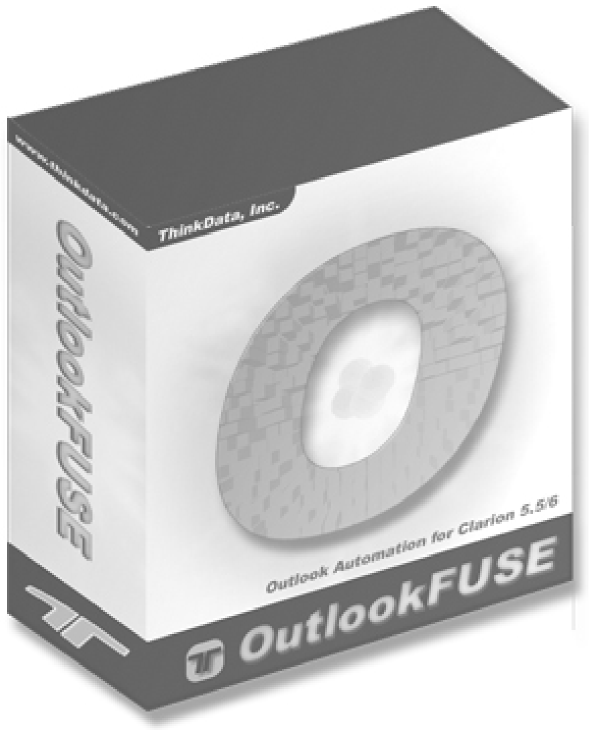
THINKDATA INC.

Clarion Third Party Add-Ons



OutlookFUSE 2.0 User Guide

CLARION THIRD PARTY ADD-ONS

OutlookFUSE 2.0 User Guide

Copyright © 2004 ThinkData Inc.
2508 Pacific Avenue • Suite 1
Venice Beach, California 90291
Phone 310.823.2571
Fax 310.943.1858
info@thinkdata.com

Table of Contents

Introduction

Overview

The OutlookFUSE product provides a native COM (Component Object Model) automation interface to Microsoft Outlook using Clarion 5.5 and Clarion 6. It relies on the Plugware COM classes which encapsulate the major portions of the COM Application Programming Interface (API) specification as designed by Microsoft. The Plugware COM classes usher in a new era of stability and performance when writing COM automation interfaces in Clarion. Full source code to these classes has been provided along with detailed examples demonstrating how to use the OutlookFUSE product. This gives the developer/end-user an opportunity to understand the underlying principles of COM while providing an open and stable platform from which to develop additional automation interfaces. The example application included is intended to demonstrate early and late bound interface functionality to Microsoft Outlook and is in no way intended to be an exhaustive demonstration of the power of the Outlook COM object model. We believe that with OutlookFUSE and a little imagination you will achieve large gains in performance, productivity and stability when writing custom automation interfaces for Outlook.

Prerequisites

This manual is targeted at intermediate Clarion 5.5 and Clarion 6 programmers with some Win32 programming experience and a basic understanding of COM. It is not intended to be a primer on COM – for that we recommend the following reading list.

Inside COM by Dale Rogerson.
This book discloses the secrets of COM programming for the advanced engineer. It includes many sample programs on CD-ROM. (Microsoft Press, ISBN 1-57231-349-8)

Essential COM by Don Box
This text covers the motivation for the design of COM and its distributed aspects. It shows how COM works and contains coverage of the core concepts of distributed COM including detailed descriptions of COM theory and remote servers. It also offers a thorough explanation of COM's basic vocabulary. (Addison Wesley ISBN 0-201-63446-5)

The Plugware COM classes and OutlookFUSE product are written exclusively for Clarion 5.5H and Clarion 6. Earlier versions of Clarion are not currently supported because of improvements in the compiler to support interfaces and passing GROUP structures by value.

Installation

To install OutlookFUSE, simply execute the supplied olfuse.exe file and follow the instructions in the installation program. OutlookFUSE consists of Clarion source files, documentation and example applications. The default installation directory (\C55) is located based on your Clarion 5.5 installation. If you choose \C55 as the installation directory the installer will place files in the following order:

\C55\Libsrc – The following source files for the Plugware COM classes and the Outlook 2003 wrapper will default to this directory:

pwapi.inc
pwapifnc.inc
pwcomdef.inc
pwcom.inc
pwcom.clw
pwheap.inc
pwheap.clw
oldef.inc
olint.inc
oliid.inc
outlook.inc
outlook.clw
outlook1.inc
outlook1.clw

\C55\Libsrc\OutlookFUSE\Outlook98 – Source files for Outlook 98 wrapper including:

oldef.inc
olint.inc
oliid.inc
outlook.inc
outlook.clw
outlook1.inc
outlook1.clw

\C55\Libsrc\OutlookFUSE\Outlook2000 – Source files for Outlook 2000 wrapper including:

oldef.inc
olint.inc
oliid.inc
outlook.inc
outlook.clw
outlook1.inc
outlook1.clw

\C55\Libsrc\OutlookFUSE\Outlook2002 – Source files for Outlook 2002 wrapper including:

oldef.inc
olint.inc
oliid.inc
outlook.inc
outlook.clw
outlook1.inc
outlook1.clw

\C55\Libsrc\OutlookFUSE\Outlook2003 – Source files for Outlook 2003 wrapper including:

oldef.inc
olint.inc
oliid.inc
outlook.inc
outlook.clw
outlook1.inc
outlook1.clw

\C55\Docs\OutlookFUSE – OutlookFUSE documentation including:

outlookfuse.pdf

\C55\Examples\OutlookFUSE – Source files for the example applications including:

olfuse.clw
olfuse.exe
olfuse.prj
olimpexp.app
olimpexp.dct
thinkdata.ico

The wrappers are named identically to make it easy to recompile your early bound Outlook COM automation application for either Microsoft Outlook 98, 2000, or 2002. Simply copy the files from the appropriate directory under \Libsrc\OutlookFUSE into your \Libsrc directory and recompile. The wrapper for Microsoft Outlook 2003 uses a new naming convention for the classes and must be used independently of the other class wrappers. The demo executables in \C55\Examples\OutlookFUSE have been compiled with the Microsoft Outlook 2003 wrapper under Clarion 5.5H.

Plugware COM Overview

The Plugware COM classes, written by Plugware Solutions.com Ltd., provide the Clarion developer with a set of classes to encapsulate the COM API natively in Clarion. Now any developer can write pure Clarion source code to interface with COM objects using native early or late binding without worrying about stability, performance, or flexibility. So without further ado let us explore the Plugware COM classes!

Plugware COM ships as seven Clarion source files and one export file for multi-DLL and hand-coded applications. These files are:

pwapi.inc – Contains a large number of the Win32 API data types and constants. It is included by the pwcomdef.inc file.

pwapifnc.inc – Contains a large number of the Win32 API function prototypes for Clarion. It is included by the pwcomdef.inc file.

pwcomdef.inc – Contains the common interfaces (IUnknown, IDispatch, ITypeInfo), common data types for implementing COM automation, and the function prototypes for Win32 API calls used by the COM classes. It is included by the pwcom.inc file.

pwcom.inc – Contains the class definitions for the early and late binding implementations of generic COM objects including COM string classes, variant factory object, safe array support and dispatch interface wrapper.

pwcom.clw – Contains the source implementation of the classes defined in pwcom.inc.

pwheap.inc – Contains the class definition for the PWHeap class which is used by the string classes to allocate memory properly. It is included by the pwcom.clw file.

pwheap.clw – Contains the source implementation of the class defined in pwheap.inc.

pwcom.exp – Contains the exports for the Plugware COM classes and methods used in multi-DLL and hand-coded Clarion application development.

String Classes

There are four classes provided to deal with strings in COM. Figure 1.1 shows the inheritance relationships of these classes.

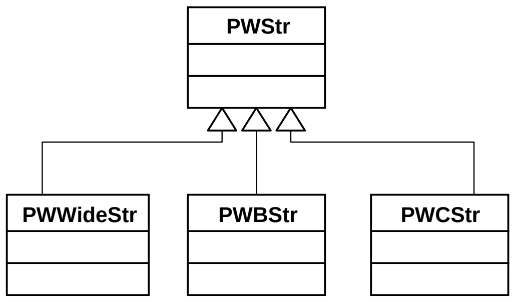


FIGURE 1.1 The relationship between the parent PWStr class and the derived classes PWWideStr, PWBStr (BSTR or binary string support in COM) and PWCStr

The PWStr is the parent class of the three other classes. It contains a virtual destructor which calls the PWStr.Release method to handle freeing the memory for the string. The PWWideStr class is used internally by the PWBStr class to allocate memory and map a character string to a wide-character (Unicode) string. The two classes of interest to the developer are the PWBStr and PWCStr classes.

The PWBStr class was written to support the COM BSTR data type. A BSTR is a pointer to a wide character string and is sometimes referred to as a Basic string or binary string. The PWBStr can take a Clarion cstring or string in its initialization method and the PWBStr.GetStr() method will be called when the developer wishes to pass a BSTR to a COM object.

The PWCStr class is designed to retrieve a Clarion CSTRING from a PWBStr object. It will be used frequently to retrieve string output parameters from calls to COM objects. It operates in the reverse of the PWWideStr by calling the WideCharToMultiByte API function to map a wide-character (Unicode) string to a character string.

A helper function called _cstr has been added to Plugware COM to support the automatic conversion of BSTR return values to a Clarion CSTRING. It is prototyped as follows:

_cstr(long bstrVal, short fFreeBStr = true),*cstring

The _cstr function is passed the BSTR parameter and returns the reference to a newly created Clarion CSTRING.

Helper Classes

There are five classes provided for making the process of interfacing to the Win32 COM API easier:

PWCOMIniter

The PWCOMIniter should be the first object instantiated on each thread using the Plugware COM classes. It calls the Win32 API function CoInitialize() to initialize COM for that thread. When its destructor is called the CoUnInitialize() API function is called. The PWCOMIniter must be instantiated before any of the other Plugware COM classes to ensure that its destructor method is called after all other classes have destructed.

PWCOMError

The PWCOMError class is responsible for translating COM errors returned as HRESULTs into something readable to the developer or end user. The method of interest to the developer inside the PWCOMError class is:

PWCOMError.GetError procedure(*cstring szErrorMsg, *long dwBufferLen, HRESULT hr),long – This method fills the passed szErrorMsg with the results from the Windows API error message for the passed HRESULT. This method uses the FormatMessage API function internally.

PWDateTime

The PWDateTime class converts Clarion dates and times into values compatible with COM objects. The methods of interest to the developer inside the PWDispatch class are:

PWDateTime.Init procedure(*tVariant vtTime) – This method takes a variant of type VT_DATE and initializes the PWDateTime structure with its contents.

PWDateTime.Init procedure(*real systime) – This method takes a Clarion real containing a COleDateTime date/time value and initializes the PWDateTime structure with its contents.

PWDateTime.Init procedure(*SYSTEMTIME systime) – This method takes a Windows API SYSTEMTIME structure and initializes the PWDateTime with its contents.

PWDateTime.Init procedure(date cwDate, time cwTime) – This method initializes the PWDateTime structure with a Clarion date and time. Use this method in conjunction with PWDateTime.GetAsCOleDateTime to pass dates to COM objects.

PWDateTime.GetAsCOleDateTime procedure(),real – This method returns the contents of the PWDateTime structure as a real which is equivalent to the COleDateTime class used in the Microsoft Foundation Classes (MFC). Most COM objects will accept dates in this format.

PWDateTime.GetAsClarionDate procedure(),date – This method returns the contents of the PWDateTime structure as a Clarion DATE. Use this method in conjunction with PWDateTime.GetAsClarionTime to get the complete date/time component.

PWDateTime.GetAsClarionTime procedure(),time – This method returns the contents of the PWDateTime structure as a Clarion TIME. Use this method in conjunction with PWDateTime.GetAsClarionDate to get the complete date/time component.

PWDateTime.Now procedure() – This method sets the PWDateTime structure to the current time using the GetLocalTime API function.

PWInvokeHelper

The PWInvokeHelper class is used in late binding automation to handle parameter passing to COM object methods via the PWDispatch.Invoke method. Plugware COM provides a number of different versions of PWDispatch.Invoke which will be discussed in the Late Binding Automation section of this documentation. These Invoke methods will cover calling methods with up to 10 parameters; therefore, one will generally not need to manipulate a PWInvokeHelper object directly. The easiest way to understand the functionality of PWInvokeHelper is to follow the code in the PWDispatch.Invoke methods in pwcom.clw.

PWSafeArray

The PWSafeArray encapsulates the functionality of the SAFEARRAY data type. A SAFEARRAY is an array which includes boundary information. This provides the developer with the size and dimensions of the array, thus eliminating the possibility of out of bounds addressing errors. The PWSafeArray class effectively wraps the SafeArray COM API functions defined in OLEAUT32.DLL. The methods of interest to the developer inside the PWCOMError class are:

PWSafeArray.Attach procedure(procedure(*tVariant vtsa, short fSelfCleaning = true),short) – This method attaches a PWSafeArray object to a variant returned from a call to a COM object method. It returns true if the attach operation succeeds and false if it fails.

PWSafeArray.Attach procedure(*_SAFEARRAY sa, short fSelfCleaning) – Attaches a PWSafeArray object to a data structure of type _SAFEARRAY. It returns true if the attach operation succeeds and false if it fails.

PWSafeArray._Create procedure(VARTYPE vt, long ulCount, long lLBound)– Creates a new array of type vt (see pwcomdef.inc for a list of valid VARTYPEs) with a size and the lower bound specification. It makes calls internally to SafeArrayCreate to create the new array descriptor and allocate and initialize the data type for the array.

PWSafeArray._Copy procedure(*HRESULT hr),*PWSafeArray - Copies the existing PWSafeArray and returns a reference to the newly created PWSafeArray.

PWSafeArray.GetType procedure,long – Returns the type of SafeArray contained in the PWSafeArray object. See pwcomdef.inc for a list of the valid VARTYPEs.

PWSafeArray.GetLowerBound procedure(long uDim = 0),long – Returns the lower bound element of the SafeArray. This return value is zero based.

PWSafeArray.GetUpperBound procedure(long uDim = 0),long – Returns the upper bound element of the SafeArray. This return value is zero based.

PWSafeArray.GetDimensions procedure(*long xElems, *long yElems, *long nDims) – Returns the number of dimensions of the array and the number of elements in the x and y dimensions.

PWSafeArray.GetDimension procedure(long nDim, *long nElems) – Returns the zero based number of elements in the dimension specified by the first parameter. Pass 0 as the first parameter to get the number of elements in the 1st dimension of the array.

PWSafeArray.AccessData procedure – Retrieves a pointer to the array data and increments the lock count of the array. You must call PWSafeArray.UnaccessData once you are finished manipulating the data after calling this method.

PWSafeArray.UnaccessData procedure – Calls SafeArrayUnaccessData internally to decrement the lock count of the array. You should call this method once you are finished manipulating the data after calling PWSafeArray.AccessData.

Early Binding Automation

The PWCOMObject class defined in pwcom.clw is responsible for handling the low level instantiation of COM objects and the proper release of those interfaces once the developer is finished using them. The PWDispatch class used in Late Binding Automation is actually derived from the PWCOMObject and will be covered in the next section.

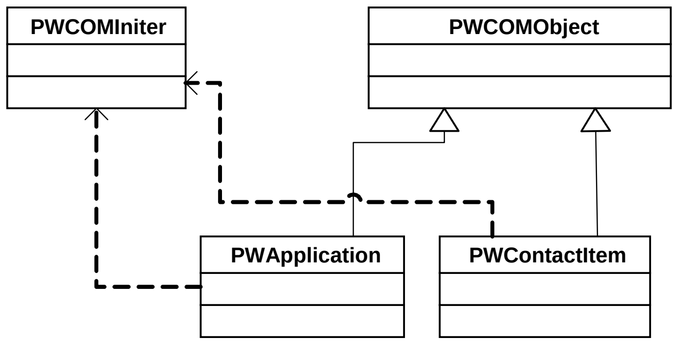


FIGURE 1.2 The relationship between the PWCOMObject and two of the Outlook COM objects, PWApplication and PWContactItem. The dotted arrows indicate that the child objects are reliant on a PWCOMIniter being instantiated for the current thread.

The methods of interest to the developer inside the PWCOMObject class are:

<u>PWCOMObject.AttachExisting procedure(RCLSID rclsid, REFIID riid)</u> - Attaches the object to an existing instance of the CLSID passed in, if it exists, and attaches itself to the IID passed as the second parameter.

<u>PWCOMObject.CreateInstance procedure(REFCLSID rclsid, REFIID riid, long dwClsContext = CLSCTX_ALL)</u> – Uses the CoCreateInstance API call to create an instance of the CLSID passed in. It attaches itself to the IID passed as the second parameter. The third parameter is the context in which the code that manages the newly created object will run in. It is taken from the CLSCTX itemized equates defined in pwcomdef.inc.

<u>PWCOMObject.GetInterface procedure(long pInterface, REFIID riid, *long pvObject, short fRelease)</u> – This method returns an interface from a pointer to an IDispatch or IUnknown interface. The first parameter contains the long pointer to the IDispatch interface passed in. The second parameter contains the IID of the interface we wish to get back from the internal call to QueryInterface. The third parameter will contain a pointer to the output interface. The fourth parameter is a true/false flag where a true will cause the internally referenced IUnknown to release. This method will be used most frequently in the early binding automation when a procedure call returns a pointer to an IDispatch (i.e. PWApplication.CreateItem, PWItems._Add, etc. both of which are defined in outlook.inc).

PWCOMObject.QueryInterface procedure(REFIID riid, *long pvObject) – This method calls QueryInterface on the object wrapped in the current PWCOMObject. This might be used when attaching PWCOMObject to IUnknown interfaces to determine what kind of interface is contained in the object.

PWCOMObject.AddRef procedure – This is equivalent to calling IUnknown.AddRef and is used internally by the PWCOMObject class.

PWCOMObject.Release procedure – This method releases the interface contained in the PWCOMObject and is equivalent to calling IUnknown.Release. It maintains an internal reference count to ensure that the object is released at the proper time.

PWCOMObject.Attach procedure(long pUnk, short fNoAddRef = false) – This method attaches the object to a passed pointer to an IUnknown interface. The first parameter contains the long pointer to the IUnknown interface we wish to attach to. The second parameter determines whether we should call IUnkown.AddRef on this interface.

Late Binding Automation

The PWDispatch class defined in pwcom.clw is the derived late bound version of the PWCOMObject class. It encapsulates much of the functionality of working with IDispatch interfaces. PWDispatch has been designed to abstract complexity away from the development of COM automation solutions by providing a simple interface with functions to perform type conversions and variant parameter passing. Figure 1.3 shows the relationship between PWDispatch and the classes it relies upon to get its job done.

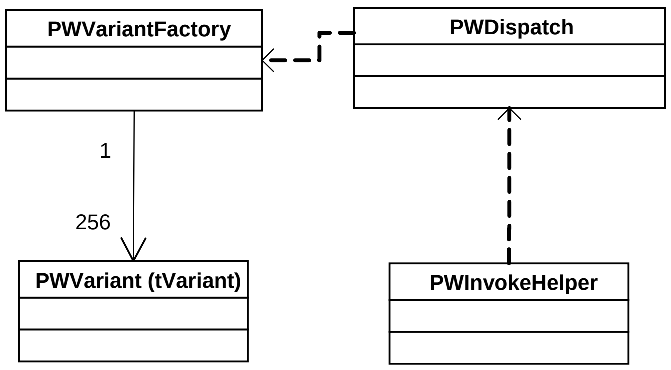


FIGURE 1.3 The PWVariantFactory object is a dependency of the PWDispatch class because variants are cached in the global PWVariantFactory when handling the Invoke calls to the COM object inside PWDispatch. PWVariantFactory can have a maximum of 256 cached variants at a time. PWInvokeHelper can be used independently of the PWVariantFactory to prepare parameter lists and call Invoke on them.

PWVariantFactory is a class used by a global helper function called _vt. The _vt function will convert Clarion data types into variants suitable for passing to COM objects. The PWVariantFactory is globally instantiated and it is responsible for caching the variants created when _vt is called. The PWVariantFactory stores a maximum of 256 cached variants to conserve memory (this can be edited in the PWVariantFactory.Construct method in pwcom.clw) so you will see references to PWVariantFactory.Release() in the examples to clear the cache for re-use.

The PWDispatch class relies on internal PWInvokeHelper objects inside its Invoke methods to handle initializing the parameter lists before calling IDispatch.Invoke. There are four versions of the PWDispatch.Invoke class which are of particular note. They take 0, 1, 5, and 10 variant parameters respectively. This allows the COM developer to pass optional parameters to procedures which take a long list. For those rare instances in which you will need to pass more than 10 parameters to a COM procedure you will need to instantiate your own PWInvokeHelper and use the self.SetParam method similarly to the way it is used in the PWDispatch.Invoke method for 10 parameters declared in pwcom.inc.

The methods of interest to the developer inside the PWDispatch class are:

PWDispatch.CreateInstance procedure(*cstring szObject) – Take a passed cstring containing the name of the object to instantiate (i.e. 'Outlook.Application' for Microsoft Outlook) and handles the internal calls into the COM API to call CoCreateInstance on the proper CLSID.

PWDispatch.CreateInstance procedure(*cstring szProgID, long flags) – An overloaded version of the CreateInstance above which takes a second parameter containing the context in which the code that manages the newly created object will run in. It is taken from the CLSCTX itemized equates defined in pwcomdef.inc.

PWDispatch.AttachExisting procedure(RCLSID rclsid) - Attaches the object to an existing instance of the CLSID passed in, if it exists. The PWDispatch will then be attached to the IDispatch interface of the object referenced by the CLSID.

PWDispatch.AttachExisting procedure(*cstring szProgID) – An overloaded version of the AttachExisting above which takes a cstring containing the name of the existing instance of an object to attach to (i.e. 'Outlook.Application' will attach to an existing instance of Microsoft Outlook).

PWDispatch.Attach procedure(long pIDisp, short fNoAddRef) – This method attaches the object to a passed pointer to an IDispatch interface. The first parameter contains the long pointer to the IDispatch interface we wish to attach to. The second parameter determines whether we should call IDispatch.AddRef on this interface.

PWDispatch.Attach procedure(*tVariant vtDisp, short fNoAddRef) – An overloaded version of the Attach above which takes a variant parameter containing an IDispatch interface instead of a long pointer to the interface. This allows us to attach a PWDispatch to an output parameter from a call to PWDispatch.Invoke.

PWDispatch.Attach procedure(*IDispatch IDisp, short fNoAddRef) – An overloaded version of the Attach above which takes an IDispatch interface parameter rather than a long pointer or a variant.

We alluded to the Invoke methods earlier and discussed the fact that there are several different versions depending on the number of parameters the COM method may take. Optional parameters are handled easily with the variant vtMissing declared globally in the Plugware COM classes. vtMissing can be used to pass empty or optional parameters to the COM methods. The next section covers the use of the PWDispatch.Invoke method (which is a wrapper around the IDispatch.Invoke COM method) and gives examples of using the vtMissing variant to pass empty or optional parameters.

PWDispatch.Invoke procedure(*cstring szMember, long wFlags, *tVariant vtRet, short fDispatchPut) – This version of PWDispatch.Invoke takes no input parameters. The first parameter is a cstring containing the name of the member method or property to invoke. The second parameter is a flag for IDispatch::Invoke and can be one of the following constants declared in pwcomdef.inc:

DISPATCH_METHOD – This Member is a COM object method to execute
DISPATCH_PROPERTYGET – This Member is a property we wish to retrieve
DISPATCH_PROPERTYPUT – This Member is a property we wish to put
DISPATCH_PROPERTYPUTREF – This Member is a property we wish to but by reference, not by value

The third parameter is a variant output parameter from the Invoke call. It will contain the return value from the call to IDispatch.Invoke once it has completed. The fourth parameter is a true/false flag and should be set to true only when the COM object expects a put-by-reference rather than a put-by-value. An example call is listed below:

```
  szMember = 'Count'
  hr = DispInboxItems.Invoke(szMember, DISPATCH_PROPERTYGET, vtIDisp)
```

<u>PWDispatch.Invoke procedure(*cstring szMember, long wFlags, *tVariant vtParam1, *tVariant vtRet, short fDispatchPut = false)</u> – This version of PWDispatch.Invoke is identical to the one above except it takes one input parameter called vtParam1. The other parameters are treated the same as the version which takes no input parameters. An example call is listed below:

```
  szMember = 'GetDefaultFolder'
  hr = DispNamespace.Invoke(szMember, DISPATCH_METHOD, _vt(lolFolderInbox), vtIDisp)
```

<u>PWDispatch.Invoke procedure(*cstring szMember, long wFlags, long nParams, *tVariant vtParam1, *tVariant vtParam2, *tVariant vtParam3, *tVariant vtParam4, *tVariant vtParam5, *tVariant vtRet, short fDispatchPut)</u> – This version of PWDispatch.Invoke is identical to the one above except it takes five input parameters called vtParam1, vtParam2, vtParam3, vtParam4, and vtParam5. This method can be used for procedures which take between 1 and 5 parameters because we can pass a vtMissing for parameters which we wish to omit. An example call listed below uses this version to call a method requiring only one parameter:

```
  szMember = 'GetDefaultFolder'
  hr = DispNamespace.Invoke(szMember, DISPATCH_METHOD, _vt(lolFolderInbox), _vt(vtMissing), _vt(vtMissing),
_vt(vtMissing), _vt(vtMissing), vtIDisp)
```

PWDispatch.Invoke procedure(*cstring szMember, long wFlags, long nParams, *tVariant vtParam1, *tVariant vtParam2, *tVariant vtParam3, *tVariant vtParam4, *tVariant vtParam5, *tVariant vtParam6, *tVariant vtParam7, *tVariant vtParam8, *tVariant vtParam9, *tVariant vtParam10, *tVariant vtRet, short fDispatchPut) – This version of PWDispatch.Invoke is identical to the one above except it takes ten input parameters called vtParam1, …, vtParam10. This method can be used for procedures which take between 1 and 10 parameters using the vtMissing variant.

Multi-DLL Considerations

Plugware COM provides support for compiling the classes into a multi-DLL project. All Plugware COM products ship with the pwcom.exp file installed into the Libsrc directory. This export file should be merged into the export file of the DLL containing your base class declarations and data. Once you have done this the Plugware COM objects can be referenced from any DLL or EXE which references this base DLL.

The linking of the classes is handled using project pragma settings similar to the way the ABC classes are exported in Clarion. A list of the pragmas for exporting the Plugware COM class definitions is described below for those who prefer to hand code their project in Clarion:

Applications Compiled with Data Local to the Module

%#pragma define(_APIDllMode_=>off)

%#pragma define(_APILinkMode=>on)

%#pragma define(_COMDllMode_=>off)

%#pragma define(_COMLinkMode_=>on)

Applications Compiled with Data External to the Module

%#pragma define(_APIDllMode_=>on)

%#pragma define(_APILinkMode=>off)

%#pragma define(_COMDllMode_=>on)

%#pragma define(_COMLinkMode_=>off)

OutlookFUSE Overview

OutlookFUSE consists of four generated wrapper classes for the Microsoft Outlook COM object model, one for Outlook 98, the second for Outlook 2000, the third for Outlook 2002, and the fourth for Outlook 2003. Each set consists of seven source files which encapsulate the interfaces and data types for Outlook automation. They are identically named to make it is easy for the developer to interchange them if a project requires an early binding interface to a different version of Outlook.

Microsoft recommends using the earliest type library for the application you wish to provide compatibility. So if you are writing an early binding application that must be compatible with Outlook 98, 2000, 2002, and 2003 you should use the Outlook 98 wrappers in OutlookFUSE. This is one of the reasons they recommend the use of late binding to resolve all versioning issues when developing for Microsoft Office. OutlookFUSE can be used either way - however, when you need maximum performance we do recommend using early binding. An excellent resource and description of the differences between early and late binding can be found on MSDN at the following link:

http://support.microsoft.com/default.aspx?scid=KB;EN-US;q245115&

We recommend doing some research and deciding on the right approach for your project. You can also visit our website and find resources via our support forum at the following link:

http://www.thinkdata.com/forum

The seven source files which form the OutlookFUSE product and the early bound version of the Microsoft Outlook COM object model are:

oliid.inc – Contains the definitions for the CLSID and IID values for the entire Microsoft Outlook COM object model. It is included by the olint.inc file.

olint.inc – Contains the Microsoft Outlook COM object model interfaces defined with Clarion compatible prototypes. It is included by the outlook.inc and outlook1.inc file.

oldef.inc – Contains the Microsoft Outlook COM object model constants defined as Clarion equates. It is included by the olint.inc file.

outlook.inc – Contains the generated Plugware COM classes to implement the interfaces defined in olint.inc.

outlook.clw – Contains the source implementation of the classes defined in outlook.inc.

outlook1.inc – Contains the second portion of the generated Plugware COM classes to implement the interfaces defined in olint.inc.

outlook1.clw – Contains the source implementation of the class defined in outlook1.inc.

Outlook Events

OutlookFUSE 2.0 supports the event interfaces published by Microsoft for Outlook 2002 and 2003. The demo application, olfuse.clw, contains examples of using the events. A developer can even keep events from happening by setting the Cancel byte to true in the methods supporting cancellation. The four classes which encapsulate the event interfaces are:

IApplicationEvents10 – Events specific to the Application object (Outlook 2002)

IApplicationEvents11 – Events specific to the Application object (Outlook 2003)

IExplorerEvents10 – Events specific to the Explorer object

IInspectorEvents10 – Events specific to the Inspector object

IItemEvents10 – Events specific to Item objects

Please study the olfuse.clw source code to get an idea of how to sink the events and derive the above classes so that your methods are called.

A detailed explanation of Outlook events can be found at Microsoft using the following link:

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/modcore/html/deovrunderstandingeventsinoutlook.asp

Example Application

The first example application which ships with OutlookFUSE is olimpexp.app. It demonstrates importing and exporting Contacts, importing appointments, tasks, and messages. The documentation does not cover this example due to its size – for questions about it you can visit our support forum. Instead, we will cover the second example application which ships with OutlookFUSE called olfuse.prj. It is a Clarion source project and you can view the source in its entirety by looking at the olfuse.clw file (most likely installed in your \C55\Examples\OutlookFUSE directory). This is the same as the free demo which can be downloaded from the ThinkData site using the following link

http://www.thinkdata.com/aspwpadmin/stattrack/includes/dltrack.asp?Title=OFUSEDEMO2&File=ofdemo.zip

The source code is commented and ships for free with the downloadable demo to illustrate how easy it is to use OutlookFUSE and the Plugware COM classes. This example demonstrates importing the Outlook Inbox into a Clarion queue using the early and late bound Plugware COM and the equivalent Clarion COM code. It also demonstrates creating an Outlook Contact using all three forms of creation. The Clarion COM code works but it is unstable and does not perform well. Furthermore, it does not give you the level of control that the Plugware COM gives (for example, Clarion does not let us know if it creating a new instance of the object or attaching to an existing one so we don't know if we can close Outlook or not!). We provided side-by-side Clarion examples for demonstration and documentation purposes.

The following listing is a subset of the code in the example with comments in bold to describe what the code is doing. Some of the comments will have a "VB Equivalent" which shows how to convert Visual Basic COM automation code into Clarion using OutlookFUSE. Enjoy!

```
! This line of code is all you need to include OutlookFUSE into your source
project
include('outlook.inc'),once

  map
    ImportInboxEarlyBound   ! Import Outlook Inbox via Early Bound Plugware COM
    ImportInboxLateBound    ! Import Outlook Inbox via Late Bound Plugware COM
    NewContactEarlyBound    ! Create a new contact via Early Bound Plugware COM
    NewContactLateBound      ! Create a new contact via Late Bound Plugware COM
    NewAppointmentEarlyBound ! Create a new appointment item in Outlook
    NewTaskEarlyBound        ! Create a new task item in Outlook
    ShowCOMError(HRESULT hrparam)    ! Display a COM error using API Messagebox
  end

! It is important to declare the COMIniter before any other Plugware COM object
! to ensure that it will be the last to destruct, otherwise you will get COM
! errors thrown if you attempt to execute COM code when CoUnInitialize() has
! been called
COMIniter           PWCOMIniter
COMError            PWCOMError
DispApplication     PWDispatch
DispNamespace       PWDispatch
DispInbox           PWDispatch
DispInboxItems      PWDispatch
DispMessage         PWDispatch
szMember            cstring(256)
pvObject            long,auto
pvTemp              long,auto
vtIDisp             like(tVariant)
vtValue             like(tVariant)



InboxQ              queue,pre(OLIN)
SenderName            cstring(256)
ToList                cstring(256)
CCList                cstring(256)
BCCList               cstring(256)
Subject               cstring(256)
Body                  cstring(256)
                    end

ImportInboxEarlyBound procedure
```

```
Outlook                TDOFApplication
Namespace              TDOFNameSpace
Inbox                  TDOFMAPIFolder
InboxItems             TDOFItems
Mail                   TDOFMailItem
BStrNamespace          PWBStr
BStrFullName           PWBStr
BStrEmail1Address      PWBStr
lCount                 long
fCloseOutlook          byte
sz                     &cstring
pbstr                  long
szNmsp                 cstring('MAPI')

  code
    free(InboxQ)
    clear(WindowMessage)
    RecordsProcessed = 0
    RecordsToProcess = 0

    ! Use the Win32 API call GetActiveObject to connect to existing
    ! instance of the Outlook Application COM Object if available
    hr = Outlook.AttachExisting(address(CLSID_Application), |
          address(IID__Application))
    if hr ~= S_OK
      ! The attach to existing instance failed so we must create one using
      ! Outlook.CreateInstance which calls the Win32 API call CoCreateInstance

      hr = Outlook.CreateInstance(address(CLSID_Application), |
           address(IID__Application), |
           bor(CLSCTX_LOCAL_SERVER, CLSCTX_INPROC_SERVER))
      if hr ~= S_OK then ShowCOMError(hr);return.

      ! We have created our own instance of Outlook so we should call the
      ! Outlook.Quit method at the end to close the application once we are
      ! done using it.  This illustrates the power of OutlookFUSE – standard
      ! Clarion COM cannot give you this level of control and stability.
      fCloseOutlook = true
    end

    ! VB Equivalent:  Outlook.GetNameSpace("MAPI")
    lLength = BStrNamespace.Init(szNmsp)
    hr = Outlook.GetNamespace(BStrNamespace.GetStr(), pvObject)
    if hr ~= S_OK then ShowCOMError(hr);return.

    ! Attach the Namespace object to the interface pointer returned
    ! from GetNamespace
    hr = Namespace.Attach(pvObject)
    if hr ~= S_OK then ShowCOMError(hr);return.

    ! VB Equivalent:  Inbox = Namespace.GetDefaultFolder(olFolderInbox)
    hr = Namespace.GetDefaultFolder(olFolderInbox, pvObject)
    if hr ~= S_OK then ShowCOMError(hr);return.

    ! Attach the Inbox object to the interface pointer returned
    ! from GetDefaultFolder
    hr = Inbox.Attach(pvObject)
    if hr ~= S_OK then ShowCOMError(hr);return.

    ! VB Equivalent:  InboxItems = Inbox.Items
    hr = Inbox.get_Items(pvObject)
    if hr ~= S_OK then ShowCOMError(hr);return.



    ! Attach the InboxItems object to the interface pointer returned
    ! from get_Items
```

```
hr = InboxItems.Attach(pvObject)
if hr ~= S_OK then ShowCOMError(hr);return.

! Get the number of messages in the InboxItems collection
hr = InboxItems.get_Count(lCount)
if hr ~= S_OK then ShowCOMError(hr);return.

RecordsToProcess = lCount
RecordsProcessed = 0

if RecordsToProcess
  loop
    RecordsProcessed += 1
    ! Check to see if we are done
    if RecordsProcessed > RecordsToProcess
      if fCloseOutlook
        ! Close Outlook since we instantiated a new copy of it above
        hr = Outlook.Quit()
      end
      break
    end

    Progress = RecordsProcessed / RecordsToProcess * 100
    WindowMessage = 'Processing message ' |
                    &RecordsProcessed&' of '&RecordsToProcess
    display

    ! vtValue will contain the index of the current MailItem
    ! and InboxItems._Item will retrieve it
    ! and place the pointer to the interface in the pvObject variable
    vtValue.vt = VT_I4
    vtValue.iVal = RecordsProcessed

    ! VB Equivalent:  MailItem = InboxItems.Item(vtValue)
    hr = InboxItems._Item(vtValue, pvObject)
    if hr ~= S_OK then ShowCOMError(hr);return.

    ! Process a record by taking the pvObject which is a
    ! pointer to the interface returned by InboxItems._Item and
    ! query it using QueryInterface to see if it is a MailItem object,
    ! if not the loop skips this record and processes the next
    hr = E_FAIL
    if pvObject
      IUnk &= (pvObject)
      hr = IUnk.QueryInterface(address(IID__MailItem), pvObject)
      IUnk.Release()
    else
      hr = S_OK
    end
    if hr = S_OK
      ! Success - we have a MailItem so attach the Mail
      ! object to it and retrieve the fields
      hr = Mail.Attach(pvObject)
      if hr = S_OK
        clear(InboxQ)

        ! Get all the fields and add to the InboxQ.
        ! Use the _cstr
        ! helper function to convert BSTR values to cstring values

        hr = Mail.get_SenderName(pbstr)
        if hr = S_OK and pbstr
          sz &= _cstr(pbstr)
          OLIN:SenderName = clip(sz)
          dispose(sz)
        end
        hr = Mail.get_To(pbstr)
        if hr = S_OK and pbstr
          sz &= _cstr(pbstr)
          OLIN:ToList = clip(sz)
          dispose(sz)
        end

        hr = Mail.get_CC(pbstr)
        if hr = S_OK and pbstr
          sz &= _cstr(pbstr)
          OLIN:CCList = clip(sz)
          dispose(sz)
        end

        hr = Mail.get_BCC(pbstr)
        if hr = S_OK and pbstr
          sz &= _cstr(pbstr)
          OLIN:BCCList = clip(sz)
          dispose(sz)
        end

        hr = Mail.get_Subject(pbstr)
        if hr = S_OK and pbstr
          sz &= _cstr(pbstr)
          OLIN:Subject = clip(sz)
          dispose(sz)
        end

        hr = Mail.get_Body(pbstr)
        if hr = S_OK and pbstr
          sz &= _cstr(pbstr)
          OLIN:Body = clip(sz)
          dispose(sz)
        end

        add(InboxQ)

        ! Release the existing message so we can re-use it in the loop
        Mail.Release()
      end
    end
  end
end

! Release the interfaces to the objects we used

InboxItems.Release()
Inbox.Release()
Namespace.Release()
Outlook.Release()

return
```

```
ImportInboxLateBound procedure

progid              cstring('Outlook.Application')
szNmsp              cstring('MAPI')
IUnknownI           &IUnknown
IDispatchI          &IDispatch
fCloseOutlook       byte
sz                  &cstring

  code
    free(InboxQ)
    clear(WindowMessage)
    RecordsProcessed = 0
    RecordsToProcess = 0

    ! Use the PWDispatch.AttachExisting to connect to existing
    ! instance of the Outlook Application COM Object if available
    hr = DispApplication.AttachExisting(progid)
    if hr ~= S_OK
      ! The attach to existing instance failed so we must create one using
      ! the PWDispatch.CreateInstance
      hr = DispApplication.CreateInstance(progid)
      if hr ~= S_OK then ShowCOMError(hr);return.
      fCloseOutlook = true
    end

    ! VB Equivalent:  Outlook.GetNameSpace("MAPI")
    szMember = 'GetNamespace'
    hr = DispApplication.Invoke(szMember, DISPATCH_METHOD, |
                             _vt(szNmsp), vtIDisp)
    if hr ~= S_OK then ShowCOMError(hr);return.

    ! Attach the Namespace object to the interface pointer
    ! returned from GetNamespace
    hr = DispNamespace.Attach(vtIDisp)
    if hr ~= S_OK then ShowCOMError(hr);return.

    ! VB Equivalent:  Inbox = Namespace.GetDefaultFolder(olFolderInbox)
    lolFolderInbox = olFolderInbox
    szMember = 'GetDefaultFolder'
    hr = DispNamespace.Invoke(szMember, DISPATCH_METHOD, |
                             _vt(lolFolderInbox), vtIDisp)
    if hr ~= S_OK then ShowCOMError(hr);return.

    ! Attach the DispInbox object to the interface pointer
    ! returned from GetDefaultFolder
    hr = DispInbox.Attach(vtIDisp)
    if hr ~= S_OK then ShowCOMError(hr);return.

    ! VB Equivalent:  InboxItems = Inbox.Items
    szMember = 'Items'
    hr = DispInbox.Invoke(szMember, DISPATCH_PROPERTYGET, vtIDisp)
    if hr ~= S_OK then ShowCOMError(hr);return.

    ! Attach the InboxItems object to the interface pointer returned from Items
    hr = DispInboxItems.Attach(vtIDisp)
    if hr ~= S_OK then ShowCOMError(hr);return.

    ! Get the number of messages in the Inbox (DispInboxItems) collection
    szMember = 'Count'
    hr = DispInboxItems.Invoke(szMember, DISPATCH_PROPERTYGET, vtIDisp)
    if hr ~= S_OK then ShowCOMError(hr);return.

    RecordsToProcess = vtIDisp.iVal
    RecordsProcessed = 0
```

```
if RecordsToProcess
  loop
    RecordsProcessed += 1
    ! Check to see if we are done
    if RecordsProcessed > RecordsToProcess then
      if fCloseOutlook
        ! Close Outlook since we instantiated a new copy of it above
        szMember = 'Quit'
        hr = DispApplication.Invoke(szMember, DISPATCH_METHOD, vtIDisp)
        if hr ~= S_OK then ShowCOMError(hr);return.
      end
      break
    end
    Progress = RecordsProcessed / RecordsToProcess * 100
    WindowMessage = 'Processing message ' |
                    &RecordsProcessed&' of '&RecordsToProcess
    display

    ! VB Equivalent:  MailItem = DispInboxItems.Item(vtValue)
    szMember = 'Item'
    hr = DispInboxItems.Invoke(szMember, DISPATCH_PROPERTYGET, |
                           _vt(RecordsProcessed), vtIDisp)
    if hr ~= S_OK then ShowCOMError(hr);return.

    ! Process a record
    clear(InboxQ)

    ! Attach the DispMessage object to the interface pointer
    ! returned from Item(vtValue)
    hr = DispMessage.Attach(vtIDisp)
    if hr ~= S_OK then ShowCOMError(hr);return.

    ! Clarion Equivalent
    ! OLIN:SenderName = clip(?OLE{clip(MessageItem) & '.SenderName'})
    szMember = 'SenderName'
    hr = DispMessage.Invoke(szMember, DISPATCH_PROPERTYGET, vtIDisp)
    if hr ~= S_OK then ShowCOMError(hr);return.

    ! After we get the field we use the _cstr helper function
    ! to convert to a Clarion cstring
    sz &= _cstr(vtIDisp.iVal)
    OLIN:SenderName = clip(sz)
    dispose(sz)

    ! Clarion Equivalent
    !  OLIN:ToList = clip(?OLE{clip(MessageItem) & '.To'})
    szMember = 'To'
    hr = DispMessage.Invoke(szMember, DISPATCH_PROPERTYGET, vtIDisp)
    if hr ~= S_OK then ShowCOMError(hr);return.

    ! After we get the field we use the _cstr helper function
    ! to convert to a Clarion cstring
    sz &= _cstr(vtIDisp.iVal)
    OLIN:ToList = clip(sz)
    dispose(sz)

    ! Clarion Equivalent
    !  OLIN:CCList = clip(?OLE{clip(MessageItem) & '.CC'})
    szMember = 'CC'
    hr = DispMessage.Invoke(szMember, DISPATCH_PROPERTYGET, vtIDisp)
    if hr ~= S_OK then ShowCOMError(hr);return.

    ! After we get the field we use the _cstr helper function
    ! to convert to a Clarion cstring
    sz &= _cstr(vtIDisp.iVal)
    OLIN:CCList = clip(sz)
    dispose(sz)
```

```
        ! Clarion Equivalent
        !  OLIN:BCCList = clip(?OLE{clip(MessageItem) & '.BCC'})
        szMember = 'BCC'
        hr = DispMessage.Invoke(szMember, DISPATCH_PROPERTYGET, vtIDisp)
        if hr ~= S_OK then ShowCOMError(hr);return.

        ! After we get the field we use the _cstr helper function
        ! to convert to a Clarion cstring
        sz &= _cstr(vtIDisp.iVal)
        OLIN:BCCList = clip(sz)
        dispose(sz)

        ! Clarion Equivalent
        !  OLIN:Subject = clip(?OLE{clip(MessageItem) & '.Subject'})
        szMember = 'Subject'
        hr = DispMessage.Invoke(szMember, DISPATCH_PROPERTYGET, vtIDisp)
        if hr ~= S_OK then ShowCOMError(hr);return.

        ! After we get the field we use the _cstr helper function
        ! to convert to a Clarion cstring
        sz &= _cstr(vtIDisp.iVal)
        OLIN:Subject = clip(sz)
        dispose(sz)

        ! Clarion Equivalent
        !  OLIN:Body = clip(?OLE{clip(MessageItem) & '.Body'})
        szMember = 'Body'
        hr = DispMessage.Invoke(szMember, DISPATCH_PROPERTYGET, vtIDisp)
        if hr ~= S_OK then ShowCOMError(hr);return.

        ! After we get the field we use the _cstr helper function
        ! to convert to a Clarion cstring
        sz &= _cstr(vtIDisp.iVal)
        OLIN:Body = clip(sz)
        dispose(sz)

        add(InboxQ)

        ! Release the existing message so we can re-use it in the loop
        DispMessage.Release()

        ! Release the VariantFactory so we don't use too much of the cache when
        ! making calls using _vt() - maximum of 256 values allowed in
        ! the VariantFactory cache (configurable in PWVariantFactory.Construct
        ! method in pwcom.clw)
        VariantFactory.Release()
      end
    end

    ! Release the interfaces to the objects we used

    DispInboxItems.Release()
    DispInbox.Release()
    DispNamespace.Release()
    DispApplication.Release()

    return
```

```
NewContactEarlyBound procedure

Outlook                 TDOFApplication
Namespace               TDOFNameSpace
Contact                 TDOFContactItem
BStrFullName            PWBStr
BStrEmail1Address       PWBStr
fCloseOutlook           byte

  code
    ! Use the Win32 API call GetActiveObject to connect to existing
    ! instance of the Outlook Application COM Object if available
    hr = Outlook.AttachExisting(address(CLSID_Application), |
         address(IID__Application))
    if hr ~= S_OK
      ! The attach to existing instance failed so we must create one using
      ! Outlook.CreateInstance which calls the Win32 API call CoCreateInstance

      hr = Outlook.CreateInstance(address(CLSID_Application), |
           address(IID__Application), |
           bor(CLSCTX_LOCAL_SERVER, CLSCTX_INPROC_SERVER))
      if hr ~= S_OK then ShowCOMError(hr);return.

      ! We have created our own instance of Outlook so we
      ! should call the Outlook.Quit method at the end
      ! to close the application once we are done using it.
      fCloseOutlook = true
    end

    ! VB Equivalent:  Contact = Outlook.CreateItem(olContactItem)
    hr = Outlook.CreateItem(olContactItem, pvObject)
    if hr ~= S_OK then ShowCOMError(hr);return.

    ! The prototype for the Outlook.CreateItem is in outlook.inc and looks like
    ! PWApplication.CreateItem(OlItemType ItemType,
    ! *long pvObject),HRESULT,virtual  !**IDispatch pvObject
    ! In general you should use QueryInterface when the return type of
    ! a procedure is an IDispatch interface
    ! because we cannot assume the returned IDispatch is a dual interface.
    ! We have added a helper function to the PWCOMObject class called
    ! GetInterface which will handle the internal
    ! QueryInterface call.  The first parameter is pvObject which is the
    ! returned interface from Outlook.CreateItem.  The second parameter is
    ! address(IID__ContactItem) which is defined in oliid.inc.  It
    ! tells QueryInterface that we want to make sure we have a PWContactItem
    ! interface.  The third parameter is pvTemp which is a long containing the
    ! interface to the PWContactItem once GetInterface is finished.  The
    ! fourth parameter is set to true which tells us to release the
    ! IUnknown interface when done with it.
    ! The GetInterface call replaces the code below it which has been
    ! commented out and delimited with ----
    hr = Outlook.GetInterface(pvObject, address(IID__ContactItem), |
                              pvTemp, true)
    if hr ~= S_OK then ShowCOMError(hr);return.
    pvObject = pvTemp
    !-----------------------------------------------------------------------
    ! This code replaced as discussed above
    !IUnk &= (pvObject)
    !hr = IUnk.QueryInterface(address(IID__ContactItem), pvObject)
    !if hr ~= S_OK then
    !  IUnk.Release()
    !  ShowCOMError(hr)
    !  return
    !end
    !IUnk.Release()
    !-----------------------------------------------------------------------
    ! Attach the Contact object to the interface pointer
    ! returned from CreateItem
    hr = Contact.Attach(pvObject)
    if hr ~= S_OK then ShowCOMError(hr);return.

    ! VB Equivalent:  Contact.FullName = szFullName
    lLength = BStrFullName.Init(szFullName)
    hr = Contact.put_FullName(BStrFullName.GetStr())
    if hr ~= S_OK then ShowCOMError(hr);return.

    ! VB Equivalent:  Contact.Email1Address = szEmail1Address
    lLength = BStrEmail1Address.Init(szEmail1Address)
    hr = Contact.put_Email1Address(BStrEmail1Address.GetStr())
    if hr ~= S_OK then ShowCOMError(hr);return.

    ! VB Equivalent:  Contact.Save
    hr = Contact.Save()
    if hr ~= S_OK then ShowCOMError(hr);return.

    if fCloseOutlook
      ! Close Outlook since we instantiated a new copy of it above
      hr = Outlook.Quit()
    end

    ! Release the interfaces to the objects we used
    Namespace.Release()
    Outlook.Release()

    return


NewContactLateBound procedure

progid              cstring('Outlook.Application')
DispContact         PWDispatch
lolContactItem      long
fCloseOutlook       byte

  code
    ! Use the PWDispatch.AttachExisting to connect to existing
    ! instance of the Outlook Application COM Object if available
    hr = DispApplication.AttachExisting(progid)
    if hr ~= S_OK
      ! The attach to existing instance failed so we must create one using
      ! the PWDispatch.CreateInstance
      hr = DispApplication.CreateInstance(progid)
      if hr ~= S_OK then ShowCOMError(hr);return.
      fCloseOutlook = true
    end

    ! VB Equivalent:  Contact = Outlook.CreateItem(olContactItem)
    lolContactItem = olContactItem
    szMember = 'CreateItem'
    hr = DispApplication.Invoke(szMember, DISPATCH_METHOD, |
                                _vt(lolContactItem), vtIDisp)
    if hr ~= S_OK then ShowCOMError(hr);return.

    ! Attach the DispContact object to the IDispatch pointer
    ! returned from CreateItem
    hr = DispContact.Attach(vtIDisp)
    if hr ~= S_OK then ShowCOMError(hr);return.

    ! VB Equivalent:  Contact.FullName = szFullName
    szMember = 'FullName'
    hr = DispContact.Invoke(szMember, DISPATCH_PROPERTYPUT, |
                            _vt(szFullName), vtIDisp)
    if hr ~= S_OK then ShowCOMError(hr);return.
    ! VB Equivalent:  Contact.Email1Address = szEmail1Address
    szMember = 'Email1Address'
    hr = DispContact.Invoke(szMember, DISPATCH_PROPERTYPUT, |
                            _vt(szEmail1Address), vtIDisp)
    if hr ~= S_OK then ShowCOMError(hr);return.

    ! VB Equivalent:  Contact.Save
    szMember = 'Save'
    hr = DispContact.Invoke(szMember, DISPATCH_METHOD, vtIDisp)
    if hr ~= S_OK then ShowCOMError(hr);return.

    if fCloseOutlook
      ! Close Outlook since we instantiated a new copy of it above
      szMember = 'Quit'
      hr = DispApplication.Invoke(szMember, DISPATCH_METHOD, vtIDisp)
      if hr ~= S_OK then ShowCOMError(hr);return.
    end

    ! Release the interfaces to the objects we used
    DispContact.Release()
    DispNamespace.Release()
    DispApplication.Release()

    return


NewAppointmentEarlyBound procedure

Outlook                 TDOFApplication
Appointment             TDOFAppointmentItem
BStrSubject             PWBStr
fCloseOutlook           byte

  code
    if ~szApptSubject or ~ApptStartDate or ~ApptStartTime |
    or ~ApptEndDate or ~ApptEndTime
      message('All Appointment fields are required','Required
Fields',ICON:Exclamation)
      return
    end

    hr = Outlook.AttachExisting(address(CLSID_Application),
address(IID__Application))
    if hr ~= S_OK
      hr = Outlook.CreateInstance(address(CLSID_Application), |
           address(IID__Application), bor(CLSCTX_LOCAL_SERVER,
CLSCTX_INPROC_SERVER))
      if hr ~= S_OK then ShowCOMError(hr);return.
      fCloseOutlook = true
    end

    ! Create an appointment item and then call GetInterface, which calls
QueryInterface internally to ensure
    ! that we have the IID__AppointmentItem interface.  Once we get that
interface we attach the Appointment
    ! object to it.
    hr = Outlook.CreateItem(olAppointmentItem, pvObject)
    if hr ~= S_OK then ShowCOMError(hr);return.
    hr = Outlook.GetInterface(pvObject, address(IID__AppointmentItem), pvTemp,
true)
    if hr ~= S_OK then ShowCOMError(hr);return.
    pvObject = pvTemp

    hr = Appointment.Attach(pvObject)
    if hr ~= S_OK then ShowCOMError(hr);return.
    ! Initialize the BSTR containing the subject for the appointment
    BstrSubject.Init(szApptSubject)
    hr = Appointment.put_Subject(BstrSubject.GetStr())
    if hr ~= S_OK then ShowCOMError(hr);return.

    ! Initialize the COMTime class with the Clarion start date and time
    COMTime.Init(ApptStartDate, ApptStartTime)
    hr = Appointment.put_Start(COMTime.GetAsCOleDateTime())
    if hr ~= S_OK then ShowCOMError(hr);return.

    ! Initialize the COMTime class with the Clarion end date and time
    COMTime.Init(ApptEndDate, ApptEndTime)
    hr = Appointment.put_End(COMTime.GetAsCOleDateTime())
    if hr ~= S_OK then ShowCOMError(hr);return.

    ! Save the Appointment
    hr = Appointment.Save()

    if fCloseOutlook
      hr = Outlook.Quit()
    end


NewTaskEarlyBound procedure

Outlook                 TDOFApplication
Task                    TDOFTaskItem
BStrSubject             PWBStr
fCloseOutlook           byte

  code
    if ~szTaskSubject or ~DueDate or ~DueTime
      message('All Task fields are required','Required
Fields',ICON:Exclamation)
      return
    end

    hr = Outlook.AttachExisting(address(CLSID_Application),
address(IID__Application))
    if hr ~= S_OK
      hr = Outlook.CreateInstance(address(CLSID_Application), |
           address(IID__Application), bor(CLSCTX_LOCAL_SERVER,
CLSCTX_INPROC_SERVER))
      if hr ~= S_OK then ShowCOMError(hr);return.
      fCloseOutlook = true
    end

    ! Create a task item and then call GetInterface, which calls QueryInterface
internally to ensure
    ! that we have the IID__TaskItem interface.  Once we get that interface we
attach the Task
    ! object to it.
    hr = Outlook.CreateItem(olTaskItem, pvObject)
    if hr ~= S_OK then ShowCOMError(hr);return.
    hr = Outlook.GetInterface(pvObject, address(IID__TaskItem), pvTemp, true)
    if hr ~= S_OK then ShowCOMError(hr);return.
    pvObject = pvTemp

    hr = Task.Attach(pvObject)
    if hr ~= S_OK then ShowCOMError(hr);return.

    BstrSubject.Init(szTaskSubject)
    hr = Task.put_Subject(BstrSubject.GetStr())
    if hr ~= S_OK then ShowCOMError(hr);return.
    ! Initialize the COMTime class with the Clarion due date and time
    COMTime.Init(DueDate, DueTime)
    hr = Task.put_DueDate(COMTime.GetAsCOleDateTime())
    if hr ~= S_OK then ShowCOMError(hr);return.

    ! Save the Appointment
    hr = Task.Save()

    if fCloseOutlook
      hr = Outlook.Quit()
    end


ShowCOMError procedure(HRESULT hrparam)

lCOMErrorlen        long
szCOMError          cstring(255)
szErrorCaption      cstring('COM Error')

  code
    ! We must pass in the length of the locally declared cstring
    lCOMErrorlen = len(szCOMError)
    if COMError.GetError(szCOMError, lCOMErrorlen, hrparam)
      if szCOMError
        szCOMError = clip(szCOMError) & ': hr = '&hrparam
      else
        szCOMError = 'Error unknown: hr = '&hrparam
      end
      ! Use the Win32 MessageBox API call to display the error message
      messagebox(0{prop:handle}, address(szCOMError), |
                 address(szErrorCaption), MB_ICONEXCLAMATION)
    end

    return
```

Links

We have provided several links which we found helpful in the development of the OutlookFUSE product and in its daily use. If you find helpful resources in your development process, or have questions for us related to OutlookFUSE, we encourage you to submit them to our support forum listed below.

ThinkData Support Forum
http://www.thinkdata.com/forum/

Microsoft Office Development Support Center
http://support.microsoft.com/support/officedev/

Outlook Spy
http://www.dimastr.com/outspy/

Outlook Exchange
http://www.outlookexchange.com

MicroEye Outlook Knowledge Base (excellent object model diagram here!)
http://www.microeye.com/resources/outlkb.htm

Slipstick Outlook Code Samples
http://www.slipstick.com/dev/code/index.htm

Summary

The examples we have shown only scratch the surface of the potential of the OutlookFUSE product. There are approximately 70 interfaces in the olint.inc include file for the Microsoft Outlook 2003 object model and the entire wrapper totals nearly 20,000 lines of code. OutlookFUSE is the only product of its kind for Clarion 5.5 and Clarion 6 providing native early and late binding to Outlook with a stable and efficient COM layer from Plugware Solutions. The sky's the limit!

We hope you enjoy OutlookFUSE and invite you to join ThinkData's support forum to ask questions and get tips on using it to its fullest potential. Thanks for your interest in OutlookFUSE and keep your eyes open for other COM automation products from our FUSE product line.