in which the attachment is currently encoded, for example MacBinary, UUENCODE, or binary.

Comments

MapiFileTagExt defines the type of an attached file for purposes such as filing it, choosing the correct application to launch when opening it, or any use that requires full information regarding the file type. For more information on using MAPI with X.400 messaging, see Chapter 26, "Using MAPI with X.400 Message Systems."

MapiFileTagExt is defined in MAPI.H.

See Also

MapiFileDesc structure

MAPIINIT_0

A **MAPIINIT_0** structure holds the per-instance global data for Extended MAPI.

Syntax

```
typedef struct {
     ULONG ulVersion;
     ULONG ulFlags;
} MAPIINIT_0, FAR *LPMAPIINIT_0;
```

Members

ulVersion
Integer value indicating the MAPIINIT_0 structure version.

ulFlags
Bitmask of flags controlling the format of the data. The following flag can be set:

MAPI_NT_SERVICE
This flag should be passed if the client is a Windows NT service. It is ignored on all platforms other than NT. Value is 0x00010000.

MAPI_MULTITHREAD_NOTIFICATIONS
This flag should be passed if the client wants MAPI notifications to happen on a separate thread, instead of Windows NT and Windows 95. Value is 1.

MAPI_INIT-VERSION
The client application must pass this flag to be sure the correct version of the structure is used.

Comments

MAPIInitialize uses this structure in its processing.

The **MAPIINIT_0** structure is defined in MAPIX.H.

See Also

MapiInitialize function

MapiMessage

A **MapiMessage** structure holds information about a message.

Syntax

```
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
Pointer to the text string describing the message subject, typically limited to 256 characters or less. If this member is empty or NULL, there is no subject text.

lpszNoteText
Pointer to a string containing the message text. If this member is empty or NULL, there is no message text.

lpszMessageType
Pointer to a string indicating the message type. The type is for use by non - interpersonal messaging (IPM) applications. Not all messaging systems support non-IPM messages; systems that do not may not provide, and will ignore, the **lpszMessageType** member. If this member is empty or NULL, the message type is IPM.

lpszDateReceived
Pointer 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
Pointer to a string identifying the conversation thread to which the message belongs. Some messaging systems may ignore and not return this member.

flFlags
Bitmask of message status flags. Flags must be zero for outbound messages and should be ignored for inbound messages. 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.

lpOriginator
Pointer to a **MapiRecipDesc** structure holding information about the sender of the message.

nRecipCount
Number of structures that describe message recipients in the array pointed to by the **lpRecips** member. The number of recipient structures in the message envelope is stored in this member. A value of zero indicates no recipients are included.

lpRecips
Pointer to an array of **MapiRecipDesc** structures, each holding information about a message recipient.

nFileCount
Number of structures that describe file attachments in the array pointed to by the **lpFiles** member. A value of zero indicates no file attachments are included.

lpFiles
Pointer to an array of **MapiFileDesc** structures, each holding information about a file attachment.

Comments

MapiMessage is defined in the MAPI header files as **MapiMessageA** for ANSI or DBCS platforms and **MapiMessageW** for Unicode platforms. Your application, however, can simply refer to the name **MapiMessage**. Be sure to define the manifest constant UNICODE and limit the platform where necessary. MAPI will interpret the platform information and internally translate **MapiMessage** to the appropriate alternative.

The **MAPIMessage** structure is defined in MAPI.H.

See Also

MapiFileDesc structure, **MapiRecipDesc** structure

MAPINAMEID

A **MAPINAMEID** structure associates a property ID in a foreign source with a **GUID** structure representing a globally unique identifier and with an object kind made up of an interface identifier and an object name.

Syntax

```
typedef struct _MAPINAMEID
{
     LPGUID lpguid;
     ULONG ulKind;
     union {
           LONG LID;
           LPWSTR lpwstrName;
     } Kind;
} MAPINAMEID, FAR *LPMAPINAMEID;
```

Members

lpguid
Pointer to a **GUID** structure defining a particular property set.

ulKind
Value describing the kind of object described in the **Kind** union.

LID
Numeric value representing the property name.

lpwstrName
Pointer to a wide character string representing the property name.

Comments

MAPINAMEID is used to make a MAPI compatible name from a name used for another implementation of the messaging system.

The **MAPINAMEID** structure is defined in MAPIDEFS.H.

See Also

GUID structure, **IMAPIProp::GetIDsFromNames** interface

MapiRecipDesc

A **MapiRecipDesc** structure holds information about a message sender or recipient.

Syntax

```
typedef struct {
     ULONG ulReserved
     ULONG ulRecipClass;
     LPTSTR lpszName;
     LPTSTR lpszAddress;
     ULONG ulEIDSize;
     LPVOID lpEntryID;
} MapiRecipDesc, FAR *lpMapiRecipDesc;
```

Members

ulReserved
Reserved; must be zero.

ulRecipClass
Message recipient or sender class. Possible values for the class are:

Value	Constant	Meaning
0	MAPI_ORIG	Indicates the 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
Pointer 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 pointer indicates an address entered by the user for a recipient not in an address book (that is, a custom recipient).

Extended MAPI defines the format of an address pointed to by the **lpszAddress** member as [*address type*][*e-mail address*]. An example of an address is FAX:206-555-1212 or SMTP:M@X.COM.

ulEIDSize
Size, in bytes, of the opaque binary data pointed to by the **lpEntryID** member.

lpEntryID
Pointer to binary entry-identifier data used by the messaging system to specify the message recipient. Unlike the address indicated by **lpszAddress**, this data is opaque and may not be printable. The messaging system returns in this member valid entry identifier data for all recipients or senders listed in accessible address books.

Comments

MapiRecipDesc is defined in the MAPI header files as **MapiRecipDescA** for ANSI or DBCS platforms and **MapiRecipDescW** for Unicode platforms. Your application, however, can simply refer to the name **MapiRecipDesc**. Be sure to define the manifest constant UNICODE and limit the platform where necessary. MAPI will interpret the platform information and internally translate **MapiRecipDesc** to the appropriate alternative.

The **MapiRecipDesc** structure is defined in MAPI.H.

MAPIUID

A **MAPIUID** structure contains a unique identifier (UID) used to identify a particular message conversation or string-to-identifier mapping.

Syntax

```
typedef struct _MAPIUID
{
     BYTE ab[16];
} MAPIUID, FAR *LPMAPIUID;
```

Members

ab
Array containing a 16-byte UID.

Comments

A MAPI UID is a globally unique identifier (GUID) put into Intel processor byte order. That is, a MAPI UID 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. Your 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 application 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 UID 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 *OLE Programmer's Reference, Volume One.*

The **IsEqualMAPIUID** macro is used inline 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 input parameters *lpuid1 and lpuid2* specify pointers to a **MAPIUID** structure identifying the first UID and second UID, respectively.

If your application or provider uses this macro, be sure to include MEMORY.H in your 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

An **MTSID** structure holds 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.

Syntax

```
typedef struct {
     ULONG cb;
     BYTE        ab[MAPI_DIM];
} MTSID, FAR *LPMTSID;
```

Members

cb
Size, in bytes, of the data in the **abEntry** member.

abEntry
Byte array to contain the MTS entry-identifier data.

Comments

The **MTSID** structure is used only for X.400 mappings of MAPI entry identifiers. It corresponds to MAPI's **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 parameter *_lpentry* is a pointer 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 parameter *_cb* is the byte size of the message transfer agent identifier. 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

A **NEWMAIL_NOTIFICATION** structure holds information about a notification event that indicates to an application a new message has arrived. MAPI uses this structure only as a member of the **NOTIFICATION** structure, which holds information about a notification event for the advise sink.

Syntax

```
typedef struct _NEWMAIL_NOTIFICATION
{
     ULONG cbEntryID;
     LPENTRYID lpEntryID;
     ULONG cbParentID;
     LPENTRYID lpParentID;
     ULONG ulFlags;
     LPTSTR lpszMessageClass;
     ULONG ulMessageFlags;
} NEWMAIL_NOTIFICATION;
```

Members

cbEntryID
Size, in bytes, of the entry identifier of the arriving message.

lpEntryID
Pointer to the entry identifier of the arriving message.

cbParentID
Size, in bytes, of the data pointed to by the **lpParentID** member.

lpParentID
Pointer to the entry identifier of the folder where the arriving message was placed.

ulFlags
Bitmask of flags describing 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 8-bit format.

lpszMessageClass
Pointer to the message class of the new message, indicating whether the messaging system is using a Unicode or an ANSI or DBCS platform.

ulMessageFlags
Copy of the PR_MESSAGE_FLAGS property for the message, which contains a bitmask of flags controlling the current state of the message object.

Comments

The **NEWMAIL_NOTIFICATION** structure is defined in MAPIDEFS.H.

See Also

NOTIFICATION structure, PR_MESSAGE_FLAGS property

NOTIFICATION

A **NOTIFICATION** structure holds information about one of several possible notification events for an advise sink.

Syntax

```
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
Type of notification event that occurred. Your 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**
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.

Comments

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

A **NOTIFKEY** structure holds the notification key for the MAPI notification engine; 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.

Syntax

```
typedef struct
{
     ULONG cb;
     BYTE        ab[MAPI_DIM];
} NOTIFKEY, FAR *LPNOTIFKEY;
```

Members

cb
  Size, in bytes, of the notification key in the **ab** member.

ab
  Byte array containing the notification key.

Comments

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

An **OBJECT_NOTIFICATION** structure holds 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.

Syntax

```
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
Size, in bytes, of the entry identifier of the object that was created or modified.

lpEntryID
Pointer to the entry identifier of the object.

ulObjType
Type of object (for example, message or folder). Possible types are:

Value	Constant	Meaning
1	MAPI_STORE	Message store.
2	MAPI_ADDRBOOK	Address book.
3	MAPI_FOLDER	Folder.
4	MAPI_ABCONT	Address book container.
5	MAPI_MESSAGE	Message.
6	MAPI_MAILUSER	Messaging user.
7	MAPI_ATTACH	Message attachment.
8	MAPI_DISTLIST	Distribution list.
9	MAPI_PROFSECT	Profile section.
A	MAPI_STATUS	Status.
B	MAPI_SESSION	Session.

cbParentID
Size, in bytes, of the data pointed to by the **lpParentID** member.

lpParentID
Pointer to the entry identifier of the parent of the object.

cbOldID
Size, in bytes, of the entry identifier for the original object.

lpOldID

Pointer to the entry identifier of the original object. This pointer is NULL if the object has not changed.

cbOldParentID
Size, in bytes, of the data pointed to by the **lpOldParentID** member.

lpOldParentID
Pointer to the entry identifier of the parent of the original object.

lpPropTagArray
Pointer to an **SPropTagArray** structure containing the tags of object properties affected by the notification event.

Comments

The **OBJECT_NOTIFICATION** structure is defined in MAPIDEFS.H.

See Also

NOTIFICATION structure, **SPropTagArray** structure

OPTIONDATA

An **OPTIONDATA** structure contains information specific to an e-mail address type supported by the transport service provider. **OPTIONDATA** address information can apply to a message or to a message recipient.

Syntax

```
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 option data flags. The following flags can be set:

MAPI_MESSAGE
Indicates message-specific options.

MAPI_RECIPIENT
Indicates recipient-specific options.

lpRecipGUID
Pointer to a **GUID** structure identifying the message recipient, if applicable.

lpszAdrType
Pointer to the e-mail address type that must be NULL.

lpszDLLName
Pointer to the name of the DLL to be loaded.

ulOrdinal
Ordinal the transport provider DLL uses to find the client application's option callback function.

cbOptionsData
Size, in bytes, of the data pointed to by the **lpbOptionsData** member.

lpbOptionsData
Pointer to transport-provider option data specific to the recipient or message.

cOptionsProps
Number of properties in the array pointed to by the **lpOptionsProps** member.

lpOptionsProps
Pointer to an array of **SPropValue** structures, each holding information about a default option property.

Comments

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 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 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**).

Syntax

```
typedef FNIDLE   *PFNIDLE;
```

Comments

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

SAndRestriction

An **SAndRestriction** structure holds a group of search restrictions that are combined using a logical AND operation.

Syntax

```
typedef struct _SAndRestriction
{
     ULONG cRes;
     LPSRestriction lpRes;
} SAndRestriction;
```

Members

cRes
Number of search restrictions in the array pointed to by the **lpRes** member.

lpRes
Pointer to an array of **SRestriction** structures to be combined with a logical AND operation.

Comments

The **SAndRestriction** structure is defined in MAPIDEFS.H.

See Also

SRestriction structure

SAppTimeArray

An **SAppTimeArray** structure holds a pointer to an array of application time values for use in an **SPropValue** structure holding information about a property.

Syntax

```
typedef struct _SAppTimeArray
{
     ULONG cValues;
     double      FAR *lpat;
} SAppTimeArray;
```

Members

cValues
Number of values in the array pointed to by the **lpat** member.

lpat
Pointer to an array of application time values.

Comments

The **SAppTimeArray** structure is defined in MAPIDEFS.H.

SAVEOPTS

A **SAVEOPTS** enumeration holds a save options enumeration variable for use with the **IMAPIForm::Close** method.

Syntax

```
typedef enum tagSAVEOPTS
{
     SAVEOPTS_SAVEIFDIRTY = 0,
     SAVEOPTS_NOSAVE = 1,
     SAVEOPTS_PROMPTSAVE = 2
} SAVEOPTS;
```

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.

Comments

The **SAVEOPTS** enumeration is defined in MAPIFORM.H.

See Also

IMAPIForm::ShutdownForm method

SBinary

An **SBinary** structure holds a pointer to a property value of type PT_BINARY for use in an **SPropValue** structure holding information about a property.

Syntax

```
typedef struct _SBinary
{
     ULONG cb;
     LPBYTE      lpb;
} SBinary, FAR *LPSBinary;
```

Members

cb
Size, in bytes, of the data pointed to by the **lpb** member.

lpb
Pointer to the PT_BINARY property value.

Comments

The **SBinary** structure is defined in MAPIDEFS.H.

See Also

SPropValue structure

SBinaryArray

An **SBinaryArray** structure holds a property value of type PT_MV_BINARY for use in an **SPropValue** structure holding information about a multivalued property.

Syntax

```
typedef struct _SBinaryArray
{
     ULONG cValues;
     SBinary     FAR *lpbin;
} SBinaryArray;
```

Members

cValues
Number of values in the array pointed to by the **lpbin** member.

lpbin
Pointer to an array of **SBinary** structures holding the multiple values for the property.

Comments

The **SBinaryArray** structure is defined in MAPIDEFS.H.

See Also

PT_MV_BINARY property type, **SBinary** structure, **SPropValue** structure

SBitMaskRestriction

An **SBitMaskRestriction** structure holds a bitmask search restriction for objects such as tables and message stores.

Syntax

```
typedef struct _SBitMaskRestriction
{
     ULONG relBMR;
     ULONG ulPropTag;
     ULONG ulMask;
} SBitMaskRestriction;
```

Members

relBMR
Relational operator that applies to all property comparison restrictions. 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 based on unequal values.

RELOP_RE
Indicates the comparison is made on LIKE (regular expression) values.

RELOPEQ
Indicates the comparison is made based on equal values.

ulPropTag
Property tag of the property being masked in the search.

ulMask
Bitmask of flags.

Comments

The **SBitmaskRestriction** structure is defined in MAPIDEFS.H.

See Also

SRestriction structure

SCommentRestriction

An **SCommentRestriction** structure holds a comment search restriction for objects such as tables and message stores.

Syntax

```
typedef struct _SCommentRestriction
{
     ULONG cValues;
     LPSRestriction lpRes;
     LPSPropValue lpProp;
} SCommentRestriction;
```

Members

cValues
Number of values in the array pointed to by the **lpProp** member.

lpRes
Pointer to RES_COMMENT in an **SRestriction** structure.

lpProp
Pointer to an array of **SPropValue** structures, each holding information about a property. The method using the **SCommentRestriction** structure ignores the values of the properties when computing the restriction result.

Comments

The **SCommentsRestriction** structure is defined in MAPIDEFS.H.

See Also

SPropValue structure, **SRestriction** structure

SComparePropsRestriction

An **SComparePropsRestriction** structure holds a search restriction that compares properties for objects such as tables and message stores.

Syntax

```
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 based on unequal values.

RELOP_RE
Indicates the comparison is made on LIKE (regular expression) values.

RELOPEQ
Indicates the comparison is made based on equal values.

ulPropTag1
Property tag of the first property to be compared.

ulPropTag2
Property tag of the second property to be compared.

Comments

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 MAPI_E_TOO_COMPLEX.

The **SComparePropsRestriction** structure is defined in MAPIDEFS.H.

See Also

SBitMaskRestriction structure, **SRestriction** structure

SContentRestriction

An **SContentRestriction** structure holds a search restriction for a search of message contents through messages that are part of objects such as tables and message stores.

Syntax

```
typedef struct _SContentRestriction
{
     ULONG ulFuzzyLevel;
     ULONG ulPropTag;
     LPSPropValue lpProp;
} SContentRestriction;
```

Members

ulFuzzyLevel

Fuzzy level defining options for a message contents search. The lower 16 bits contain a code with the following possible values:

FL_FULLSTRING
The comparison deals with property types PT_STRING8 and PT_BINARY. The function checks to see if the property value indicated by *lpSPropValueSrc* is contained as a full string in the other property.

FL_PREFIX
The comparison deals with property types PT_STRING8 and PT_BINARY. The function compares the values of the two properties only up to the length of the property indicated by *lpSPropValueSrc*.

FL_SUBSTRING
The comparison deals with property types PT_STRING8 and PT_BINARY. The function checks to see if the property value indicated by *lpSPropValueSrc* is contained as a substring in the other property.

The upper 16 bits of the fuzzy level make up a bitmask of flags defining different options for comparing character strings within messages. The following flags can be set:

FL_IGNORECASE
The comparison deals only with PT_STRING8 properties. The function makes the comparison in case-insensitive fashion.

FL_IGNORENONSPACE
The comparison deals only with PT_STRING8 properties. The function makes the comparison so as to ignore Unicode-defined "nonspacing characters," for example, diacritical marks.

FL_LOOSE
The comparison deals only with PT_STRING8 properties. The service provider performs as many fuzzy level heuristics of types FL_IGNORECASE and FL_IGNORESPACE as it has been designed to handle.

ulPropTag

Property tag for the property pointed to by the **lpProp** member. MAPI requires only the property type within this tag and ignores the property identifier portion.

lpProp

Pointer to the **SPropValue** structure holding information about a property.

Comments

The error value MAPI_E_TOO_COMPLEX is returned if the data type of the property to be checked in the **ulPropTag** member doesn't match the data type indicated by the **lpProp** pointer.

The **SContentRestriction** structure is defined in MAPIDEFS.H.

See Also

SPropValue structure, **SRestriction** structure

SCurrencyArray

An **SCurrencyArray** structure holds a property value of type PT_MV_CURRENCY for use in an **SPropValue** structure holding information about a multivalued property.

Syntax

```
typedef struct _SCurrencyArray
{
     ULONG cValues;
     CURRENCY FAR *lpcur;
} SCurrencyArray;
```

Members

cValues
Number of values in the array pointed to by the **lpcur** member.

lpcur
Pointer to an array of **CURRENCY** structures holding the multiple values for the property.

Comments

The **SCurrencyArray** structure is defined in MAPIDEFS.H.

See Also

PT_MV_CURRENCY property type, **SPropValue** structure

SDateTimeArray

An **SDateTimeArray** structure holds a property value of type PT_MV_SYSTIME for use in an **SPropValue** structure holding information about a multivalued property.

Syntax

```
typedef struct _SDateTimeArray
{
     ULONG cValues;
     FILETIME FAR *lpft;
} SDateTimeArray;
```

Members

cValues
Number of values in the array pointed to by the **lpft** member.

lpft
Pointer to an array of **FILETIME** structures holding the multiple values for the property.

Comments

The **SDateTimeArray** structure is defined in MAPIDEFS.H.

See Also

FILETIME structure, PT_MV_SYSTIME property type, **SPropValue** structure

SDoubleArray

An **SDoubleArray** structure holds a property value of type PT_MV_DOUBLE for use in an **SPropValue** structure holding information about a multivalued property.

Syntax

```
typedef struct _SDoubleArray
{
     ULONG cValues;
     double      FAR *lpdbl;
} SDoubleArray;
```

Members

cValues
Number of values in the array pointed to by the **lpdbl** member.

lpdbl
Pointer to the array of double values making up the property.

Comments

The **SDoubleArray** structure is defined in MAPIDEFS.H.

See Also

PT_MV_DOUBLE property type, **SPropValue** structure

SExistRestriction

An **SExistRestriction** structure holds a search restriction for a search to determine whether a property exists for an object such as a table or message store.

Syntax

```
typedef struct _SExistRestriction
{
     ULONG ulReserved1;
     ULONG ulPropTag;
     ULONG ulReserved2;
} SExistRestriction;
```

Members

ulReserved1
Reserved; must be zero.

ulPropTag
Property tag of the property to be tested.

ulReserved2
Reserved; must be zero.

Comments

The **SExistRestriction** structure is defined in MAPIDEFS.H.

See Also

SRestriction structure

SGuidArray

An **SGuidArray** structure holds a property value of type PT_MV_CLSID for use in an **SPropValue** structure holding information about a multivalued property.

Syntax

```
typedef struct _SGuidArray
{
     ULONG cValues;
     GUID  FAR *lpguid;
} SGuidArray;
```

Members

cValues
Number of values in the array pointed to by the **lpguid** member.

lpguid
Pointer to an array of property values for use in the **SPropValue** structure.

Comments

The **SGUIDArray** structure is defined in MAPIDEFS.H.

See Also

GUID structure, PT_MV_CLSID property type, **SPropValue** structure

SLargeIntegerArray

An **SLargeIntegerArray** structure holds a property value of type PT_MV_I8 for use in an **SPropValue** structure holding information about a multivalued property.

Syntax

```
typedef struct _SLargeIntegerArray
{
     ULONG cValues;
     LARGE_INTEGER     FAR *lpli;
} SLargeIntegerArray;
```

Members

cValues
Number of values in the array pointed to by the **lpli** member.

lpli
Pointer to an array of **LARGE_INTEGER** structures holding the multiple values for the property.

Comments

The **SLargeIntegerArray** structure is defined in MAPIDEFS.H.

See Also

LARGE_INTEGER structure, PT_MV_I8 property type, **SPropValue** structure

SLongArray

An **SLongArray** structure holds a property value of type PT_MV_LONG for use in an **SPropValue** structure holding information about a multivalued property.

Syntax

```
typedef struct _SLongArray
{
     ULONG cValues;
     LONG   FAR *lpl;
} SLongArray;
```

Members

cValues
Number of values in the array pointed to by the **lpl** member.

lpl
Pointer to the array of long values making up the property.

Comments

The **SLongArray** structure is defined in MAPIDEFS.H.

See Also

PT_MV_LONG property type, **SPropValue** structure

SLPSTRArray

An **SLPSTRArray** structure holds a property value of type PT_MV_STRING8 for use in an **SPropValue** structure holding information about a multivalued property.

Syntax

```
typedef struct _SLPSTRArray
{
     ULONG cValues;
     LPSTR FAR *lppszA;z
} SLPSTRArray;
```

Members

cValues
Number of values in the array pointed to by the **lppszA** member.

lppszA
Pointer to the array of null-terminated 8-bit character strings making up the property.

Comments

The **SLPSTRArray** structure is defined in MAPIDEFS.H.

See Also

PT_MV_STRING8 property type, **SPropValue** structure

SMAPIFormInfoArray

Contains a list of pointers to **IMAPIFormInfo** interfaces.

Syntax

```
typedef struct
{
     ULONG cForms;
     LPMAPIFORMINFO aFormInfo[MAPI_DIM];
} SMAPIFormInfoArray, FAR * LPSMAPIFORMINFOARRAY;
```

Members

cForms
Number of **IMAPIFormInfo** interface pointers in this structures list.

aFormInfo
Names the first pointer in the list of interface pointers.

Comments

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 that are required to hold an **SMAPIFormInfoArray** containing a specified number of **IMAPIFormInfo** interface pointers. The syntax is:

int CbMAPIFormInfoArray (**int** *_c*)

The parameter *_c* specifies the number of **IMAPIFormInfo** interface pointers. This macro returns the number of bytes of memory needed to hold a **SMAPIFormInfoArray** containing *_c* pointers.

The **SMAPIFormInfoArray** structure is defined in MAPIFORM.H.

See Also

IMAPIFormContainer::ResolveMultipleMessageClasses method, **IMAPIFormMgr::CalcFormPropSet** method, **IMAPIFormMgr::ResolveMultipleMessageClasses** method, **IMAPIFormMgr::SelectMultipleForms** method

SMAPIFormProp

An **SMAPIFormProp** structure holds a form property used with form interfaces.

Syntax

```
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

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
Pointer to the display name of the property.

nSpecialType
Special type tag defined in the **FORMPROPSPECIALTYPE** enumeration. This tag applies to the **u** union.

nmidIdx
MAPINAMEID structure containing the identifier for the interface implementing the property.

cfpevAvailable
Number of **SMAPIFormPropEnumVal** structures in the array pointed to by the **pfpevAvailable** member.

pfpevAvailable
Pointer to an array of **SMAPIFormPropEnumVal** structures, each of which holds a value for a form property.

ulFlags
Undefined.

Comments

The **SMAPIFormProp** structure is defined in MAPIDEFS.H.

See Also

FORMPROPSPECIALTYPE structure, **MAPINAMEID** structure, **SMAPIFormPropEnumVal** structure

SMAPIFormPropArray

An **SMAPIFormPropArray** structure contains a list of form properties, i.e., **SMAPIFormProp** structures.

Syntax

```
typedef struct
{
     ULONG cProps;
     ULONG ulPad;
     SMAPIFormProp aFormProp[MAPI_DIM];
} SMAPIFormPropArray, FAR * LPMAPIFORMPROPARRAY;
```

Members

cProps
Number of verbs in the list.

ulPad
An eight byte pad used for alignment of the **SMAPIFormPropArray.aFormProp**
Names the first form property in the list.

Comments

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, i.e. a specified number of **SMAPIFormProp** structures. The syntax is:

int CbMAPIFormPropArray (**int** _*c*)

The parameter _*c* specifies the number of form properties. This macro returns the number of bytes of memory needed to hold an **SMAPIFormPropArray** containing _*c* form properties.

The **SMAPIFormPropArray** structure is defined in MAPIFORM.H.

See Also

CbMAPIFormPropArray macro, **IMAPIFormContainer::CalcFormPropSet** method, **IMAPIFormInfo::CalcFormPropSet** method, **IMAPIFormMgr::CalcFormPropSet** method, **SMAPIFormProp** structure

SMAPIFormPropEnumVal

An **SMAPIFormPropEnumVal** structure holds a value for a form property for use as part of the **SMAPIFormProp** structure.

Syntax

```
typedef struct _SMAPIFormPropEnumVal
{
     SPropValue  val;
     ULONG nVal;
} SMAPIFormPropEnumVal;
```

Members

val
  SPropValue structure holding information about the form property.

nVal
  Enumeration value for the structure in the **val** member.

Comments

The **SMAPIFormProp** structure holds form property information used in forms definitions.

The **SMAPIFormPropEnumVal** structure is defined in MAPIDEFS.H.

See Also

SMAPIFormProp structure, **SPropValue** structure

SMAPIVerb

An **SMAPIVerb** structure contains arrays of MAPI verbs.

Syntax

```
typedef struct
{
     LONG lVerb;
     LPTSTR szVerbname;
     DWORD fuFlags;
     DWORD grfAttribs;
     ULONG ulFlags;                     /* Either 0 or MAPI_UNICODE */
} SMAPIVerb, FAR * LPMAPIVERB;
```

Members

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 8-bit format.

Comments

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

An **SMAPIVerbArray** structure contains a list of MAPI verbs, i.e. **SMAPIVerb** structures.

Syntax

```
typedef struct
{
     ULONG cMAPIVerb;
     SMAPIVerb aMAPIVerb[MAPI_DIM];
} SMAPIVerbArray, FAR * LPMAPIVERBARRAY;
```

Members

cForms
Number of verbs in the list.

aFormInfo
Names the first verb in the list.

Comments

This structure is passed as a parameter in the method **IMAPIFormInfo::CalcVerbSet**.

Use the **cbMAPIVerbArray** macro to determine the memory allocation requirements of an **SMAPIVerbArray** containing a specified number of verbs, i.e. a specified number of **SMAPIVerb** structures. The syntax is:

int CbMAPIVerbArray (**int** _*c*)

The parameter _*c* specifies the number of verbs. This macro returns the number of bytes of memory needed to hold an **SMAPIVerbArray** containing _*c* verbs.

The **SMAPIVerbArray** is defined in MAPIFORM.H.

See Also

IMAPIFormInfo::CalcVerbSet method, **SMAPIVerb** structure

SMessageClassArray

An **SMessageClassArray** structure contains a list of message class string pointers.

Syntax

```
typedef struct
{
     ULONG cValues;
     LPCSTR aMessageClass[MAPI_DIM];
} SMessageClassArray, FAR * LPSMESSAGECLASSARRAY;
```

Members

cValues
The number of message class string pointers in this structure.

aMessageClass
Names the first pointer in the list of message class string pointers.

Comments

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 that are required to hold an **SMessageClassArray** containing a specified number of message class string pointers. The syntax is:

int CbMessageClassArray (**int** _*c*)

The parameter _c specifies the number of message class string pointers. This macro returns the number of bytes of memory needed to hold a **SMessageClassArray** containing _*c* pointers.

The **SMessageClassArray** structure is defined in MAPIDEFS.H.

See Also

IMAPIFormContainer::ResolveMultipleMessageClasses method,
IMAPIFormMgr::ResolveMultipleMessageClasses method

SNotRestriction

An **SNotRestriction** structure holds a group of search restrictions to which a logical NOT operation has been applied.

Syntax

```
typedef struct _SNotRestriction
{
     ULONG ulReserved;
     LPSRestriction lpRes;
} SNotRestriction;
```

Members

ulReserved
Reserved; must be zero.

lpRes
Pointer to an array of **SRestriction** structures holding the restrictions to be included in the logical NOT operation.

The **SNotRestriction** structure is defined in MAPIDEFS.H.

See Also

SRestriction structure

SOrRestriction

An **SOrRestriction** structure holds a group of search restrictions combined using a logical OR operation.

Syntax

```
typedef struct _SOrRestriction
{
     ULONG cRes;
     LPSRestriction lpRes;
} SOrRestriction;
```

Members

cRes
Number of structures in the array pointed to by the **lpRes** member.

lpRes
Pointer to an array of **SRestriction** structures defining the restrictions to be combined using the logical OR operation.

Comments

The **SOeRestriction** structure is defined in MAPIDEFS.H.

See Also

SRestriction structure

SPropAttrArray

An **SPropAttrArray** structure holds a list of attributes for a MAPI property.

Syntax

```
typedef struct
{
     ULONG cValues;
     ULONG aPropAttr[MAPI_DIM];
} SPropAttrArray, FAR *LPSPropAttrArray;
```

Members

cValues
Number of property attributes in the array in the **aPropAttr** member.

aPropAttr
Array of property attributes. Attributes that can be placed in this list are:

PROPATTR_MANDATORY
PROPATTR_READABLE
PROPATTR_WRITEABLE
PROPATTR_NOT_PRESENT

Comments

Use the **CbSPropAttrArray** macro to determine the number of bytes occupied by an existing **SPropAttrArray**. The syntax is:

int CbSPropAttrArray (**LPSPROPATTRARRAY** *_lparray*)

The *_lparray* parameter is a pointer 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 parameter *_c* specifies the number of property attributes. This macro returns the number of bytes of memory needed to hold an **SPropAttrArray** containing *_c* property attributes.

The **SPropAttrArray** structure is defined in IMESSAGE.H.

SPropertyRestriction

An **SPropertyRestriction** structure holds a search restriction that compares a property to a constant for a search on an object such as a table or message store.

Syntax

```
typedef struct _SPropertyRestriction
{
     ULONG relop;
     ULONG ulPropTag;
     LPSPropValue lpProp;
} SPropertyRestriction;
```

Members

relop
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 based on unequal values.

RELOP_RE
Indicates the comparison is made on LIKE (regular expression) values.

RELOP_EQ
Indicates the comparison is made based on equal values.

ulPropTag
Property tag for the property involved.

lpProp
Pointer to an **SPropValue** structure holding the property. MAPI requires only the property type within this **SPropValue** structure and ignores the property identifier portion.

Comments

The comparison order is *(property value) (relational operator) (constant value)*.

The method returns the error value MAPI_E_TOO_COMPLEX when the type of the property to be checked does not match the tag in the **ulPropTag** member.

The **SPropertyRestriction** structure is defined in MAPIDEFS.H.

See Also

SPropValue structure, **SRestriction** structure

SPropProblem

An **SPropProblem** structure holds an SCODE error value that is a result of an operation attempting to modify or delete a MAPI property.

Syntax

```
typedef struct _SPropProblem {
     ULONG ulIndex;
     ULONG ulPropTag;
     SCODE scode;
} SPropProblem, FAR *LPSPropProblem;
```

Members

ulIndex
Index in an array of property tags of the property for which a problem is indicated.

ulPropTag
Property tag for the property.

scode
Error code indicating the property problem encountered during the update. This code can be any SCODE value.

Comments

The **SPropProblem** structure is defined in MAPIDEFS.H.

See Also

SCODE data type, **SPropProblemArray** structure

SPropProblemArray

An **SPropProblemArray** structure holds 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.

Syntax

```
typedef struct _SPropProblemArray {
     ULONG cProblem;
     SPropProblem aProblem[MAPI_DIM];
} SPropProblemArray, FAR *LPSPropProblemArray;
```

Members

cProblem
Number of problem structures in the array indicated by the **aProblem** member.

aProblem
Array of **SPropProblem** structures, each holding a property error.

Comments

The **SPropProblemArray** structure is defined in MAPIDEFS.H.

See Also

SCODE data type, **SPropProblem** structure

SPropTagArray

An **SPropTagArray** structure contains an array of property tags.

Syntax

```
typedef struct _SPropTagArray {
     ULONG cValues;
     ULONG aulPropTag[MAPI_DIM];
} SPropTagArray, FAR *LPSPropTagArray;
```

Members

cValues
Number of property tags in the array indicated by the **aulPropTag** member.

aulPropTag
Array of property tags.

Comments

The **CbSPropTagArray** macro determines the number of bytes occupied by an existing **SPropTagArray**. The syntax is:

int CbSPropTagArray (**LPSPropTagArray** *_lparray*)

The *_lparray* specifies a pointer to an **SPropTagArray**. 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** 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 *_ctag* property tags.

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 parameter *_ctag* 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 a sized property tag array pointer *lpSizedSPropTagArray* 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

An **SPropValue** structure holds a MAPI property, including its property tag and property value.

Syntax

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

Members

ulPropTag
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
Padding bytes to properly align the information indicated by the **Value** member.

Value
Property value from the **_UPV** union.

Comments

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.

Five macros to set data types, flags, or to return property or type or property identifier value.

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 ID 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 parameter *tag* 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 logical or of the MV_FLAG and the MV_INSTANCE flags.

The MV_FLAG indicates a multi-valued property. The MV_INSTANCE flag is used in table operations to request that a multi-valued 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 multi-valued types.

The **PROP_ID** macro returns the property ID value from the supplied property tag. The syntax is:

ULONG PROP_ID (**ULONG** *ulPropTag*)

The parameter ulPropTag specifies a property tag for which you want to obtain the property ID. This macro returns the property ID in the low-order word (bits 0-15) and zeroes 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 contains the type and the high-order 16 bits of the returned tag contains the property identifier. The syntax is:

ULONG   PROP_TAG (*ulPropType, ulPropID***)**

The parameter *ulPropType* specifies the property type to be used in the property tag. The parameter *ulPropID* 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 parameter ulPropTag 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 0. For example, PROP_TYPE(PR_ENTRYID) returns the value PT_BINARY.

The **SPropValue** structure is defined in MAPIDEFS.H.

See Also

Extended MAPI Property Types

SRealArray

An **SRealArray** structure holds a property value of type PT_FLOAT for use in an **SPropValue** structure holding information about a multivalued property.

Syntax

```
typedef struct _SRealArray
{
     ULONG cValues;
     float FAR *lpflt;
} SRealArray;
```

Members

cValues
Number of values in the array pointed to by the **lpflt** member.

lpflt
Pointer to an array of float values making up the property.

Comments

The **SRealArray** structure is defined in MAPIDEFS.H.

See Also

PT_FLOAT property type **SPropValue** structure

SRestriction

An **SRestriction** structure holds a search restriction or a set of search restrictions.

Syntax

```
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

Restriction type. Possible values are:

Value	Level	Meaning
0	RES_AND	A logical AND restriction, defined by an **SAndRestriction** structure.
1	RES_OR	A logical OR restriction, defined by an **SOrRestriction** structure.
2	RES_NOT	A logical NOT restriction, defined by an **SNotRestriction** structure.
3	RES_CONTENT	A message content restriction, defined by an **SContentRestriction** structure.
4	RES_PROPERTY	A property restriction, defined by an **SPropertyRestriction** structure.
5	RES_COMPAREPROPS	A property comparison restriction, defined by an **SComparePropsRestriction** structure.
6	RES_BITMASK	A bitmask restriction, defined by an **SBitMaskRestriction** structure.
7	RES_SIZE	A size restriction, defined by an **SSizeRestriction** structure.
8	RES_EXIST	A property existence restriction, defined by an **SExistRestriction** structure.
9	RES_SUBRESTRICTIO	A subrestriction restriction, defined by

	N	an **SSubRestrictionRestriction** structure.
A	RES_COMMENT	A comment restriction, defined by an **SCommentRestriction** structure.

resCompareProps
An **SComparePropsRestriction** structure. This structure is the first in the union to accommodate static initializations of three-value restrictions.

resAnd
An **SAndRestriction** structure.

resOr
An **SOrRestriction** structure.

resContent
An **SContentRestriction** structure.

resProperty
An **SPropertyRestriction** structure.

resBitMask
An **SBitMaskRestriction** structure.

resSize
An **SSizeRestriction** structure.

resExist
An **SExistRestriction** structure.

resSub
An **SSubRestriction** structure.

resComment
An **SCommentRestriction** structure.

Comments

A client application uses **SRestriction** structures in calls to the **IMAPITable::Restrict** and (for search-results folders) **IMAPIContainer::SetSearchCriteria** methods. An application 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**. During a search or restriction operation, a client application 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.

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

An **SRow** structure holds a table row containing selected properties for a specific object.

Syntax

```
typedef struct _SRow {
     ULONG ulAdrEntryPad;
     ULONG cValues;
     LPSPropValue lpProps;
} SRow, FAR *LPSRow;
```

Members

ulAdrEntryPad
Padding bytes to properly align the information pointed to by the **lpProps** member.

cValues
Number of values in the array pointed to by **lpProps**.

lpProps
Pointer to an array of **SPropValue** structures. Each SPropValue structure represents column within the **SRow**.

Comments

SRows 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

An **SRowSet** structure holds a set of table rows; each row in this set of rows contains selected properties for a specific object.

Syntax

```
typedef struct _SRowSet
{
     ULONG cRows;
     SRow aRow[MAPI_DIM];
} SRowSet, FAR *LPSRowSet;
```

Members

cRows
Number of rows in the array indicated by the **aRow** member.

aRow
Array of **SRow** structures, one for each table row.

Comments

The MAPI function **HrQueryAllRows** retrieves MAPI table rows into this structure.

The structure member types and allocation rules for **SRows** and **ADRENTRY**s are identical. **SRowSet** structures can be cast into **ADRLISTs** 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 parameter _*lpSRowSet* 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 parameter _*crow* specifies the number of rows, i.e., the number of elements of type **SRow** in the **aRow** member. This macro returns the number of bytes of memory that an **SRowSet** with _*crow* rows 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 parameter _*crow* 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

An **SShortArray** structure holds a property value of type PT_MV_SHORT for use in an **SPropValue** structure holding information about a multivalued property.

Syntax

```
typedef struct _SShortArray
{
     ULONG cValues;
     short int   FAR *lpi;
} SShortArray;
```

Members

cValues
Number of values in the array pointed to by the **lpi** member.

lpi
Pointer to an array of short values making up the property.

Comments

The **SRowSet** structure is defined in MAPIDEFS.H.

See Also

PT_MV_SHORT property type, **SPropValue** structure

SSizeRestriction

An **SSizeRestriction** structure holds a size search restriction for objects such as tables and message stores.

Syntax

```
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 based on unequal values.

RELOP_RE
Indicates the comparison is made on LIKE (regular expression) values.

RELOPEQ
Indicates the comparison is made based on equal values.

ulPropTag
Property tag.

cb
Size, in bytes, of the property value.

Comments

The **SSizeRestriction** structure is defined in MAPIDEFS.H.

See Also

SRestriction structure

SSortOrder

A **SSortOrder** structure defines the sort order of data in a table.

Syntax

```
typedef struct _SSortOrder
{
     ULONG ulPropTag;
     ULONG ulOrder;
} SSortOrder, FAR *LPSSortOrder;
```

Members

ulPropTag
Property tag of the column on which the table is to be sorted.

ulOrder
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 category to its left. Using this flag reduces the number of category rows which are displayed. This flag can be used on multiple adjacent columns for multiple combinations.

TABLE_SORT_DESCEND
Sorts the table in descending order.

Comments

The **SSortOrder** structure is defined in MAPIDEFS.H.

See Also

SSortOrderSet structure

SSortOrderSet

An **SSortOrderSet** structure defines a set of sort orders for a table, each order indicating on which column the table is to be sorted.

Syntax

```
typedef struct _SSortOrderSet
{
     ULONG cSorts;
     ULONG cCategories;
     ULONG cExpanded;
     SSortOrder aSort[MAPI_DIM];
} SSortOrderSet, FAR *LPSSortOrderSet;
```

Members

cSorts
Number of columns on which to sort the table in the array in the **aSort** member.

cCategories
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
Number of categories that start in an expanded condition. . Possible values include zero.

aSort
Array of **SSortOrder** structures, each defining a sort order.

Comments

Use the **CbSSortOrderSet** macro to determine the number of bytes of memory occupied by an existing **SSortOrderSet** structure. The syntax is:

int CbSSortOrderSet (**LPSSortOrderSet** *_lpSSortOrderSet*)

The parameter *lpSSortOrderSet* 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:

int CbNewSSortOrderSet (**int** *_csort*)

The *_csort* parameter specifies the number of sort orders, i.e., the number of elements of type **SSortOrder** in the **aSort** member. This macro returns the number of bytes of memory that an **SSortOrderSet** with *_csort* sort orders would occupy.

The **SizedSSortOrderSet** macro creates a structure definition identical to that of **SSortOrderSet** but specifies the number columns to sort on. Use the **SizedSSortOrderSet** macro to create sort order sets with explicit bounds. The syntax is:

SizedSSortOrderSet (**int** *_csort*, *_name*)

The parameter *_crow* 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 a **LPSSortOrderSet** pointer, perform the following cast:

```
lpSSortOrderSet = (LPSSortOrderSet) lpSizedSSortOrderSet.
```

The **SSortOrderSet** structure is defined in MAPIDEFS.H.

SSubRestriction

An **SSubRestriction** structure holds a search subrestriction for subobjects of table entries.

Syntax

```
typedef struct _SSubRestriction
{
     ULONG ulSubObject;
     LPSRestriction lpRes;
} SSubRestriction;
```

Members

ulSubObject
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
Pointer to RES_SUBRESTRICTION restriction type.

Comments

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

A **STATUS_OBJECT_NOTIFICATION** structure holds 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.

Syntax

```
typedef struct
{
     ULONG cbEntryID;
     LPENTRYID lpEntryID;
     ULONG cValues;
     LPSPropValue lpPropVals;
} STATUS_OBJECT_NOTIFICATION;
```

Members

cbEntryID
Size, in bytes, of the entry identifier of the changed status object.

lpEntryID
Pointer to the entry identifier of the changed status object.

cValues
Number of **SPropValue** structures in the array pointed to by the **lpPropVals** member.

lpPropVals
Pointer to an array of **SPropValue** structures, one for each changed property of the status object.

Comments

The **STATUS_OBJECT_NOTIFICATION** structure is defined in MAPIDEFS.H.

See Also

NOTIFICATION structure, **SPropValue** structure

STnefProblem

An **STnefProblem** structure holds information about a property or attribute processing problem that occurred during the encoding or decoding of a Transport-Neutral Encapsulation Format (TNEF) stream.

Syntax

```
typedef struct _STnefProblem
{
     ULONG ulComponent;
     ULONG ulAttribute;
     ULONG ulPropTag;
     SCODE scode;
} STnefProblem;
```

Members

ulComponent
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
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
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
Error code indicating the problem encountered during processing. The error can be any SCODE value as listed in Appendix A, "Return Codes." The error value MAPI_E_UNABLE_TO_COMPLETE is returned when the processing problem occurs in an encapsulation block.

Comments

If an **STnefProblem** structure is not generated during the processing of an attribute or property, your application can continue under the assumption 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** is defined in TNEF.H.

See Also

STnefProblemArray structure

STnefProblemArray

An **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.

Syntax

```
typedef struct _SPropProblemArray {
     ULONG cProblem;
     SPropProblem aProblem[MAPI_DIM];
} SPropProblemArray, FAR *LPSPropProblemArray;
```

Members

cProblem
Number of elements in the array in the **aProblem** member.

aProblem
Array of **STnefProblem** structures. Each structure holds information about a property or attribute processing problem.

Comments

If a problem occurs during attribute or property processing, an output parameter in the method **ITnef::ExtractProps** and in the method **ITnef::Finish** each receive a pointer to a **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 a **STnefProblemArray** was generated.

If an **STnefProblem** structure is not generated during the processing of an attribute or property, your application can continue under the assumption 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 **STnefProblemArray** structure is defined in TNEF.H.

See Also

ITnef::ExtractProps method, **ITnef::Finish** method, **STnefProblem** structure

SWStringArray

An **SWStringArray** structure holds a property value of type PT_MV_UNICODE for use in an **SPropValue** structure holding information about a multivalued property.

Syntax

```
typedef struct _SWStringArray
{
     ULONG cValues;
     LPWSTR      FAR *lppszW;
} SWStringArray;
```

Members

cValues
Number of values in the array pointed to by the **lppszW** member.

lppszW
Pointer to the array of null-terminated Unicode string values making up the property.

Comments

The **SWStringArray** structure is defined in MAPIDEFS.H.

See Also

PT_MV_UNICODE property type, **SPropValue** structure

TABLE_NOTIFICATION

The **TABLE_NOTIFICATION** data structure holds 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.

Syntax

```
typedef struct _TABLE_NOTIFICATION
{
     ULONG ulTableEvent;
     HRESULT hResult;
     SPropValue propIndex;
     SPropValue propPrior;
     SRow row;
} TABLE_NOTIFICATION;
```

Members

ulTableEvent
Bitmask of flags representing 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.

In response to this flag, the client application can reread the entire table to get current data.

TABLE_ERROR
Indicates an error has occurred during asynchronous table processing.

TABLE_RELOAD
Indicates a need to re-read the data and start over. When clients receive this event, they should assume that nothing about the table is still valid. All bookmarks, instance keys, status and positioning information are obsolete.

TABLE_RESTRICT_DONE
Indicates a search has completed the table.

TABLE_ROW_ADDED
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 zero.

TABLE_ROW_DELETED
Indicates the **propIndex** member that the client application has deleted.

Because the client application handles notifications asynchronously, notification of deletion of part or all of a table might arrive after the application is already aware of the change.

TABLE_ROW_MODIFIED
Indicates the **propIndex** member identifies the row that the application has modified.

Because the client application handles notifications asynchronously from regular processing, notification of a table modification might arrive after the application is already aware of the change.

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 zero.

TABLE_SETCOL_DONE
Indicates the table's columns have been set.

TABLE_SORT_DONE
Indicates the table has been sorted.

hResult
HRESULT value for the TABLE_ERROR event listed for the **ulTableEvent** member.

propIndex
SPropValue structure giving the index of the table row that has changed, that is, the current row's index.

propPrior
SPropValue structure giving the index of the row preceding the current one.

row
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.

Comments

TABLE_ERROR can be sent as a result of asynchronous calls to the **Sort**, **IMAPITable::Restrict**, or **IMAPITable::SetColumns** method. 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 zero.

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, your 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, **GetLastErrorSort**

Simple Data Types

This section contains a reference entry for each data type.

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.

Syntax

```
typedef ULONG BOOKMARK;
```

Comments

MAPI defines the bookmarks listed following.

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.

Your application can create additional bookmarks. Created bookmarks are only useful while a table is open. Your application 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.

Syntax

```
typedef unsigned char  BYTE;
```

HRESULT

HRESULT is a data type that is a 32-bit error or warning code.

Syntax

```
typedef LONG            HRESULT;
```

Comments

An HRESULT 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.

A HRESULT type returned as an error code 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 codes of these two types. To create an HRESULT code from an SCODE code, use **ResultFromScode(SCODE)**. To convert an SCODE code to HRESULT form, use **GetScode(HRESULT)**. For details about **ResultFromScode** and **GetScode** and for faster ways of making the conversions just mentioned, see *OLE Programmer's Reference, Volume One.* . 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.

Syntax

```
typedef unsigned long LHANDLE, FAR *LPLHANDLE;
```

LONG

LONG is a data type that is a 32-bit signed integer.

Syntax

```
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.

Syntax

```
typedef ULONG    SCODE;
```

Comments

All MAPI functions and methods return SCODE values; some MAPI functions also return warnings, which are non zero HRESULT values.

To obtain an SCODE value from an HRESULT value, your 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.

Syntax

```
typedef char           TCHAR;
typedef WCHAR  TCHAR;
```

Comments

Your 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.

Syntax

```
typedef unsigned long  ULONG;
```

WCHAR

WCHAR is a data type that is a Unicode character string.

Syntax

```
typedef WORD            WCHAR;
```

Additional Extended MAPI Functions

This chapter documents a variety of utility functions for Extended MAPI that are occasionally useful for application development. For a more general discussion of how Extended MAPI functions work, see MA*PI Programmer's Guide.* For information on required functions, refer to Chapter 4, "Essential Extended MAPI Functions."

ChangeIdleRoutine

[New - Windows 95]

Changes some or all of the characteristics of an idle function (that is, a function based on the **FNIDLE** function prototype) included in a client application or service provider.

Syntax

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 new priority for the idle function. priorities for implementation-defined functions are greater than or less than zero, but not zero. The zero priority is reserved for a user event such as a mouse click..

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 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 is to enable the idle function when the **FtgRegisterIdleRoutine** function registers it.

FIROINTERVAL
Indicates that the time indicated by *csecIdle* is the minimum interval between successive calls.

FIROWAIT
Indicates that the time indicated by *csecIdle* is the minimum user idle time that must elapse before the MAPI idle engine will call the idle function for the first time.

ircIdle
Input parameter containing a bitmask of flags used to indicate 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 (*csecIdle*).

FIRCIRO
: Indicates a change to the options for the idle function (*iroIdle*).

FIRCPFN
: Indicates a change to the idle function pointer (*pfnIdle*).

FIRCPRI
: Indicates a change to the priority of the idle function (*priIdle*).

FIRCPV
: Indicates a change to the memory block of the idle function (*pvIdleParam*).

Comments

The **ChangeIdleRoutine** function is defined in MAPIUTIL.H.

See Also

FNIDLE function prototype, **FtgRegisterIdleRoutine** function

CloseIMsgSession

[New - Windows 95]

Closes an **IMessage** session.

Syntax

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.

Comments

A client application uses this function along with **OpenIMsgSession** to wrap the creation of messages inside a message session. When the 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 the **CloseIMsgSession** function, it closes all these objects.

The **CloseIMsgSession** function is defined in IMESSAGE.H.

See Also

IMessage : IMAPIProp interface, **OpenIMsgSession** function

DeregisterIdleRoutine

[New - Windows 95]

Removes a client application or service provider function based on the **FNIDLE** function prototype from the idle table.

Syntax

VOID DeregisterIdleRoutine(FTG *ftg***)**

Parameters

ftg
Input parameter containing a function tag that identifies the idle function to be removed.

Comments

A client application or service provider can only remove an idle function (that is, a function based on **FNIDLE**) 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.

The application 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.

The **DeregisterIdleRoutine** function is defined in MAPIUTIL.H.

See Also

FNIDLE function prototype, **FtgRegisterIdleRoutine** function

EnableIdleRoutine

[New - Windows 95]

Enables or disables an idle function (that is, a function based on the **FNIDLE** function prototype).

Syntax

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, and FALSE if the idle engine should disable it.

Comments

The **EnableIdleRoutine** function is defined in MAPIUTIL.H.

See Also

FNIDLE function prototype, **FtgRegisterIdleRoutine** function

FBadColumnSet

[New - Windows 95]

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.

Syntax

ULONG FBadColumnSet(LPSPropTagArray *lpptaCols***)**

Parameters

lpptaCols
Input parameter pointing to an array of **SPropTagArray** structures defining the table column sets to validate.

Return Values

Returns TRUE if the specified column sets are invalid, and FALSE otherwise.

Comments

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.

The **FBadColumnSet** function is defined in MAPIVAL.H.

See Also

IMAPITable::SetColumns method, **SPropTagArray** structure

FBadEntryList

[New - Windows 95]

Validates a list of MAPI entry identifiers.

Syntax

BOOL FBadEntryList(LPENTRYLIST *lpEntryList***)**

Parameters

lpEntryList
Input parameter pointing to an **ENTRYLIST** structure defining the list of entry identifiers to be validated.

Comments

This 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.

The **FBadEntryList** function is defined in MAPIVAL.H.

Return Values

Returns TRUE if any of the listed entry identifiers are invalid, and FALSE otherwise.

See Also

ENTRYLIST structure

FBadProp

[New - Windows 95]

Validates a specified property.

Syntax

ULONG FBadProp(LPSPropValue *lpprop***)**

Parameters

lpprop
Input parameter containing an **SPropValue** structure defining the property to be validated.

Return Values

Returns TRUE if the specified property is invalid, and FALSE otherwise.

Comments

A service provider can call this function for several reasons, for example to prepare for a call to the **IMAPIProp::SetProps** method setting a property. The **FBadProp** function 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.

The **FBadProp** function is defined in MAPIVAL.H.

See Also

FBadPropTag function, **IMAPIProp::SetProps** method, **SPropValue** structure

FBadPropTag

[New - Windows 95]

Validates a specified property tag.

Syntax

ULONG FBadPropTag(ULONG *ulPropTag***)**

Parameters

ulPropTag
  Input parameter containing the property tag to be validated.

Return Values

Returns TRUE if the specified property tag is not a valid MAPI property tag, and FALSE otherwise.

Comments

A service provider calls the **FBadPropTag** function.

The **FBadPropTag** function is defined in MAPIVAL.H.

See Also

FBadProp function

FBadRestriction

[New - Windows 95]

Validates a restriction used to limit a table.

Syntax

ULONG FBadRestriction(LPSRestriction *lpres***)**

Parameters

lpres
Input parameter containing an **SRestriction** structure defining the restriction to be validated.

Return Values

Returns TRUE if the specified restriction or any of its subrestrictions is invalid, and FALSE otherwise.

Comments

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.

A **FBadRestriction** function is defined in MAPIVAL.H.

See Also

IMAPIContainer : IMAPIProp interface, **IMAPITable::FindRow** method, **IMAPITable::Restrict** method, **SRestriction** structure

FBadRgIpNameID

[New - Windows 95]

Validates an array of pointers that specify name identifier structures and verifies their allocation.

Syntax

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

Returns TRUE if any one of the specified name identifiers is invalid, and FALSE otherwise.

Comments

The **FBadRgIpNameID** function is defined in MAPIVAL.H.

See Also

IMAPIProp::GetIDsFromNames method, **IMAPIProp::GetNamesFromIDs** method, **MAPINAMEID** structure

FBadRglpszW

[New - Windows 95]

Validates all strings in an array of Unicode strings.

Syntax

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

Returns TRUE if any of the strings in the specified array are invalid, and FALSE otherwise.

Comments

The **FBadRglpszW** function is defined in MAPIVAL.H

FBadRow

[New - Windows 95]

Validates a row in a table.

Syntax

ULONG FBadRow(LPSRow *lprow***)**

Parameters

lprow
Input parameter pointing to an **SRow** structure identifying the row to be validated.

Return Values

Returns TRUE if the specified row is invalid, and FALSE otherwise.

Comments

A service provider calls the **FBadRow** function.

The **FBadRow** function is defined in MAPIVAL.H.

See Also

FBadRowSet function, **SRow** structure

FBadRowSet

[New - Windows 95]

Validates all table rows included in a set of table rows.

Syntax

BOOL FBadRowSet(LPSRowSet *lpRowSet***)**

Parameters

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

Returns TRUE if any row of the specified row set is invalid or if the row set itself is invalid, and FALSE otherwise.

Comments

A service provider calls the **FBadRowSet** function.

The **FBadRowSet** function is defined in MAPIVAL.H.

See Also

FBadRow function, **SRowSet** structure

FBadSortOrderSet

[New - Windows 95]

Validates a sort order set by verifying its memory allocation.

Syntax

ULONG FBadSortOrderSet(LPSSortOrderSet *lpsos***)**

Parameters

lpsos
Input parameter pointing to an **SSortOrderSet** structure identifying the sort order set to be validated.

Return Values

Returns TRUE if the specified sort order set is invalid, and FALSE otherwise.

Comments

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.

The **FBadSortOrderSet** function is defined in MAPIVAL.H.

See Also

IMAPITable::SortTable method, **SSortOrderSet** structure

FBinFromHex

[New - Windows 95]

Converts a string representation of a hexadecimal number to binary data.

Syntax

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 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.

Return Values

Returns TRUE if the function successfully converts the string into a binary number, and FALSE if the input string contains invalid ASCII hexadecimal characters.

Comments

The **FBinFromHex** function is defined in MAPIUTIL.H.

See Also

ScBinFromHexBounded function

FEqualNames

[New - Windows 95]

Determines whether two MAPI name identifiers are equal.

Syntax

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

Returns TRUE if the two name identifiers are equal, and FALSE otherwise.

Comments

This 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.

The **FEqualNames** function is defined in MAPIUTIL.H.

See Also

MAPINAMEID structure

FPropCompareProp

[New - Windows 95]

Compares two properties using a binary relational operator.

Syntax

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

Returns TRUE if it succeeds in comparing the input properties, and FALSE otherwise.

Comments

The order of comparison is *lpSPropValue1*, *ulRelOp*, *lpSPropValue2*. If the property types of the properties 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. **FPropContainsProp** can be used to prepare restrictions for generating a table.

The **FPropCompareProp** function is defined in MAPIUTIL.H.

See Also

SPropValue structure

FPropContainsProp

[New - Windows 95]

Compares two property values, generally strings or binary arrays, to see if one value contains the other.

Syntax

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 the **FPropContainsProp** function 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; 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*.

FL_SUBSTRING
The comparison deals with property types PT_STRING8 and PT_BINARY. The function checks to see if the property value indicated by *lpSPropValueSrc* is contained as a substring in the other property.

Return Values

This function returns 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 *lpSPropValueSrc* is contained as a substring in the property indicated by *lpSPropValueDst*.
- 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*.

The function returns FALSE if the properties being compared are not both of the same type, if one or both of the properties is not of either the PT_STRING8 or PT_BINARY type, or if the input fuzzy level is not one of those listed for *ulFuzzyLevel*.

Comments

For comparisons of properties of type PT_STRING8 not covered by one of the values for *ulFuzzyLevel*, **FPropContainsProp** compares the input property values as fuzzy level 1.

The comparison method this function 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.

The **FPropContainsProp** function is defined in MAPIUTIL.H.

See Also

FPropCompareProp function, **SPropValue** structure

FPropExists

[New - Windows 95]

Searches for a given property tag in an **IMAPIProp** interface or an interface derived from **IMAPIProp**, such as **IMessage** or **IMAPIFolder**.

Syntax

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

Returns TRUE if the function finds a match for the given property tag, and FALSE otherwise.

Comments

If the given property tag has type PT_UNSPECIFIED, the function finds a match only for the property identifier. Otherwise, the match is for the entire property tag, including the type.

The **FPropExists** function is defined in MAPIUTIL.H.

See Also

IMAPIFolder : IMAPIContainer interface, **IMAPIProp : IUnknown** interface, **IMessage : IMAPIProp** interface

FtAddFt

[New - Windows 95]

Adds one unsigned 64-bit integer to another.

Syntax

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.

Return Values

Returns a **FILETIME** structure containing the sum of the two integers.

Comments

The **FtAddFt** function is defined in MAPIUTIL.H.

See Also

FILETIME structure, **FtNegFt** function, **FtSubFt** function

FtgRegisterIdleRoutine

[New - Windows 95]

Adds a client application or service provider function based on the **FNIDLE** function prototype to the idle table.

Syntax

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, a WM_PAINT message, and so on).

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 *csecIdle* parameter represents an input to **FtgRegisterIdleRoutine**.

The meaning of the initial time value varies, depending on the calling implementation's requirements. It can be:

- The minimum user idle time that must elapse before the MAPI idle engine will call the idle function (FIROWAIT flag, described following for the *iroIdle* parameter). After this time, the idle engine can call the function as often as necessary.
- The minimum interval between calls to the idle function (FIROINTERVAL flag, described following for *iroIdle*).

iroIdle
Input parameter containing a bitmask of flags used to 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 the **FtgRegisterIdleRoutine** function registers it.

FIROINTERVAL
Indicates that the time specified by *csecIdle* is the minimum interval between successive calls to the idle function.

FIROWAIT
Indicates that the time specified by *csecIdle* 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.

Return Values

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 application or service provider, for example because of memory problems, it returns NULL.

Comments

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, your client 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 is defined in MAPIUTIL.H.

See Also

ChangeIdleRoutine function, **DeregisterIdleRoutine** function, **FNIDLE** function prototype

FtgRegisterIdleRoutine, FNIDLE

[New - Windows 95]

The **FNIDLE** prototype function represents a client application or service provider idle function that the MAPI idle engine calls periodically according to priority. The specific functionality of the idle function is defined by the application or provider.

Syntax

BOOL (STDAPICALLTYPE FNIDLE)(*LPVOID***)**

Parameters

LPVOID

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

An idle function with the **FNIDLE** prototype should always return FALSE.

Comments

The 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 **FtgRegisterIdleRoutine**. Then the application or provider can use these other idle engine functions as needed during idle operations:

- **EnableIdleRoutine**
- **ChangeIdleRoutine**
- **DeregisterIdleRoutine**
- **Idle_DeInitDLL**
- **FIsIdleExit**
- **FDoNextIdleTask**

The **FNIDLE** function is defined in MAPIUTIL.H.

See Also

ChangeIdleRoutine function, **DeregisterIdleRoutine** function, **EnableIdleRoutine** function, **FtgRegisterIdleRoutine** function

FtMulDw

[New - Windows 95]

Multiplies an unsigned 64-bit integer indicating a time value by an unsigned 32-bit integer in doubleword format.

Syntax

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.

Return Values

Returns a **FILETIME** structure containing the product of the input values.

Comments

The **FtMulDw** function is defined in MAPIUTIL.H.

See Also

FILETIME structure, **FtAddFt** function, **FtMulDwDw** function, **FtNegFt** function

FtMulDwDw

[New - Windows 95]

Multiplies one unsigned 32-bit integer in doubleword format by another.

Syntax

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.

Return Values

Returns a **FILETIME** structure containing the product of the input values.

Comments

The **FtMulDwDw** function is defined in MAPIUTIL.H.

See Also

FILETIME structure, **FtMulDw** function, **FtNegFt** function

FtNegFt

[New - Windows 95]

Computes the two's complement of an unsigned 64-bit integer indicating a time value.

Syntax

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.

Return Values

Returns a **FILETIME** structure containing the two's complement of the input value.

Comments

The **FtNegFt** function is defined in MAPIUTIL.H.

See Also

FILETIME structure, **FtAddFt** function, **FtSubFt** function

FtSubFt

[New - Windows 95]

Subtracts one unsigned 64-bit integer indicating a time value from another.

Syntax

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.

Return Values

Returns a **FILETIME** structure containing the results of the subtraction.

See Also

FILETIME structure, **FtAddFt** function, **FtNegFt** function

GetInstance

[New - Windows 95]

Copies one value within a multivalued property to a single-valued property of the same type.

Syntax

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.

Comments

If the value copied is too large for the allocated memory, the **GetInstance** function only copies pointers rather than allocating new memory.

Comments

The **GetInstance** function is defined in MAPIUTIL.H.

See Also

SPropValue structure

HexFromBin

[New - Windows 95]

Converts a binary number into a string representation of a hexadecimal number.

Syntax

VOID HexFromBin(LPBYTE *pb*, **int** *cb*, **LPTSTR** *sz***)**

Parameters

pb
  Input parameter pointing to the binary number 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 number is stored.

Return Values

Returns TRUE if the function successfully converts the binary number into a string, and FALSE otherwise.

Comments

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, for which you have allocated memory, a representation of this binary information in hexadecimal numbers. If the byte value was 10, for example, the hexadecimal string would be 0A, so one byte becomes two bytes long in the string.

The **HexFromBin** function is defined in MAPIUTIL.H.

See Also

ScBinFromHexBounded function

HrAllocAdviseSink

[New - Windows 95]

Creates an advise sink object, given a context specified by the calling implementation and a callback function to be triggered by an event notification.

Syntax

HrAllocAdviseSink(LPNOTIFCALLBACK *lpfnCallback*, **LPVOID** *lpvContext*, **LPMAPIADVISESINK FAR*** *lppAdviseSink*)

Parameters

lpfnCallback
Input parameter pointing to the callback function (that is, the function based on the **NOTIFCALLBACK** function prototype) 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 variable where the returned advise sink object is stored.

Comments

To use this function, your client 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 a method of your object. If the object generating the notification is not present, the application 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.

The **HrAllocAdviseSink** function is defined in MAPIUTIL.H

See Also

NOTIFCALLBACK function prototype

HrAllocAdviseSink, NOTIFCALLBACK

[New - Windows 95]

The **NOTIFCALLBACK** prototype function defines a client application or service provider callback function that the MAPI calls to send an event notification. This callback function can only be used when wrapped in an advise sink object created by calling **HrAllocAdviseSink**.

Syntax

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, *lpvContext* represents a pointer to the address of a C++ object.

cNotification
Input parameter specifying the number of event notifications in the array indicated by *lpNotifications*.

lpNotifications
Output parameter specifying a pointer to the location to which this function writes an array of **NOTIFICATION** structures containing the event notifications.

Return Values

The set of valid return values depends on whether a client or a provider is implementing the NOTIFCALLBACK function. 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.

Comments

The **NOTIFCALLBACK** function prototype is defined in MAPIDEFS.H.

See Also

HrAllocAdviseSink function, **IMAPIAdviseSink::OnNotify** method, **NOTIFICATION** structure

HrComposeEID

[New - Windows 95]

Creates a compound entry identifier for an object, usually a message in a message store.

Syntax

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 holding 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 *cbStoreRecordKey* parameter value 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.

Comments

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.

This function, primarily for use with CMC, enables CMC-based applications to work with objects in multiple stores through the use of compound entry identifiers.

The HrComposeEID **function is defined in MAPIUTIL.H. See Also**

HrComposeMsgID function, **HrDecomposeEID** function, **HrDecomposeMsgID** function

HrComposeMsgID

[New - Windows 95]

Creates an ASCII entry-identifier string for an object, usually a message in a message store.

Syntax

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 holding 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 holding 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 noncompound entry identifier converted to text.

Comments

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 *cbStoreRecordKey* is zero), the object's entry identifier is simply copied and converted into a string.

The **HrComposeMsgID** function enables 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.

The **HrComposeMsgID** function is defined in MAPIUTIL.H.

See Also

HrComposeEID function, **HrDecomposeEID** function, **HrDecomposeMsgID** function, **HrSzFromEntryID** function

HrDecomposeEID

[New - Windows 95]

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.

Syntax

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 is stored the returned size, in bytes, of the entry identifier of the message store containing the object. 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 *pEID* 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 objec is stored. If *pEID* 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.

Comments

If the identifier identified by *pEID* 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 copied. The compound identifier taken apart is usually one created by the **HrComposeEID** function.

The memory that holds the *pEID* pointer is released upon successful completion of this function. The calling implementation is responsible for freeing memory for the output parameters.

The **HrDecomposeEID** function is defined in MAPIUTIL.H.

See Also

HrComposeEID function, **HrComposeMsgID** function, **HrDecomposeMsgID** function

HrDecomposeMsgID

[New - Windows 95]

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.

Syntax

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 is stored the returned size, in bytes, of the entry identifier string of the message store containing the object. 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 is stored the returned entry identifier of the message store containing the object. If *szMsgID* 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 *szMsgID* 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 pointer to the entry identifier of the object (within its store). If *szMsgID* is a noncompound entry identifier string.

Comments

Noncompound entry-identifier strings are accepted. The compound identifier string taken apart is usually one created by the **HrComposeMsgID** function.

The memory that holds the compound entry-identifier string is released upon successful completion of this function.

The **HrDecomposeMsgEID** function is defined in MAPIUTIL.H.

See Also

HrComposeEID function, **HrComposeMsgID** function, **HrDecomposeEID** function

HrEntryIDFromSz

[New - Windows 95]

Creates an entry identifier from an ASCII-encoded string, and allocates memory for the entry identifier.

Syntax

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 a variable storing the size, in bytes, of the entry identifier pointed to by the *ppentry* parameter.

ppentry
Output parameter pointing to the **ENTRYID** structure to which this function writes the entry identifier.

Comments

The **HrEntryIDFromSz** function is defined in MAPIUTIL.H.

See Also

ENTRYID structure, **HrSzFromEntryID** function

HrGetOneProp

[New - Windows 95]

Retrieves the value of a single property from an **IMAPIProp** interface or an interface derived from **IMAPIProp**.

Syntax

HrGetOneProp(LPMAPIPROP *pmp*, **ULONG** *ulPropTag*, **LPSPropValue FAR*** *ppprop***)**

Parameters

pmp
Input parameter pointing to the interface from which the property value is retrieved.

ulPropTag
Input parameter containing the property tag of the property to be retrieved.

ppprop
Output parameter pointing a variable where the returned **SPropValue** structure defining the retrieved property value is stored.

Return Values

MAPI_E_NOT_FOUND
The requested object does not exist.

Comments

Unlike the method **IMAPIProp::GetProps**, **HrGetOneProp** does not return a warning. Because it retrieves only one property, it simply either succeeds or fails. To retrieve multiple properties, your client application or service provider should call **GetProps**.

The **HrGetOneProp** function is defined in MAPIUTIL.H.

See Also

HrSetOneProp function, **IMAPIProp : IUnknown** interface, **IMAPIProp::GetProps** method, **SPropValue** structure

HrSetOneProp

[New - Windows 95]

Changes one property of an object.

Syntax

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.

Comments

Unlike the method **IMAPIProp::SetProps**, the **HrSetOneProp** function does not return a warning. Because it sets only one property, it simply either succeeds or fails. To change multiple properties, use the faster **SetProps**.

The **HrSetOneProp** function is defined in MAPIUTIL.H.

See Also

HrGetOneProp function, **IMAPIProp : IUnknown** interface, **IMAPIProp::SetProps** method, **SPropValue** structure

HrSzFromEntryID

[New - Windows 95]

Encodes an entry identifier into an ASCII string, and allocates memory for the string.

Syntax

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.

Comments

To use Simple MAPI to look at messages in a folder other than the Inbox folder (the only folder recognized by Simple MAPI), your application must use Extended MAPI methods. It calls these methods to enumerate the messages in the folder, looking through the contents table for the folder, which stores entry identifiers. When the application finds the entry identifier for the desired message, it can call the **HrSzFromEntryID** function to convert the identifier to a Simple MAPI message identifier. Then your application can call the Simple MAPI **MAPIReadMail** function to retrieve the required information about the message.

The **HrSzFromEntryID** function is defined in MAPIUTIL.H.

See Also

ENTRYID structure, **HrComposeEID** function, **HrEntryIDFromSz** function, **MAPIReadMail** function

HrValidateIPMSubtree

[New - Windows 95]

Adds one or more standard interpersonal message (IPM) folders to a message store.

Syntax

HrValidateIPMSubtree(LPMDB *lpMDB*, **ULONG** *ulFlags*, **ULONG FAR*** *lpcValues*, **LPSPropValue FAR*** *lppProps*, **LPMAPIERROR FAR*** *lppMapiError***)**

Parameters

lpMDB
Input parameter pointing to a message store object.

ulFlags
Input parameter containing a bitmask of flags used to indicate options. The following flags can be set:

MAPI_FORCE_CREATE
Requests that a folder be verified before creation, even if message store properties indicate that they are valid. An application would typically set this flag when an error indicates that the folder structure has been damaged.

MAPI_FULL_IPM_TREE
Requests that the following folders be created in addition to the standard ones: Folder and Common Views, and the Inbox, Outbox and Sent Items folders under the IPM-subtree root folder. MAPI_FULL_IPM_TREE is normally set when the message store is the default store.

lpcValues
Output parameter containing the number of values returned in the *lppProps* property value array. The value of the *lpcValues* parameter can be zero if *lppProps* is NULL.

lppProps
Output parameter pointing to an array of **SPropValue** structures indicating values for PR_VALID_FOLDER_MASK and related properties. The IPM Inbox's entry identifier is included, with a special property tag equal to PROP_TAG(PT_BINARY, PROP_ID_NULL). The *lppProps* parameter can be NULL, indicating that the calling implementation is not interested in getting a property value array returned.

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.

Comments

MAPI uses the **HrValidateIPMSubtree** function internally to construct the standard IPM folder tree in a message store when it is first opened, or when it is made the default store.

HrValidateIPMSubtree always creates the IPM subtree root folder, the Deleted Items folder under the IPM subtree root, and the finders folder in the message store root.

.

HrValidateIPMSubtree sets the property PR_VALID_FOLDER_MASK to indicate whether each folder has a valid entry identifier. The following related entry-identifier properties on the message store are set to the entry identifiers of the corresponding folders and returned in *lppProps* 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(PT_BINARY, PROP_ID_NULL) – placeholder for IPM Inbox

The **HrValidateIPMSubtree** function is defined in MAPIUTIL.H.

See Also

IMAPISession::OpenMsgStore method, **MAPIERROR** structure, PR_VALID_FOLDER_MASK property, **SPropValue** structure

LPropCompareProp

[New - Windows 95]

Compares two property values to determine if they are equal.

Syntax

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.

Return Values

This function returns one of the following for most property types:

- < zero if the value indicated by the *lpSPropValueA* parameter is less than that indicated by the *lpSPropValueB* parameter.
- > 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 value is nonzero and consistent across calls.

Comments

Use **LPropCompareProp** only if the types of two properties are the same.

Before calling **LPropCompareProp**, your client application or service provider must first retrieve the properties for comparison with a call to the **IMAPIProp::GetProps** method. When the application 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 values are unequal, this function determines which one is the greater. The properties that **LPropCompareProp** compares do not have to belong to the same object.

The **LPropCompareProp** function is defined in MAPIUTIL.H.

See Also

IMAPIProp::GetProps method, **SPropValue** structure

MAPIDeInitIdle

[New - Windows 95]

Shuts down the DLL for the idle engine. Your 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.

Syntax

VOID MAPIDeInitIdle(*void***)**

Comments

The **MAPIDeInitIdle** function is defined in MAPIUTIL.H.

See Also

MAPIInitIdle, **FNIDLE**

MAPIInitIdle

[New - Windows 95]

Initializes the DLL for the idle engine. A client application or service provider must call **MAPIInitIdle** before calling any other idle engine function.

Syntax

LONG MAPIInitIdle(LPVOID *lpvReserved***)**

Parameter

lpvReserved
    Reserved, must be zero.

Return Values

MAPIInitIdle returns zero if initialization is successful, and -1 otherwise. When the **MAPIInitIdle** function is called multiple times, only the first call succeeds.

Comments

The **MAPIInitIdle** function is defined in MAPIUTIL.H.

See Also

MAPIDeinitIdle, **FNIDLE**

MapStorageSCode

[New - Windows 95]

Maps an HRESULT return value from an OLE storage object to a MAPI return value of the SCODE type.

Warning There is no guarantee that this function will exist in shipped versions of the message dynamic-link library (DLL).

Syntax

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

This function returns an SCODE value corresponding to the value of the *StgSCode* parameter. If this function cannot find a matching value, it returns MAPI_E_CALL_FAILED.

Comments

Extended MAPI provides this function for the internal use of MAPI SDK 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.

The **MapStorageSCode** is defined in IMESSAGE.H.

OpenIMsgSession

[New - Windows 95]

Groups the creation of messages within a session, so that closing the session also closes all messages created within that session.

Syntax

SCODE OpenIMsgSession(LPMALLOC *lpMalloc*, **ULONG** *ulFlags*, **LPMSGSESS FAR*** *lppMsgSess***)**

Parameters

lpMalloc
Input parameter pointing to a standard OLE memory allocator.

ulFlags
Reserved; must be zero.

lppMsgSess
Output parameter pointing to a variable where the returned message-session object is stored.

Comments

To establish a message session, a client application should call the **OpenIMsgSession** function to obtain a message session pointer before calling the **OpenIMsgOnIStg** function for the first time in the session. When finished with the message session, the application should call the **CloseIMsgSession** function to end it.

This function is used by clients that want to deal with several related message on storage objects. If only a single message is to be open at a time, the application does not need to track multiple messages. Thus it has little cause to create a message session and no reason to call **OpenIMsgSession**.

The **OpenIMsgSession** function is defined in IMESSAGE.H.Use of OLE memory allocators is described in *Inside OLE, Second Edition,* by Kraig Brockschmidt, and *OLE Programmer's Reference, Volume One* and *Volume Two*.

See Also

CloseIMsgSession function, **IMAPISession : IUnknown** interface, **OpenIMsgOnIStg** function

PpropFindProp

[New - Windows 95]

Searches for a specified property in a property set.

Syntax

PpropFindProp(LPSPropValue *rgprop*, **ULONG** *cprop*, **ULONG** *ulPropTag***)**

Parameters

rgprop
: Input parameter containing an array of **SPropValue** structures that define the properties to be searched through.

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 *rgprop*.

Return Values

Returns an **SPropValue** structure defining the property that matches the input property tag, and NULL if there is no match.

Comments

If the given property tag indicates a property of type PT_UNSPECIFIED, this 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.

The **PpropFindProp** function is defined in MAPIUTIL.H.

See Also

SPropValue structure

PreprocessMessage

[New - Windows 95]

The **PreprocessMessage** function prototype, implemented by transport providers, preprocesses message contents or the format of a message.

Syntax

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, this parameter points to the folder that contains messages to be preprocessed. On output, this parameter points to the folder where preprocessed messages have been placed.

lpAllocateBuffer
: Input parameters pointing to the **MAPIAllocateBuffer** function, to be used to allocate memory.

lpAllocateMore
: Input parameters pointing to the **MAPIAllocateMore** function, to be used to allocate additional memory where required.

lpFreeBuffer
: Input parameters pointing to the **MAPIFreeBuffer** function, to be used to free memory.

lpcOutbound
: Output parameter pointing to the location to which this function writes the number of messages in the array indicated by the *lppMessage* parameter.

lpppMessage
: Output parameter pointing to the location to which this function writes the address of an array of messages to which this function has added the preprocessed message and any other additional messages it has generated.

lppRecipList
: Output parameter pointing to an optional **ADRLIST** structure listing preprocessor-detected recipients for which the message is undeliverable.

Comments

A transport-provider message preprocessor can present a progress user interface. However, it should never use a dialog box requiring user interaction.

Your preprocessor must take care when adding large amounts of data to an outbound message. This type of message might be stored in a server-based message store, causing the preprocessor to have to access the remote store unnecessarily. The preprocessor should have an option that allows it to store data that takes a large amount of space in a local message store and provide a reference to that local store in the message.

Note that your preprocessor should not release any of the objects originally input to **PreprocessMessage**.

Before the client application can call this function, the transport provider must have registered the function in a call to the **IMAPISupport::RegisterPreprocessor** method. After calling **PreprocessMessage**, the client application cannot continue submitting a message until the function returns.

Before returning, **PreprocessMessage** writes the preprocessed message and any additional messages that it generates to a message array. If the function detects unreachable message recipients during operation, it writes them in a list to the client application.

The **PreprocessMessage** function is defined in MAPISPI.H.

See Also

ADRLIST structure, **IAddrBook : IMAPIProp** interface, **IMAPIFolder : IMAPIContainer** interface, **IMAPISupport : IUnknown** interface, **IMAPISupport::RegisterPreprocessor** method

PropCopyMore

[New - Windows 95]

Copies a single property value from a source location to a destination location.

Syntax

SCODE PropCopyMore(LPSPropValue *lpSPropValueDest*, **LPSPropValue** *lpSPropValueSrc*, **ALLOCATEMORE ****lpfAllocMore*, **LPVOID** *lpvObject***)**

Parameters

o

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.

Comments

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.

The function does not need to allocate memory unless the property value copied is of a type, such as STRING8, that does not fit in an **SPropValue** structure. For these large properties, the function allocates memory using the **MAPIAllocateMore** indicated on input.

Injudicious use of **PropCopyMore** fragments memory; consider using the **ScCopyProps** function instead.

The **PropCopyMore** function is defined in MAPIUTIL.H.

See Also

MAPIAllocateMore function, **ScCopyProps** function, **SPropValue** structure

RemovePreprocessInfo

[New - Windows 95]

Removes from a message preprocessed information written by a function based on the **PreprocessMessage** function prototype.

Syntax

HRESULT RemovePreprocessInfo(LPMESSAGE *lpMessage***)**

Parameters

lpMessage
Input parameter pointing to the preprocessed message from which information is to be removed.

Comments

A client application calls the **RemovePreprocessInfo** function. A transport provider registers **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 **PreprocessMessage**. A client usually calls **RemovePreprocessInfo** after sending a message containing preprocessed information.

The **RemovePreprocessInfo** function is defined in MAPISPI.H.

See Also

IMAPISupport::RegisterPreprocessor method, **PreprocessMessage** function prototype

ScBinFromHexBounded

[New - Windows 95]

Converts the specified portion of a string representation of a hexadecimal number into a binary number.

Syntax

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, **HexFromBin** function

ScCopyNotifications

[New - Windows 95]

Copies a group of event notifications to a single block of memory.

Syntax

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 the location to which this function copies the notifications.

pcb
Output parameter pointing to the size, in bytes, of the array indicated by *rgntf*.

Comments

If *pcb* is NULL, nothing happens; if non-null, the **ScCopyNotifications** function stores the size and bytes of the array. The *pvDst* parameter must be large enough to accommodate the entire array.

The **ScCopyNotifications** function is defined in MAPIUTIL.H.

See Also

NOTIFICATION structure, **ScCountNotifications** function, **ScRelocNotifications** function

ScCopyProps

[New - Windows 95]

Copies the properties defined by an array of **SPropValue** structures to a new destination.

Syntax

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 into 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
Output parameter containing the size, in bytes, of the block of memory indicated by the *pvDst* parameter.

Comments

The new array and its data will 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. Note that prior to this adjustment, the pointers will not be valid.

ScCopyProps maintains the original property order for the copied property array.

The **ScCopyProps** function is defined in MAPIUTIL.H.

See Also

ScDupPropset function, **SPropValue** structure, **ScRelocProps** function

ScCountNotifications

[New - Windows 95]

Determines the size, in bytes, of an array of event notifications, and validates the memory associated with the array.

Syntax

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
Output parameter pointing to a variable where the size, in bytes, is stored of the array pointed to by *rgntf*.

Comments

If NULL is passed in the *pcb* parameter, 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 accommodate the entire array.

The **ScCountNotifications** function is defined in MAPIUTIL.H.

See Also

NOTIFICATION structure, **ScCopyNotifications** function, **ScRelocNotifications** function

ScCountProps

[New - Windows 95]

Determines the size, in bytes, of a property value array, and validates the memory associated with the array.

Syntax

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
Output parameter pointing to a variable where the size, in bytes, is stored of the property array.

Comments

If any property in the property value array has a property identifier of PROP_ID_NULL or PROP_ID_INVALID, the **ScCountProps** function returns MAPI_E_INVALID_PARAMETER. **ScCountProps** also returns this value if the property array contains a multivalued property with no property values.

As it is counting, this function validates the memory associated with the array. **ScCountProps** only works with properties about which MAPI has information.

The **ScCountProps** function is defined in MAPIUTIL.H.

See Also

PropCopyMore function, **SPropValue** structure

ScLocalPathFromUNC

[New - Windows 95]

Locates a local path counterpart to the given UNC path.

Syntax

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.

Comments

The **ScLocalPathFromUNC** function is defined in MAPIUTIL.H.

See Also

ScUNCFromLocalPath function

ScRelocNotifications

[New - Windows 95]

Adjusts a pointer within a specified event notification array.

Syntax

SCODE ScRelocNotifications(int *cntf*, **LPNOTIFICATION** *rgntf*, **LPVOID** *pvBaseOld*, **LPVOID** *pvBaseNew*,**ULONG FAR*** *pcb***)**

Parameters

cntf
Input parameter containing the number of notifications in the array indicated by the *rgntf* parameter.

rgntf
Input parameter pointing to an array of **NOTIFICATION** structures defining event notifications.

pvBaseOld
Input parameter pointing to the original base address of the array indicated by *rgntf*.

pvBaseNew
Input parameter containing the location to which this function writes the new base address of the array indicated by *rgntf*.

pcb
Output parameter pointing to the location to which this function writes the number of bytes needed to contain the array indicated by the *pvBaseNew* parameter.

Comments

The **ScRelocNotifications** is defined in MAPIUTIL.H.

See Also

NOTIFICATION structure, **ScCopyNotifications** function, **ScCountNotifications** function

ScRelocProps

[New - Windows 95]

Adjusts the pointers in a **SPropValue** array after the array and its underlying data have been copied or moved to a new location.

Syntax

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 indicated 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 *rgprop.* For the **ScRelocProps** function to work properly, the array and its underlying data must initially reside together in memory allocated with a single call to the **MAPIAllocateBuffer** function.

pvBaseNew
Input parameter pointing to the new base address of the array indicated by *rgprop*. Note that the array and data must have been moved or copied together into a buffer created with a single **MAPIAllocateBuffer** call**.**

pcb
Output parameter pointing to the number of bytes needed to contain the array indicated by *pvBaseNew*parameter.

Comments

ScRelocProps assumes that the property value array was originally in a single allocation like the one in the **ScCopyProps** function. If using a property value that is built from disjoint blocks of memory, use **ScCopyProps** instead.

To maintain the validity of pointers in an **SPropValue** array when writing it to and reading it from a disk, perform the following:

1. Before writing the array and data to disk, use **ScRelocProps** on the array with *pvBaseNew* pointing to some standard value (zero, for instance).
2. After reading the array and data from a disk, use **ScRelocProps** on the array with the *pvBaseOld* parameter equal to the same standard value used in 1. Note that the array and data must be read into a buffer created with a single allocation.

The **ScRelocProps** function is defined in MAPIUTIL.H.

See Also

MAPIAllocateBuffer function, **ScCopyProps** function, **ScCountProps** function, **ScDupPropset** function, **SPropValue** structure

ScUNCFromLocalPath

[New - Windows 95]

Locates a UNC path counterpart to the given local path.

Syntax

SCODE ScUNCFromLocalPath(**LPSTR** *szLocal*, **LPSTR** *szUNC*, **UINT** *cchUNC*)

Parameters

szLocal
Output parameter containing a DRIVE:PATH name of a file or directory.
szUNC
Input parameter containing a \\SERVER\SHARE\PATH of the same file or directory.
cchUNC
Input parameter containing the size of the buffer for the output string.

Comments

The **ScUNCFromLocalPath** function is defined in MAPIUTIL.H.

See Also

ScLocalPathFromUNC function

UIAddRef

[New - Windows 95]

Provides an alternative way to invoke the method **IUnknown::AddRef**.

Syntax

ULONG UIAddRef(LPVOID *punk***)**

Parameters

punk
Input parameter pointing to an interface derived from the **IUnknown** interface, in other words any MAPI interface.

Return Values

Returns the value returned by **IUnknown::AddRef**, which is the new value of the reference count for the interface. The value is nonzero.

Comments

The **UIAddRef** function generates less code than the **AddRef** method and can be used in situations requiring a minimum of code.

The **UIAddRef** function is defined in MAPIUTIL.H.

See Also

UIRelease function

UlPropSize

[New - Windows 95]

Obtains the size of a single property value.

Syntax

ULONG UlPropSize(LPSPropValue *lpSPropValue***)**

Parameters

lpSPropValue
Input parameter pointing to an **SPropValue** structure defining the property to be measured.

Return Values

Returns the size, in bytes, of the property value for the specified property. It disregards the size of the remainder of the **SPropValue** structure.

Comments

The **UlPropSize** function is defined in MAPIUTIL.H.

See Also

SPropValue structure

UlRelease

[New - Windows 95]

Provides an alternative way to invoke the OLE method **IUnknown::Release**.

Syntax

ULONG UlRelease(LPVOID *punk***)**

Parameters

punk
Input parameter pointing to an interface derived from the **IUnknown** interface, in other words any MAPI interface.

Return Values

This function returns the value returned by **IUnknown::Release**.

Comments

The return value can be equal to the reference count for the object to be released. The reference count is the number of existing pointers to the object.

If the *punk* parameter specifies NULL, the function returns immediately without calling **IUnknown::Release**. This feature can save code if your 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.

The **UlRelease** function is defined in MAPIUTIL.H.

See Also

UlAddRef function

MAPISVC.INF File Format and WIN.INI File Format

This chapter contains information about the MAPISVC.INF file format and the WIN.INI file format.

MAPISVC.INF File Format

The MAPISVC.INF file acts as the central database for MAPI message service configuration information.MAPISVC.INF 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, 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. For example, some service providers require a password to be supplied at logon time. Rather than forcing the user to enter the password with every logon, a service provider might store the password in the profile

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 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.

Figure 8.1, following, illustrates 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 start up 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, group10896 0 /a "SDKEX.bmp"}

All MAPISVC.INF files will 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 may be more or less service provider sections and possibly some special sections. Each of these types of sections in described in more detail below.

[Help File Mappings] Section

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
```

[Services] Section

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].

[Default Services] Section

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 in Figure 8.1:

```
[Default Services]
AB=Default Address Book
MsgService=My Own Service
```

Message Service Sections

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 configuration user interface or property sheet under the control of a client application.

The information provided for the PR_SERVICE_DLL_NAME entry is the name of the dynamic-link library (DLL) that contains the message service. The information provided for the PR_SERVICE_ENTRY_POINT entry is 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*

Section List Entries

There are two types of section list entries: one that service provider sections for each 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 illustration in Figure 8.1, 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

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 contains 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. For example, the PR_PROVIDER_DISPLAY property is an optional property that service providers include if they have two different display names.

Typical service provider sections include the following entries, all of which are required except for PR_PROVIDER_DISPLAY:

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. 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
```

WIN.INI File Format

The Windows initialization file, WIN.INI, contains information about programs installed on a computer and options about their use. MAPI specifies a few entries that might need to be added to WIN.INI by the program that you write for installing your message service, depending on the computer's MAPI installation. These entries indicate the existence of various MAPI components and provide information about their configuration. Because there are default settings for each WIN.INI entry, your setup program can exclude those entries for which the default setting is appropriate.

[Mail] Section

The entries to be added to this section specify the types of MAPI client application programming interfaces (APIs) that are installed on a computer. These types include Simple MAPI, Extended MAPI, OLE Messaging, and CMC. If CMC is installed, another entry exists to identify the name of the CMC DLL file that should be loaded for your platform.

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 is set to 0. The MAPI entry identifies a Simple MAPI installation, the MAPIX entry identifies an Extended MAPI installation, and the OLEMessaging 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 looks 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, Extended MAPI, and OLE Messaging installations always reside in a standard DLL file. For Extended 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 (Windows 95 and Windows NT operating systems) if there is a Simple MAPI installation available for at least one of the platforms.

If OLE Messaging 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 to know 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 Extended MAPI. Client applications can load the Extended 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 Extended 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.

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

[MAPI] Section

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. You can specify your versions of these components in the [MAPI] section of the WIN.INI file. 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 Extended 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 machine rather than locally, allowing them to be accessed from anywhere in the network.

[MAPI 1.0 Time Zone] Section

The [MAPI 1.0 Time Zone] section stores information that is used by MAPI's 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 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;
```

The TIME_ZONE_INFORMATION structure specifies information specific to the time zone.

Members

Bias

Specifies the current bias, in minutes, for local time translation on this computer. The bias is the difference, in minutes, between Coordinated Universal Time (UTC) and local time. All translations between UTC and local time are based on the following formula:

UTC = local time + bias

This member is required.

StandardName

Specifies a null-terminated string associated with standard time on this operating system. For example, this parameter could contain "EST" to indicate Eastern Standard Time. This string is not used by the operating system, so anything stored there by using the SetTimeZoneInformation function is returned unchanged by the GetTimeZoneInformation function. This string can be empty.

StandardDate

Specifies a SYSTEMTIME structure that contains a date and UTC when the transition from daylight time to standard time occurs on this operating system. If this date is not specified, the wMonth member in the SYSTEMTIME structure must be zero. If this date is specified, the DaylightDate value in the TIME_ZONE_INFORMATION structure must also be specified. Local time translations done during the standard-time range are relative to the supplied StandardBias value.

This member supports two date formats. Absolute format specifies an exact date and time when standard time begins. In this form, the wYear, wMonth, wDay, wHour, wMinute, wSecond, and wMilliseconds members of the SYSTEMTIME structure are used to specify an exact date.

Day-in-month format is specified by setting the wYear member to zero, setting the wDayOfWeek member to an appropriate weekday, and using a wDay value in the range 1 through 5 to select the correct day in the month. Using this notation, the first Sunday in April can be specified, as can the last Thursday in October (5 is equal to "the last").

StandardBias

Specifies a bias value to be used during local time translations that occur during standard time. This member is ignored if a value for the StandardDate member is not supplied.

This value is added to the value of the Bias member to form the bias used during standard time. In most time zones, the value of this member is zero.

DaylightName

Specifies a null-terminated string associated with daylight time on this operating system. For example, this parameter could contain "PDT" to indicate Pacific Daylight Time. This string is not used by the operating system, so anything stored there by using the SetTimeZoneInformation function is returned unchanged by the GetTimeZoneInformation function. This string can be empty.

DaylightDate

Specifies a SYSTEMTIME structure that contains a date and UTC when the transition from standard time to daylight time occurs on this operating system. If this date is not specified, the wMonth member in the SYSTEMTIME structure must be zero. If this date is specified, the StandardDate value in the TIME_ZONE_INFORMATION structure must also be specified. Local time translations during the daylight-time range are relative to the supplied DaylightBias value. This member supports the absolute and day-in-month time formats described for the StandardDate member.

DaylightBias

Specifies a bias value to be used during local time translations that occur during daylight time. This member is ignored if a value for the DaylightDate member is not supplied.

This value is added to the value of the Bias member to form the bias used during daylight time. In most time zones, the value of this member is -60.

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 body. 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 body. 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

[New - Windows 95]

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

[New - Windows 95]

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

[New - Windows 95]

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

[New - Windows 95]

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

[New - Windows 95]

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

[New - Windows 95]

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

[New - Windows 95]

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

[New - Windows 95]

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

[New - Windows 95]

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

[New - Windows 95]

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

[New - Windows 95]

The **RENDDATA** structure describes how and where the attachment is rendered in the message body.

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.*

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 the rest of this chapter.

The [Description] Section

The [Description] section lists all attributes of the form visible in client application'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 registry. The remaining entries are optional. The format of the [Description] section is shown here.

[Description]

MessageClass = *string*

Clsid = *guid*

DisplayName = *displayed string*

;optional entries

Category = *displayed string*

Subcategory = *displayed string*

Comment = *displayed string*

Owner = *displayed string*

Number = *displayed string*

SmallIcon = *path*

LargeIcon = *path*

Version = *integer*

Locale = *string*

Hidden = *integer*

DesignerToolName = *string*

DesignerToolGuid = *clsid*

DesignerRuntimeGuid = *clsid*

ComposeInFolder = *integer*

ComposeCommand = *string*

The Category and Subcategory entries are used by forms 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 subcategory. This categorization can then be used by viewer applications to display folder hierarchies. 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 variants of the same form version. 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 "Comment 1" as the key, and so on.

The SmallIcon and LargeIcon 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 wherethe configuration file is installed. The Version entry is used to indicate the version number of the form. Locale is the 3-letter language identifier of the destination registry. The Hidden entry indicates whether the form is displayed in registry 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.

```
[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

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 or call the **IMAPIProp::GetProps** method and requesting all form 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 conntaining named strings are not supported. The format of the [Extension] section is shown here.

[Extension*string***]**

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

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 the string code of the platform. Each string corresponds to the CPU entry of an individual [Platforms] sections. Each entry in a [Platforms] section 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. Additionally, either a File entry or a LinkTo entry must be specified. The File entry lists the form server application executable file that the forms registry maintains and loads into a new subdirectory in the disk cache when the form is launched. Instead of a File entry, a platform can have a LinkTo entry. The LinkTo entry contains a single key that names the platform hosting the files.

The Registry entry is used whenever the File entry is used, it identifies the registry key for the registry where the executable file for the form server application is stored. Strings preceded by a backslash (\ ) are placed at the root of the \\HKEY_CLASSES_ROOT\ registry key. Strings not preceded by a backslash are placed at the root of 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 containing the downloaded form files. 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 shown here.

[Platform.*platform string***]**

CPU = *string*

OSVersion = *string*

File = *path*

LinkTo = *string*

Registry = *string*

Following are shown 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 forms registry, 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.

The [Properties] Section

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*

Following is an example of a [Properties] section.

```
[Properties]
Property1 = Fire Hazard
Property2 = Safe
```

Each [**Property.***string*] section describes a single property. The Type entry specifies the MAPI property type, for example 3(PT_14), of the property. The NmidPropset entry is optional; together with either the NmidString entry or the NmidInteger entry, which are mutually exclusive, the NmidPropset entry gives the name of the property. 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 NmidPropset 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.

The format of a [**Property.***string*] section is shown here.

[Property.*string***]**

Type = *integer*

NmidPropset = *guid*

NmidString = *string*

NmidInteger = *integer*

DisplayName = *string*

Flags = integer

SpecialType = 0|1

Enum1 = *string*

Following is an example of a [**Property.***string*] section.

```
[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. Following is the format for the [**Enum1.***string*] section.

[**Enum1.***string*]

NmidPropset = *guid*

NmidString = *string*

NmidInteger = *integer*

EnumCount = *integer*

Val.*integer***.Display** = *string*

Val.*integer***.Index** = *integer*

Here 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, soprting could be done based on Low-Medium-High order rather than High-Medium-Low order.

The [Verbs] Section

The [Verbs] section lists the complete set of verbs supported by the form. The format of the [Verbs] section is shown following.

[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 shown here.

[**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 and then passed to the **IMAPIForm::DoVerb** method.

Glossary

A

address book
A collection of one or more recipients. Recipients are messaging users or distribution lists.

address book container
A MAPI address book object that holds recipients, such as distribution lists and messaging users, and implements the **IABContainer** interface.

address book provider
A service provider that manages one or more address book containers, enabling users to address messages and create recipients.

advise sink
A MAPI object that is told of changes affecting other MAPI objects. Advise sinks register with specific objects to be told of specific types of changes. Advise sinks implement the **IMAPIAdviseSink** interface.

ambiguous name resolution
See name resolution.

application registry
A MAPI folder object that an application designates to store a form-server executable file, a form configuration file, and all incoming messages for one or more message classes. An application registry implements the **IMAPIFormContainer** interface and is a form registry that is specific to a particular application. *See also* form registry, personal registry.

associated information
Hidden data that relates to a standard folder and enables providers to order information in that folder; examples of such data include a view and the definition of a form.

attachment
A MAPI object that contains data appended to a message and that implements the **IAttach** interface. Most attachments are either files, OLE objects, other messages, or binary data such as sound or images.

attachment table
A MAPI table object that provides access to information about a message's attachments.

B

bookmark
A marker that identifies a position within a table.

C

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 Document Integration (EDI) data standard. Now called the International Telecommunications Union (ITU). The acronym stands for Comite Consultatif International Telegraphique et Telephonique, the committee's French name. *See also* Electronic Document Integration, 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. These functions and methods are known collectively as the client interface. *See also* client interface, messaging application.

client extension
A program component that is used with a client application to add to the application's feature set, for example to add a handler for custom commands.

client interface
The set of interfaces and functions that are implemented or used by client applications. MAPI's client interface has three components: Common Messaging Calls (CMC), Simple MAPI, and Extended MAPI.

CMC
See Common Messaging Calls.

common dialog box
A Microsoft Windows dialog box that is provided with the MAPI SDK. Client applications use common dialog boxes to promote an appearance both consistent within the application and consistent with other Windows-based applications.

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 part of the MAPI subsystem's client application interface. CMC is a cross-platform function set that enables client applications to be independent of the actual messaging system, operating system, or hardware used. *See also* Extended MAPI, Simple MAPI.

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 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 Simple MAPI client applications to open messages from a nondefault message store provider. *See also* entry

identifier.

configuration file
A formatted file created by form developers to define a form. Because configuration files are used by form servers to load forms, each form must be defined using a configuration file. Configuration files must have the .CFG filename extension.

connection number
A number that uniquely identifies a notification registration.

container
A MAPI object that holds one or more other MAPI objects and that implements the **IMAPIContainer** interface. Address books and folders are two types of containers. *See also* address book container, folder, form container.

contents table
A MAPI table object that provides access to what is contained within another MAPI object. Contents tables are used by message store providers, for example, to display information about messages within a folder.

conversation key
An identifier that, applied to a message, identifies it as one of a series of messages about a particular topic and that enables messages to be sorted by topic. Like other keys, the conversation key is a binary reference or tag that can be compared with other binary references or tags. How messages are sorted using conversation keys is determined by a client application and is not affected by user actions. The message property that holds the conversation key is PR_CONVERSATION_KEY.

conversation thread
A collection of messages that pertain to the same topic.

custom recipient
An address book entry that represents a recipient that is not in the current address book. A custom recipient is created by the user of a client application with a template supplied by the address book provider. Also referred to as a one-off address.

custom recipient table
A MAPI table object that provides access to information about supported custom recipients.

D

data extension
See Common Messaging Calls data extension.

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. The default profile is stored under the default profile key in the MAPISVC.INF file.

delegate access
The ability of one user to allow another user to send messages as the first user and to access the first user's message store.

Deleted Items folder
A standard folder within the interpersonal message (IPM) subtree that is usually designated to hold items, such as messages, that are marked for deletion. *See also* IPM subtree.

delivery report
An announcement from the MAPI spooler or a transport provider to a message sender that indicates a recipient has received a message. The recipient might or might not have read the message.

disk instance
A temporary file that contains executable files for a form if those executable files are not yet on the user's local disk.

display name
A character string that represents a MAPI object in the user interface. 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 has data about one property of the control, such as its location, size, or type.

distribution list
A MAPI address book object that represents a set of individual recipients addressed together as a group and that implements the **IDistList** interface. Distribution lists can contain other distribution lists and individual recipients. A distribution list is one of the two types of MAPI address book objects that can receive messages; the other is a mail user object. *See also* mail user object, recipient.

DR
See delivery report.

E

Electronic Document Integration
(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. *See also* X.435.

entry identifier
A unique piece of data that distinguishes a MAPI object from all other objects of the same type. 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, long-term entry identifier, short-term entry identifier.

event
A change in an object's state that can generate a notification, such as a critical error occurring or a table object being modified. *See also* notification.

event mask
A bitmask that is used to indicate one or more types of notifications. Advise sink objects register for specific notifications using the event mask.

Extended MAPI
A set of functions and object-oriented interfaces that forms the foundation for the MAPI subsystem's client and service provider interfaces. Extended MAPI is intended for complex messaging-based applications, such as advanced workgroup programs, and for service providers. *See also* Common Messaging Calls, Simple MAPI.

extension
See client extension, Common Messaging Calls data extension.

F

flag
A constant that indicates a particular option for an operation. A flag is used either alone or with other flags to set a bitmask parameter, structure member, or property. When multiple flags must be set, use the logical-OR operator.

folder
A MAPI object in a message store that can contain other objects, such as messages and other folders. Also referred to as a standard folder. *See also* IPM subtree, receive folder, root folder, search-results folder.

foreign system
An X.400 messaging term indicating a messaging system outside the realm of the local X.400 network.

form
A MAPI object that is used to display a message with a particular message class in a structured format based on MAPI properties. User interaction with the form is controlled with a form server application. MAPI form objects implement the **IMAPIForm** and **IMAPIFormInfo** interfaces.

form activation
The process of starting a form server so as to enable a user to work with a particular form within a client application.

form container
A MAPI folder object that an application designates to store a form-server executable file, a form configuration file, and all incoming messages for one or more message classes. A form container implements the **IMAPIFormContainer** interface. Also referred to as an application registry. *See also* application registry, form registry.

form registry
A MAPI folder object that contains information about a form, such as the form's executable file, its configuration file, and all incoming messages for one or more message classes. A form registry can be any folder into which a form is installed. *See also* application registry, personal registry.

form registry provider
A service provider that manages form registries, that implements the **IMAPIFormMgr** interface, and that like most service providers is a dynamic-link library (DLL).

form resolution
The process of mapping the message classes of particular messages to the class identifiers of the form servers for those messages. 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. *See also* message class handler.

form viewer
A type of client application that can display a MAPI form.

fuzzy level
A value that describes the degree of exactness or looseness desired when searching for a target word in a text selection. Lower levels of fuzziness indicate more exact matching; higher levels return matches for more varied forms of the target word and usually require more execution time.

G

gateway
Software that links two or more dissimilar messaging systems (that is, two or more messaging systems that use different protocols). A gateway provides protocol and storage conversion.

H

hands-off state
A condition in which a form's storage object is unavailable for either read-only or read-write access. Forms are placed in the hands-off state during a save operation to enable the save operation to occur uninterrupted. *See also* normal state, no-scribble state.

hierarchy table
A MAPI table object that provides hierarchical access to information within a MAPI container. For example, a hierarchy table is used by message store providers to display the folder hierarchy.

hook provider
See message hook provider.

I

IID
See interface identifier.

Inbox
A standard folder within the interpersonal message (IPM) subtree that is usually designated as the destination for incoming messages. *See also* IPM subtree, receive folder.

information service
See message service.

information store
The default message store provider for the Microsoft Exchange Server.

input parameter
A structure, pointer, or variable that is allocated, initialized, and freed by a process that calls a public function or interface method. *See also* input-output parameter, output parameter.

input-output parameter
A structure, pointer, or variable that is initially allocated, initialized, and eventually freed by a process that calls a public function or interface method. Special memory allocation rules apply to input-output parameters. Input-output parameters are typically used to reset a value specified by the calling process. *See also* input parameter, output parameter.

interface identifier
(IID) A constant that represents a particular interface and is used to request a pointer to the interface in a variety of method calls. For example, the IID for the **IAddrBook** interface is IID_IADDRBOOK.

International Telecommunications Union
(ITU) *See* CCITT.

interpersonal message
(IPM) A message that belongs to a message class for which messages are meant to be read by a user rather than by an application or process.

IPM
See interpersonal message.

IPM subtree
(interpersonal message subtree) A set of standard folders that is typically created by MAPI to handle incoming, outgoing, deleted, and sent messages. The default names for these folders are the Inbox, Outbox, Sent Items, and Deleted Items folders.

K

key
A binary reference or tag that can be compared with another binary reference or tag. *See also* conversation key, record 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 the MAPI subsystem. Logon involves the verification of the user's name and password and the user's selection of a valid profile.

logoff
The process by which a messaging user ends a session with the MAPI subsystem.

long-term entry identifier
An entry identifier that is stored for later use and that can be used for multiple operations. *See also* short-term entry identifier.

M

mail user object
A MAPI address book container object that represents an individual messaging user and that implements the **IMailUser** interface. A mail user object is one of two types of MAPI address book objects that can receive messages; the other is a distribution list object. *See also* distribution list, recipient.

MAPI
(Messaging Application Programming Interface) A messaging architecture that enables multiple applications to interact with multiple messaging systems across a variety of hardware platforms. *See also* Extended MAPI, MAPI subsystem, Simple MAPI.

MAPI interface
A set of related functions that describe the behavior of an object in the MAPI environment. All MAPI interfaces derive from the OLE base interface, **IUnknown**.

MAPI object
An object that supports the OLE component object model, implementing one or more interfaces derived from the **IUnknown** interface.

MAPI spooler
A MAPI process that queues messages until ready to send and handles the sending of messages between client applications and messaging systems.

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 several support objects.

message
A MAPI message store object that is used for sending data and receiving data by means of a messaging system and that implements the **IMessage** interface.

message class
A string property that is assigned to each MAPI message, identifying the type of message.

message class handler
A MAPI object that facilitates limited interaction between a form and its form viewer and that can be implemented as either a dynamic link library (DLL) or an executable file. A form server enables more robust form interaction than a message handler does. *See also* form server.

message hook provider
A service provider that operates on either outbound messages or inbound messages. Outbound hook providers process messages before the MAPI spooler routes them to the transport provider. Inbound hook providers are used to intercept messages and reroute them. For example, a message hook provider might move messages from the default Sent Items folder to an alternate folder. Message hook providers implement the **ISpoolerHook** interface. Also referred to as a spooler hook or a hook provider. *See also* message postprocessor, message preprocessor.

message postprocessor

A transport provider component that operates on messages immediately after they have been sent by the MAPI spooler. For example, a message postprocessor might delete the local copy of a message made by a message preprocessor. *See also* message hook provider, message preprocessor.

message preprocessor
A transport provider component that operates on messages at some time before they are sent. For example, a message preprocessor might make a local copy of every outgoing message. *See also* message hook provider, message postprocessor.

message service
A group of one or more related service providers. Defining a message service simplifies installation and configuration. Rather than working with individual service providers, client applications work with the message service. The message service has information about each of its service providers.

message service table
A MAPI table object that provides access to information about currently loaded message services, such as a service's name and associated files.

message site
A MAPI object that provides an area for form display and that implements the **IMAPIMessageSite** interface.

message store
A MAPI object that contains messages and folders organized hierarchically and that implements the **IMsgStore** interface.

message store provider
A service provider that manages a message store, handling message distribution, organization, and storage.

message stores table
A MAPI table object that provides access to information about currently loaded message stores.

messaging application
A program that uses the MAPI subsystem's client application 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. *See also* client application.

messaging service
See message service.

messaging system
A product that enables electronic communication over a network, such as sending and receiving information.

messaging transport provider
See transport provider.

multivalued property

A property for which the data structure's property type contains the flag MV_FLAG and the property value contains multiple values of the specified property type. *See also* property value, single-valued property.

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 resolved before the message can be sent. *See also* resolved recipient, unresolved recipient.

NDR
See nondelivery report.

no-scribble state
A condition in which a form's storage object is accessible only for reading; read-write access is prohibited. *See also* hands-off state, normal state.

non-IPM message
(noninterpersonal message) A message that is meant to be read by an application rather than by a human user, such as a notification message sent from a workgroup scheduling application.

nondelivery report
An announcement to a message sender that indicates a message could not be delivered to a recipient. Many situations can cause nondelivery reports to be generated, including when the message address is inaccurate, when the message address cannot be reached using the available transport providers, and when the network is not working.

nonread notification
An announcement to a message sender that indicates a message was not read by the recipient (that is, the application did not display the message contents to the recipient) before the recipient deleted the message or before the specified expiration time. Nonread notifications for a message are generated by a message store provider when the provider first processes the message.

nonread report
See nonread notification.

nontransmittable property
An X.400 property that is not sent with the message of which it is a part. Nontransmittable properties are typically those properties that only apply to a message before it has been sent. *See also* transmittable property.

normal state
A condition in which a form's storage object is accessible for both reading and writing. Form storage is typically in the normal state unless a save operation is occurring. *See also* hands-off state, no-scribble state.

notification
A pronouncement of change to a MAPI object. The object receiving the notification is referred to as the advise sink. Advise sinks register for a type of notification from a specific object by calling the object's **Advise** method. *See also* advise sink.

NRN
See nonread notification.

O

OLE Messaging
A programming interface, used primarily by Visual Basic and Visual C and C messaging client developers. OLE Messaging provides programmable objects that, like Microsoft Excel objects and Microsoft Access objects, make available properties and methods that can then be managed by Visual Basic and Visual Basic for Applications programs.

OLE Messaging collection
An object that contains zero or more OLE Messaging objects of the same type. OLE Messaging 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 exposed by OLE Messaging, such as the Session object, Message object, or Folder object.

one-off address
See custom recipient.

Outbox
A standard folder within the interpersonal message (IPM) subtree that is usually designated as the holding place for outgoing messages before they are sent. *See also* IPM subtree.

output parameter
A structure, pointer, or variable that is allocated, initialized, and freed by a process that is called by a public function or interface method and used to send information back to the calling process. Special memory allocation rules apply to output parameters. *See also* input parameter, input-output parameter.

P

PAB
See personal address book.

personal address book
A MAPI container object that holds recipient entries either created by the user or copied from other address book containers. Personal address book objects commonly are contained in files that have the filename extension .PAB.

personal message store
A MAPI message-store object that is created by a user and is stored in a file that has the .PST filename extension.

personal registry
A MAPI folder object that a user designates to store local form server executable and configuration files. A personal registry is a type of form registry. *See also* application registry, form registry.

primary identity
A string that represents the user of a MAPI session. The primary identity of the user sending a message is placed in that message's PR_RECIPIENT_FROM property when the message is sent.

probe
An X.400 messaging term indicating a message that has a specific message-class identifier but no content. Probes are used by X.400 message senders to determine whether a message has been sent to recipients and, if it has been sent, how.

profile
Configuration information about a session's set of message services. Profiles are created from information stored in the MAPI configuration file, MAPISVC.INF, and can be modified by service providers or by the MAPI configuration application.

profile name
A text string that identifies a particular profile.

profile section
A portion of a profile that is represented by a unique identifier. A profile section can contain information provided or used by MAPI, by a particular message service, or by a particular service provider.

profile table
A MAPI table object that provides access to the names and status of profiles on a particular computer.

property
A MAPI object that describes an attribute of another MAPI object and that implements the **IMAPIProp** interface. The information that defines a MAPI property is contained in a structure that holds a unique identifier, a data type, and a value. *See also* property tag, property value.

property identifier

A unique 16-bit number that identifies a particular property. The property identifier is contained in the high-order 16 bits of the property tag. *See also* property, property tag, property value.

property page
One section of a property sheet. *See also* property sheet.

property set
A group of user-defined properties.

property sheet
A 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. *See also* property page.

property tag
A 32-bit unsigned integer within a property structure that contains the property's unique identifier and data type. The high-order 16 bits contain the unique identifier, and the low-order 16 bits contain the type. *See also* property, property identifier, property type, property value.

property type
A constant that describes the data type for a property and represents a single value or multiple values. The property type is contained in the low-order 16 bits of the property tag. *See also* property, property tag, property value.

property value
The actual data of a property, a part of that property's defining structure. Property values can be a single piece of data or multiple pieces of data. *See also* multivalued property, property, property tag, single-valued property.

provider
See service provider.

provider table
A MAPI table object that provides access to information about currently loaded service providers.

proxy address
An address that is used by directories from different X.400 messaging systems to refer to MAPI objects, such as distribution lists or faxes.

PST
See personal message store.

R

read flag
A constant, MSGFLAG_READ, that is used to mark a message as having been read. The read flag for a message can be set in a variety of situations; its being set does not necessarily indicate the intended recipient has read the message.

read notification
A announcement to a message sender that indicates the read flag has been set for a message. A read notification can be sent in a variety of situations; its being sent does not necessarily indicate the intended recipient has read the message. Read notifications are generated by a message store provider when a message is first processed.

read receipt
See read notification.

read report
See read notification.

receive folder
A MAPI folder object that is designated as the destination for a particular set of incoming messages. Typically, the receive folder is the Inbox in the interpersonal message (IPM) subtree. *See also* Inbox, IPM subtree.

recipient
The user who receives a particular message. Recipients are individual messaging users, implemented as MAPI mail user objects, or collections of messaging users, implemented as MAPI distribution list objects. *See also* custom recipient, distribution list, mail user object.

recipient list
A collection of zero or more individual messaging users, groups of messaging users, or a combination of both types of users that are selected to receive a message.

recipient table
A MAPI table object that provides access to information about a message's recipient list.

recipient box
An editable field in the MAPI common dialog box for addressing into which names of recipients for a message being composed are placed. There must be at least one recipient box in the addressing dialog box for recipients that receive the message directly. There can be one or two other recipient boxes if either carbon copy (CC) or blind carbon-copy (BCC) recipients are supported.

record key
A unique identifier that is a binary reference or tag that can be compared with other binary reference or tags. Objects assign record keys by setting their PR_RECORD_KEY property.

registration
A request on the part of a client application or MAPI to receive notifications about a specific type of event. *See also* notification.

resolution
See 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 test conducted against a table to filter the rows of the table, limiting a user's view of data to only those rows that passed the test.

RN
See read notification.

root folder
The MAPI folder object that appears at the top of the 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. A root folder can contain messages and other folders.

rule
An operation implemented by a message hook provider. *See also* message hook provider.

S

search-results folder
A MAPI folder object that contains a list of links to messages that match specified search criteria. A search-results folder is invisible to the user and is valid only for the duration of the current session. It cannot contain other folders or messages, and other folders or messages cannot be created in or moved into it.

section
A portion of either a session profile or a form configuration file. *See also* configuration file, profile section.

secure property
A property that has a property tag in the range 0x67F0 through 0x67FF and is encrypted and invisible. 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 standard folder within the interpersonal message (IPM) subtree that is usually designated to hold copies of outgoing messages. 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 dynamic-link library (DLL) that links client applications with services supplied by messaging systems. A service provider typically offers address book, form management, message store, message preprocessing, message postprocessing, or transport services. *See also* address book provider, form registry provider, message hook provider, message postprocessor, message preprocessor, 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. The session identifies the available messaging operations and the service providers available to handle the operations.

session handle
A handle to a session initiated by a Simple MAPI client application. Session handles are returned by the **MAPILogon** function at logon time.

session identifier
A pointer to an Extended MAPI session object that implements the **IMAPISession** interface. Session identifiers are returned by the **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 that is used for a single operation only and that is not stored for later use. *See*

also 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 part of the MAPI subsystem's client interface and that enables basic messaging features to be added to client applications. *See also* Common Messaging Calls, Extended MAPI.

single-valued property
A property for which the data structure contains one property value of the specified property type. *See also* multivalued property, property value.

SPI
See service provider interface.

spooler
See MAPI spooler.

spooler hook
See message hook provider.

status table
A MAPI table object that provides access to information about each of the service providers in the active profile, describing 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.

T

table
A MAPI object that provides access to data in row and column format and that supports the **IMAPITable** interface. MAPI table columns always refer to specific properties of a row. The values of table rows and columns vary depending on the type of table. *See also* attachment table, contents table, custom recipient table, display table, hierarchy table, message service table, message stores table, profile table, provider table, recipient 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 dialog box that is used for creating custom recipients.

template identifier
An entry identifier for a recipient that is managed by an address book provider that is not part of the current profile.

tightly coupled
Describes a relationship between a message store provider and transport provider that enables messages to be passed directly between the two, bypassing the MAPI spooler.

TNEF
See Transport-Neutral Encapsulation Format.

transmittable property
An X.400 property that is sent with the message of which it is a part. An attachment is an example of a transmittable message properties. *See also* nontransmittable property.

Transport-Neutral Encapsulation Format
(TNEF) A standard method defined by MAPI of passing message text and binary data. The method involves wrapping unsupported MAPI properties in a binary attachment file that accompanies a message through the transport process and decoding the file on the receiving side to reconstitute the MAPI properties.

transport provider
A service provider that carries messages, resolving MAPI addresses to the format used by the messaging system that delivers the messages.

U

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

view
A MAPI table object that presents a particular way of looking at data in a table, such as a subset of data based on a particular sort order. Some applications enable views to be saved.

view context
A MAPI object implemented by a client application that can act as a form viewer. The view context object is used to handle form commands such as Next, Previous, and Delete.

W

wastebasket
See Deleted Items folder.

Windows Messaging System
(WMS) The MAPI subsystem that is part of the Microsoft Windows 95 operating system.

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 X.400 Application Programming Interface Association (XAPIA). 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 Document Integration (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 X.400 communications.

Extended MAPI Return Values

Throughout the descriptions for Extended MAPI methods and functions in *MAPI Programmer's Reference, Volume 1,* and *MAPI Programmer's Reference, Volume 2,* only those return values that client applications and service providers can handle are documented. For example, MAPI_E_NOT_FOUND is documented as the result of a nonexistent profile being named when a session is opened; in this case, error handling can include displaying a dialog box requesting that the user either provide the name of an existing profile or create a new profile.

Extended MAPI methods and functions can also return values representing error conditions that cause method and function calls to fail and that cannot be handled in such a fashion. These values are documented in this appendix. For additional information about the MAPI return value philosophy, see *MAPI Programmer's Guide*.

Values Indicating Parameter Validation Errors

Client applications and service providers should use the parameter validation functions provided by MAPI to validate parameters. For more information on how to use the MAPI parameter validation model in your programs, see the *MAPI Programmer's Guide*. The following error values can be returned by methods and functions using MAPI parameter validation:

MAPI_E_INVALID_PARAMETER
: One or more parameters passed by the calling application were not valid.

MAPI_E_UNKNOWN_FLAGS
: Reserved or unsupported flags were used; therefore, the operation did not complete.

Values Indicating Resource-Related Errors

The following error values can be returned to indicate the failure of a disk, of memory, or of system resources:

MAPI_E_DISK_ERROR
A disk error prevented successful completion of the operation.

MAPI_E_NOT_ENOUGH_DISK
Insufficient disk space was available to complete the operation.

MAPI_E_NOT_ENOUGH_MEMORY
Insufficient memory was available to complete the operation.

MAPI_E_NOT_ENOUGH_RESOURCES
Insufficient system resources were available to complete the operation.

Values Indicating Unicode Errors

The following error value can be returned by all MAPI string methods and functions:

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 supports only Unicode.

Unicode implementation is optional when developing programs with MAPI. However, many MAPI methods and functions enable client applications and service providers to specify whether text strings should be in Unicode format; they do so by taking a *ulFlags* parameter and allowing calling processes to pass MAPI_UNICODE in it to indicate passed-in or returned strings, or both, should be Unicode. Implementations that do not support Unicode, or that only support Unicode, return MAPI_E_BAD_CHARWIDTH when the presence or absence of MAPI_UNICODE does not match the implementation's requirements. For example, message class names cannot be in Unicode format, so passing MAPI_UNICODE to a function that also takes a message class name as a parameter always results in an error.

Values Indicating Unknown Error Conditions

If an error occurs for which an implementation cannot determine the origin, the following error value can be returned:

MAPI_E_CALL_FAILED
An error of unexpected or unknown origin prevented the operation from completing.

Extended MAPI Versions of 32-Bit Windows Functions

This appendix documents functions from the Win32 application programming interface (API) useful to 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 such functions have limitations relative to their Win32 SDK counterparts. Such limitations are noted section 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.

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 Extended 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.*

In general, error codes returned from these functions or from **GetLastError** come from DOS and may not always match the Win32 counterpart.

CloseHandle

Syntax

BOOL CloseHandle(**HANDLE** *hObject*)

Limitations

This function will close only file handles.

CompareFileTime

Syntax

LONG CompareFileTime(CONST FILETIME * *lpft1*, **CONST FILETIME *** *lpft2***)**

CompareStringA

int CompareStringA(LCID *Locale*, **DWORD** *dwCmpFlags*, **LPCSTR** *lpString1*, **int** *cchCount1*,
LPCSTR *lpString2*, **int** *cchCount2***)**

CompareStringW

Syntax

int CompareStringW(LCID *lcid*, **DWORD** *fdwStyle*, **LPCWSTR** *lpString1*, **int** *cch1*, **LPCWSTR** *lpString2*, **int** *cch2***)**

Limitations

Only accurate for less than 128 characters.

CopyFile

Syntax

BOOL CopyFile(LPCTSTR *lpszExistingFile*, **LPCTSTR** *lpszNewFile*, **BOOL** *fFailIfExists***)**

CopyMemory

Syntax

VOID CopyMemory (PVOID *Destination*, **CONST VOID *** *Source*,**DWORD** *Length***)**

CreateDirectory

Syntax

BOOL CreateDirectory(LPCTSTR *lpszPath*, **LPSECURITY_ATTRIBUTES** *lpsa***)**

Limitations:

LPSECURITY_ATTRIBUTES is not supported; fails if non-null.

CreateFile

Syntax

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

Syntax

BOOL DeleteFile(LPCTSTR *lpszFileName***)**

DosDateTimeToFileTime

Syntax

BOOL DosDateTimeToFileTime(WORD *wDOSDate*, **WORD** *wDOSTime*, **LPFILETIME** *lpft***)**

FBadReadPtr

Syntax

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

Syntax

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

Syntax

BOOL FileTimeToDosDateTime(CONST FILETIME * *lpft*, **LPWORD** *lpwDOSDate*, **LPWORD** *lpwDOSTime***)**

FileTimeToSystemTime

Syntax

BOOL FileTimeToSystemTime(CONST FILETIME * *lpft*, **LPSYSTEMTIME** *lpst***);**

FillMemory

Syntax

VOID FillMemory (PVOID *Destination*,**DWORD** *Length*,**BYTE** *Fill*)

FindClose

Syntax

BOOL FindClose(HANDLE *hFindFile***)**

FindFirstFile

Syntax

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

Syntax

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

Syntax

UINT GetACP(VOID)

GetCurrentProcessID

Syntax

DWORD GetCurrentProcessId(VOID)

Limitations

Returns HTASK.   Value is subject to reuse by the operating system.

GetFileAttributes

Syntax

DWORD GetFileAttributes(LPCTSTR *lpFileName***)**

Limitations

This function won't work on Novell NetWare without FILESCAN rights.

GetFileSize

Syntax

DWORD GetFileSize(HANDLE *hFile*, **LPDWORD** *lpdwFileSizeHigh***)**

GetFileTime

Syntax

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

Syntax

DWORD GetFullPathName(LPCTSTR *lpszFile***, DWORD** *cchPath***, LPTSTR** *lpszPath***, LPTSTR ****ppszFilePart***)**

Limitations

This function does not handle ".." path components in *lpFileName*.

GetLastError

Syntax

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

Syntax

VOID GetLocalTime(LPSYSTEMTIME *lpst***)**

Limitations

This function returns the current time as local time.

GetSystemTime

Syntax

VOID GetSystemTime(LPSYSTEMTIME *lpst***)**

Limitations

This function depends on the time zone and eturns 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

UINT GetTempFileName(LPCTSTR *lpszPath*, **LPCTSTR** *lpszPrefix*, **UINT** *uUnique*, **LPTSTR** *lpszTempFile***)**

GetTempFileName32

UINT WINAPI GetTempFileName32 (LPCSTR *lpPathName*, **LPCSTR** *lpPrefixString*, **UINT** *uUnique*, **LPSTR** *lpTempFileName***)**

GetTempPath

Syntax

DWORD GetTempPath(DWORD *cchBuffer*, **LPTSTR** *lpszTempPath***)**

GetTimeZoneInformation

Syntax

DWORD GetTimeZoneInformation(LPTIME_ZONE_INFORMATION *lptzi***)**

GetUserDefaultLCID

Syntax

LCID GetUserDefaultLCID(VOID)

InterlockedDecrement

Syntax

LONG InterlockedDecrement(LPLONG *lplVal***)**

Limitations

This function relies on cooperative multitasking.

InterlockedIncrement

Syntax

LONG InterlockedIncrement(LPLONG *lplVal***)**

Limitations

This function relies on cooperative multitasking.

IsBadBoundedStringPtr

BOOL WINAPI IsBadBoundedStringPtr(const void FAR* *lpsz*, **UINT** *cchMax***)**

IsBadReadPtr

Syntax

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

Syntax

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

Syntax

int lstrlenW(LPCWSTR *lpszString***)**

lstrcmpW

int lstrcmpW(LPCWSTR *lpszString1*, **LPCWSTR** *lpszString2***)**

lstrcpyW

LPWSTR lstrcpyW(LPWSTR *lpszString1*, **LPCWSTR** *lpszString2***)**

MoveFile

Syntax

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

Syntax

VOID MoveMemory (PVOID *Destination*, **CONST VOID *** *Source*,**DWORD** *Length***)**

MulDiv32

Syntax

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

Syntax

int MultiByteToWideChar(UINT *CodePage*, **DWORD** *dwFlags*, **LPCSTR** *lpMultiByteStr*, **int** *cchMultiByte*, **LPWSTR** *lpWideCharStr*, **int** *cchWideChar***)**

Limitations

This function only works for less than 128 characters; does not support Unicode single-byte conversion; only works reliably for ASCII characters.

ReadFile

Syntax

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

Syntax

BOOL RemoveDirectory(LPCTSTR *lpszDir***)**

SetEndOfFile

Syntax

BOOL SetEndOfFile(HANDLE *hFile***)**

SetFilePointer

Syntax

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

Syntax

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

Syntax

VOID Sleep(DWORD *cMilliseconds***)**

Limitations

This function does not handle the ALT+TAB and ALT+ESC key combinations for task switching.

SystemTimeToFileTime

Syntax

BOOL SystemTimeToFileTime(CONST SYSTEMTIME * *lpst*, **LPFILETIME** *lpft***)**

WideCharToMultiByte

Syntax

int WideCharToMultiByte(UINT *CodePage*, **DWORD** *dwFlags*, **LPCWSTR** *lpWideCharStr*, **int** *cchWideChar*, **LPSTR** *lpMultiByteStr*, **int** *cchMultiByte*, **LPCSTR** *lpDefaultChar*, **LPBOOL** *lpUsedDefaultChar***)**

Limitations

This function only works for less than 128 characters; does not support Unicode single-byte conversion; works reliably only for ASCII characters.

WriteFile

Syntax

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

Syntax

VOID ZeroMemory (PVOID *Destination*,**DWORD** *Length***)**

Mapping of X.400 P2 Attributes to Extended 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 chapter describes mappings between the recommended implementation's P2 attributes and Extended MAPI properties.

For a description of how to write providers and applications that use these mappings, see the *MAPI Programmer's Guide*.

The reference information is presented in two different ways in this chapter:

- 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 sections contain tables for each object listing the mappings from X.400 P2 attributes to their corresponding Extended MAPI properties.

OMP_O_IM_C_BD_PRT

[New - Windows 95]

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

[New - Windows 95]

The attributes of the class OMP_O_IM_C_BILAT_DEF_BD_PRT map to Extended MAPI properties as follows:

MH ID/Type	MAPI Property
IM_BILATERAL_DATA	PR_ATTACH_DATA_BIN

OMP_O_IM_C_EXTERN_DEF_BD_PRT

[New - Windows 95]

The attributes of the class OMP_O_IM_C_EXTERN_DEF_BD_PRT map to Extended 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

[New - Windows 95]

The attributes of the class OMP_O_IM_C_G3_FAX_BD_PRT are not mapped to Extended MAPI properties.

MH ID/Type	MAPI Property
IM_G3_FAX_NBPS	Not mapped to an Extended MAPI property.
IM_IMAGES	Not mapped to an Extended MAPI property.

OMP_O_IM_C_G4_CLASS_1_BD_PRT

[New - Windows 95]

The attributes of the class OMP_O_IM_C_G4_CLASS_1_BD_PRT are not mapped to Extended MAPI properties.

MH ID/Type	MAPI Property
IM_G4_CLASS_1_DOCUMENT	Not mapped to an Extended MAPI property.

OMP_O_IM_C_GENERAL_TEXT_BD_PRT

[New - Windows 95]

The attributes of the class OMP_O_IM_C_GENERAL_TEXT_BD_PRT are not mapped to Extended MAPI properties.

MH ID/Type	**MAPI Property**
IM_CHAR_SET_REG	Not mapped to an Extended MAPI property.
IM_TEXT	Not mapped to an Extended MAPI property.

OMP_O_IM_C_IA5_TEXT_BD_PRT

[New - Windows 95]

The attributes of the class OMP_O_IM_C_IA5_TEXT_BD_PRT are mapped to the following Extended MAPI properties:

MH ID/Type	MAPI Property
IM_REPERTOIRE	Not mapped to an Extended MAPI property.
IM_TEXT	PR_BODY

OMP_O_IM_C_INTERPERSONAL_MSG

[New - Windows 95]

The attributes of the class OMP_O_IM_C_INTERPERSONAL_MSG map to Extended 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	Extended 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

[New - Windows 95]

The attributes of the class OMP_O_IM_C_INTERPERSONAL_NOTIF map to Extended 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

[New - Windows 95]

The attributes of the class OMP_O_IM_C_IPM_IDENTIFIER map to Extended 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

[New - Windows 95]

The attributes of the class OMP_O_IM_C_ISO_6937_TEXT_BD_PRT are mapped to the following Extended MAPI properties:

MH ID/Type	**MAPI Property**
IM_REPERTOIRE	Not mapped to an Extended MAPI property.
IM_TEXT	PR_ATTACH_DATA_BIN

OMP_O_IM_C_MESSAGE_BD_PRT

[New - Windows 95]

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 an Extended MAPI property.
IM_IPM	Mapped to an IMessage object embedded in the message.

OMP_O_IM_C_MIXED_MODE_BD_PRT

[New - Windows 95]

The attributes of the class OMP_O_IM_C_MIXED_MODE_BD_PRT are not mapped to any Extended MAPI properties.

MH ID/Type	MAPI Property
IM_MIXED_MODE_DOCUMENT	Not mapped to an Extended MAPI property.

OMP_O_IM_C_NATIONAL_DEF_BD_PRT

[New - Windows 95]

The attributes of the class OMP_O_IM_C_NATIONAL_DEF_BD_PRT are not mapped to Extended MAPI properties.

MH ID/Type	MAPI Property
IM_NATIONAL_DATA	Not mapped to an Extended MAPI property.

OMP_O_IM_C_NON_RECEIPT_NOTIF

[New - Windows 95]

The attributes of the class OMP_O_IM_C_NON_RECEIPT_NOTIF map to Extended 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

[New - Windows 95]

The attributes of the class OMP_O_IM_C_ODA_BD_PRT are not mapped to Extended MAPI properties.

MH ID/Type	MAPI Property
IM_APPLICATION_PROFILE	Not mapped to an Extended MAPI property.
IM_ARCHITECTURE_CLASS	Not mapped to an Extended MAPI property.
IM_ODA_DOCUMENT	Not mapped to an Extended MAPI property.

OMP_O_IM_C_OR_DESCRIPTOR

[New - Windows 95]

The attributes of the class OMP_O_IM_C_OR_DESCRIPTOR map to Extended 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

[New - Windows 95]

The attributes of the class OMP_O_IM_C_RECEIPT_NOTIF map to Extended 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

[New - Windows 95]

The attributes of the class OMP_O_IM_C_OR_DESCRIPTOR map to Extended 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 Extended MAPI properties:

IM_NOTIFICATION_REQUEST Value	Extended 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

[New - Windows 95]

The attributes of the class OMP_O_IM_C_TELETEX_BD_PRT are mapped to the following Extended MAPI properties:

MH ID/Type	MAPI Property
IM_TELETEX_COMPATIBLE	Not mapped to an Extended MAPI property.
IM_TELETEX_DOCUMENT	PR_BODY
IM_TELETEX_NBPS	Not mapped to an Extended MAPI property.

OMP_O_IM_C_UNIDENTIFIED_BD_PRT

[New - Windows 95]

The attributes of the class OMP_O_IM_C_UNIDENTIFIED_BD_PRT are not mapped to Extended MAPI properties.

MH ID/Type	MAPI Property
IM_UNIDENTIFIED_DATA	Not mapped to an Extended MAPI property.
IM_UNIDENTIFIED_TAG	Not mapped to an Extended MAPI property.

OMP_O_IM_C_USA_NAT_DEF_BD_PRT

[New - Windows 95]

The attributes of the class OMP_O_IM_C_USA_NAT_DEF_BD_PRT are mapped to the following Extended MAPI properties:

MH ID/Type	**MAPI Property**
IM_BODY_PART_NUMBER	Not mapped to an Extended MAPI property.
IM_USA_DATA	Not mapped to an Extended MAPI property.

OMP_O_IM_C_VIDEOTEX_BD_PRT

[New - Windows 95]

The attributes of the class OMP_O_IM_C_VIDEOTEX_BD_PRT are not mapped to Extended MAPI properties.

MH ID/Type	MAPI Property
IM_VIDEOTEX_DATA	Not mapped to an Extended MAPI property.
IM_VIDEOTEX_SYNTAX	Not mapped to an Extended MAPI property.

Comprehensive X.400 Attributes Reference

The following table contains mappings between X.400 P2 attributes and Extended 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 an Extended MAPI property.
IM_ARCHITECTURE_CLASS (OMP_O_IM_C_ODA_BD_PRT)	Not mapped to an Extended 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 an Extended MAPI property.
IM_CHAR_SET_REG (OMP_O_IM_C_GENERAL_TEXT_BD_PRT)	Not mapped to an Extended 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 an Extended 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 an Extended MAPI property.
IM_G4_CLASS_1_DOCUMENT (OMP_O_IM_C_G4_CLASS_1_BD_PRT)	Not mapped to an Extended MAPI property.
IM_IMAGES (OMP_O_IM_C_G3_FAX_BD_PRT)	Not mapped to an Extended 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 an Extended MAPI property.
IM_NATIONAL_DATA (OMP_O_IM_C_NATIONAL_DEF_BD_PRT)	Not mapped to an Extended 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 an Extended 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 (OMP_O_IM_C_RECEIPT_NOTIF)	PR_RECEIPT_TIME
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 an Extended MAPI property.
IM_REPERTOIRE (OMP_O_IM_C_ISO_6937_TEXT_BD_PRT)	Not mapped to an Extended 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 an Extended 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 an Extended MAPI property.
IM_TEXT (OMP_O_IM_C_GENERAL_TEXT_BD_PRT)	Not mapped to an Extended 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 (OMP_O_IM_C_UNIDENTIFIED_BD_PRT)	Not mapped to an Extended MAPI property.
IM_UNIDENTIFIED_TAG (OMP_O_IM_C_UNIDENTIFIED_BD_PRT)	Not mapped to an Extended MAPI property.
IM_USA_DATA	Not mapped to an Extended MAPI property.

(OMP_O_IM_C_USA_NAT_DEF_BD_PRT)	
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 an Extended MAPI property.
IM_VIDEOTEX_SYNTAX (OMP_O_IM_C_VIDEOTEX_BD_PRT)	Not mapped to an Extended MAPI property.

About This Book

MAPI Programmer's Reference, Volume 2, one of three books accompanying the Messaging Application Programming Interface (MAPI) Software Development Kit (SDK), provides a complete reference to Extended MAPI functions, data structures, data types, properties, and return values and to the Simple MAPI and Common Messaging Calls (CMC) programming models. It also documents how to develop key configuration files for Extended MAPI; illustrates Transport-Neutral Encapsulation Format (TNEF), a serialization of Extended MAPI properties; describes Extended MAPI implementations of 32-bit Windows® functions; and provides a glossary of MAPI terms. This book should be used together with *MAPI Programmer's Reference, Volume 1,* and *MAPI Programmer's Guide* to develop messaging-based applications and services.

Intended Audience

MAPI Programmer's Reference, Volume 2, is written for C and C++ developers with a range of needs and experience with messaging. For those developers who want to use MAPI to augment their applications with messaging features, no specific prerequisite knowledge is required. However, for workgroup developers who plan to use MAPI to create full-scale messaging applications, extensions or customized solutions for existing products, or message services, a background in OLE programming is recommended.

How This Book Is Organized

This book's contents are organized as follows:

- Chapter 1, "Introduction," introduces concepts basic to using Extended MAPI functions, data structures, macros, data types, properties, and return values, including usage of input and output function parameters, function parameter commonality, and flag usage.
- Chapter 2, "Common Messaging Calls (CMC)," provides a complete reference to the CMC application programming interface (API), including documentation on CMC functions, data structures, data types, and data extensions.
- Chapter 3, "Simple MAPI," provides a complete reference to Simple MAPI, including documentation for Simple MAPI functions and structures for C and C++ and functions and data types for Visual Basic®.
- Chapter 4, "Essential Extended MAPI Functions," documents those Extended MAPI functions that are essential to implementing and using Extended MAPI. Functions are described in alphabetic order.
- Chapter 5, "Extended MAPI Properties," documents the Extended MAPI properties in alphabetic order.
- Chapter 6, "Extended MAPI Structures and Simple Data Types," documents the Extended MAPI data structures and data types in alphabetic order. It also provides documentation for macros that create structures with the structures they create.
- Chapter 7, "Additional Extended MAPI Functions," documents Extended MAPI functions specific to certain purposes that are not of general use in Extended MAPI development. Functions are described in alphabetic order.
- Chapter 8, "MAPISVC.INF File Format and WIN.INI File Format," documents the MAPISVC.INF file, the central database for MAPI configuration information. It also covers MAPI entries to the WIN.INI initialization file for the Microsoft® Windows operating system; these entries indicate the existence of various MAPI components and provide information about their configuration.
- Chapter 9, "Transport-Neutral Encapsulation Format (TNEF)," documents TNEF and includes examples of TNEF encoding.
- Chapter 10, "Configuration File Format," describes the format of a MAPI configuration file, a file created by a form developer to define a MAPI form.
- Appendix A, "Extended MAPI Return Values," provides information on the Extended MAPI return values that represent error conditions that cause method and function calls to fail and that cannot be dealt with using conventional error handling. Other Extended MAPI error values are documented with the methods or functions that return them.
- Appendix B, "Extended MAPI Versions of 32-Bit Windows Functions" documents MAPI implementations of certain 32-bit Windows functions.
- Appendix C, "Mapping of X.400 P2 Attributes to Extended MAPI Properties," maps attributes used by the X.400 message-handling standard to the corresponding Extended MAPI properties.
- The glossary defines the MAPI terms used most often throughout the MAPI SDK documentation.

Document Conventions

The following typographical conventions are used throughout this book:

Typographical convention	Meaning
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, **IMAPISupport::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, *ulFlags* is a MAPI function parameter. In addition, italics are used to highlight the first use of terms and to emphasize meaning.
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.
monospace	Indicates source code, structure syntax, examples, user input, and program output. For example: `ptbl->SortTable(pSort, TBL_BATCH);`
SMALL CAPITALS	Indicate the names of keys on the keyboard. When you see a plus sign (+) between two key names, you should hold down the first key while pressing the second. The carriage return key, sometimes marked as a bent arrow on the keyboard, is called ENTER.

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 the null-terminated string name *ProfileName*. For more information about Hungarian notation, see *Programming Windows 3.1* by Charles Petzold, and *Code Complete,* by Steve McConnell.

For More Information

For information on MAPI not covered in this book, see *MAPI Programmer's Guide* and *MAPI Programmer's Reference, Volume 1*, both contained within the MAPI SDK. *MAPI Programmer's Guide* provides a conceptual overview of the MAPI architecture and describes a wide range of programming tasks illustrated with extensible and reusable sample code. *MAPI Programmer's Reference, Volume 1,* provides a complete reference to the interfaces and methods used with Extended MAPI objects.

For more information about OLE programming, see *Inside OLE, Second Edition*, by Kraig Brockschmidt (Redmond, WA: Microsoft Press®, 1995), and *OLE Programmer's Reference, Volume One* and *Volume Two,* in the Microsoft Win32® SDK, published on CD-ROM by the Microsoft Developer Network (MSDN) and by Microsoft Visual C++™.

For more information about programming for 32-bit Windows, see the Win32 SDK. For more information about programming for 16-bit Windows, see the Microsoft Windows SDK for Windows version 3.1, published on CD-ROM by MSDN and by Visual C++. Another good general source on Windows programming is *Programming Windows 3.1,* by Charles Petzold (Redmond, WA: Microsoft Press, 1992), and a good handbook on practical software development in *Code Complete,* by Steve McConnell (Redmond, WA: Microsoft Press, 1993).

For more information about C and C++ programming, see the documentation for Visual C++, also published on CD-ROM by MSDN and by Visual C++.

Microsoft Win32 Developer’s Reference

You have requested information from the **Microsoft Win32 Developer’s Reference**. One or more of these help files is not available on your system.