CIMAppObject™ Notifications Package User Guide

Last Updated: May 12, 2000

Document ID:

Table of Contents

Notification Package Overview

The Cimetrix Notifications Package solves a number of problems and issues facing all programmers today. It is responsible for allowing components to notify each other of state or data transitions. Transitions may include errors, events, exceptions, warnings, status, trace information, internal data changes, etc. This generic notification mechanism allows any form of data to be communicated between components. Components may be COM (Component Object Model) objects, C++ classes, Visual Basic clients, Windows controls or even functionally oriented blocks of C code.

The Cimetrix Notifications Package and its internal functionality are the foundation of component communication in the next generation of Cimetrix products. Programmers who need a mechanism for components to communicate with each other should use this package if they want to take full advantage of the Cimetrix product features.

The package provides a number of features implemented using COM interfaces. Microsoft's COM technology is rapidly dominating the Windows programming industry today. COM provides a definition, but not an implementation of how components and objects communicate with each other. The Cimetrix Notifications Package extends that communication definition and actually provides a simple and easy to use implementation while hiding the complexities of the underlying COM sub-system. It is important to understand this distinction: COM defines how components *may* interact, but COM does not provide an implementation as to how they *will* interact. The Cimetrix Notifications Package implements a real, concrete, and very powerful solution. It provides a completely extensible and generic mechanism for component communication.

The Cimetrix Notifications Package has also been carefully designed and implemented in order to leverage Microsoft's next version of COM. COM+ will extend the current COM standard and provide additional and more powerful features. The new inheritance and event handling features in COM+ will make the Cimetrix Notifications Package even more powerful.

Clients who wish to use the Notifications Package may not be familiar with or comfortable with COM. Therefore, the Notifications Package provides a number of helper classes that completely hide the details of COM. Clients are free to use the Notifications Package via its COM interfaces and/or via its C++ class library.

The Cimetrix Notifications Package also leverages state of the art Object-Oriented (OO) techniques. It implements a well-known and very popular OO design pattern. The Subject-Observer (a.k.a.: Publish-Subscribe) design pattern[1-293] defines the relationship between components and the roles and

[1] *Design Patterns – Elements of Reusable Object-Oriented Software*, Published by Addison-Wesley, Authors: Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides

responsibilities of a subject and an observer. The Cimetrix implementation of this design pattern provides a number of enhancements and new features as well.

Definitions

To better understand the features of the Cimetrix Notifications Package, a number of terms must be understood. The following definitions are integral parts of the Notifications Package:

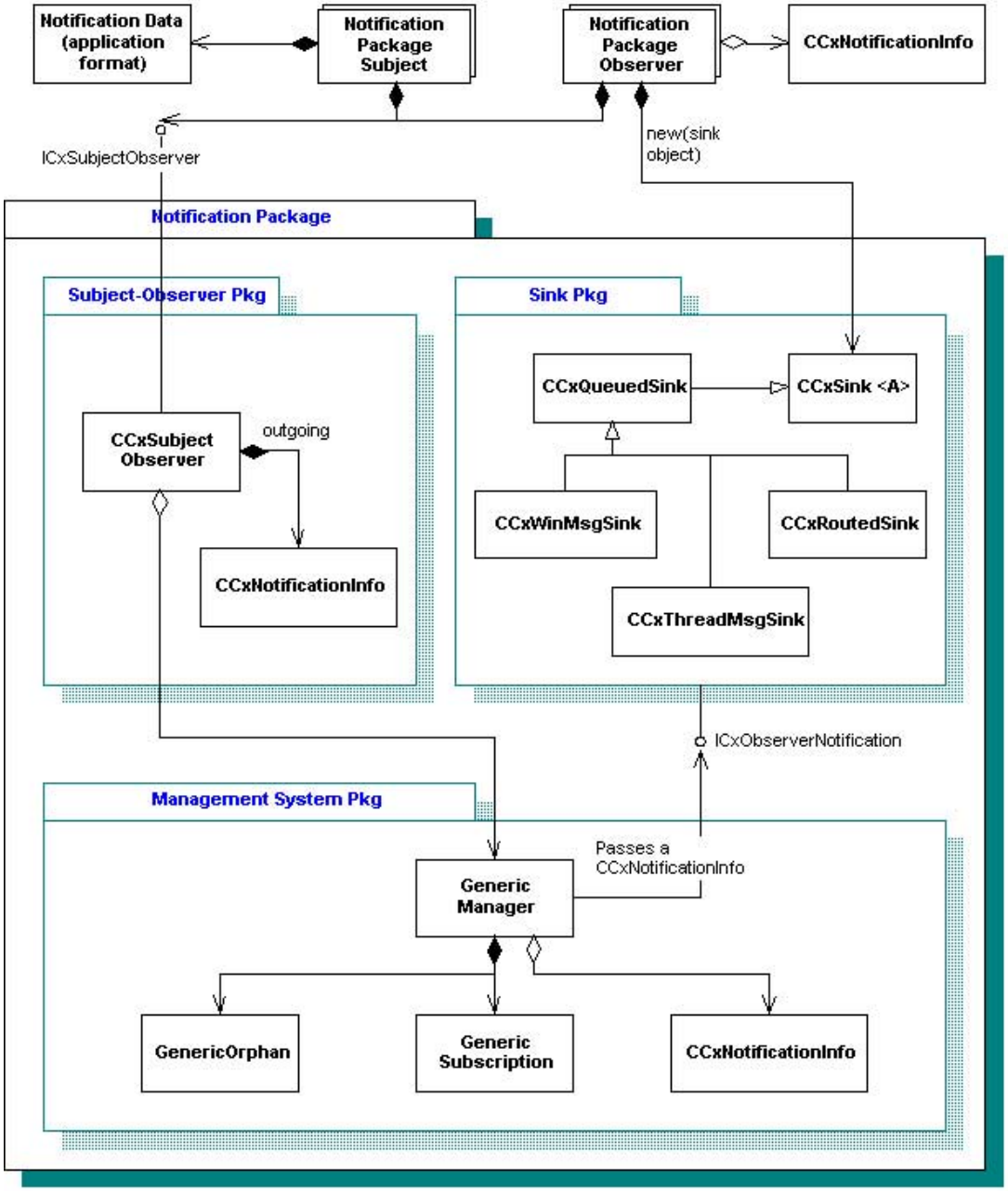
- ***Subject***: Any component that communicates its internal state to other components. The Subject component is responsible for notifying its Observer(s) of internal state transitions.
- ***Observer***: Any component that is dependent on or observes a Subject and is interested in being notified of the Subject's state or data transitions. An Observer is responsible for reacting appropriately to a Subject notification.
- ***Subscription***: The logical connection between a Subject and Observer. Observers subscribe to one or more Subjects. Subjects can have one or more Observers.
- ***Orphan Subscription***: A subscription from an Observer to a Subject that has not yet been instantiated. Once the Subject is instantiated, the orphan subscription automatically becomes a regular subscription.
- ***Notification***: The act of sending details and data, from a Subject to an Observer.
- ***Management Sub-system*:** Responsible for maintaining lists of subscriptions, orphans, component names, thread-safety, and for passing notifications from a Subject to its Observer(s) via the Observer Sink.
- ***Observer Sink:*** Each Observer MUST implement a Sink object. The Sink Object receives notifications from the Management Subsystem.
- ***Windows NT Events:*** The Notifications Package uses Windows NT Events for its own internal communication as well as for broadcasting status to external entities. NT Events provide an extremely fast method of communication between applications, objects, or any other component. Unfortunately, NT events do not carry data with them. NT Events can be thought of as a Notification *without* data. Conversely, a notification can be thought of as an NT event *with* data. Developers are free to use NT Events for their own communication or they can rely on the power of the Notifications package and its own internal use of Windows NT Events.

Conceptual Overview

The Notification Package is broken into three major functional areas:

1. Subject-Observer Subpackage
2. Sink Subpackage
3. Management System Subpackage

These sub-packages and their major components are illustrated in the following UML diagram. This diagram is intended to provide a high level view of data flow through the Notifications Package. It is not meant to show all of the classes that are exposed and available in the Notifications Package.



The major components illustrated in the Conceptual Overview Diagram above are defined as follows:

- ***Notification Data (Application Format)*:** This represents any format of data that lives inside clients of the Notifications Package.
- ***Notification Package Subject***: Any application or object that wishes to send notifications.
- ***Notification Package Observer***: Any application or object that wishes to receive notifications. SPECIAL NOTE: The Subject and Observer can be the same object.
- ***CCxSubjectObserver*:** This class implements the ICxSubjectObserver interface. This is the interface that provides for Subject-Observer subscription requests, un-subscribing, error handling, and notification information passing. This is the "front-end" to the GenericManager Management System. Note that the relationship to the CCxNotificationInfo class is an outgoing relationship. This means that the notification information came into the CCxSubjectObserver and is going out to the GenericManager in the form or a CCxNotificationInfo class.
- ***Sink Package*:** This package implements the ICxObserverNotification interface. This is where Advise Sinks are defined. This is the "back-end" of the system where notification data from a Subject gets routed back to an Observer. The notification information is coming into the Sink package from the management system. The Sink package defines and implements a number of different sink types. The specific kind of derived class that the Observer instantiates determines how notification data gets back into the Observer. Each and every Sink object in the system gets its own thread created with it. The creator of the sink object has the ability to control the priority of the thread. The sink's thread receives each of the notifications and depending on its derived class capabilities, may forward the notification data directly to the Observer, or may wait for the Observer to explicitly request the data.
- ***CCxSink:*** Abstract base class that all sink objects are derived from. All communication from the Management System to the sink object is through the abstract interface defined by the CCxSink Object. The CCxSink object's COM interface is ICxObserverNotification.
- ***CCxRoutedSink:*** A routed or callback sink object. An Observer uses this sink object if the Observer wants a callback containing the notification information.
- ***CCxWinMsgSink:*** A Windows message sink. Observers who would like to receive their notification information passed via a call to PostMessage use the message sink. The Observer can use it's own Windows message queue to receive the notification objects.
- ***CCxThreadMsgSink:*** A thread message sink. Observers who would like to receive their notification information passed via a call to PostThreadMessage use the message sink.
- ***CCxQueuedSink:*** Observers who want to request the notification information from the sink as needed use a queued sink. Observers using this sink MUST explicitly "pull" the data out of the sink object. This feature would be useful for Observers that prefer to retrieve notification data when they are ready to process it. For example, an Observer may require that some processing or an event occur *before* a notification is processed.
- ***CCxNotificationInfo*:** This is the notification data. This object contains the details of a Subject's transition. The notification can be made up of flags,

data, or details. A notification may also be telling an Observer that a Subject has been destroyed in the system.

- ***GenericSubscription***: A single connection from an Observer to a Subject. Note that an Observer may have multiple subscriptions to the same subject.
- ***GenericOrphan***: An orphan is a subscription request to a Subject that does not yet exist in the system. By allowing Orphans to be created, timing issues regarding the order of instantiation can be addressed and handled. In other words, Observers do not need to know when a Subject is alive, only that the Subject may become alive at some future point.
- ***GenericManager*:** The GenericManager is the "middle" of the Notifications Package. The GenericManager performs the following functions:
    1. The GenericManager maintains the list of subscriptions between subjects and observers. This alleviates the Subject from having to maintain and loop through its own list of observers when sending a notification. The GenericManager handles this looping and the dispersal of notifications to all observers who may be interested in the notification.
    2. Maintains the list of orphan subscriptions.
    3. Maintains thread safety between subjects and observers. Notifications will most likely consist of data that initiates in one thread and gets passed around to other threads. The GenericManager efficiently copies the data from the Subject and passes the copied data to appropriate Observers. The GenericManager has its own high-priority thread that receives subscription and notification requests.
    4. Forwarding a notification from a Subject to ALL subscribed Observers. The Subject sends a single notification that may get propagated to numerous Observers. The Subject does not have to worry about how many Observers it has. The GenericManager maintains these relationships.
    5. Keeps track of object names and enforces unique names in the notification system.

To COM or not to COM

Many clients evaluating the Notifications Package may not be familiar with using COM in a C++ application.

The Notifications Package provides two distinct and interchangeable ways for application programmers to create Subjects and Observers and then have those objects communicate with each other via the Notifications Package. Clients using the Notifications Package can:

1. Use COM exclusively to create Subjects and Observers.
2. Rely on the Notifications Package C++ classes only. The developer requires absolutely no COM knowledge. The C++ classes provided by the Notifications Package hide everything beneath a pure C++ class hierarchy.
3. Use a combination of C++ and COM to create Subjects and Observers.

Fundamental to the Notifications Package are Subjects and Observers. Creating Subjects and Observers ultimately requires the creation of COM objects. Whether using the COM API's to create the COM objects or using the Notifications Package C++ class hierarchy classes, at the core, COM is used. The only difference is that the Notifications Package C++ class hierarchy hides all of the COM issues and API's from the application programmer.

The subsections below show exactly how to create Subjects and Observers using either mechanism.

Using COM

For application programmers familiar with COM interfaces, CoCreateInstance, COM threading models, and various other COM concepts, it should be very easy to use the Notifications Package.

When using COM to create an object that acts only as a Subject, only the following step is required:

1. Using CoCreateInstance or CoCreateInstanceEx, create an ICxSubjectObserver COM object.

When using COM to create an object that is both a Subject and an Observer, the following steps are required:

1. Using CoCreateInstance or CoCreateInstanceEx, create an ICxSubjectObserver COM object.
2. Using the C++ new operator, C's malloc, ATL's CComObject helper class, or a number of other mechanisms, create a sink object that derives from the Notifications Package CCxSink class.
3. Depending on the type of sink object created in step 3 above, create either a hidden window object, a worker thread object, or a notification route object. For examples of these objects refer to the CCxNotificationWnd, CCxNotificationThread, or the CCxNotificationRoute classes respectively. Note that if a client creates a CCxQueuedSink object in step 2 above, this step is not necessary.
4. Finally, connect the sink object to the ICxSubjectObserver COM object.

The following source code example illustrates the creation of a "routed" Observer. This Observer will use a CCxNotificationRoute as it's callback mechanism for receiving a notification.

```
HRESULT CMyClass::CreateRoutedSubjectObserver(CString strName)
{
      HRESULT hr;

      // Our class definition defines m_pSO and m_pSink as
      // follows:
      // ICxSubjectObserver   *m_pSO;
      // CCxQueuedSink *m_pSink;

      // ***STEP 1***
      hr = CoCreateInstance(CLSID_CCxSubjectObserver,
            NULL,
            (m_bInProcess?  CLSCTX_INPROC_SERVER : \
                            CLSCTX_LOCAL_SERVER),
            IID_ICxSubjectObserver,
           reinterpret_cast<void**>(&m_pSO));

      if (FAILED(hr))
      {
            m_pSO = NULL;

            // Poor debugging, but you get the idea...
            ASSERT(FALSE);
      }
```

```
	// Create the sink and connect it to the subject-observer
	// This source code uses ATL's CComObject to easily create
	// a sink object.  There are a number of other ways this
	// could be done.  For more information refer to MSDN and
	// search for "Connection point sinks".
	if( m_pSO )
	{
		// ***STEP 2***
	hr = CComObject<CCxRoutedSink>::CreateInstance(&m_pSink);
		if (FAILED(hr))
			ASSERT(FALSE);

		// ***STEP 3***
		// We can do this because this object inherits from
		// CxNotificationRoute.  This is what tells
		// the notification system how to get back to
		// us and call our OnNotifyNOTIFY(),
		// OnNotifyNOTIFY_SUBJECTBROKEN(),
		// OnThreadInit(), and OnThreadCleanup() methods
		m_pSink->SetNotificationRoute(this);

		// We use the ATL "AtlAdvise" call to connect the
		// ICxSubjectObserver to the sink
		// ***STEP 4***
		DWORD dwCookie = -1;
		hr = AtlAdvise(m_pSubjectObserver,
				m_pSink ->GetUnknown(),
				IID_ICxObserverNotification,
				&dwCookie);

		if (FAILED(hr))
			ASSERT(FALSE);

		// SPECIAL NOTE:  We haven't called SetMyName()
		//  yet!!! Better do it now
		hr = m_pSO->SetMyName(CComBSTR(strName),
					MY_PERSONAL_PKGID);

	}

	return hr;
}
```

Note the comments in the source code:
`// ***STEP n***`
Each of these comments refers to the steps outlined at the beginning of this section. The block of code following the "***STEP n***" text implements the required step.

The following source code example illustrates the creation of an Observer that prefers to receive notifications via a window.

```
HRESULT CMyClass::CreateWindowSubjectObserver(CString strName)
{
	HRESULT hr;

	// Our class definition defines m_pSO and m_pSink as
```

```
// follows:
// ICxSubjectObserver   *m_pSO;
// CCxQueuedSink *m_pSink;

// ***STEP 1***
hr = CoCreateInstance(CLSID_CCxSubjectObserver,
        NULL,
        (m_bInProcess?  CLSCTX_INPROC_SERVER : \
                        CLSCTX_LOCAL_SERVER),
        IID_ICxSubjectObserver,
        reinterpret_cast<void**>(&m_pSO));

if (FAILED(hr))
{
      m_pSO = NULL;

      // Poor debugging, but you get the idea...
      ASSERT(FALSE);
}

// Create the sink and connect it to the subject-observer
// This source code uses ATL's CComObject to easily create
// a sink object.  There are a number of other ways this
// could be done.  For more information refer to MSDN and
// search for "Connection point sinks".
if( m_pSO )
{
      // ***STEP 2***
      CComObject<CCxWinMsgSink> *pNotifier;

hr = CComObject<CCxWinMsgSink>::CreateInstance(&pNotifier);
      if (FAILED(hr))
            ASSERT(FALSE);

      // Now, tell the sink how to get data back to
      // us.  This is what tells the notification
      // system how to send the notification
      // back to us.  Here, we are telling the
      // notification system to send the notification
      // directly to our main dialog window.  We
      // will "catch" the notification and handle it
      // there.  We could make a call like the following
      // that's commented out if we wanted
      // the notification system to forward the
      // notification directly to our main MFC dialog
      // window.
      // pNotifier->SetWindowHandle(
      //                AfxGetMainWnd()->m_hWnd );

      // Instead of simply sending our notification
      // data directly to an MFC window, we
      // "catch" it with another window, and then
      // have the m_pWnd window format it and forward it
      // to the dialog window.  Of course, using the
      // CCxNotificationWnd is entirely optional.  We
      // could have used our application's main window
      // if we wanted.
```

```
            // ***STEP 3***
            m_pWnd = new CCxNotificationWnd;
            m_pWnd->Create(NULL, strName, WS_CHILD ,
                   CRect(0,0,0,0), AfxGetMainWnd(), 0, NULL);
            ASSERT(m_pWnd->m_hWnd);
            pNotifier->SetWindowHandle( m_pWnd->m_hWnd );

            // We use the ATL call to connect the
            // ICxSubjectObserver to the sink
            // ***STEP 4***
            DWORD dwCookie = -1;
            hr = AtlAdvise(m_pSO,pNotifier->GetUnknown(),
                        IID_ICxObserverNotification,&dwCookie);
            if (FAILED(hr))
                  ASSERT(FALSE);

            // SPECIAL NOTE:  We haven't called SetMyName()
            //  yet!!! Better do it now
            hr = m_pSO->SetMyName(CComBSTR(strName),
                              MY_PERSONAL_PKGID);
      }


      return hr;
}
```

Once again, note the comments in the source code:
`// ***STEP n***`
Each of these comments refers to the steps outlined at the beginning of this section. The block of code following the "***STEP n***" text implements the required step.

The following source code example illustrates the creation of an Observer that prefers to receive notifications via a worker thread.

```
HRESULT CMyClass::CreateThreadSubjectObserver(CString strName)
{
      HRESULT hr;

      // Our class definition defines m_pSO and m_pSink as
      // follows:
      // ICxSubjectObserver   *m_pSO;
      // CCxQueuedSink *m_pSink;

      // ***STEP 1***
      hr = CoCreateInstance(CLSID_CCxSubjectObserver,
              NULL,
              (m_bInProcess?  CLSCTX_INPROC_SERVER : \
                              CLSCTX_LOCAL_SERVER),
              IID_ICxSubjectObserver,
              reinterpret_cast<void**>(&m_pSO));

      if (FAILED(hr))
      {
            m_pSO = NULL;

            // Poor debugging, but you get the idea...
```

```
            ASSERT(FALSE);
      }

      // Create the sink and connect it to the subject-observer
      // This source code uses ATL's CComObject to easily create
      // a sink object.  There are a number of other ways this
      // could be done.  For more information refer to MSDN and
      // search for "Connection point sinks".
      if( m_pSO )
      {
            // ***STEP 2***
            CComObject<CCxThreadMsgSink> *pNotifier;

      hr = CComObject<CCxWinMsgSink>::CreateInstance(&pNotifier);
            if (FAILED(hr))
            {
                  ASSERT(FALSE);
                  return E_FAIL;
            }

            // ***STEP 3***
            // Now, tell the sink how to get data back to
            // us.  This is what tells the notification
            // system how to send the notification back to us
            m_pThread = new CCxNotificationThread;

            If (!m_pThread)
            {
                  ASSERT(FALSE);
                  return E_FAIL;
            }

            DWORD dwThreadID = m_pThread->GetThreadID();
            BOOL bRet = pNotifier->SetThreadHandle(dwThreadID);
            ASSERT(bRet);

            // We use the ATL call to connect the
            // ICxSubjectObserver to the sink
            // ***STEP 4***
            DWORD dwCookie = -1;
            hr = AtlAdvise(m_pSO,pNotifier->GetUnknown(),
                        IID_ICxObserverNotification,&dwCookie);
            if (FAILED(hr))
            {
                  ASSERT(FALSE);
                  return E_FAIL;
            }

            // SPECIAL NOTE:  We haven't called SetMyName()
            //  yet!!! Better do it now
            hr = m_pSO->SetMyName(CComBSTR(strName),
                              MY_PERSONAL_PKGID);
      }

      return hr;
}
```

Using C++ Classes

For application programmers who are comfortable with C++ and class libraries, the Notifications Package provides an extremely easy to use and powerful solution. The Notifications Package provides a number of C++ classes that wrap and completely hide COM from the user.

Prior to actually instantiating one of the C++ classes, of course it is necessary to determine the needed functionality. There are a number of C++ classes that provide Subject and Observer capabilities. There is only one kind of Subject, but there are a number of Observers. Observer choices include a routed Observer, a windows message Observer, a thread message Observer, or a simple queued Observer. Refer to the C++ Subjects and Observers section for information on the Subject and Observer helper class capabilities.

When using the Notifications Package C++ classes to create an object that acts only as a Subject, only the following step is required:

1. Instantiate the CCxSubjectOnly class.

When using the Notifications Package C++ classes to create an object that is both a Subject and an Observer, the following steps are required:

1. Determine the type of Observer that is needed by your application. Depending on the type of Observer, be prepared with a CCxNotificationRoute pointer, an HWND, or a DWORD thread ID. Refer to the CCxNotificationWnd hidden window object, the CCxNotificationThread worker thread object, or the CCxNotificationRoute object for information on how you can create your own window, thread, or route.
2. Instantiate the CCxQueuedSubjectObserver, CCxRoutedSubjectObserver, CCxWinMsgSubjectObserver, or CCxThreadMsgSubjectObserver and pass the appropriate values in the constructor. Refer to each class for the constructor parameters.

The following source code example illustrates the creation of a CCxSubjectOnly object:

```
HRESULT CMyClass::CreateSubjectOnly(CString strName)
{
	// Initialize our member variable.  Note that m_pObj
	// is defined in our class definition as:
	// CCxSubjectOnly *m_pObj;
	m_pObj = new CCxSubjectOnly(strName);
	ASSERT(m_pObj);
}
```

The following source code example illustrates the creation of a CCxQueuedSubjectObserver object:

```
HRESULT CMyClass::CreateQueuedSubjectObserver(CString strName)
{
	// Initialize our member variable.  Note that m_pObj
	// is defined in our class definition as:
	// CCxQueuedSubjectObserver *m_pObj;
	m_pObj = new CCxQueuedSubjectObserver(strName);
	ASSERT(m_pObj);
```

```
}
```

The following source code example illustrates the creation of a CCxRoutedSubjectObserver object:

```
HRESULT CMyClass::CreateRoutedSubjectObserver(CString strName)
{
      // Assume that CMyClass inherits from
      // CCxNotificationRoute and implements the
      // necessary virtual functions.

      // Initialize our member variable.  Note that m_pObj
      // is defined in our class definition as:
      // CCxRoutedSubjectObserver *m_pObj;
      m_pObj = new CCxRoutedSubjectObserver(this, strName);
      ASSERT(m_pObj);
}
```

The following source code example illustrates the creation of a CCxThreadMsgSubjectObserver object. Note that the creation of a CCxNotificationThread is optional if we want to use a different worker thread that has already been created. The thread ID of an existing worker thread could be used as the first parameter of the CCxThreadMsgSubjectObserver constructor. This example assumes there is no other worker thread and that we need to create our own.

```
HRESULT CMyClass::CreateThreadMsgSubjectObserver(CString strName)
{
      // Assume that we have a member variable for our
      // worker thread declared as follows:
      // CCxNotificationThread *m_pThread;
      m_pThread = new CCxNotificationThread;
      ASSERT(m_pThread && m_pThread->GetThreadID());

      // Initialize our member variable.  Note that m_pObj
      // is defined in our class definition as:
      // CCxThreadMsgSubjectObserver *m_pObj;
      m_pObj = new CCxThreadMsgSubjectObserver(
                              m_pThread->GetThreadID(),
                              strName);
      ASSERT(m_pObj);
}
```

The following source code example illustrates the creation of a CCxWinMsgSubjectObserver object. Note that the creation of a CCxNotificationWnd is optional if we want to use a different HWND. For example, if CMyClass already inherits from a CWnd or a CDialog, instead of creating a CCxNotificationWnd, we could use the CMyClass::m_hWnd member variable as the first parameter to the constructor in CCxWinMsgSubjectObserver.

```
HRESULT CMyClass::CreateWinMsgSubjectObserver(CString strName)
{
      // Assume that we have a member variable for our
      // hidden window declared as follows:
```

```
        // CCxNotificationWnd *m_pWnd;
        m_pWnd = new CCxNotificationWnd;
        m_pWnd->Create(NULL,
                        strName,
                        WS_CHILD ,
                        CRect(0,0,0,0),
                        AfxGetMainWnd(),
                        0,
                        NULL);
        ASSERT(m_pWnd->m_hWnd);

        // Initialize our member variable.  Note that m_pObj
        // is defined in our class definition as:
        // CCxWinMsgSubjectObserver *m_pObj;
        m_pObj = new CCxWinMsgSubjectObserver(m_pWnd->m_hWnd,
                                              strName);
        ASSERT(m_pObj);
    }
```

C++ Subjects and Observers

The Notifications Package provides a number of helper classes that can be used to create Subjects and Observers. These helper classes require absolutely no COM knowledge in order to create them. The purpose of this section is to help the reader determine which C++ helper class to use.

This section contains a class hierarchy diagram (below) that illustrates the relationship between all of the helper classes. Note that the diagram shows Subjects using the ICxSubjectObserver COM interface directly and also using the CCxSubjectOnly class. The diagram also shows Observers using the ICxSubjectObserver COM interface and using the helper "Observer" classes (CCxQueuedSubjectObserver, CCxRoutedSubjectObserver, CCxWinMsgSubjectObserver, and CCxThreadMsgSubjectObserver). Remember that Subjects and Observers are free to use either method (COM or C++). For details and source code examples on how to use COM versus C++, refer to the To COM or Not to COM section.

Before discussing the different kinds of Subjects and Observers and when they should be used, let's review the flow of data from a Subject to an Observer. Remember that Subjects send notifications and Observers receive notifications. When a Subject sends a notification, the following steps occur:

1. A Subject sends a notification to the Notifications Package Management System. The Subject has no idea if there are any Observers subscribed to it.
2. The Notifications Package Management System determines if there are any subscribed Observers. If so, the Management System forwards the notification object on to the Observer's sink object.
3. An Observer's sink object receives the notification from the Management System. It is important to understand that the Observer does not yet "hold" the notification, the Observer's sink object holds the notification in its internal queue. Remember that all sink objects inherit from CCxQueuedSink. Therefore, they all have an internal queue.
4. If the Observer's sink object is a CCxRoutedSink, a CCxWinMsgSink, or a CCxThreadMsgSink, the sink physically delivers the notification to the Observer object. Special Note: if the Observer's sink object is a CCxQueuedSink, then it is the responsibility of the Observer to "pull" the notification out of the CCxQueuedSink queue.

The following C++ classes can be instantiated by a client of the Notifications Package.

- **CCxSubjectOnly**: Use this class if you only need to send notifications.
- **CCxQueuedSubjectObserver**: Use this class if you would like to *send or receive* notifications and have the notifications "pile up" in a queue. The creator of the CCxQueuedSubjectObserver object must pull notifications out of the queue.
- **CCxRoutedSubjectObserver**: Use this class if you would like to *send or receive* notifications in a queue and then have each notification automatically forwarded on to a handler function that you provide. The notification is automatically forwarded to an object that you provide that inherits from CCxNotificationRoute.

- **CCxWinMsgSubjectObserver**: Use this class if you would like to *send or receive* notifications in a queue and then have each notification automatically forwarded on to a window procedure that you provide via an HWND. The notification is automatically forwarded to a window with a windows message loop. The window can either be an existing application window or a window created using the CCxNotificationWnd helper class. The window it gets forwarded to is determined by the creator of the CCxWinMsgSubjectObserver and by the HWND passed in its constructor.
- **CCxThreadMsgSubjectObserver**: Use this class if you would like to *send or receive* notifications in a queue and then have each notification automatically forwarded on to a worker thread that you provide via a thread ID. Worker threads can be created via the WIN32 CreateThread method or by using the CCxNotificationThread helper class. The notification is automatically forwarded to the worker thread with a message loop. The thread it gets forwarded to is determined by the creator of the CCxThreadMsgSubjectObserver and by the DWORD thread ID passed in its constructor.

The following class hierarchy diagram illustrates the relationships between the Subject and Observer classes and the methods in each class. Each of these classes is publicly exported by the CxNotifyClient DLL.

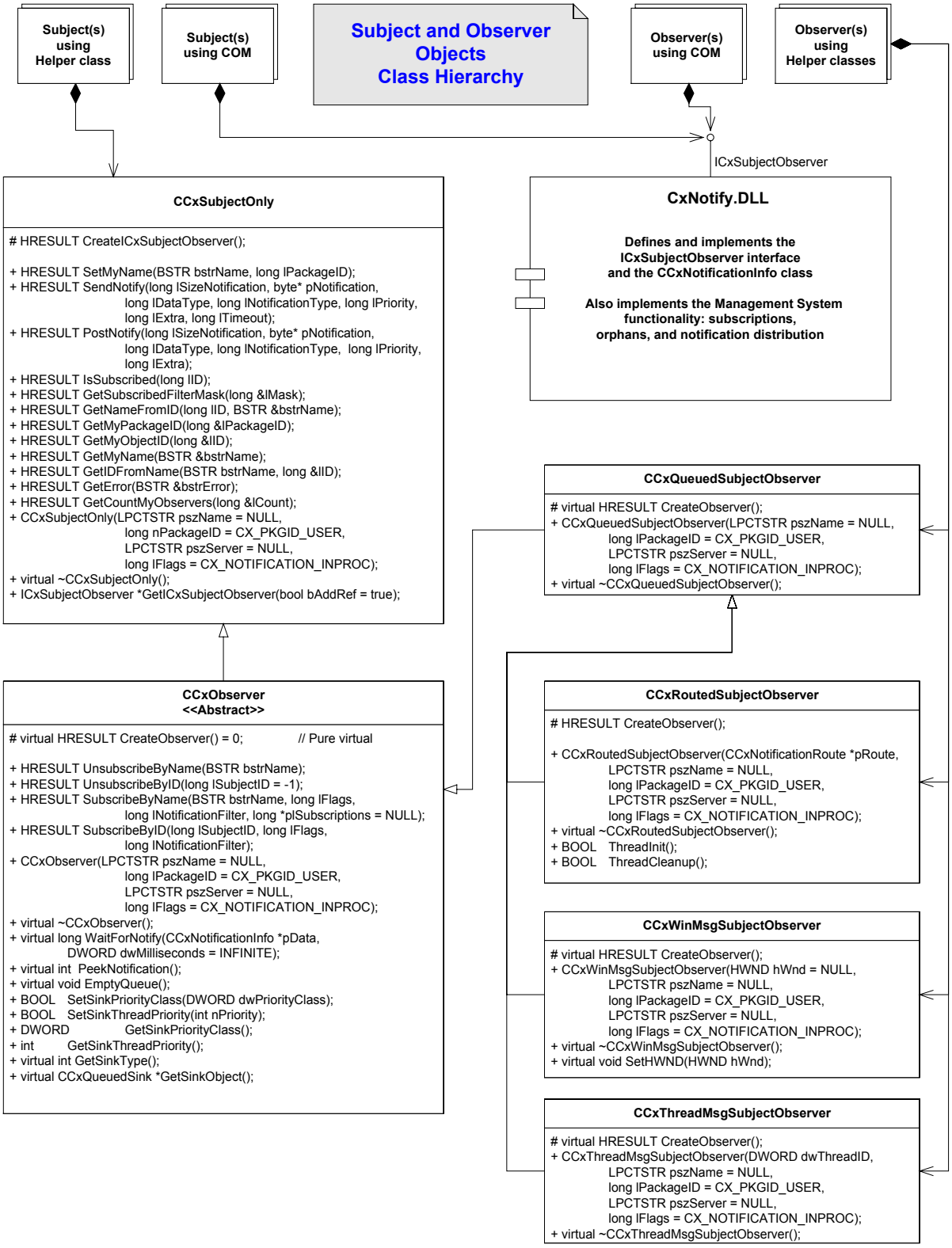


Diagram: Subject and Observer class hierarchy

One thing that the diagram above does NOT show (because of space limitation) is the relationship between the Observer classes and their associated and hidden sink objects. Each class that derives from CCxObserver contains an associated sink object. The sink object is created in the Observer's CreateObserver() method. For example, the CCxRoutedSubjectObserver::CreateObserver() creates a CCxRoutedSink, the CCxWinMsgSubjectObserver::CreateObserver() creates a CCxWinMsgSink, the CCxThreadMsgSubjectObserver::CreateObserver() creates a CCxThreadMsgSink, and the CCxQueuedSubjectObserver::CreateObserver() creates a CCxQueuedSink. Creation and destruction of the sink objects is completely handled by the classes that derive from CCxObserver. A client who creates one of the objects that derives from CCxObserver requires no knowledge of COM or sinks. The sole purpose of these sink objects is:

1. Define and implement a queue that holds notifications from the Management System.
2. Depending on the kind of sink, the sink may forward the notification on to another object that knows how to handle the notification.

Pay particular attention to each of the Observer types in the diagram. Each type of Observer requires a specific parameter in its constructor. For example, the routed Subject Observer constructor requires a pointer that can be a "callback", the window Subject Observer constructor requires an HWND, and the thread message Subject Observer constructor requires a DWORD thread ID. The Notifications Package provides a number of helper classes that can be used to create objects that provide these parameters. Refer to the CCxNotificationWnd hidden window object, the CCxNotificationThread worker thread object, or the CCxNotificationRoute object. Note that if a client wants the CCxQueuedSubjectObserver capabilities, no special "helper" class is needed.

Installed Files

The Notifications Package is installed as part of CIMAppObjects™. The CIMAppObjects packages are part of the Cimetrix CODE product line.

During installation of the CODE product, the user is prompted for the location of the installation directory. For the purposes of discussion, we will assume that the user has selected `C:\Program Files\Cimetrix` for the location of the CODE installation. Assuming this installation path, the following Notifications Package files are installed:

Header Files	Installation Subdirectory	Description
CxCriticalArray.h	.\include	Defines and implements the CCxCriticalArray class. This class provides a critical section array. It provides critical section locking via it's Lock() and Unlock() methods.
CxCriticalList.h	.\include	Defines and implements the CCxCriticalList class. This class is a critical section list. It provides critical section locking via the Lock() and Unlock() methods.
CxCriticalQueue.h	.\include	Defines and implements the CCxCriticalQueue class. This class is a critical section STL priority queue. It provides critical section locking via the Lock() and Unlock() methods.
CxDataTypes.h	.\include	The CCxDataTypes class provides helper methods that allow the caller to use predefined data types. These data types are completely optional. Using these data types are NOT required by the Notification package. They are provided only as a convenience for the user. Once again: These data types are NOT required by the notification package. They are 100% optional.
CxDiffTime.h	.\include	Defines the CCxDiffTime class. This class is used internally by the CCxNotificationInfo object to track the time it takes for

		delivery of a notification to occur.
CxNotification.h	.\include	Defines the CCxNotification class. This class provides a COM interface to a notification object. Note that this class is more limited than its' C++ counterpart CCxNotificationInfo class.
CxNotificationHelper.h	.\include	Defines the CCxNotificationHelper class. This class can be used to extract and format notification data.
CxNotificationInfo.h	.\include	Defines the CCxNotificationInfo class. This ***IS*** the notification object that gets delivered to an Observer.
CxNotificationRoute.h	.\include	Defines the CCxNotificationRoute class. This class is used as a base class for any object wishing to receive routed notifications via the CCxRoutedSink sink object.
CxNotificationThread.h	.\include	Defines the CCxNotificationThread class. This class can be used as a helper class to create a worker thread that receives and handles notifications.
CxNotificationTypes.h	.\include	Defines the CCxNotificationTypes class. The CCxNotificationTypes class provides helper methods that allow the caller to use predefined notification types. These notification types are completely optional. Using these notification types are NOT required by the Notification package. They are provided only as a convenience for the user. Once again: These notification types are NOT required by the notification package. They are 100% optional.
CxNotificationWnd.h	.\include	Defines the CCxNotificationWnd class. This class is a helper class that can be used to "catch" notifications via a windows message loop.

CxNotify.h	.\include	MASTER HEADER FILE: Single header file that #includes ALL other necessary headers for the entire notification package. This file includes everything necessary to use the Notifications Package. Include it in your stdafx.h file or wherever you need access to the notification package objects or its' COM interfaces.
CxNotifyClient.h	.\include	Single header file that #includes ALL other necessary headers for the CxNotifyClient DLL
CxNotifyServer.h	.\include	Single header file that defines everything needed by the CxNotify COM Server DLL. It contains a number of #defines used throughout the Notifications Package.
CxNotifySupport.h	.\include	Single header file that #includes ALL other necessary headers for the CxNotifySupport DLL.
CxObserver.h	.\include	Defines the CCxObserver object. This object can be used to create an Observer object.
CxQueuedSO.h	.\include	Defines the CCxQueuedSubjectObserver class. This class can be used to create a queued Subject-Observer object.
CxRoutedSO.h	.\include	Defines the CCxRoutedSubjectObserver class. This class can be used to create a routed Subject-Observer object.
CxSink.h	.\include	Provides the definition for all sink objects. The sink objects define how notifications are delivered to an Observer object. These classes are used by the Management System to deliver notification data to an Observer. This header file defines the CCxSink, CCxQueuedSink, CCxRoutedSink, CCxThreadMsgSink, and CCxWinMsgSink classes.
CxSinkData.h	.\include	Defines a number of classes used by the Management

		System. The Management System uses these classes to pass information to Observer Sink objects.
CxSinkTypes.h	.\include	Defines the CCxSinkTypes class. This class can be used to enumerate all of the sink type objects supported by the Notifications Package.
CxSubject.h	.\include	Defines the CCxSubjectOnly class. This class can be used to create a Subject object.
CxThreadMsgSO.h	.\include	Defines the CCxThreadMsgSubjectObserver class. This class can be used to create a thread message Subject-Observer object.
CxWinMsgSO.h	.\include	Defines the CCxWinMsgSubjectObserver class. This class can be used to create a Windows Message Subject-Observer object.
Notify.h	.\include	Notify.h is a header file generated by the MIDL compiler that defines and exposes all of the COM interfaces for the Notifications Package.
Notify_i.c	.\include	This file contains the actual definitions of the IIDs and CLSIDs provided by the Notifications Package.

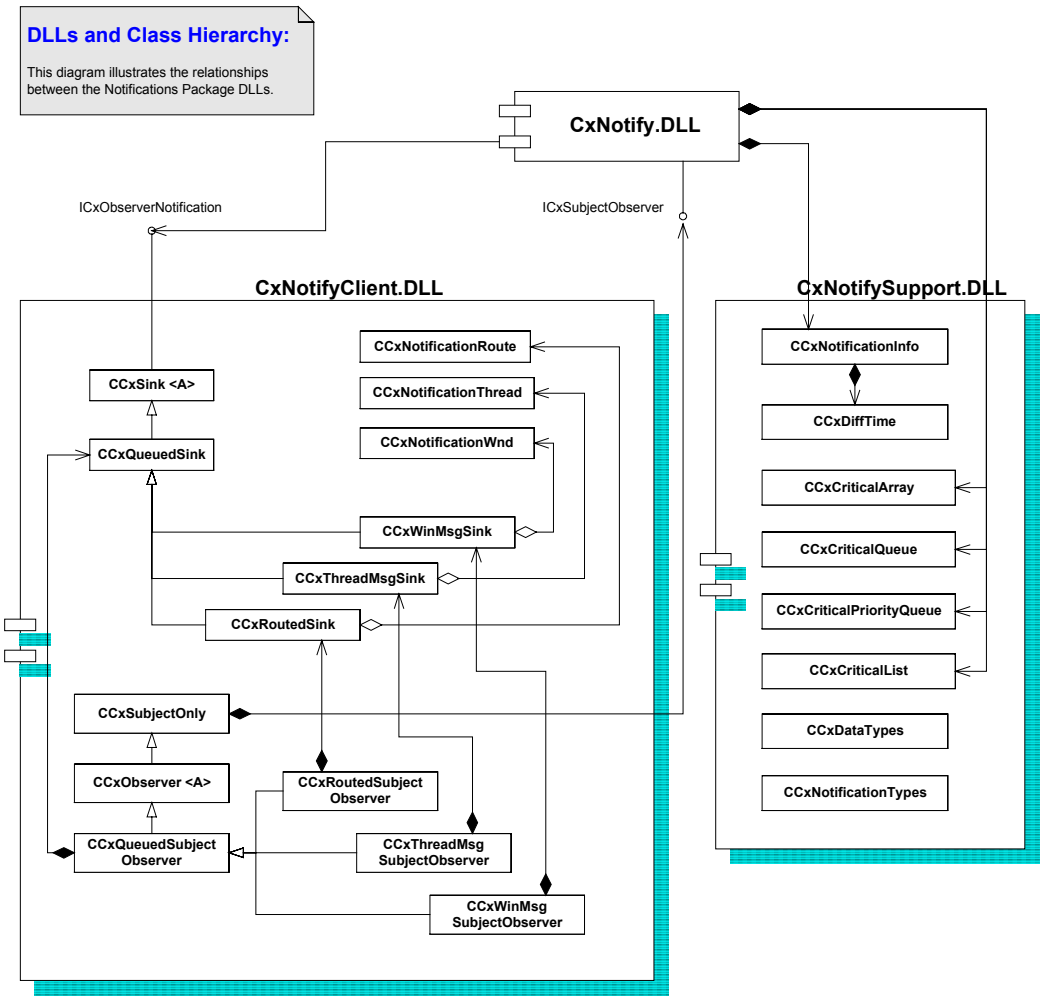
The following binary files are distributed with the Notifications Package.

Binary Files	**Installation Subdirectory**	**Description**
CxNotify.dll	.\bin	COM Server DLL that contains ALL of the COM functionality for the Notifications Package. Note that there is no link library (CxNotify.lib) for this DLL. Application do NOT need to directly link with this DLL. The COM subsystem in Windows NT provides the "connection" with the DLL.
CxNotify.tlb	.\lib	Notifications Package type library.
CxNotifyClient.dll	.\bin	DLL that provides the implementation of all sink objects, C++ wrapper classes for the COM server objects, and

		miscellaneous helper classes.
CxNotifyClient.lib	.\lib	Link library for the CxNotifyClient.dll. Any application that uses the Notifications Package will need to link with this library.
CxNotifySupport.dll	.\bin	DLL that provides the implementation of actual notification object and it's associated helper classes.
CxNotifySupport.lib	.\lib	Link library for the CxNotifySupport.dll. Any application that uses the Notifications Package will need to link with this library.

DLLs and Public Classes

The following UML diagram illustrates the class hierarchy relationships of classes within and between each DLL. Inheritance, composition, and aggregation class relationships are shown.



Note that the CxNotify DLL does not expose any public C++ classes. It exposes only the ICxSubjectObserver COM interface which is used by the CxNotifyClient DLL.

Programmer's Reference

The Notifications Package contains a number of binary files as DLLs. Each DLL exports classes and/or COM interface definitions and implementation. This section documents each publicly exported class and COM object. Each subsection within the Programmer's Reference defines the following:

1. Class or COM Interface name
2. Brief explanation of the purpose and content of the class or COM interface
3. Method Name
4. Name of the DLL that exports the class or COM interface
5. Name of the header file(s) needed to use the class or COM interface
6. Prototype of all public and protected methods for each class
7. Description of each parameter for each method
8. Return value and explanation for each method
9. Programming Example for each method
10. Optional related links to other methods, classes, or COM interfaces

The following tables show each DLL provided by the Notifications Package and provides a list of each publicly exported classes or COM components provided by the DLL.

The header file that defines each class is also documented. To have visibility for all classes, simply #include the CxNotify.h header file. This header file includes all others.

DLL Name	Public Classes	Header File
CxNotifyClient.DLL	CCxNotificationHelper	CxNotificationHelper.h
	CCxNotificationRoute	CxNotificationRoute.h
	CCxNotificationThread	CxNotificationThread.h
	CCxNotificationWnd	CxNotificationWnd.h
	CCxObserver	CxObserver.h
	CCxQueuedSink	CxSink.h
	CCxQueuedSubjectObserver	CxQueuedSO.h
	CCxRoutedSink	CxSink.h
	CCxRoutedSubjectObserver	CxRoutedSO.h
	CCxSink	CxSink.h
	CCxSinkTypes	CxSinkTypes.h
	CCxSubjectOnly	CxSubject.h
	CCxThreadMsgSink	CxSink.h
	CCxThreadMsgSubjectObserver	CxThreadMsgSO.h
	CCxWinMsgSink	CxSink.h
	CCxWinMsgSubjectObserver	CxWinMsgSO.h

DLL Name	Public Classes	Header File
CxNotifySupport.DLL	CCxCriticalArray	CxCriticalArray.h
	CCxCriticalList	CxCriticalList.h
	CCxCriticalPriorityQueue	CxCriticalQueue.h
	CCxCriticalQueue	CxCriticalQueue.h
	CCxDataTypes	CxDataTypes.h

CCxDiffTime	CxDiffTime.h
CCxNotificationInfo	CxNotification.h
CCxNotificationTypes	CxNotificationTypes.h

DLL Name	Public COM Interfaces	Description
CxNotify.DLL	ICxObserverNotification	This is a source interface that in implemented by all sink objects. A source interface according to COM, is defined by the DLL, but **NOT** implemented. The sink objects that actually implement this interface are publicly exported from the CxNotifyClient.DLL. Any third party developer who wishes to implement a Notifications Package sink object MUST implement this COM interface. Notify.h provides the definition of this interface. Notify_i.c provides the CLSID and IID of this COM interface. CxNotify.tlb provides the type library for this COM interface.
	ICxSubjectObserver	This is the "main" COM interface for all objects that need either a Subject or an Observer. This single COM interface provides ALL capabilities of Subjects and Observers. Notify.h provides the definition of this interface. Notify_i.c provides the CLSID and IID of this COM interface. CxNotify.tlb provides the type library for this COM interface.

CCxCriticalArray Class

The CCxCriticalArray class inherits from and extends MFC's CArray template class. Additional member functions were added to make the class thread-safe. For detailed documentation on the CArray template class and its members, refer to the MFC documentation.

The following table summarizes the available methods

Method Name	Description
`GetCount`	Maintains a count of the number of items in the array.
`Lock`	Locks the array so that no other thread can manipulate the data it holds.
`Unlock`	Unlocks the array

The following table summarizes the member variables

Name	Data Type	Scope	Description
m_csLock;	CCriticalSection	Protected	Maintains the lock on the array
m_lCtr	long	Protected	Adds an easy to use count of the number of items in the array

CCxCriticalArray::GetCount

Prototype

```
long GetCount();
```

Parameters

None	

Scope

`Public`

Definition

```
Dynamic Link Library        CxNotifySupport.DLL
Header Files                #include <CxCriticalArray.h>
```

Description

This method locks returns the value of our m_lCtr member variable.

Return Values

`long`	Number of items in the array

See Also

CCxCriticalArray::Lock

Prototype

```
BOOL Lock();
```

Parameters

None	

Scope

```
Public
```

Definition

```
Dynamic Link Library          CxNotifySupport.DLL
Header Files                  #include <CxCriticalArray.h>
```

Description

This method locks the critical array object so that no other thread can access it. Once the array is locked, other member functions can be utilized in a thread-safe manner. This method uses a critical section locking object to limit access.

Once the user is done adding, removing, or updating elements in the array, the Unlock() method MUST be called.

Return Values

BOOL	TRUE if successful, else FALSE

Source Code Example

```
#include <CxCriticalArray.h>

typedef CCxCriticalArray<void*, void*>    VOIDPTR_ARRAY;
typedef CCxCriticalArray<long, long>      LONG_ARRAY;

class CMyClass
{
protected:
      LONG_ARRAY        m_alUsedIDs;
      VOIDPTR_ARRAY     m_apObjects;

public:
      void AddItem(long lNextID, void *pObject);
}


void CMyClass::AddItem(long lNextID, void *pObject)
{
      m_alUsedIDs.Lock();
      m_apObjects.Lock();

      m_alUsedIDs.Add(lNextID);
```

```
        m_apObjects.Add(pObject);

        m_alUsedIDs.Unlock();
        m_apObjects.Unlock();

    }
```

See Also

Unlock()
MFC CArray template class documentation

CCxCriticalArray::Unlock

Prototype

```
BOOL Unlock();
```

Parameters

None	

Scope

```
Public
```

Definition

```
Dynamic Link Library        CxNotifySupport.DLL
Header Files                #include <CxCriticalArray.h>
```

Description

This method unlocks the critical array object so that others threads are free to access it. The Unlock method MUST always be called after locking the array. Otherwise a deadlock situation will occur. Other threads will wait forever to manipulate the array because the array will remain locked indefinitely.

Return Values

BOOL	TRUE if successful, else FALSE

Source Code Example

Refer to the source code example in the Lock method documentation.

See Also

Lock()
MFC CArray template class documentation

CCxCriticalList Class

The CCxCriticalList class inherits from and extends MFC's CList template class. Additional member functions were added to make the class thread-safe. For detailed documentation on the CList template class and its members, refer to the MFC documentation.

The following table summarizes the available methods

Method Name	Description
Lock	Locks the list so that no other thread can manipulate the data it holds.
Unlock	Unlocks the list

The following table summarizes the member variables

Name	Data Type	Scope	Description
m_csLock;	CCriticalSection	Protected	Maintains the lock on the list

CCxCriticalList::Lock

Prototype

```
BOOL Lock();
```

Parameters

None	

Scope

```
Public
```

Definition

```
Dynamic Link Library        CxNotifySupport.DLL
Header Files                #include <CxCriticalList.h>
```

Description

This method locks the critical list object so that no other thread can access it. Once the list is locked, other member functions can be utilized in a thread-safe manner. This method uses a critical section locking object to limit access.

Once the user is done adding, removing, or updating elements in the list, the Unlock() method MUST be called.

Return Values

BOOL	TRUE if successful, else FALSE

Source Code Example

```
#include <CxCriticalList.h>
```

```
typedef CCxCriticalList<void*, void*>     VOIDPTR_LIST;
typedef CCxCriticalList<long, long>       LONG_LIST;

class CMyClass
{
protected:
      LONG_LIST         m_listUsedIDs;
      VOIDPTR_LIST      m_listObjects;

public:
      void AddItem(long lNextID, void *pObject);
}


void CMyClass::AddItem(long lNextID, void *pObject)
{
      m_listUsedIDs.Lock();
      m_listObjects.Lock();

      m_listUsedIDs.Add(lNextID);
      m_listObjects.Add(pObject);

      m_listUsedIDs.Unlock();
      m_listObjects.Unlock();

}
```

See Also

Unlock()
MFC CList template class documentation

CCxCriticalList::Unlock

Prototype

```
BOOL Unlock();
```

Parameters

None	

Scope

```
Public
```

Definition

```
Dynamic Link Library          CxNotifySupport.DLL
Header Files                  #include <CxCriticalArray.h>
```

Description

This method unlocks the critical list object so that others threads are free to access it. The Unlock method MUST always be called after locking the list.

Otherwise a deadlock situation will occur. Other threads will wait forever to manipulate the list because the list will remain locked indefinitely.

Return Values

BOOL	TRUE if successful, else FALSE

Source Code Example

Refer to the source code example in the Lock method documentation.

See Also

Lock()
MFC CList template class documentation

CCxCriticalPriorityQueue Class

The CCxCriticalPriorityQueue class inherits from and extends the STL's priority_queue template class. Additional member functions were added to make the class thread-safe. For detailed documentation on the STL's priority_queue template class and its members, refer to the STL documentation.

The following table summarizes the available methods

Method Name	Description
Lock	Locks the queue so that no other thread can manipulate the data it holds.
Unlock	Unlocks the queue

The following table summarizes the member variables

Name	Data Type	Scope	Description
m_csLock;	CCriticalSection	Protected	Maintains the lock on the queue

CCxCriticalPriorityQueue::Lock

Prototype

```
BOOL Lock();
```

Parameters

None	

Scope

```
Public
```

Definition

```
Dynamic Link Library        CxNotifySupport.DLL
Header Files                #include <CxCriticalQueue.h>
```

Description

This method locks the queue object so that no other thread can access it. Once the queue is locked, other member functions can be utilized in a thread-safe manner. This method uses a critical section locking object to limit access.

Once the user is done adding, removing, or updating elements in the queue, the Unlock() method MUST be called.

Return Values

BOOL	TRUE if successful, else FALSE

Source Code Example

```
#include <CxCriticalQueue.h>
```

```
class CMyClass
{
protected:

#ifdef _PRIORITY_QUEUE
CCxCriticalPriorityQueue<void *, deque<void *> > m_qObjects;
#else
      CCxCriticalQueue<void * > m_qObjects;
#endif

public:
      void AddItem(void *pObject);
}


void CMyClass::AddItem(void *pObject)
{
      m_qObjects.Lock();

      m_qObjects.push(pObject);

      m_qObjects.Unlock();

}
```

See Also

Unlock()
STL's priority_queue template class documentation

CCxCriticalPriorityQueue::Unlock

Prototype

```
BOOL Unlock();
```

Parameters

None	

Scope

```
Public
```

Definition

```
Dynamic Link Library          CxNotifySupport.DLL
Header Files                  #include <CxCriticalQueue.h>
```

Description

This method unlocks the queue object so that others threads are free to access it. The Unlock method MUST always be called after locking the queue. Otherwise a deadlock situation will occur. Other threads will wait forever to manipulate the queue because the queue will remain locked indefinitely.

Return Values

BOOL	TRUE if successful, else FALSE

Source Code Example

Refer to the source code example in the Lock method documentation.

See Also

Lock()
STL's priority_queue template class documentation

CCxCriticalQueue Class

The CCxCriticalQueue class inherits from and extends the STL's queue template class. Additional member functions were added to make the class thread-safe. For detailed documentation on the STL's queue template class and its members, refer to the STL documentation.

The following table summarizes the available methods

Method Name	Description
Lock	Locks the queue so that no other thread can manipulate the data it holds.
Unlock	Unlocks the queue

The following table summarizes the member variables

Name	Data Type	Scope	Description
m_csLock;	CCriticalSection	Protected	Maintains the lock on the queue

CCxCriticalQueue::Lock

Prototype

```
BOOL Lock();
```

Parameters

None	

Scope

Public

Definition

```
Dynamic Link Library        CxNotifySupport.DLL
Header Files                #include <CxCriticalQueue.h>
```

Description

This method locks the queue object so that no other thread can access it. Once the queue is locked, other member functions can be utilized in a thread-safe manner. This method uses a critical section locking object to limit access.

Once the user is done adding, removing, or updating elements in the queue, the Unlock() method MUST be called.

Return Values

BOOL	TRUE if successful, else FALSE

Source Code Example

```
#include <CxCriticalQueue.h>
```

```
class CMyClass
{
protected:

#ifdef _PRIORITY_QUEUE
CCxCriticalPriorityQueue<void *, deque<void *> > m_qObjects;
#else
      CCxCriticalQueue<void * > m_qObjects;
#endif

public:
      void AddItem(void *pObject);
}


void CMyClass::AddItem(void *pObject)
{
      m_qObjects.Lock();

      m_qObjects.push(pObject);

      m_qObjects.Unlock();

}
```

See Also

Unlock()
STL's queue template class documentation

CCxCriticalQueue::Unlock

Prototype

```
BOOL Unlock();
```

Parameters

None	

Scope

```
Public
```

Definition

```
Dynamic Link Library          CxNotifySupport.DLL
Header Files                  #include <CxCriticalQueue.h>
```

Description

This method unlocks the queue object so that others threads are free to access it. The Unlock method MUST always be called after locking the queue. Otherwise a deadlock situation will occur. Other threads will wait forever to manipulate the queue because the queue will remain locked indefinitely.

Return Values

BOOL	TRUE if successful, else FALSE

Source Code Example

Refer to the source code example in the Lock method documentation.

See Also

Lock()
STL's queue template class documentation

CCxDataTypes Class

The CCxDataTypes class is a helper class that provides data type flags that could be passed as part of a notification. This class is completely optional and provided only as a reference to programmers using the Notifications Package.

The class defines a few data types and provides helpful class methods for enumerating the data types.

Once again, this class is provided as a helper only. The Notifications Package does NOT attempt to define and support specific data types. The Notifications Package is completely unaware of data types as a notification is passed from a Subject to an Observer.

The following table summarizes the available methods

Method Name	Description
`FindDataTypeID`	Given a string find its associated data type id.
`FindDataTypeID`	Given a data type id find its associated string representation.

The following table summarizes the member variables

Name	Data Type	Scope	Description
m_aDataTypes	structDataTypes *	Protected	An array of structures of data types. The structure is defined as: typedef struct structDataTypes { int m_id; LPCTSTR m_str; } structDataTypes;

This class "wraps" the following enumerated data types

```
enum
{
      CX_DTYPE_CHARSTAR = 0,
      CX_DTYPE_INT,
      CX_DTYPE_LONG,
      CX_DTYPE_DOUBLE,
      CX_DTYPE_END_VALUE            // NTYPE_END_VALUE must be
the last value
};
```

The structure defined in the member variable table holds both the enumerated value of a data type and a corresponding string representation of the data type. The string representation can be used to populate a list box in a GUI for example.

The following code is used in our constructor to load the m_aDataTypes member variable:

```
CCxDataTypes::CCxDataTypes()
```

```
{
      m_aDataTypes = new structDataTypes[CX_DTYPE_END_VALUE + 1];

      static structDataTypes tmpDataTypes[] =
      {
            { CX_DTYPE_CHARSTAR,    _T( "Char *" ) },
            { CX_DTYPE_INT,         _T( "Integer" ) },
            { CX_DTYPE_LONG,        _T( "Long" ) },
            { CX_DTYPE_DOUBLE,      _T( "Double" ) },
            { CX_DTYPE_END_VALUE,   _T( "") }   // must be last
      };

      // Fill the structure
      for (int i=CX_DTYPE_CHARSTAR; i <= CX_DTYPE_END_VALUE; i++)
      {
            m_aDataTypes[i].m_id = tmpDataTypes[i].m_id;
            m_aDataTypes[i].m_str = tmpDataTypes[i].m_str;
      }
}
```

As you can see, this is a severely limited subset of data types. Most programmers using the Notifications Package will need to support substantially more data types. They will probably also want to support more complex data types as well.

CCxDataTypes::FindDataTypeID

Prototype

```
int FindDataTypeID(CString strDType);
```

Parameters

CString strDType	[in] Data type string to search for. This string must be one of the values that was loaded into our structDataTypes array member variable.

Scope

```
Public
```

Definition

```
Dynamic Link Library          CxNotifySupport.DLL
Header Files                  #include <CxDataTypes.h>
```

Description

This method searches for the string passed in strDType. If it finds the string, the method returns the ID of the string. The ID is one of the enumerated data type values defined above.

For example, if the string "Integer" is passed into this method, we return the CX_DTYPE_INT enumerated value.

Return Values

`int`	Enumerated value found in the m_aDataTypes member variable, else CX_DTYPE_END_VALUE if the value is not found

See Also

FindDataTypeString

CCxDataTypes::FindDataTypeString

Prototype

```
LPCTSTR FindDataTypeString(int id);
```

Parameters

int id	[in] ID to search for. The ID number must be in our m_aDataTypes member variable.

Scope

```
Public
```

Definition

```
Dynamic Link Library          CxNotifySupport.DLL
Header Files                  #include <CxDataTypes.h>
```

Description

This method searches for the id passed in. If it finds the id, the method returns the string representation of the id.

For example, if the id CX_DTYPE_INT is passed in, then we return "Integer".

Return Values

`LPCTSTR`	Enumerated value found in the m_aDataTypes member variable, else "" if the value is not found

See Also

FindDataTypeID

CCxDiffTime Class

The CCxDiffTime class is used within a CCxNotificationInfo object to keep track of the amount of time it takes to deliver a notification.

When this object gets created, time tracking is automatically started.

The following table summarizes the available methods

Method Name	Description
`DeltaTicks`	Returns the number of CPU ticks it took for the notification to be delivered. The value is calculated by subtracting the start counter (m_u64nStartCtr) from the end counter value (m_u64nEndCtr).
`DeltaTime`	Returns the number of microseconds for a notification to be delivered. This value is equivalent to (DeltaTicks() * m_lResolution) / m_u64nFreq
`GetEndTicks`	Returns the value of the m_u64nEndCtr member variable.
`GetEndTime`	Returns the notification's ending time as a time_t
`GetResolution`	Returns the default resolution (microseconds)
`GetResolutionString`	Returns a string for the default resolution
`GetStartTicks`	Returns the value in the m_u64nStartCtr member variable.
`GetStartTime`	Returns the notification's starting time as a time_t.
`Reset`	Resets all counters and member variables to default values
`SetEnd`	Forces a read of the CPU tick counter. We store this value in our m_u64nEndCtr member variable. This method is called at the end of the delivery of a notification.
`SetStart`	Forces a read of the CPU tick counter. We store this value in our m_u64nStartCtr member variable. This method is called at the beginning of the delivery of a notification.

The following table summarizes the member variables

Name	Data Type	Scope	Description
m_StartTime	time_t	Protected	Start time of the notification
m_EndTime	time_t	Protected	End time of the notification
m_lResolution	long	Protected	Time resolution - default is microseconds
m_u64nStartCtr	unsigned __int64	Protected	Counter that stores CPU ticks from a Pentium CPU. The start counter is stored for later use to be subtracted from the end

			counter
m_u64nEndCtr	unsigned __int64	Protected	Counter that stores CPU ticks from a Pentium CPU. The end counter is stored for later use to subtract from the start counter
m_u64nFreq	unsigned __int64	Protected	CPU clock speed

CCxDiffTime::CCxDiffTime

Prototype

```
CCxDiffTime(long lResolution = CX_MICROSECONDS);
```

Parameters

long lResolution	[in] default resolution to use. For a list of available resolutions, refer to CxDiffTime.h

Scope

Public

Definition

```
Dynamic Link Library          CxNotifySupport.DLL
Header Files                  #include <CxDiffTime.h>
```

Description

This method is our default constructor and initializes the resolution to microseconds.

Return Values

None	

See Also

CCxNotificationInfo class

CCxDiffTime::CCxDiffTime

Prototype

```
CCxDiffTime(CCxDiffTime &refTime);
```

Parameters

CCxDiffTime &refTime	[in] Reference to an existing CCxDiffTime object.

Scope

Public

Definition

```
Dynamic Link Library          CxNotifySupport.DLL
Header Files                  #include <CxDiffTime.h>
```

Description

This method is our copy constructor. It initializes our internal member variables with the values from refTime.

Return Values

None	

See Also

CCxNotificationInfo class

CCxDiffTime::~ CCxDiffTime

Prototype

```
virtual ~CCxDiffTime();
```

Parameters

None	

Scope

Public

Definition

```
Dynamic Link Library          CxNotifySupport.DLL
Header Files                  #include <CxDiffTime.h>
```

Description

This method is our destructor and performs all necessary cleanup.

Return Values

None	

See Also

CCxNotificationInfo class

CCxDiffTime::DeltaTicks

Prototype

```
inline DWORD DeltaTicks();
```

Parameters

None	

Scope

Public

Definition

```
Dynamic Link Library          CxNotifySupport.DLL
Header Files                  #include <CxDiffTime.h>
```

Description

This method returns the difference between the start and end time counters that are tracked by this object. These counters track the number CPU ticks that have occurred since the computer was last rebooted. These CPU ticks can be used to determine time deltas with a very high degree of resolution. The default resolution is CX_MICROSECONDS defined in our header file.

Return Values

DWORD	The difference between the start and end time counters.

See Also

CCxNotificationInfo class

CCxDiffTime::DeltaTime

Prototype

```
inline double DeltaTime();
```

Parameters

None	

Scope

```
Public
```

Definition

```
Dynamic Link Library          CxNotifySupport.DLL
Header Files                  #include <CxDiffTime.h>
```

Description

This method returns a real time value. The value is determined as follows: DeltaTicks() * m_lResolution / CPU speed.

Return Values

double	Returns the time delta between the start and end counters. The default resolution is microseconds.

See Also

CCxNotificationInfo class

CCxDiffTime::GetEndTicks

Prototype

```
inline DWORD GetEndTicks()
```

Parameters

None	

Scope

Public

Definition

```
Dynamic Link Library          CxNotifySupport.DLL
Header Files                  #include <CxDiffTime.h>
```

Description

This method returns the value of our m_u64nEndCtr member variable.

Return Values

DWORD	Returns the value of our m_u64nEndCtr member variable.

See Also

CCxNotificationInfo class

CCxDiffTime::GetEndTime

Prototype

```
inline time_t GetEndTime();
```

Parameters

None	

Scope

Public

Definition

```
Dynamic Link Library          CxNotifySupport.DLL
Header Files                  #include <CxDiffTime.h>
```

Description

This method returns the value of our m_EndTime member variable.

Return Values

time_t	Returns the end time as a time_t.

See Also

CCxNotificationInfo class

CCxDiffTime::GetResolution

Prototype

```
inline double GetResolution();
```

Parameters

None	

Scope

Public

Definition

```
Dynamic Link Library          CxNotifySupport.DLL
Header Files                  #include <CxDiffTime.h>
```

Description

This method returns the clock resolution. Default value is in microseconds (1/1000000).

Return Values

double	Returns the value of our m_lResolution member variable

See Also

CCxNotificationInfo class

CCxDiffTime::GetResolutionString

Prototype

```
inline LPCTSTR GetResolutionString();
```

Parameters

None	

Scope

Public

Definition

```
Dynamic Link Library          CxNotifySupport.DLL
Header Files                  #include <CxDiffTime.h>
```

Description

This method returns the string representation of the m_lResolution member variable.

Supported resolutions and their string representations are: "nanoseconds", "microseconds", "milliseconds", and "seconds".

Return Values

LPCTSTR	Returns one of the values listed in the description section

See Also

CCxNotificationInfo class

CCxDiffTime::GetStartTicks

Prototype

```
inline DWORD GetStartTicks()
```

Parameters

None	

Scope

Public

Definition

```
Dynamic Link Library          CxNotifySupport.DLL
Header Files                  #include <CxDiffTime.h>
```

Description

This method returns the value of our m_u64nStartCtr member variable.

Return Values

DWORD	Returns the value of our m_u64nStartCtr member variable.

See Also

CCxNotificationInfo class

CCxDiffTime::GetStartTime

Prototype

```
inline time_t GetStartTime();
```

Parameters

None	

Scope

Public

Definition

```
Dynamic Link Library          CxNotifySupport.DLL
Header Files                  #include <CxDiffTime.h>
```

Description

This method returns the start time as a time_t value. The method returns the time when this object was first created or when SetStart was last called.

Return Values

Time_t	Returns the time when this object was first created or when SetStart was called.

See Also

CCxNotificationInfo class
SetStart

CCxDiffTime::operator =

Prototype

```
CCxDiffTime & operator =(const CCxDiffTime &refObj);
```

Parameters

CCxDiffTime &refObj	[in] reference to an existing CCxDiffTime object.

Scope

Public

Definition

```
Dynamic Link Library          CxNotifySupport.DLL
Header Files                  #include <CxDiffTime.h>
```

Description

This method assigns the values in refObj to this object. This method can be used to assign the values of one CCxDiffTime object to another CCxDiffTime object.

Return Values

CCxDiffTime &	Returns a reference to this object.

See Also

CCxNotificationInfo class

CCxDiffTime::Reset

Prototype

```
inline void Reset();
```

Parameters

None	

Scope

Public

Definition

```
Dynamic Link Library          CxNotifySupport.DLL
Header Files                  #include <CxDiffTime.h>
```

Description

This resets all internal member variables to default values. All timers and counters are reinitialized to zero.

Return Values

None	

See Also

CCxNotificationInfo class

CCxDiffTime::SetEnd

Prototype

```
inline DWORD SetEnd();
```

Parameters

None	

Scope

```
Public
```

Definition

```
Dynamic Link Library          CxNotifySupport.DLL
Header Files                  #include <CxDiffTime.h>
```

Description

This method sets the m_u64nEndCtr member variable to the current value of the CPU tick counter. For details on how this is done, refer to the WIN32 API documentation on QueryPerformanceCounter.

Return Values

DWORD	The newly set value of m_u64nEndCtr.

See Also

CCxNotificationInfo class

CCxDiffTime::SetStart

Prototype

```
inline DWORD SetStart();
```

Parameters

None	

Scope

Public

Definition

Dynamic Link Library	CxNotifySupport.DLL
Header Files	#include <CxDiffTime.h>

Description

This method sets the m_u64nStartCtr member variable to the current value of the CPU tick counter. For details on how this is done, refer to the WIN32 API documentation on QueryPerformanceCounter.

Return Values

DWORD	The newly set value of m_u64nStartCtr.

See Also

CCxNotificationInfo class

CCxNotificationHelper Class

The CCxNotificationHelper class…

Method Name	Description
`CCxNotificationHelper`	Class constructor
`~CCxNotificationHelper`	Class destructor
`Extract`	Allows the input of an existing CCxNotificationInfo object. All internal member variables are initialized with the data from the notification object.
`GetAvgElapsedTimeString`	Calculate, format and return our average elapsed time as a string
`GetAvgElapsedTimeValue`	Returns the average elapsed time value as a double
`GetDataAsString`	Formats and returns the notification byte buffer as a string
`GetDataType`	Returns the notification data type
`GetElapsedTimeString`	Format and return our elapsed time as a string
`GetElapsedTimeValue`	Returns the amount of time it took for the notification data to be sent from a Subject to an Observer.
`GetEndTimeString`	Returns the time when the notification object was delivered
`GetExtraData`	Returns the value of the extra parm on the notification object
`GetMaxTime`	Since this object can be reused, it can remember the maximum time a notification took to be delivered. It returns this value.
`GetMaxTimeString`	Format and return our maximum notification time string
`GetMinTime`	Since this object can be reused, it can remember the minimum time a notification took to be delivered. It returns this value.
`GetMinTimeString`	Format and return our minimum notification time string
`GetNotificationsRcvd`	Returns the number of notifications that this object has processed
`GetNotificationsRcvdString`	Format and return our count of notifications received string
`GetNotificationType`	Returns the notification type
`GetSize`	Returns the size of the notification data
`GetStartTimeString`	Returns the time when the notification was initially sent.
`Reset`	The Reset function resets all of our counters and internal data.

Name	Data Type	Scope	Description
m_dblElapsedTime	double	Protected	Elapsed time of the current notification.
m_dblMaxTime	double	Protected	Maximum time for all notifications that have arrived
m_dblMinTime	double	Protected	Minimum time for all notifications that have arrived
m_dblTotalElapsedTime	double	Protected	Grand total of all elapsed time - used to compute averages
m_lDataTypeRcvd	long	Protected	Data type of the last notification
m_lExtraRcvd	long	Protected	Extra parameter of the last notification
m_lNotificationsRcvd	long	Protected	Number of notifications received - used to compute averages
m_lNTypeRcvd	long	Protected	Notification type of the last notification
m_lSizeRcvd	long	Protected	Current size (in bytes) of the notification received
m_strData	CString	Protected	Notification data formatted as a string (if possible)
m_strEndTime	CString	Protected	Ending time of the notification as a string
m_strStartTime	CString	Protected	Starting time of the notification as a string

CCxNotificationHelper::CCxNotificationHelper

Prototype

```
CCxNotificationHelper();
```

Parameters

None

Scope

```
Public
```

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
```

```
Header Files                    #include <CxNotificationHelper.h>
```

Description

The constructor take no parameters for this class. The Extract method is the one and only place where the object's internal member variables can be manipulated. Each time the Extract method is called, a notification object is passed in. The notification object is used to set all other internal member variables on this object.

See Also

CCxNotificationHelper::Extract

CCxNotificationHelper::~CCxNotificationHelper

Prototype

```
virtual ~CCxNotificationHelper;
```

Parameters

None

Scope

Public

Definition

```
Dynamic Link Library            CxNotifyClient.DLL
Header Files                    #include <CxNotificationHelper.h>
```

Description

This is the destructor.

CCxNotificationHelper::Extract

Prototype

```
virtual void Extract(CCxNotificationInfo* pNotification);
```

Parameters

CCxNotificationInfo* pNotification	[in] This is the notification object. The notification object was sent from a Subject to an Observer.

Scope

Public

Definition

```
Dynamic Link Library            CxNotifyClient.DLL
Header Files                    #include <CxNotificationHelper.h>
```

Description

This method is the front end to the entire class. A notification object is passed in to the Extract method as an input parameter. The Extract method sets the values of its internal member variables when the notification object is passed in. Each time a notification object is passed in, counters are incremented, averages are updated, and maximum and minimum delivery time values are stored. Note that all time values are in microseconds because the CCxNotificationInfo class uses microseconds for all of its elapsed time values.

This object can be reused any number of times. It will calculate averages and also maintain minimum and maximum notification delivery times. To reset the entire object call the CCxNotificationHelper::Reset method.

Return Values

`Void`	None

See Also

CCxNotificationHelper::Reset

CCxNotificationHelper::GetAvgElapsedTimeString

Prototype

```
virtual CString GetAvgElapsedTimeString();
```

Parameters

None	

Scope

`Public`

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxNotificationHelper.h>
```

Description

Calculate, format and return our average elapsed time in microseconds. Each time the Extract method is called on this object, another notification object is passed in. The delta time for each of the notification object is added to and stored in the m_dblTotalElapsedTime member variable and the m_lNotificationsRcvd member variable is incremented by 1.

The time value that gets added to m_dblTotalElapsedTime comes from the CCxNotificationInfo::DeltaTime() method. The CCxNotificationInfo::DeltaTime method returns the number of elapsed microseconds it took for the notification to be sent from the Subject to the Observer.

The average elapsed time is the average of all notification objects that this helper class has processed. Each time the Reset method is called, the average is reset.

Return Values

`CString`	Returns the elapsed delivery time for the notification object as a string. Time is in microseconds

See Also

CCxNotificationInfo::DeltaTime
CCxDiffTime::DeltaTime

CCxNotificationHelper::GetAvgElapsedTimeValue

Prototype

```
virtual double GetElapsedTimeValue()
```

Parameters

none	

Scope

`Public`

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxNotificationHelper.h>
```

Description

Calculate, format and return our average elapsed time in microseconds. Each time the Extract method is called on this object, another notification object is passed in. The delta time for each of the notification object is added to and stored in the m_dblTotalElapsedTime member variable and the m_lNotificationsRcvd member variable is incremented by 1.

The time value that gets added to m_dblTotalElapsedTime comes from the CCxNotificationInfo::DeltaTime() method. The CCxNotificationInfo::DeltaTime method returns the number of elapsed microseconds it took for the notification to be sent from the Subject to the Observer.

The average elapsed time is the average of all notification objects that this helper class has processed. Each time the Reset method is called, the average is reset.

Return Values

`double`	Returns the elapsed delivery time for the notification object as a double. Time is in microseconds

See Also

CCxNotificationInfo::DeltaTime
CCxDiffTime::DeltaTime

CCxNotificationHelper::GetDataAsString

Prototype

```
virtual CString GetDataAsString()
```

Parameters

none	

Scope

```
Public
```

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxNotificationHelper.h>
```

Description

This method returns the notification object's byte buffer formatted as a string. This method is particularly helpful for Subjects who send notification data as text.

The byte buffer comes from the CCxNotificationInfo::Bytes() method of the notification object passed in to the Extract method.

Return Values

`CString`	Returns the notification object's byte buffer formatted as a CString.

See Also

CCxNotificationInfo::Bytes
CCxNotificationHelper::Extract

CCxNotificationHelper::GetDataType

Prototype

```
virtual long GetDataType()
```

Parameters

None	

Scope

```
Public
```

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxNotificationHelper.h>
```

Description

This method returns the data type value from the notification object. The data type comes from CCxNotificationInfo::DataType method. Data types are long values that are defined and agreed to by Subjects and Observers. Observers use the data type value to help them format or parse the notification data buffer.

Keep in mind that the Notifications Package does not attempt to define all supported data types. Note that the Notifications Package does provide the CCxDataTypes class as an optional group of data types. These data types are optional and only intended as a "helper" class for the user.

Return Values

`long`	Returns the notification object's data type

See Also

CCxNotificationInfo::DataType
CCxDataTypes class

CCxNotificationHelper::GetElapsedTimeString

Prototype

```
virtual CString GetElapsedTimeString();
```

Parameters

None	

Scope

`Public`

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxNotificationHelper.h>
```

Description

Format and return the notification object's elapsed time as a string. The elapsed time is the number of microseconds it took for the notification object to be sent from the Subject and received by the Observer.

Return Values

`CString`	The elapsed time in microseconds that it took for delivery of the notification. The elapsed time is formatted and returned as a CString. Note that the time is NOT a date formatted string.

See Also

CCxNotificationInfo::DeltaTime

CCxNotificationHelper::GetElapsedTimeValue

Prototype

```
virtual double GetElapsedTimeValue();
```

Parameters

none	

Scope

Public

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxNotificationHelper.h>
```

Description

This method returns the number of microseconds it took for delivery of the notification object. The delivery time is measured from when the Subject sent the notification to when the Observer received the notification.

Return Values

double	The elapsed time in microseconds it took for delivery of the notification.

See Also

CCxNotificationInfo::DeltaTime

CCxNotificationHelper::GetEndTimeString

Prototype

```
virtual CString GetEndTimeString()
```

Parameters

none	

Scope

Public

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxNotificationHelper.h>
```

Description

Returns a formatted time string. The time string is when the notification was initially sent by the Subject.

The WIN32 **asctime** function is used to format the time. The string result produced by **asctime** contains exactly 26 characters and has the form `Wed Jan 02 02:03:55 1980\n\0`. A 24-hour clock is used by **asctime**.

Return Values

`CString`	A formatted time string of when the notification object was received by an Observer. The text string returned is formatted by the WIN32 asctime function and has the form `Wed Jan 02 02:03:55 1980\n\0`. A 24-hour clock is used by **asctime.**

See Also

CCxNotificationHelper::GetExtraData

Prototype

```
virtual long GetExtraData()
```

Parameters

none	

Scope

`Public`

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxNotificationHelper.h>
```

Description

This method returns the value from CCxNotificationInfo::Extra(). The extra data parameter can be anything that the sender of the notification wants. The only limitation of the extra parameter is that it must be a long and can NOT be a pointer. If a Subject attempts to send a pointer as the extra parameter, the results are undefined.

Return Values

`long`	Returns the value from CCxNotificationInfo::Extra()

See Also

CCxNotificationInfo::Extra()

CCxNotificationHelper::GetMaxTime

Prototype

```
virtual double GetMaxTime()
```

Parameters

none	

Scope

Public

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include < CxNotificationHelper.h >
```

Description

The CCxNotificationHelper class is designed to be reused a number of times. Each time a notification object is passed in to the Extract method, the maximum time value may be updated. The maximum time is the greatest time (in microseconds) that it has taken for a notification object to be sent from a Subject to an Observer. The maximum time is stored in our m_dblMaxTime member variable.

Each time the Reset method is called, this m_dblMaxTime member variable is reset to 0.

Return Values

long	Returns the maximum delivery time in microseconds that it took for a notification to be sent from a Subject to an Observer.

See Also

CCxNotificationHelper::GetMaxTimeString

Prototype

```
virtual CString GetMaxTimeString();
```

Parameters

none	

Scope

Public

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include < CxNotificationHelper.h >
```

Description

The CCxNotificationHelper class is designed to be reused a number of times. Each time a notification object is passed in to the Extract method, the maximum time (in microseconds) value may be updated. The maximum time is the greatest time that it has taken for a notification object to be sent from a Subject to an Observer.

Each time the Reset method is called, this maximum time is reset to 0.

Return Values

`CString`	Returns the maximum delivery time in microseconds that it took for a notification to be sent from a Subject to an Observer. The maximum delivery time m_dblMaxTime member variable is formatted as a CString object and returned to the caller.

See Also

CCxNotificationHelper::GetMinTime

Prototype

```
virtual double GetMinTime()
```

Parameters

None	

Scope

`Public`

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxNotificationHelper.h>
```

Description

The CCxNotificationHelper class is designed to be reused a number of times. Each time a notification object is passed in to the Extract method, the minimum time value may be updated. The minimum time is the smallest time (in microseconds) that it has taken for a notification object to be sent from a Subject to an Observer. The minimum time is stored in the m_dblMinTime member variable.

Each time the Reset method is called, this m_dblMinTime member variable is reset to 0.

Return Values

double	Returns the minimum delivery time in microseconds that it took for a notification to be sent from a Subject to an Observer. The m_dblMinTime member variable holds the value that is returned.

See Also

CCxNotificationHelper::GetMinTimeString

Prototype

```
virtual CString GetMinTimeString();
```

Parameters

None	

Scope

Public

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include < CxNotificationHelper.h >
```

Description

The CCxNotificationHelper class is designed to be reused a number of times. Each time a notification object is passed in to the Extract method, the minimum time (in microseconds) value may be updated. The minimum time is the smallest time that it has taken for a notification object to be sent from a Subject to an Observer.

Each time the Reset method is called, this minimum time is reset to 0.

Return Values

CString	Returns the minimum delivery time in microseconds that it took for a notification to be sent from a Subject to an Observer. The minimum delivery time member variable m_dblMinTime is formatted as a CString and returned.

See Also

CCxNotificationHelper::GetNotificationsRcvd

Prototype

```
virtual long GetNotificationsRcvd()
```

Parameters

none	

Scope

Public

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include < CxNotificationHelper.h >
```

Description

This method returns a count of the number of notification objects that have been processed by this object. The m_lNotificationsRcvd member variable stores this value. m_lNotificationsRcvd is used in all functions that return averages.

Each time the Reset method is called, m_lNotificationsRcvd is reset to 0.

Return Values

long	Returns the number of notification objects that have been processed by this class. The m_lNotificationsRcvd member variable holds this value.

See Also

CCxNotificationHelper::Reset

CCxNotificationHelper::GetNotificationsRcvdString

Prototype

```
virtual CString   GetNotificationsRcvdString();
```

Parameters

none	

Scope

Public

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxNotificationHelper.h>
```

Description

This method returns a string representation of the count of the number of notification objects that have been processed by this object. The m_lNotificationsRcvd member variable stores this count. m_lNotificationsRcvd is used in all functions that return averages.

Each time the Reset method is called, m_lNotificationsRcvd is reset to 0.

Return Values

CString	Returns the number of notification objects that have been processed by this class. The return value is formatted as a CString.

See Also

CCxNotificationHelper::Reset

CCxNotificationHelper::GetNotificationType

Prototype

```
virtual long GetNotificationType()
```

Parameters

none	

Scope

Public

Definition

```
Dynamic Link Library        CxNotifyClient.DLL
Header Files                #include <CxNotificationHelper.h>
```

Description

This method returns the data type from the object passed to Extract. The data type comes from the CCxNotificationInfo::DataType() method. Data types are used by Observers to determine how to parse or process the byte buffer of a notification object. The byte buffer is returned by calling CCxNotificationInfo::Bytes().

Return Values

long	Returns the data type of the notification object.

See Also

CCxNotificationInfo::DataType
CCxNotificationInfo::Bytes

CCxNotificationHelper::GetSize

Prototype

```
virtual long GetSize()
```

Parameters

None	

Scope

Public

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxNotificationHelper.h>
```

Description

This method returns the size of the byte buffer in the notification object passed to Extract. The size value comes from the CCxNotificationInfo::Size() method.

Return Values

long	Returns the size of the byte buffer of the notification object.

See Also

CCxNotificationInfo::Bytes
CCxNotificationInfo::Size

CCxNotificationHelper::GetStartTimeString

Prototype

```
virtual CString GetStartTimeString()
```

Parameters

none	

Scope

Public

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxNotificationHelper.h>
```

Description

Returns a formatted time string. The time string is when the notification was initially sent by the Subject. The WIN32 **asctime** function is used to format the time. The string result produced by **asctime** contains exactly 26 characters and has the form `Wed Jan 02 02:03:55 1980\n\0`. A 24-hour clock is used by **asctime**.

Return Values

CString	A formatted time string of when the notification object was received by an Observer. The text string returned is formatted by the WIN32 asctime function and has the form `Wed Jan 02 02:03:55 1980\n\0`. A 24-hour clock is used by

	asctime.

See Also

CCxNotificationHelper::Reset

Prototype

```
virtual void Reset();
```

Parameters

none	

Scope

Public

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include < CxNotificationHelper.h >
```

Description

The Reset function resets all of our counters and internal data. Calling the Reset method sets the object's internal state to the same values as when the object was initially created.

Return Values

Void	None

See Also

CCxNotificationHelper::CCxNotificationHelper

CCxNotificationInfo Class

A CCxNotificationInfo object is the actual notification object sent from a Subject to an Observer. Any Observer receiving a notification get a CCxNotificationInfo object passed to it.

The following class members are defined exposed by the CCxNotificationInfo class.

`Bytes`	This method returns a pointer to the data passed in the notification.
`DataType`	This method returns the data type of the data sent in the notification.
`DeltaTicks`	This method returns the number of CPU instructions that have elapsed since the notification was sent by the Subject.
`DeltaTime`	This method returns the actual elapsed time.
`Extra`	This method returns the value of the "extra" parameter passed as part of the notification
`GetEndTime`	This method returns the end time as a time_t parameter.
`GetMustCopyFlag`	This method returns the value of the "must copy" flag. This flag determines whether or not a notification object MUST be copied by someone who wishes to forward the notification on to someone/something else.
`GetProcessedHandle`	This member function returns the event handle for the "Processed" event.
`GetStartTime`	This method returns the start time as a time_t parameter. The start time is when the notification began its journey through the notification package. This equates to when the Subject sent the notification by calling ICxSubjectObserver::PostNotify(…) or ICxSubjectObserver::SendNotify(…).
`NotificationType`	This method returns the type of notification.
`operator =`	The equal operator allows the caller to initialize the notification object with the values from another notification object
`PackageID`	This method returns the package ID of the Subject who sent the notification.
`Priority`	This method returns the priority of the notification object.
`Release`	This method will delete the notification object immediately. It is similar to the COM Release method, but there is no reference count maintained by the notification object. Therefore, the object

	will be immediately deleted.
ResetTime	This method will reset the start and end times of the notification.
SenderObjectID	Used to return the ID of the Subject that sent the notification.
SetBytes	This method allows the caller to put a byte buffer into the notification object.
SetMustCopyFlag	This method allows the caller to set our internal must copy flag variable.
SetProcessed	This method can be called to tell anyone listening via WaitForMultipleObjects or WaitForSingleObject that the notification has been processed.
Size	This method returns the size of the notification data byte buffer.

The following member variables are part of the CCxNotificationInfo class.

Name	Data Type	Scope	Description
m_bDeleteBytes	bool	Protected	
m_bMustCopy	bool	Protected	
m_cTime	CCxDiffTime	Protected	
m_evtProcessed	CEvent	Protected	
m_lDataType	long	Protected	
m_lExtra	long	Protected	
m_lNotificationType	long	Protected	
m_lPackageID	long	Protected	
m_lPriority	long	Protected	
m_lSenderObject	long	Protected	
m_lSize	long	Protected	
m_pBytes	byte *	Protected	

CCxNotificationInfo::Bytes

Prototype

```
byte *Bytes()
```

Parameters

None

Scope

Public

Definition

```
Dynamic Link Library        CxNotifySupport.DLL
Header Files                #include <CxNotification.h>
```

Description

This method returns a pointer to the data passed in the notification. The bytes pointer is an address that points to the beginning of the notification data. The length of the data is returned by calling the Size() method. The contents of the byte pointer buffer are completely undefined. The Subject sending the

notification determines the format and content of the byte buffer. Observers are required to understand this format in order to effectively get the notification data.

Return Values

None

Source Code Example:

```
// code goes here…
```

See Also

CCxNotificationInfo::NotificationType(), CCxNotificationInfo::Size()

CCxNotificationInfo::DataType

Prototype

```
long DataType()
```

Parameters

None

Scope

Public

Definition

Dynamic Link Library	CxNotifySupport.DLL
Header Files	#include <CxNotification.h>

Description

This method returns the data type of the data sent in the notification. This value MUST be agreed upon between the Subject and Observer. The notification package does not explicitly define and support only certain kinds of data types. Any data types can be used as long as the Subject and Observer agree to the types.

The Notifications Package does provide a "helper" class called CCxDataTypes. This class provides a number of predefined data types that can optionally be used by Subjects and Observers if they do not wish to define their own data types. These data types are COMPLETELY optional and provided only as a convenience to the users of the Notifications Package.

Return Values

long	The data type value passed into the SendNotify or PostNotify method by the Subject sending the notification

Source Code Example:

```
// code goes here…
```

See Also

CCxDataTypes class

CCxNotificationInfo::DeltaTicks

Prototype

```
DWORD DeltaTicks()
```

Parameters

None

Scope

Public

Definition

```
Dynamic Link Library          CxNotifySupport.DLL
Header Files                  #include <CxNotification.h>
```

Description

This method returns the number of CPU instructions that have elapsed since the notification was sent by the Subject. The Notifications Package uses the WIN32 QueryPerformanceCounter() function to track the time it takes for a notification to be sent from a Subject to its' Observer(s).

The time can be formatted by using the CCxNotificationHelper class.

Return Values

DWORD	Number of elapsed ticks between when the notification was sent to when it was received

Source Code Example:

```
// code goes here…
```

See Also

CCxDiffTime::DeltaTicks(),CCxNotificationHelper

CCxNotificationInfo::DeltaTime

Prototype

```
double DeltaTime()
```

Parameters

None

Scope

Public

Definition

```
Dynamic Link Library          CxNotifySupport.DLL
Header Files                  #include <CxNotification.h>
```

Description

This method returns the elapsed time based on our start and end counters and the machine frequency (CPU speed). All values are returned in microseconds

The CCxNotificationHelper class can be used to get a formatted time string.

Return Values

`double`	Number of microseconds from when the notification was sent to when it was received

Source Code Example:

```
// code goes here…
```

See Also

CCxDiffTime::DeltaTicks(),CCxNotificationHelper

CCxNotificationInfo::Extra

Prototype

```
long Extra()
```

Parameters

None

Scope

Public

Definition

```
Dynamic Link Library          CxNotifySupport.DLL
Header Files                  #include <CxNotification.h>
```

Description

This method returns the value of the "extra" parameter passed as part of the notification. Subjects must not use this parameter to pass a pointer through the notification.

Return Values

`long`	The 'extra' value passed into the SendNotify or PostNotify method by the Subject sending the notification

Source Code Example:

```
// code goes here…
```

See Also

CCxNotificationInfo::GetEndTime

Prototype

```
time_t GetEndTime()
```

Parameters

None

Scope

Public

Definition

```
Dynamic Link Library          CxNotifySupport.DLL
Header Files                  #include <CxNotification.h>
```

Description

This method returns the end time as a `time_t` parameter. The end time is when the notification is delivered to the Observer.

Return Values

`time_t`	The end time when the notification was received by the Observer

Source Code Example:

```
// code goes here…
```

See Also

CCxNotificationInfo::GetMustCopyFlag

Prototype

```
bool GetMustCopyFlag()
```

Parameters

None

Scope

Public

Definition

```
Dynamic Link Library          CxNotifySupport.DLL
Header Files                  #include <CxNotification.h>
```

Description

This method returns the value of the m_bMustCopy member variable. The m_bMustCopy member variable determines whether this notification object needs to be copied before passing it on to other Observers or objects. This variable is primarily used for optimization in the Notifications Package. The Notifications Package determines if there is only a single Observer, then there is no need to copy the notification object.

Return Values

bool	true if the object MUST be copied, else false.

Source Code Example:
```
// code goes here…
```

See Also

CCxNotificationInfo::GetProcessedHandle

Prototype

```
HANDLE GetProcessedHandle()
```

Parameters

None

Scope

Public

Definition

```
Dynamic Link Library          CxNotifySupport.DLL
Header Files                  #include <CxNotification.h>
```

Description

This member function returns the event handle for the "Processed" event. Each notification object contains an NT event object which can be used to determine when the notification has been processed. When the notification object is deleted, the destructor for the notification automatically sets the processed event handle. Observers can also call the `SetProcessed()` method to signal the processed event.

Return Values

HANDLE	The NT event handle for this notification object. This handle becomes signaled when the notification object is either deleted by the Observer or the Observer calls the SetProcessed() method on the notification object.

Source Code Example:
```
// code goes here…
```

See Also

CCxNotificationInfo::SetProcessed

CCxNotificationInfo::GetStartTime

Prototype

```
time_t GetStartTime()
```

Parameters

None

Scope

Public

Definition

```
Dynamic Link Library          CxNotifySupport.DLL
Header Files                  #include <CxNotification.h>
```

Description

This method returns the starting time of the notification as a time_t data structure. The start time is when the Subject calls either ICxSubjectObserver::SendNotify or ICxSubjectObserver::PostNotify.

Return Values

time_t	Time value when the notification was sent by the Subject

Source Code Example:

```
// code goes here…
```

See Also

CCxNotificationInfo::GetEndTime, ICxSubjectObserver::SendNotify, ICxSubjectObserver::PostNotify

CCxNotificationInfo::NotificationType

Prototype

```
long NotificationType()
```

Parameters

None

Scope

Public

Definition

```
Dynamic Link Library          CxNotifySupport.DLL
Header Files                  #include <CxNotification.h>
```

Description

This method returns the type of notification. Notification types are used as a contract between a Subject and its Observer(s). The notification package does not attempt to define valid notification types. Notification types are used as a bit-mask in the Management System within the Notifications Package. Therefore, a particular Subject can define up to 32 valid notification types (a long is currently 32 bits).

When an Observer subscribes to a subject via the ICxSubjectObserver::SubscribeByName or ICxSubjectObserver::SubscribeByID, the Observer MUST pass a notification type value. This value is stored as part of the subscription by the Notifications Package management system. When a notification is sent by a

Subject, the Subject MUST provide a notification type with the notification data. The management system compares this notification type from the Subject with the notification type subscribed to by the Observer to see if they are bit-wise compatible (a bit-wise AND '&' is performed). If the bit-wise AND '&' is successful (true), then the notification is passed on to the Observer.

The Notifications Package does provide a "helper" class called CCxNotificationTypes. This class provides a number of predefined notification types that can optionally be used by Subjects and Observers if they do not wish to define their own notification types. These notification types are COMPLETELY optional and provided only as a convenience to the users of the Notifications Package.

Return Values

long	The notification type value passed into the `SendNotify` or `PostNotify` method by the Subject sending the notification

Source Code Example:

```
// code goes here…
```

See Also

ICxSubjectObserver::SubscribeByName, ICxSubjectObserver::SubscribeByID, ICxSubjectObserver::PostNotify, ICxSubjectObserver::SendNotify

CCxNotificationInfo::operator =

Prototype

```
CCxNotificationInfo& operator = (CCxNotificationInfo &refObj);
```

Parameters

`CCxNotificationInfo &refObj`	[in] Reference to an existing CCxNotificationInfo object. This values from this input object will be used to set the values of the receiving object.

Scope

`Public`

Definition

```
Dynamic Link Library          CxNotifySupport.DLL
Header Files                  #include <CxNotification.h>
```

Description

This method assigns the values from the `refobj` parameter into the receiving object.

Return Values

`CCxNotificationInfo&`	A reference to the current object

Source Code Example:
```
// code goes here…
```

See Also

CCxNotificationInfo::PackageID

Prototype

```
long PackageID()
```

Parameters

None

Scope

Public

Definition

Dynamic Link Library	CxNotifySupport.DLL
Header Files	#include <CxNotification.h>

Description

This method returns the package ID of the Subject who sent the notification. Package ID's can be used to group Subjects. The CxNotifyServer.h header file provides Cimetrix defined Package IDs. Clients are fee to define their own Package Ids using a value or CX_PKGID_USER + n. A Subject sets its PackageID with a call to ICxSubjectObserver::SetMyName.

Package ID's are very powerful because they extend the notification types that can be handled by an Observer. Remember that a particular Subject can send up to 32 different notification types. Generally, Observers will want to handle more than 32 different notification types. The combination of PackageID with a notification type allows an Observer to determine which package generated a notification and then determine which Notification type must be handled. This allows different packages to use the same notification type numbers. This also allows a single Observer to react differently to a notification generated by two different subjects from different packages that happen to use the same notification type.

Return Values

long	The package ID value passed into the `SendNotify` or `PostNotify` method by the Subject sending the notification

Source Code Example:
```
// code goes here…
```

See Also

ICxSubjectObserver::SetMyName

CCxNotificationInfo::Priority

Prototype

```
long Priority()
```

Parameters

None

Scope

Public

Definition

```
Dynamic Link Library          CxNotifySupport.DLL
Header Files                  #include <CxNotification.h>
```

Description

This method returns the priority of the notification object. Subjects have the ability to define the priority of the notification they send. The priority of the notification determines the order that notifications get sent to Observers. The higher the priority number of notification, the higher the priority and the faster the notification gets sent on to Observers.

Return Values

long	The priority value passed into the SendNotify or PostNotify method by the Subject sending the notification

Source Code Example:

```
// code goes here…
```

See Also

CCxNotificationInfo::Release

Prototype

```
void Release(bool bDeleteBytes = true)
```

Parameters

Bool bDeleteBytes	[in] Passing in a value of 'true' tells the notification object to also delete its' Bytes data. Passing in a value of 'false' does not delete the bytes data. The default value of this parameter is true.

Scope

Public

Definition

```
Dynamic Link Library          CxNotifySupport.DLL
Header Files                  #include <CxNotification.h>
```

Description

This method will destroy the notification object by calling 'delete this'. This is an alternative way of destroying a notification object. It provides a way to delete the notification object without also destroying its associated byte buffer. This can be useful if an Observer wishes to continue using the byte buffer, but no longer needs the actual notification object.

Return Values

None

Source Code Example:

```
// code goes here…
```

See Also

CCxNotificationInfo::ResetTime

Prototype

```
void ResetTime()
```

Parameters

None

Scope

Public

Definition

```
Dynamic Link Library          CxNotifySupport.DLL
Header Files                  #include <CxNotification.h>
```

Description

This method will reset the start and end times of the notification. Subsequent calls to GetDeltaTicks will always return zero after this call completes.

Return Values

None

Source Code Example:

```
// code goes here…
```

See Also

CCxNotificationInfo::GetStartTime, CCxNotificationInfo::GetEndTime, CCxNotificationInfo::GetDeltaTicks

CCxNotificationInfo::SenderObjectID

Prototype

```
long SenderObjectID()
```

Parameters

None

Scope

Public

Definition

```
Dynamic Link Library          CxNotifySupport.DLL
Header Files                  #include <CxNotification.h>
```

Description

This method returns the ID of the Subject who sent the notification. The ID is a unique number in the notification system. Each Subject and Observer in the notification system is guaranteed to have a unique number associated with it. The ICxSubjectObserver::GetNameFromID interface method can be used to get the string name from the ID.

Return Values

long	ID of the Subject who sent the notification

Source Code Example:

```
// code goes here…
```

See Also

ICxSubjectObserver::GetNameFromID

CCxNotificationInfo::SetMustCopyFlag

Prototype

```
void SetMustCopyFlag(bool bCopy)
```

Parameters

bool bCopy	[in] Passing in a value of 'true' forces the management system to make a copy of the notification object before passing it on to any other object.

Scope

Public

Definition

```
Dynamic Link Library          CxNotifySupport.DLL
Header Files                  #include <CxNotification.h>
```

Description

This method can be used to force the management system to make a copy of the notification object prior to forwarding a notification on to an Observer.

Note that most customers will not have a need to use this method because they will not have access to the Management system API's and classes.

Return Values

None

Source Code Example:

```
// code goes here…
```

See Also

`CCxNotificationInfo::byte *&Bytes`

CCxNotificationInfo::SetProcessed

Prototype

```
void SetProcessed()
```

Parameters

None

Scope

Public

Definition

```
Dynamic Link Library          CxNotifySupport.DLL
Header Files                  #include <CxNotification.h>
```

Description

This method can be called to tell anyone listening via WaitForMultipleObjects or WaitForSingleObject that the notification has been processed. This method is very important for synchronous notifications. Subjects generate a synchronous notification by calling ICxSubjectObserver::SendNotify(). Synchronous notifications will wait until the notification is handled. A notification is handled when either the SetProcessed method is called on the notification, or the notification is deleted. SetProcessed is automatically called by the destructor or the CCxNotificationInfo class.

Return Values

None

Source Code Example:

```
// code goes here…
```

See Also

ICxSubjectObserver::SendNotify()

CCxNotificationInfo::Size

Prototype

```
long Size()
```

Parameters

None

Scope

Public

Definition

```
Dynamic Link Library          CxNotifySupport.DLL
Header Files                  #include <CxNotification.h>
```

Description

This method returns the size of the notification data byte buffer. The size of the buffer is passed in when a Subject generates a notification with a call to either ICxSubjectObserver::SendNotify() or ICxSubjectObserver::PostNotify().

Return Values

`long`	Size of the byte buffer

Source Code Example:

```
// code goes here…
```

See Also

CCxNotificationInfo::Bytes(), ICxSubjectObserver::SendNotify(), ICxSubjectObserver::PostNotify()

CCxNotificationRoute Class

The CCxNotificationRoute class is used by the CCxRoutedSink object to deliver notifications. The CCxNotificationRoute class should ALWAYS be used in conjunction with the CCxRoutedSink class. A user of these two classes must do the following:

1. Create a new class that inherits from CCxNotificationRoute
2. Implement one or more of the CCxNotificationRoute virtual methods according to the individual needs of the user
3. Create a CCxRoutedSink object
4. Call the CCxRoutedSink::SetNotificationRoute method and pass a pointer to the class created in step 1.

The CCxNotificationRoute class is essentially a virtual base class that provides 4 methods with empty implementations. The 4 methods are described in the following table.

Method Name	Description
OnNotifyNOTIFY	All notifications from a Subject to a routed sink Observer are delivered to this method.
OnNotifyNOTIFY_SUBJECTBROKEN	Each time a Subject is deleted from the system, all subscribed Observers of this Subject are notified via this method
OnThreadCleanup	Allows the creator of this object to provide a "callback" function that the CCxRoutedSink object will call to perform thread cleanup on the sink thread.
OnThreadInit	Allows the creator of this object to provide a "callback" function that the CCxRoutedSink object will call to perform thread initialization on the sink thread.

The CCxRoutedSink object makes calls on these 4 methods. It has this ability because whenever a CCxRoutedSink object is created, its SetNotificationRoute method should be called and passed a pointer to an object that inherits from CCxNotificationRoute.

Each of the CCxNotificationRoute methods are implemented as follows. This source code is the actual code implemented by the Notifications Package default implementation:

```
void CCxNotificationRoute::OnNotifyNOTIFY(
                  CCxNotificationInfo* pNotification)
{
      // Here's a default implementation
      if (pNotification)
            delete pNotification;
}
```

```
void CCxNotificationRoute::OnNotifyNOTIFY_SUBJECTBROKEN(
                  long lSubjectID)
{
      // Here's a default implementation
}

HRESULT CCxNotificationRoute::OnThreadInit()
{
      return E_NOTIMPL;
}

HRESULT CCxNotificationRoute::OnThreadCleanup()
{
      return E_NOTIMPL;
}
```

Note that there is essentially no implementation for these 4 methods. It is completely up to the derived class to determine their particular needs and their particular implementation of these methods. The one exception to this is in the OnNotifyNOTIFY method. The OnNotifyNOTIFY method must always delete the notification object that it receives otherwise memory leaks will occur.

CCxNotificationRoute::CCxNotificationRoute

Prototype

```
CCxNotificationRoute::CCxNotificationRoute()
```

Parameters

None	

Scope

Public

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxNotificationRoute.h>
```

Description

This is the default constructor. All initialization is performed in this constructor.

See Also

CCxNotificationRoute::~CCxNotificationRoute

Prototype

```
virtual ~CCxNotificationRoute()
```

Parameters

None	

Scope

Public

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxNotificationRoute.h>
```

Description

This is the destructor. All object cleanup is performed in this method.

See Also

CCxNotificationRoute::OnNotifyNOTIFY

Prototype

```
virtual void OnNotifyNOTIFY(CCxNotificationInfo* pNotification);
```

Parameters

CCxNotificationInfo* pNotification	[in] pNotification is the actual notification object from a Subject.

Scope

Public

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxNotificationRoute.h>
```

Description

This method is used by the Notifications Package for delivery of a notification. The pNotification object is an actual notification from a Subject.

Note that this method is defined as virtual. The default implementation provided in the Notifications Package contains no logic. This method MUST be overridden by an object that inherits from CCxNotificationRoute. The overridden implementation and its functionality are solely dependent upon the needs of the Observer.

This method is where the notification object is "caught" and handled. The implementer of this method is responsible for extracting any needed data from the pNotification object. For details on getting information out of the notification object, refer to the documentation on the CCxNotificationInfo class. Refer to the source code example below for details on how to implement your own OnNotifyNOTIFY() method.

The OnNotifyNOTIFY member function is where we are told that a notification has occurred. Any and all subjects that we have subscribed to, will send notifications back to us through this member function.

WARNING: Be very careful about what you do in your implementation of OnNotifyNOTIFY. If you are making GUI calls, and the GUI window has been destroyed or is in the process of being destroyed, your application will hang or generate an access violation in your implementation of OnNotifyNOTIFY.

WARNING: Also be very careful about time sensitive operations in your implementation of OnNotifyNOTIFY. This method is running on the event sink thread. If you are in this method for a long time, you are blocking other notifications from being handled by this object.

SPECIAL NOTE: Once the pNotification object is no longer needed it MUST be deleted in this method. Otherwise, memory leaks will occur. There are two ways that the notification object can be deleted. The first is by calling the C++ delete method as follows:

```
delete pNotification;
```

The second is to use the CCxNotificationInfo::Release() method as follows:

```
pNotification->Release();
```

Return Values

`void`	None

Source Code Example:

The following source code illustrates the definition of a class that inherits from CCxNotificationRoute and an implementation of the OnNotifyNOTIFY method. Note that most of this source code is cut and pasted from the MultiNotify sample application provided with the Notifications Package. The source code came from CxExample.h and CxRoutedExample.cpp.

Also note that most of the functionality in our implementation of OnNotifyNOTIFY can be found in the CCxNotificationHelper class.

```
#include <CxNotificationRoute.h>


Class CMyClass : public CCxNotificationRoute
{
public:
      CMyClass();
      virtual ~CMyClass();
      virtual HRESULT CreateSubjectObserver(CString strName);

      // CxNotificationRoute virtuals
      void OnNotifyNOTIFY(CCxNotificationInfo* pNotification);
      void OnNotifyNOTIFY_SUBJECTBROKEN(long dwSubjectID);
```

```
protected:
      double      m_dblMinTime;
      double      m_dblMaxTime;
      double      m_dblTotalElapsedTime;
      double      m_dblElapsedTime;
      long        m_lNotificationsRcvd;
      long        m_lSizeRcvd;
      long        m_lDataTypeRcvd;
      long        m_lNTypeRcvd;
      long        m_lExtraRcvd;
      int         m_nSinkType;
      CString     m_strEndTime;
      CString     m_strStartTime;
      CString     m_strData;
      ICxSubjectObserver      *m_pSO;
      CComObject<CCxRoutedSink> *m_pSink;
}
```

The following source code provides an example implementation of OnNotifyNOTIFY().

```
CMyClass::OnNotifyNOTIFY(CCxNotificationInfo *pNotification)
{
      // Do this as soon as possible
      m_dblElapsedTime  = pNotification->DeltaTime();
      TRACE("CMyClass::HandleNotificationMessage - delta time =
%.0f\n", pNotification->DeltaTime());

      // Keep track of the total elapsed time so we can
      // figure averages
      m_dblTotalElapsedTime += m_dblElapsedTime;
      m_lNotificationsRcvd++;

      char buff[100];
      CString s;
      int nData, i;
      long lData;
      double dblData;
      time_t tTime;
      struct tm tmTime;

      // Get the start time and store it in a string variable
      tTime       = pNotification->GetStartTime();
      tmTime      = *localtime( &tTime );
      sprintf( buff, "%s", asctime( &tmTime ) );
      m_strStartTime    = buff;

      // Get the end time and store it in a string variable
      tTime       = pNotification->GetEndTime();
      tmTime      = *localtime( &tTime );
      sprintf( buff, "%s", asctime( &tmTime ) );
      m_strEndTime      = buff;

      // Get the rest of the data
      m_lSizeRcvd       = pNotification->Size();
      m_lDataTypeRcvd   = pNotification->DataType();
```

```
m_lNTypeRcvd      = pNotification->NotificationType();
m_lExtraRcvd      = pNotification->Extra();

// Keep track of our min and max values
if ((m_dblElapsedTime < m_dblMinTime)||(m_dblMinTime == 0))
      m_dblMinTime = m_dblElapsedTime;
if (m_dblElapsedTime > m_dblMaxTime)
      m_dblMaxTime = m_dblElapsedTime;

// Do this for our TRACE statements
BSTR bstrName;

// Note that this call will fail with a 0x800401F0
// saying that "CoInitialize has not yet been called"
// if you did not initialize your application with the
// CoInitializeEx(NULL, COINIT_MULTITHREADED) call.
// Refer to the CMultiNotifyApp::InitInstance method
// for our COM initialization information.
HRESULT hr = m_pSubjectObserver->GetMyName(&bstrName);
ASSERT(SUCCEEDED(hr));

memset(buff, 0, 100);
m_strData.Empty();

// Convert the data from the format it was sent in
// to a string so anyone can display it.
switch(m_lDataTypeRcvd)
{
      case CX_DATA_STRING:    // Same as CHARSTAR
      case CX_DTYPE_CHARSTAR:
            for(i = 0; i < pNotification->Size(); i++ )
                  m_strData += *(pNotification->Bytes()+i);
            TRACE("CCxExampleBase::OnNotifyNOTIFY() - \
                  Observer: %s has received char * data:\
                   %s\n", CString(bstrName), m_strData);
            break;

      case CX_DTYPE_INT:
            memcpy(&nData, pNotification->Bytes(),
                   m_lSizeRcvd);
            TRACE("CCxExampleBase::OnNotifyNOTIFY() -\
                   Observer: %s has received integer data:\
                   %d\n", CString(bstrName), nData);

            if (sprintf(buff, "%d", nData) > 0)
                  m_strData = buff;
            break;

      case CX_DTYPE_LONG:
            memcpy(&lData, pNotification->Bytes(),
                   m_lSizeRcvd);
            TRACE("CCxExampleBase::OnNotifyNOTIFY() - \
                  Observer: %s has received long data: \
                  %d\n", CString(bstrName), lData);

            if (sprintf(buff, "%d", lData) > 0)
                  m_strData = buff;
```

```
                  break;

            case CX_DTYPE_DOUBLE:
                  memcpy(&dblData, pNotification->Bytes(),
                        m_lSizeRcvd);
                  TRACE("CCxExampleBase::OnNotifyNOTIFY() - \
                        Observer: %s has received double data: \
                        %e\n", CString(bstrName), dblData);

                  if (sprintf(buff, "%e", dblData) > 0)
                        m_strData = buff;
                  break;

            default:
                  // Error
                  AfxMessageBox("CCxExampleBase::OnNotifyNotify()
- Notification Failure: Invalid data type received");
                  m_strData.Empty();
                  break;
      }

      // We MUST delete the notification object when we are
      // finished with it.
      delete pNotification
}
```

See Also

CCxNotificationInfo class
CCxNotificationHelper class
CCxNotificationRoute::OnNotifyNOTIFY_SUBJECTBROKEN

CCxNotificationRoute::OnNotifyNOTIFY_SUBJECTBROKEN

Prototype

```
virtual void OnNotifyNOTIFY_SUBJECTBROKEN(long lSubjectID);
```

Parameters

long lSubjectID	[in] long lSubjectID - The Subject ID of the object that has been destroyed.

Scope

Public

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxNotificationRoute.h>
```

Description

This method is called by a Notification Package sink object whenever a Subject that this Observer is subscribed to is removed from the Notification

Package. It allows an Observer to perform special handling or cleanup whenever a Subject "goes away". The Notification Package Management System "notifies" all Observer(s) that are subscribed to a Subject whenever the Subject is destroyed. To determine the name of the Subject that was destroyed, call the ICxSubjectObserver::GetNameFromID method or the CCxSubjectOnly::GetNameFromID method.

Note that the default implementation provided by CCxNotificationRoute::OnNotifyNOTIFY_SUBJECTBROKEN does nothing. Refer to the comments at the beginning of this section for the source code of the default implementation.

Clients that need to perform any special handling or cleanup upon Subject destruction should implement this method according to their own needs.

Return Values

`void`	None

Source Code Example:

The following source code illustrates one potential implementation of the OnNotifyNOTIFY_SUBJECTBROKEN method. This example builds upon the source code example shown above in the OnNotifyNOTIFY method.

```
CMyClass::OnNotifyNOTIFY_SUBJECTBROKEN(
                        long lSubjectID)
{
      // Use our ICxSubjectObserver object to get the
      // name of the Subject whose ID is lSubjectID
      if (m_pSO)
      {
            BSTR bstrName;
            m_pSO->GetNameFromID(lSubjectID , &bstrName);
            TRACE("Subject id: %d and name: %s has left \
                  the building.\n",
                  dwSubjectID,
                  CString(bstrName));
      }

}
```

See Also

CCxNotificationInfo class
ICxSubjectObserver::GetNameFromID
CCxSubjectOnly::GetNameFromID

CCxNotificationRoute::OnThreadCleanup

Prototype

```
virtual HRESULT OnThreadCleanup();
```

Parameters

none	

Scope

`Public`

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxNotificationRoute.h>
```

Description

This method is provided to allow a client who has created a CCxRoutedSink object, the ability to cleanup the CCxRoutedSink object's thread. The CCxRoutedSink object exposes a method called ThreadCleanup() that can be called by a client. Whenever a client calls CCxRoutedSink::ThreadCleanup, the routed sink object's thread will "callback" into this method. This allows the caller to cleanup the thread in any way they see fit. Typically, this method will perform cleanup on any actions performed in the ThreadInit implementation.

Remember that the CCxRoutedSink object itself knows about a derived class's implementation of CCxNotificationRoute because when the CCxRoutedSink object was first created, it was passed a pointer to a class that implements the CCxNotificationRoute class. For an example of this, refer to the "*ICxSubjectObserver COM Interface*" section. The example in that section shows how to instantiate a CCxRoutedSink object and call its `SetNotificationRoute` method.

Return Values

`HRESULT`	The default implementation returns E_NOTIMPL. This method is virtual and therefore can be overridden by a derived class. The derived class is free to return any HRESULT they see fit.

Source Code Example:

The following example shows one possible implementation of OnThreadCleanup. It builds upon the CMyClass definition shown above in the OnNotifyNOTIFY source code example.

```
HRESULT CMyClass::OnThreadCleanup()
{
      // Because we have an OnThreadInit and we called
      // CoInitializeEx there, we MUST call CoUninitialize here
      CoUninitialize();
      return S_OK;
}
```

In order to force the CCxRoutedSink object to call our implementation of OnThreadCleanup, we must call the CCxRoutedSink object's ThreadCleanup method. Take a look at the source code example in the OnThreadInit method below. That source code example shows how we create our

ICxSubjectObserver object and then create a CCxRoutedSink and save the pointer in our member variable m_pSink. When we want the CCxRoutedSink's thread to call our OnThreadCleanup we make the following call:

```
m_pSink->ThreadCleanup();
```

See Also

CCxRoutedSink class
CCxRoutedSink::ThreadCleanup
OnThreadInit

CCxNotificationRoute::OnThreadInit

Prototype

```
virtual HRESULT OnThreadInit()
```

Parameters

none	

Scope

```
Public
```

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxNotificationRoute.h>
```

Description

Since the CCxNotificationRoute object is provided solely for the use and interaction with the CCxRoutedSink, this method is provided to allow a client who has created a CCxRoutedSink object, the ability to initialize the CCxRoutedSink object's thread. The CCxRoutedSink object exposes a method called ThreadInit() that can be called by a client. Whenever a client calls CCxRoutedSink::ThreadInit, the routed sink object's thread will "callback" into this method. This allows the caller to initialize the thread in any way they see fit. The thread priority, thread class, thread settings for COM, or any other settings can be put into the implementation of this method.

Remember that the CCxRoutedSink object itself knows about a derived class's implementation of CCxNotificationRoute because when the CCxRoutedSink object was first created, it was passed a pointer to a class that implements the CCxNotificationRoute class. For an example of this, refer to the "*ICxSubjectObserver COM Interface*" section. The example in that section shows how to instantiate a CCxRoutedSink object and call its `SetNotificationRoute` method.

Return Values

`HRESULT`	The default implementation returns E_NOTIMPL. This method is virtual and therefore can be overridden by a derived class. The derived class is free to return any HRESULT they see fit.

Source Code Example:

The following example shows one possible implementation of OnThreadInit. It builds upon the CMyClass definition shown above in the OnNotifyNOTIFY source code example.

```
HRESULT CMyClass::OnThreadInit()
{
      // Initialize this thread as a COM
      // Multi-threaded apartment (MTA) thread
      HRESULT hr = CoInitializeEx(NULL, COINIT_MULTITHREADED);

      // TODO: Add any other thread initialization code here


      return hr;
}
```

In order to force the CCxRoutedSink object to call our implementation of OnThreadInit, we must call the CCxRoutedSink object's ThreadInit method. Note that the source code example provided in our OnNotifyNOTIFY method defined the CMyClass class. In the definition of that class we had a method called CreateSubjectObserver. In our CreateSubjectObserver method we could call the CCxRoutedSink::ThreadInit method to force the routed sink's thread to call back into our implementation of OnThreadInit as follows:

```
HRESULT CMyClass::CreateSubjectObserver(CString strName)
{
      HRESULT hr;

      hr = CoCreateInstance(CLSID_CCxSubjectObserver,
                            NULL,
                            CLSCTX_LOCAL_SERVER,
                            IID_ICxSubjectObserver,
                           reinterpret_cast<void**>(&m_pSO));

      if (FAILED(hr))
      {
            m_pSO = NULL;

            // Poor debugging, but you get the idea...
            ASSERT(FALSE);
      }


      // Create the sink and connect it to the subject-observer
      if( m_pSO )
      {
      hr = CComObject<CCxRoutedSink>::CreateInstance(&m_pSink);
            if (FAILED(hr))
                  ASSERT(FALSE);

            // We can do this because we inherit from
            // CxNotificationRoute.  This is what tells
            // the notification system how to get back to
```

```
            // us and call our OnNotifyNOTIFY(),
            // OnNotifyNOTIFY_SUBJECTBROKEN(),
            // OnThreadInit(), and OnThreadCleanup() methods
            m_pSink->SetNotificationRoute(this);

            // Tell the sink object's thread to call our
            // OnThreadInit method
            hr = m_pSink->ThreadInit();
            if (FAILED(hr))
                  ASSERT(FALSE);

            // We use the ATL call to connect the
            // ICxSubjectObserver to the sink
            DWORD dwCookie = -1;
            hr = AtlAdvise(m_pSubjectObserver,
                        m_pSink ->GetUnknown(),
                        IID_ICxObserverNotification,
                        &dwCookie);

            if (FAILED(hr))
                  ASSERT(FALSE);
      }

      return hr;
}
```

See Also

CCxRoutedSink class
CCxRoutedSink::ThreadInit
OnThreadCleanup

CCxNotificationThread Class

The CCxNotificationThread class is a helper class that creates a worker thread with a message loop.

The CCxNotificationThread class can be used with the CCxThreadMsgSink object to receive notifications from the sink. A user of these two classes must do the following:

1. Create a new class that inherits from CCxNotificationThread
2. Implement one or more of the CCxNotificationThread virtual methods according to the individual needs of the user. There are two virtual methods that can be overridden - HandleNotification and HandleNotificationSubjectBroken.
3. Create a CCxThreadMsgSink object and attach it to an ICxSubjectObserver object.
4. Call the CCxThreadMsgSink::SetThreadHandle method and pass the thread ID that was created by instantiating the class that derives from the CCxNotificationThread object in step 1.

The CCxNotificationThread class is a helper class that "catches" thread message notifications and allows a derived class to implement their own handling of these notifications. The derived class implements the HandleNotification and/or HandleNotificationSubjectBroken in any way they require.

Method Name	Description
`CCxNotificationThread`	Default constructor
`~CCxNotificationThread`	Destructor
`FinalConstruct`	Performs final construction of the object
`FinalRelease`	Performs final cleanup of the object
`GetThreadID`	Returns the ID of the thread that was created.
`HandleNotification`	Virtual method that receives and handles a notification from a Subject.
`HandleNotificationSubjectBroken`	Virtual method that receives and handles a subject broken notification from the Notifications Package management system.
`ThreadProc`	This is the thread procedure for the object.
`ThreadStarter`	Static function call that invokes Threadproc.

Name	Data Type	Scope	Description
m_dwThreadID	DWORD	Protected	Holds the thread ID that gets created by this object.
m_evtStartStopThread	CEvent	Protected	An MFC CEvent

			object that is signaled when the object's thread has begun.

The CCxThreadMsgSink object sends notifications to the derived CCxNotificationThread class by using the WIN32 PostThreadMessage API. Our CCxNotificationThread::ThreadProc has built in handlers for two thread messages:

1. WM_CXNOTIFICATION - sent with a CCxNotificationInfo object
2. WM_CXSUBJECTBROKEN - sent with the ID of the Subject that is being destroyed.

The default implementations of HandleNotification and HandleNotificationSubjectBroken are as follows:

```
void CCxNotificationThread::HandleNotification(
                        CCxNotificationInfo *pNotification)
{
      if (pNotification)
      {
            delete pNotification;
            pNotification = NULL;
      }
}


void CCxNotificationThread::HandleNotificationSubjectBroken(
                                                  long lSubjectID)
{
      // Simply return and do nothing
}
```

Note that there is essentially no implementation for these two methods. It is completely up to the derived class to determine their particular needs and their particular implementation of these methods. The one minor implementation detail is in the HandleNotification method. The HandleNotification method must always delete the notification object that it receives otherwise memory leaks will occur. The derived method should also delete the notification object.

CCxNotificationThread::CCxNotificationThread

Prototype

```
CCxNotificationThread::CCxNotificationThread()
```

Parameters

none	

Scope

Public

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxNotificationThread.h>
```

Description

The default constructor creates the CCxNotificationThread object by calling the protected FinalConstruct() method. FinalConstruct() is where all of the creation work actually takes place.

Return Values

None	

See Also

FinalConstruct()

CCxNotificationThread::~CCxNotificationThread

Prototype

```
CCxNotificationThread::~CCxNotificationThread()
```

Parameters

None	

Scope

Public

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxNotificationThread.h>
```

Description

The destructor cleans up the object by calling the protected FinalRelease() method. FinalRelease() attempts to shutdown our worker thread.

Return Values

None	

See Also

FinalRelease()

CCxNotificationThread::FinalConstruct

Prototype

```
HRESULT FinalConstruct();
```

Parameters

None	

Scope

Protected

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxNotificationThread.h>
```

Description

This method creates a worker thread by calling the WIN32 API CreateThread(). CreateThread is passed a pointer to our static method CCxNotificationThread::ThreadStarter and also a pointer to the 'this' object. ThreadStarter in turn calls our ThreadProc method. By doing the roundabout startup, we are able to call ThreadProc and actually have a 'this' pointer available in ThreadProc. Static functions required by CreateThread would not have a 'this' pointer available. Our call to CreateThread looks as follows:

```
CreateThread(NULL, 0, ThreadStarter, this, 0, &m_dwThreadID );
```

Our worker thread will be destroyed when either a WM_QUIT message is posted to our thread procedure or FinalRelease is called.

Note that this method does NOT return until the newly spawned thread has actually begun execution. After creating the thread, we perform a WaitForSingleObject on the m_evtStartStopThread member variable. This member variable gets signaled once the thread has begun execution in our ThreadProc method. By waiting on the m_evtStartStopThread event, we can be assured that we do not return control back to our caller until after the worker thread has begun execution.

Return Values

HRESULT	S_OK if the worker thread was successfully created, else E_FAIL

See Also

FinalRelease, ThreadStarter, ThreadProc

CCxNotificationThread::FinalRelease

Prototype

```
void FinalRelease();
```

Parameters

None	

Scope

Protected

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxNotificationThread.h>
```

Description

This method creates simply posts a WM_QUIT message to our worker thread by making the following WIN32 call:

```
PostThreadMessage(m_dwThreadID, WM_QUIT, 0, 0);
```

Return Values

Void	None

See Also

FinalConstruct

CCxNotificationThread::GetThreadID

Prototype

```
DWORD GetThreadID();
```

Parameters

None	

Scope

Public

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxNotificationThread.h>
```

Description

This method returns the value in our m_dwThreadID member variable. This should be used by the derived class and the DWORD value returned from this method should be passed in to the CCxThreadMsgSink::SetThreadHandle method.

The value in the m_dwThreadID member variable is set during execution of the FinalConstruct method which creates our worker thread.

Return Values

Void	None

See Also

FinalConstruct
CCxThreadMsgSink::SetThreadHandle

CCxNotificationThread::HandleNotification

Prototype

```
virtual void HandleNotification(
                  CCxNotificationInfo *pNotification);
```

Parameters

CCxNotificationInfo *pNotification	[in] pointer to the notification object.

Scope

Protected

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxNotificationThread.h>
```

Description

This method is virtual and should be overridden by a derived class. The derived class' implementation should be coded according to the needs of the user.

The default implementation of this method simply deletes the notification object. The derived implementation must also delete the notification object once it is finished with it. For details on deleting a notification, refer to the CCxNotificationInfo class documentation.

The default implementations of HandleNotification is as follows:

```
void CCxNotificationThread::HandleNotification(
                        CCxNotificationInfo *pNotification)
{
      if (pNotification)
      {
            delete pNotification;
            pNotification = NULL;
      }
}
```

Note that there very minimal implementation of this method. It is completely up to the derived class to determine their particular needs and their particular implementation of this method. The HandleNotification method must always delete the notification object that it receives otherwise memory leaks will occur. The derived method should also delete the notification object.

Return Values

Void	None

See Also

ThreadProc
CCxNotificationInfo class

CCxNotificationThread::HandleNotificationSubjectBroken

Prototype

```
virtual void HandleNotificationSubjectBroken(long lSubjectID);
```

Parameters

long lSubjectID	[in] ID of the Subject who we are no longer subscribed to.

Scope

Protected

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxNotificationThread.h>
```

Description

This method is virtual and should be overridden by a derived class. The derived class' implementation should be coded according to the needs of the user.

The default implementation of this method does absolutely nothing. For implementations that need to handle being "disconnected" from a Subject - otherwise known as destroying our subscription to the Subject, this method should be overridden and implemented.

The default implementations of HandleNotificationSubjectBroken is as follows:

```
void CCxNotificationThread::HandleNotificationSubjectBroken(
                                                    long lSubjectID)
{
      // Simply return and do nothing
}
```

Return Values

void	None

See Also

CCxNotificationThread::ThreadProc

Prototype

```
DWORD ThreadProc();
```

Parameters

None	

Scope

Protected

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxNotificationThread.h>
```

Description

This method provides the main loop for our worker thread. It simply performs a while loop that intercepts notifications messages and forwards them on to either the HandleNotification method or the HandleNotificationSubjectBroken method.

This method will terminate executing when it receives a WM_QUIT message.

The default CCxNotificationThread::ThreadProc implementation looks as follows:

```
DWORD CCxNotificationThread::ThreadProc()
{
      MSG msg;

      m_evtStartStopThread.SetEvent();
      while (GetMessage(&msg, 0, 0, 0))
      {
            switch(msg.message)
            {
                  case WM_CXNOTIFICATION:
                        {
                              CCxNotificationInfo *pData = \
                              (CCxNotificationInfo *)msg.wParam;

                              // Call the virtual function that
                              // our derived class should
                              // implement
                              HandleNotification(pData);
                        }
                        break;

                  case WM_CXSUBJECTBROKEN:
                        {
                              long lSubjectID = (long)msg.wParam;

                              // Call the virtual function that
                              // our derived class should
```

```
                        //implement
                  HandleNotificationSubjectBroken(lSubjectID);
                        }
                        break;

                  default:
                        DispatchMessage(&msg);
                        break;
            }
      }

      return 0;

}
```

The ThreadProc simply catches notifications from the CCxThreadMsgSink and forwards them on to the virtual HandleNotification or HandleNotificationSubjectBroken.

Return Values

DWORD	Always 0 - the thread exists with a return code of 0.

See Also

ThreadStarter, HandleNotification, HandleNotificationSubjectBroken, FinalConstruct

CCxNotificationThread::ThreadStarter

Prototype

```
static DWORD _stdcall ThreadStarter( LPVOID lpParam );
```

Parameters

LPVOID lpParam	[in] lpParam is a pointer to our calling CCxNotificationThread object.

Scope

Protected

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxNotificationThread.h>
```

Description

This static method simply invokes the ThreadProc method as follows:
`return ((CCxNotificationThread*)lpParam)->ThreadProc();`

Note how we cast the lpParam parameter back to our original CCxNotificationThread object. The purpose of doing this is so that our worker thread has the 'this' pointer available to it in the ThreadProc method. Static

methods do not have a 'this' pointer. We need the 'this' pointer in order to call our HandleNotification or HandleNotificationSubjectBroken virtual methods.

Refer to the FinalConstruct method to see how we pass the this pointer in with the call to the WIN32 API CreateThread.

Return Values

DWORD	Always 0 - the thread exists with a return code of 0.

See Also

ThreadProc
HandleNotification
HandleNotificationSubjectBroken

CCxNotificationTypes Class

The CCxNotificationTypes class is a helper class that provides notification type flags that could be passed as part of a notification. This class is completely optional and provided only as a reference to programmers using the Notifications Package.

The class defines a few notification types and provides helpful class methods for enumerating the notification types.

Once again, this class is provided as a helper only. The Notifications Package does NOT attempt to define and support specific notification types. The Notifications Package is completely unaware of notification types as a notification is passed from a Subject to an Observer.

The following table summarizes the available methods

Method Name	Description
`FindNotificationTypeVal`	Given a string find its associated notification type value.
`FindNotificationTypeString`	Given a notification type value find its associated string representation.

The following table summarizes the member variables

Name	Data Type	Scope	Description
m_aNotificationTypes	structNotificationTypes *	Protected	An array of structures of data types. The structure is defined as: typedef struct structNotificationTypes { int m_id; int m_val; LPCTSTR m_str; } structNotificationTypes;

This class "wraps" the following enumerated data types. Each of these enumerated values is placed into the m_id member of the structNotificationTypes structure.

```
enum  // Local enums
{
      LOCAL_NTYPE_ALL   = 0,
      LOCAL_NTYPE_READY,
      LOCAL_NTYPE_STARTED,
      LOCAL_NTYPE_HALTED,
      LOCAL_NTYPE_PAUSED,
      LOCAL_NTYPE_RESUMED,
      LOCAL_NTYPE_COMPLETED,
      LOCAL_NTYPE_EVENT,
      LOCAL_NTYPE_DATA,
      LOCAL_NTYPE_LOGINFO,
      LOCAL_NTYPE_ALIVE,
      LOCAL_NTYPE_WINMSG,
```

```
      LOCAL_NTYPE_THREADMSG,
      LOCAL_NTYPE_STARTING,
      LOCAL_NTYPE_HALTING,
      LOCAL_NTYPE_CMD,
      LOCAL_NTYPE_END_VALUE         // NTYPE_END_VALUE must be
the last value
};
```

The m_val elements are filled with the following #define'd values

```
#define CX_NTYPE_READY          0x00000001L
#define CX_NTYPE_STARTED        0x00000002L
#define CX_NTYPE_HALTED         0x00000004L
#define CX_NTYPE_PAUSED         0x00000008L
#define CX_NTYPE_RESUMED        0x00000010L
#define CX_NTYPE_COMPLETED      0x00000020L
#define CX_NTYPE_EVENT          0x00000040L
#define CX_NTYPE_DATA           0x00000080L
#define CX_NTYPE_LOGINFO        0x00000100L
#define CX_NTYPE_ALIVE          0x00000200L
#define CX_NTYPE_WINMSG         0x00000400L
#define CX_NTYPE_THREADMSG      0x00000800L
#define CX_NTYPE_STARTING       0x00001000L
#define CX_NTYPE_HALTING        0x00002000L
#define CX_NTYPE_CMD            0x00004000L

#define CX_NTYPE_ALL            0xFFFFFFFFL
```

The structure defined in the member variable table holds both the enumerated value of a notification type, a #define'd value, and a corresponding string representation of the notification type. The string representation can be used to populate a list box in a GUI for example.

The following code is used in our constructor to load the m_aNotificationTypes member variable:

```
CCxNotificationTypes:: CCxNotificationTypes ()
{
      // Allocate the array or structures
      m_aNotificationTypes = new
structNotificationTypes[LOCAL_NTYPE_END_VALUE + 1];

      static structNotificationTypes tmpNT[] =
      {
      { LOCAL_NTYPE_ALL,  CX_NTYPE_ALL, _T("All") },
      { LOCAL_NTYPE_READY,       CX_NTYPE_READY,     _T("Ready") },
      { LOCAL_NTYPE_STARTED,     CX_NTYPE_STARTED,   _T("Started") },
      { LOCAL_NTYPE_HALTED,      CX_NTYPE_HALTED,    _T("Halted") },
      { LOCAL_NTYPE_PAUSED,      CX_NTYPE_PAUSED,    _T("Paused") },
      { LOCAL_NTYPE_RESUMED,     CX_NTYPE_RESUMED,   _T("Resumed") },
      { LOCAL_NTYPE_COMPLETED,   CX_NTYPE_COMPLETED,_T("Completed" ) },
      { LOCAL_NTYPE_EVENT,       CX_NTYPE_EVENT,     _T("Event") },
      { LOCAL_NTYPE_DATA,        CX_NTYPE_DATA,      _T("Data" )},
      { LOCAL_NTYPE_LOGINFO,     CX_NTYPE_LOGINFO,   _T("Log Info") },
      { LOCAL_NTYPE_ALIVE,       CX_NTYPE_ALIVE,     _T("Alive") },
      { LOCAL_NTYPE_WINMSG,  CX_NTYPE_WINMSG,_T("Windows Message" )},
      { LOCAL_NTYPE_THREADMSG,CX_NTYPE_THREADMSG,_T("Thread Message") },
      { LOCAL_NTYPE_STARTING,    CX_NTYPE_STARTING,_T("Starting") },
```

```
        { LOCAL_NTYPE_HALTING,     CX_NTYPE_HALTING,_T("Halting") },
        { LOCAL_NTYPE_CMD,  CX_NTYPE_CMD,_T("Command") },
        { LOCAL_NTYPE_END_VALUE,0x10000000L,    _T( "") }
        };

        // Fill the structure
        for (int i = LOCAL_NTYPE_ALL; i <= LOCAL_NTYPE_END_VALUE; i++)
        {
              m_aNotificationTypes[i].m_id  = tmpNT [i].m_id;
              m_aNotificationTypes[i].m_val = tmpNT [i].m_val;
              m_aNotificationTypes[i].m_str = tmpNT [i].m_str;
        }
}
```

As you can see, this is a severely limited subset of notification types. Most programmers using the Notifications Package will need to support substantially more notification types.

CCxNotificationTypes::FindNotificationTypeVal

Prototype

```
int FindNotificationTypeVal(CString strNType);
```

Parameters

CString strNType	[in] Notification type string to search for. This string must be one of the values that was loaded into our structNotificationTypes array member variable.

Scope

Public

Definition

```
Dynamic Link Library          CxNotifySupport.DLL
Header Files                  #include <CxNotificationTypes.h>
```

Description

This method searches for the string passed in strNType. If it finds the string, the method returns the ID of the string. The ID is one of the enumerated notification type values defined above.

For example, if the string "Paused" is passed into this method, we return the CX_NTYPE_PAUSED enumerated value.

Return Values

int	Enumerated value found in the m_aNotificationTypes member variable, else CX_NTYPE_END_VALUE if the value is not found

See Also

FindNotificationTypeString

CCxNotificationTypes::FindNotificationTypeString

Prototype

```
LPCTSTR FindNotificationTypeString(int id);
```

Parameters

int id	[in] ID to search for. The ID number must be in our m_aNotificationTypes member variable.

Scope

```
Public
```

Definition

```
Dynamic Link Library        CxNotifySupport.DLL
Header Files                #include <CxNotificationTypes.h>
```

Description

This method searches for the notification id passed in. If it finds the id, the method returns the string representation of the id.

For example, if the id CX_NTYPE_PAUSED is passed in, then we return "Paused".

Return Values

LPCTSTR	Enumerated value found in the m_aNotificationTypes member variable, else "" if the value is not found

See Also

FindNotificationTypeVal

CCxNotificationWnd Class

The CCxNotificationWnd class is a helper class that creates a hidden window that can be used to receive notifications. This class inherits from MFC's CWnd class.

The CCxNotificationWnd class can be used with the CCxWinMsgSink object to receive notifications from the sink. A user of these two classes must do the following:

1. Create a new class that inherits from CCxNotificationWnd
2. Implement one or more of the CCxNotificationWnd virtual methods according to the individual needs of the user. There are two virtual methods that can be overridden - HandleNotification and HandleNotificationSubjectBroken.
3. Create a CCxWinMsgSink object and attach it to an ICxSubjectObserver object. See the sample source code below for instructions on doing this.
4. Call the CCxWinMsgSink::SetWindowHandle method and pass the window handle (HWND) that was created by instantiating the class that derives from the CCxNotificationWnd object in step 1.

The CCxNotificationWnd class is a helper class that "catches" windows message notifications and allows a derived class to implement their own handling of these notifications. The derived class implements the HandleNotification and/or HandleNotificationSubjectBroken in any way they require.

Method Name	Description
`CCxNotificationWnd`	Default constructor
`~CCxNotificationWnd`	Destructor
`DefWindowProc`	Intercepts the notifications Windows messages.
`HandleNotification`	Virtual method that receives and handles a notification from a Subject.
`HandleNotificationSubjectBroken`	Virtual method that receives and handles a subject broken notification from the Notifications Package management system.

Note that there are no additional member variables needed in this class. We rely on the CWnd::m_hWnd member variable to give us the HWND we need.

The CCxWinMsgSink object sends notifications to the derived CCxNotificationWnd class by using the WIN32 PostMessage API. Our CCxNotificationWnd::DefWindowProc has built in handlers for two windows messages. These messages are defined and exposed by the Notifications Package. Refer to the documentation on the CCxWinMsgSink object for details about these two messages. The messages are defined in CxNotifyServer.h:

1. WM_CXNOTIFICATION - sent with a CCxNotificationInfo object

2. WM_CXSUBJECTBROKEN - sent with the ID of the Subject that is being destroyed.

The sole purpose of our DefWindowProc method is to receive notifications messages and forward them on to either the HandleNotification or HandleNotificationSubjectBroken methods implemented by the derived class.

CCxNotificationWnd::CCxNotificationWnd

Prototype

```
CCxNotificationWnd::CCxNotificationWnd()
```

Parameters

None	

Scope

Public

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxNotificationWnd.h>
```

Description

This is the default constructor. No additional or special code is executed in the constructor. Since we inherit from MFC's CWnd class, we rely on the base class CWnd for all initialization.

Return Values

None	

See Also

MFC's CWnd class

CCxNotificationWnd::~CCxNotificationWnd

Prototype

```
CCxNotificationWnd::~ CCxNotificationWnd ()
```

Parameters

None	

Scope

Public

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxNotificationWnd.h>
```

Description

The destructor cleans up the object.

Return Values

None	

See Also

CCxNotificationWnd::DefWindowProc

Prototype

```
virtual LRESULT DefWindowProc(UINT message, WPARAM wParam, LPARAM
lParam);
```

Parameters

UINT message	[in]
WPARAM wParam	[in]
LPARAM	[in]

Scope

Protected

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxNotificationWnd.h>
```

Description

This method intercepts notifications messages and forwards them on to either the HandleNotification method or the HandleNotificationSubjectBroken method.

This method is called whenever a windows message is posted to our HWND.

The default CCxNotificationWnd::DefWindowProc implementation looks as follows:

```
LRESULT CCxNotificationWnd::DefWindowProc(UINT message,
                                    WPARAM wParam,
                                    LPARAM lParam)
{
      switch (message)
      {
            case WM_CXNOTIFICATION:
                  {
                        CCxNotificationInfo *pN;
                        pN = (CCxNotificationInfo *)wParam;
```

```
                        // HandleNotification() is the virtual
                        // function that a derived class needs
                        // to implement.
                        HandleNotification(pN);

                  }
                  break;

            case WM_CXSUBJECTBROKEN:
                  {
                        long lSubjectID = (long)wParam;

                        // HandleNotificationSubjectBroken() is
                        // the virtual function that a derived
                        // class needs to override if they want
                        // to do something special with the
                        // subject broken information.
                  HandleNotificationSubjectBroken(lSubjectID);
                  }
                  break;

            default:
                  return CWnd::DefWindowProc(message,
                                          wParam, lParam);

      }

      return 0;
}
```

Note that the WM_CXNOTIFICATION and WM_CXSUBJECTBROKEN windows messages are defined in the Notifications Package CxNotifyServer.h header file.

Return Values

LRESULT	If we catch and handle the message, we return 0, otherwise we execute: `return CWnd::DefWindowProc(message, wParam, lParam);`

See Also

CWnd::DefWindowProc documentation in the SDK or MSDN

CCxNotificationWnd::HandleNotification

Prototype

```
virtual void HandleNotification(
                  CCxNotificationInfo *pNotification);
```

Parameters

CCxNotificationInfo *pNotification	[in] pointer to the notification object.

Scope

Protected

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxNotificationWnd.h>
```

Description

This method is virtual and should be overridden by a derived class. The derived class' implementation should be coded according to the needs of the user.

The default implementation of this method simply deletes the notification object. The derived implementation must also delete the notification object once it is finished with it. For details on deleting a notification, refer to the CCxNotificationInfo class documentation.

The default implementations of HandleNotification is as follows:

```
void CCxNotificationWnd::HandleNotification(
                        CCxNotificationInfo *pNotification)
{
      if (pNotification)
      {
            delete pNotification;
            pNotification = NULL;
      }
}
```

Note that there very minimal implementation of this method. It is completely up to the derived class to determine their particular needs and their particular implementation of this method. The HandleNotification method must always delete the notification object that it receives otherwise memory leaks will occur. The derived method should also delete the notification object.

Return Values

Void	None

See Also

ThreadProc
CCxNotificationInfo class

CCxNotificationWnd::HandleNotificationSubjectBroken

Prototype

```
virtual void HandleNotificationSubjectBroken(long lSubjectID);
```

Parameters

long lSubjectID	[in] ID of the Subject who we are no longer subscribed to.

Scope

```
Protected
```

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxNotificationWnd.h>
```

Description

This method is virtual and should be overridden by a derived class. The derived class' implementation should be coded according to the needs of the user.

The default implementation of this method does absolutely nothing. For implementations that need to handle being "disconnected" from a Subject - otherwise known as destroying our subscription to the Subject, this method should be overridden and implemented.

The default implementations of HandleNotificationSubjectBroken is as follows:

```
void CCxNotificationWnd::HandleNotificationSubjectBroken(
                                                    long lSubjectID)
{
      // Simply return and do nothing
}
```

Return Values

void	None

See Also

CCxObserver Class

The CCxObserver class is an abstract base class for all of our Observer helper classes. CCxObserver inherits from CCxSubjectOnly. To view the UML class hierarchy diagram, refer to the CCxSubjectOnly section in this document. The diagram shows Observers using the COM interface and using the helper "Observer" classes.

This class can not be instantiated because of a pure virtual method definition. The pure virtual method is CreateObserver. The CreateObserver method MUST be implemented by each derived class. Each class that inherits from CCxObserver will have its own unique implementation of CreateObserver because the type of Observer is defined by the kind of Sink object that must be instantiated.

One thing that the class hierarchy diagram does NOT show (because of space limitiation) is the relationship between the Observer classes and their associated sink objects. Note that the CCxObserver class has a method called GetSinkObject. Each class that derives from CCxObserver contains an associated sink object. The GetSinkObject method returns a pointer to an internal member variable. This internal member variable is the sink object for a particular type of sink. For example, the CCxRoutedSubjectObserver contains a CCxRoutedSink, the CCxWinMsgSubjectObserver contains a CCxWinMsgSink, and the CCxThreadMsgSubjectObserver contains a CCxThreadMsgSink. Creation and destruction of the sink objects is completely handled by the classes that derive from CCxObserver. A client who creates one of the objects that derives from CCxObserver requires no knowledge of COM or sinks.

CCxObserver is the base class for a number of "helper" classes in the Notifications Package. These helper classes provide a pure C++ implementation of an Observer. Absolutely no COM knowledge is needed in order to create and use these derived Observer classes.

The CCxObserver class defines the following methods:

Method Name	Description
`~CCxObserver`	Cleans up the Observer object
`CCxObserver`	Handles all initialization and creation for this object.
`CreateObserver`	Pure virtual method that MUST be implemented by a derived class.
`EmptyQueue`	Clears all notifications from a queue
`GetSinkObject`	Returns a pointer to the internal sink object
`GetSinkPriorityClass`	Returns the priority class of the sink thread
`GetSinkThreadPriority`	Returns the thread priority of the sink thread
`GetSinkType`	Returns an enumerated value that says which type of sink this object has
`PeekNotification`	Returns the number of notifications currently available in the sink object's

	queue, else it returns -1
`SetSinkPriorityClass`	Allows the client to set the priority class of the sink's thread
`SetSinkThreadPriority`	Allows the client to set the thread priority of the sink's thread.
`SubscribeByID`	Implements the ICxSubjectObserver::SubscribeByID method. Observers can use this method to subscribe to a Subject via its ID.
`SubscribeByName`	Implements the ICxSubjectObserver::SubscribeByName method. Observers can use this method to subscribe to a Subject via its string name
`UnsubscribeByID`	Implements the ICxSubjectObserver::UnsubscribeByID method. Observers can use this method to unsubscribe from a Subject via its ID
`UnsubscribeByName`	Implements the ICxSubjectObserver::UnsubscribeByName method. Observers can use this method to unsubscribe from a Subject via its string name
`WaitForNotify`	Allows a caller to block their thread of execution until a notification arrives. Once the notification arrives, the notification will be returned to the caller.

The CCxObserver class exposes the following member variables:

Name	Data Type	Scope	Description
m_nSinkType	int	Protected	
m_pSink	CCxQueuedSink *	Protected	

CCxObserver::CCxObserver

Prototype

```
CCxObserver(LPCTSTR pszName = NULL,
            long nPackageID = CX_PKGID_USER,
            LPCTSTR pszServer = NULL,
            long lFlags = CX_NOTIFICATION_INPROC);
```

Parameters

`LPCTSTR pszName`	[in] Name of the Observer. Default value is NULL if the default constructor is used. If NULL is passed, then the SetMyName method MUST be used to set the name of this Observer.
`long nPackageID`	[in] Package ID of this Observer. Default value is CX_PKGID_USER if the default constructor is used. If no value is passed, the SetMyName method must be used to set the Package ID.

LPCTSTR pszServer	[in] Used for DCOM creation of the ICxSubjectObserver COM object. If non-NULL, the name of the server computer where the CxNotify COM server DLL is registered and available for use.
long lFlags	[in] Creation flags. Currently this parameter is used only to determine whether to instantiate the ICxSubjectObserver object in process or out of process. The in process flag is CX_NOTIFICATION_INPROC, the out of process flag is CX_NOTIFICATION_OUTOFPROC.

Scope

Public

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxObserver.h>
```

Description

The constructor creates a Notification Package Observer and automatically instantiates the ICxSubjectObserver COM object by calling the base class CCxSubjectOnly constructor. Constructor parameters can be used to efficiently initialize the object. The name and package ID parameters replace calling the ICxSubjectObserver::SetMyName method. The lFlags parameter handles creation of the object in process or out of process as opposed to using the CoCreateInstanceEx WIN32 API. Note that by default, the COM object is created in process.

Note that this class is an abstract base class because the CreateObserver method is defined as pure virtual. A derived class MUST implement the CreateObserver method and then call that method from within their constructor.

Return Values

None	

See Also

ICxSubjectObserver COM interface
ICxSubjectObserver::SetMyName

CCxObserver::~CCxObserver

Prototype

```
virtual ~CCxObserver();
```

Parameters

None	

Scope
Public

Definition
Dynamic Link Library CxNotifyClient.DLL
Header Files #include <CxObserver.h>

Description
This method handles all cleanup for this object.

Return Values

None	

See Also
ICxSubjectObserver COM interface

CCxObserver::CreateObserver

Prototype
```
virtual HRESULT CreateObserver() = 0;
```

Parameters

None	

Scope
Protected

Definition
Dynamic Link Library CxNotifyClient.DLL
Header Files #include <CxObserver.h>

Description
This method is pure virtual and contains no implementation. Derived classes MUST implement this function. A derived class' implementation will instantiate a sink object and store a pointer to the object in our m_pSink member variable.

Return Values

HRESULT	Refer to the derived class documentation for this method

See Also
ICxSubjectObserver COM interface

CCxObserver::EmptyQueue

Prototype

```
virtual void EmptyQueue();
```

Parameters

None	

Scope

Public

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxObserver.h>
```

Description

This method delegates to the our sink object's EmptyQueue method.

Return Values

None	

See Also

ICxSubjectObserver COM interface
CCxQueuedSink::EmptyQueue()

CCxObserver::GetSinkObject

Prototype

```
virtual CCxQueuedSink *GetSinkObject()
```

Parameters

None	

Scope

Public

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxObserver.h>
```

Description

This method returns a pointer to our sink object. Note that the return value is of type CCxQueuedSink because all other sink types inherit from CCxQueuedSink.

Return Values

CCxQueuedSink *	Pointer to our internal sink object

See Also

ICxSubjectObserver COM interface

CCxQueuedSink class

CCxObserver::GetSinkPriorityClass

Prototype

```
DWORD GetSinkPriorityClass();
```

Parameters

None	

Scope

Public

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxObserver.h>
```

Description

This method returns the sink thread's priority class. Refer to the WIN32 API GetPriorityClass or to the CCxSink::SetSinkPriorityClass documentation for valid values that can be returned.

Return Values

DWORD	See the WIN32 API GetPriorityClass or to the CCxSink::SetSinkPriorityClass documentation for valid values

See Also

ICxSubjectObserver COM interface

CCxObserver::GetSinkThreadPriority

Prototype

```
int GetSinkThreadPriority();
```

Parameters

None	

Scope

Public

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxObserver.h>
```

Description

This method returns the sink thread's priority. Refer to the WIN32 API GetThreadPriority or to the CCxSink::SetSinkThreadPriority documentation for valid values that can be returned.

Return Values

int	See the WIN32 API GetThreadPriority or to the CCxSink::SetSinkThreadPriority documentation for valid values

See Also

ICxSubjectObserver COM interface
CCxSink::SetSinkThreadPriority

CCxObserver::GetSinkType

Prototype

```
virtual int GetSinkType()
```

Parameters

None	

Scope

Public

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxObserver.h>
```

Description

This method returns the enumerated value that is a sink type. Refer to the CCxSinkTypes class for the enumerated values that can be returned.

Return Values

int	Returns the enumerated value that is a sink type

See Also

ICxSubjectObserver COM interface
CCxSinkTypes class

CCxObserver::PeekNotification

Prototype

```
virtual int  PeekNotification();
```

Parameters

None	

Scope

Public

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxObserver.h>
```

Description

This method returns the number of notifications currently in the sink object's queue.

Return Values

`int`	Number of notifications currently in the sink's queue. -1 is returned if there are no notifications currently in the queue.

See Also

ICxSubjectObserver COM interface

CCxObserver::SetSinkPriorityClass

Prototype

```
BOOL SetSinkPriorityClass(DWORD dwPriorityClass);
```

Parameters

`DWORD`	See the WIN32 API SetPriorityClass or to the CCxSink::SetSinkPriorityClass documentation for valid values

Scope

```
Public
```

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxObserver.h>
```

Description

This method allows the caller to set the priority class of the sink object's thread. Note that the sink object's thread makes the actual call to the WIN32 API SetPriorityClass. This method simply assigns the dwPriorityClass value into a member variable that is used by the sink thread in its call to SetPriorityClass.

Return Values

`BOOL`	TRUE if successful, else FALSE

See Also

ICxSubjectObserver COM interface
CCxSink::SetSinkPriorityClass

CCxObserver::SetSinkThreadPriority

Prototype

```
BOOL SetSinkThreadPriority(int nPriority);
```

Parameters

int nPriority	See the WIN32 API SetThreadPriority or to the CCxSink::SetSinkThreadPriority documentation for valid values

Scope

Public

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxObserver.h>
```

Description

This method allows the caller to set the thread priority of the sink object's thread. Note that the sink object's thread makes the actual call to the WIN32 API SetThreadPriority. This method simply assigns the nPriority value into a member variable that is used by the sink thread in its call to SetThreadPriority.

Return Values

BOOL	TRUE if successful, else FALSE

See Also

ICxSubjectObserver COM interface
CCxSink::SetSinkThreadPriority

CCxObserver::SubscribeByID

Prototype

```
HRESULT SubscribeByID(long lSubjectID,
                      long lFlags,
                      long lNotificationFilter);
```

Parameters

long lSubjectID	[in] Subscribe to a specific subject ID
long lFlags	[in] Bit-mask field that can be used to set special handling and Orphan creation flags. The available flags are: • **CX_STYPE_ORPHANONFAIL** - Create an orphan only if the subscription request fails • **CX_STYPE_ORPHANONSUCCESS** - Create an orphan only if the subscription request succeeds • **CX_STYPE_ORPHANALWAYS** - Create an orphan if the subscription succeeds or fails • **CX_STYPE_ORPHAN4EVER** - Leave as an orphan even

	after the subject comes alive • **CX_STYPE_REPLACE** - Replace any existing subscriptions between the requesting Observer and the Subject specified by lSubjectID These flags are defined in CxNotifyServer.h. Any combination of these flags can be bit-wise OR'd '|' together. Pass a 0 if no special orphan handling is required.
`long lNotificationFilter`	[in] Bit-mask field that is used to filter notifications. Allowable values are anything that can be bitwise AND'd with the filter value passed during a notification call (see PostNotify or SendNotify). Pass -1 (0xffffffff) to subscribe to all notification types.

Scope

`Public`

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxObserver.h>
```

Description

Notification Package Observers who wish to use a Subject's ID to request a subscription should use this method. If an Observer knows the name of a Subject, the Observer can use the ICxSubjectObserver::GetIDFromName method to retrieve a Subject's ID.

To remove a subscription, use ICxSubjectObserver::UnsubscribeByID or ICxSubjectObserver::UnsubscribeByName.

Return Values

`HRESULT`	S_OK on creation of either a successful subscription or on creation of an orphan subscription
	E_FAIL on error. For information about the error, call ICxSubjectObserver::GetError

See Also

ICxSubjectObserver COM interface

CCxObserver::SubscribeByName

Prototype

```
HRESULT SubscribeByName(BSTR bstrName,
                        long lFlags,
                        long lNotificationFilter,
                        long *plSubscriptions = NULL);
```

Parameters

`BSTR bstrName`	[in] Name of the subject you wish to subscribe to. This may also be a wildcard string. For example, "abc*" will subscribe to all subjects hose names begin with "abc". For a complete explanation of the wildcard capabilities of the Notifications Package, see the "Description" section below.
`long lFlags`	[in] Bit-mask field that can be used to set special handling and Orphan creation flags. The available flags are: • **CX_STYPE_ORPHANONFAIL** - Create an orphan only if the subscription request fails • **CX_STYPE_ORPHANONSUCCESS** - Create an orphan only if the subscription request succeeds • **CX_STYPE_ORPHANALWAYS** - Create an orphan if the subscription succeeds or fails • **CX_STYPE_ORPHAN4EVER** - Leave as an orphan even after the subject comes alive • **CX_STYPE_REPLACE** - Replace any existing subscriptions between the requesting Observer and the Subject named by bstrName These flags are defined in CxNotifyServer.h. Any combination of these flags can be bit-wise OR'd '|' together. Pass a 0 if no special orphan handling is required.
`long lNotificationFilter`	[in] Bit-mask field that is used to filter notifications. Allowable values are anything that can be bitwise AND'd with the filter value passed during a notification call (see PostNotify or SendNotify). Pass -1 (0xffffffff) to subscribe to all notification types.
`long *plSubscriptions`	[out] Pointer to a long that receives the number of successful subscriptions

Scope

`Public`

Definition

`Dynamic Link Library  CxNotifyClient.DLL`

```
Header Files                    #include <CxObserver.h>
```

Description

Notification clients who know the name of a subject or wish to use wildcard subscriptions can call this method to subscribe to one or more subjects.

The following examples illustrate the use of wildcard subscriptions:

- "*" - Subscribe to any Subject
- "Security*" - 'Securityabc', 'SecurityDEF', etc.
- "Security[1-5]" - Security1, Security2, … , Security5
- "Security?" - 'Security' followed any character
- "Security$" - 'MySecurity', 'YourSecurity', etc.
- "Security[a | f]" - 'Securitya' or 'Securityf'

Nested expressions can also be used. For example:

- "Security(([1-3])[a | b])" - Security1a, Security1b, … , Security3a, Security 3b would be matching names.

To remove a subscription, use ICxSubjectObserver::UnsubscribeByID or ICxSubjectObserver::UnsubscribeByName.

Return Values

HRESULT	S_OK on success
	E_FAIL on error. For information about the error, call ICxSubjectObserver::GetError

See Also

ICxSubjectObserver COM interface

CCxObserver::UnsubscribeByID

Prototype

```
HRESULT UnsubscribeByID(long lSubjectID = -1);
```

Parameters

long lSubjectID	[in] Unsubscribe from this subject only

Scope

```
Public
```

Definition

```
Dynamic Link Library            CxNotifyClient.DLL
Header Files                    #include <CxObserver.h>
```

Description

Notification clients who have the ID of a subject can call this method to unsubscribe to the subject.

Note that if multiple subscriptions have been created between the calling Observer and the Subject specified by lID, ALL subscriptions are removed.

Return Values

HRESULT	S_OK on success
	E_FAIL on error. For information about the error, call ICxSubjectObserver::GetError

See Also

ICxSubjectObserver COM interface

CCxObserver::UnsubscribeByName

Prototype

```
HRESULT UnsubscribeByName(BSTR bstrName);
```

Parameters

BSTR bstrName	[in] Subject name or wildcard.

Scope

```
Public
```

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxObserver.h>
```

Description

Notification clients who have the name of a subject or who wish to use wildcard names can call this method to unsubscribe from one or more subjects.

Note that if multiple subscriptions have been created between the calling Observer and the Subject specified by bstrName, ALL subscriptions are removed. When wildcards are used in bstrName, ALL subscriptions are removed for each matching name.

For a detailed description of wildcards, refer to ICxSubjectObserver::SubscribeByName.

Return Values

HRESULT	S_OK on success
	E_FAIL on error. For information about the error, call ICxSubjectObserver::GetError. Note that if there are no Subjects matching the passed in name, E_FAIL will be returned.

See Also

ICxSubjectObserver COM interface

CCxObserver::WaitForNotify

Prototype

```
virtual long WaitForNotify(CCxNotificationInfo *pData,
                           DWORD dwMilliseconds = INFINITE);
```

Parameters

CCxNotificationInfo *&pData	[out] pointer to a CCxNotificationInfo object that will receive the notification object.
DWORD dwMilliseconds	[in] Number of milliseconds to wait for a notification. If this amount of time expires, the method

Scope

`Public`

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxObserver.h>
```

Description

This method can be used by Observers who wish to block their thread of execution until a notification is placed in the queue. When a notification is placed in the queue, the `WaitForNotify` method will return with the notification data in the `pData` parameter.

There are three scenarios involving the kind of notification and the return values this method sends back. The scenarios and return values are as follows:

1. The m_qNotifications member variable holds a CCxNotificationInfo notification: In this scenario, a pointer to a CCxNotificationInfo object is returned in the pData parameter. If a pointer is returned in pData, then the return value will be set to the ID of the Subject who sent the Notification.
2. The m_qNotifications member variable holds a "subject broken" notification. If a Subject has been removed from the notifications package, a "subject broken" notification is automatically sent to all subscribed Observers. If the notification queue holds one of these notifications, pData will be set to NULL, and the return value will be set to the ID of the Subject who was destroyed.
3. The timeout value expires before receiving a notification: In this scenario, pData is set to NULL and the return value is WAIT_TIMEOUT.

Return Values

`long`	If pData is Non-NULL, then the return value is the Subject ID who sent the notification. If dwMilliseconds expires, then we return WAIT_TIMEOUT from the WaitForMultipleObjects call.

See Also

ICxSubjectObserver COM interface

CCxQueuedSink Class

The CCxQueuedSink class inherits from the CCxSink class. Refer to the CCxSink class for a class hierarchy diagram of all the sink classes.

The CCxQueuedSink class implements the queuing capabilities for all of its derived sink objects. It creates a queue that is used to hold notifications until the Observer is ready to retrieve the notification.

Note that an Observer **MUST** explicitly pull a notification from the queue if the Observer directly instantiated a CCxQueuedSink object. If the Observer instantiates one of the derived sink objects, then the Observer is NOT required to explicitly pull a notification from the queue. The ability to pull a notification from the queue is optional for all classes that derive from CCxQueuedSink.

The following methods are defined by the CCxQueuedSink class:

Method Name	Description
`EmptyQueue`	Empties the queue and deletes all objects currently in the queue
`NotificationMessage`	Can be used by a derived class' ThreadProc method to dispatch notifications from the internal queue member variable to the final destination in an Observer.
`OnNotify`	Provides the one and only implementation of the ICxObserverNotification::OnNotify method. This method is called by the Management System for delivery of a CCxNotificationInfo object from the Management System to a sink's internal queue.
`OnNotifyNOTIFY`	Virtual function implemented by derived classes that is used by the sink object thread to deliver the CCxNotificationInfo object to the Observer.
`OnNotifySubjectBroken`	Provides the one and only implementation of the ICxObserverNotification::OnNotify method. This method is called by the Management System for delivery of a subject broken notification from the Management System to a sink's internal queue.
`OnNotifyNOTIFY_SUBJECTBROKEN`	Virtual function implemented by derived classes that is used by the sink object thread to deliver the subject broken notification object to the Observer.
`PeekNotification`	Provided to allow the user to "peek"

	at the queue. The method returns the number of notification objects currently in the queue.
`ThreadProc`	Virtual function that implements any special threading needs for this sink object. Derived classes implement their own version of ThreadProc according to the special needs of the sink object.
`WaitForNotify`	Provided to allow the caller to block their thread of execution until a notification is placed into their queue.

The following member variables are exposed by the CCxQueuedSink class:

Name	Data Type	Scope	Description
m_evtBreakWait	CEvent	Protected	Signaled to tell the WaitForNotify() method to break out of the wait
m_qNotifications	CCxCriticalQueue <CCxNotificationFromServer * >	Protected	Queue that holds ALL incoming notification objects.

CCxQueuedSink::EmptyQueue

Prototype

```
void EmptyQueue()
```

Parameters

None	

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxSink.h>
```

Description

This method can be used to delete all notifications that are currently in the queue.

Return Values

None	

See Also

CCxQueuedSink::NotificationMessage

Prototype

```
virtual void NotificationMessage(
                  CCxNotificationFromServer * pNotification);
```

Parameters

CCxNotificationFromServer * pNotification	[in] Pointer to an object that comes directly from the Management System. The CCxNotificationFromServer object can wrap either a CCxNotificationInfo or simply a "Subject Broken" notification consisting of the Subject's ID.

Scope

Protected

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxSink.h>
```

Description

This method determines the type of notification object that is being received by the Management System. Derived classes can use this method after they take a notification object out of their queue. This method will determine the type of notification (CCxNotificationInfo or "Subject broken") and call the derived class' implementation of either OnNotifyNOTIFY or OnNotifyNOTIFY_SUBJECTBROKEN. These two derived methods implement the final delivery of the notification object to the Observer.

Return Values

None	

See Also

CCxQueuedSink::OnNotify

Prototype

```
HRESULT OnNotify(long lSize,
                  byte* pObject,
                  long lBytes,
                  byte *pBytes,
                  VARIANT *pReturn);
```

Parameters

long lSize	[in] size of the entire original CCxNotificationInfo object.
byte* pObject	[in] a pointer to the original CCxNotificationInfo

	object.
long lBytes	[in] size of the byte buffer in the original CCxNotificationInfo object.
byte *pBytes,	[in] a pointer to the original CCxNotificationInfo notification data buffer.
VARIANT *pReturn	[out] - used to return the processed HANDLE back to the caller. This HANDLE is used solely for synchronous notifications.

Scope

Public

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxSink.h>
```

Description

This method provides the one and only implementation of the ICxObserverNotification::OnNotify method.

The implementation of this method actually "glues" a CCxNotificationInfo object back together. Because the Notifications Package support both in-process and out-of-process notifications, it was necessary to send the entire CCxNotificationInfo object separately from it's embedded byte buffer containing the notifications data. This is necessary because COM will not automatically pass an object across an application boundary if the object contains an embedded pointer to some other data.

In short, this method takes a CCxNotificationInfo object pointer and embeds a marshaled (by COM) pointer back into the CCxNotificationInfo object.

Finally, this method puts the notification object into the m_qNotifications member variable.

This method is called by the Notifications Package Management System to deliver a notification to an Observers queue. This method is used solely to put a notification object ***INTO*** the Observer's sink object queue. The delivery of this notification out of this queue and to an Observer occurs on the sink object's thread. Refer to the sink object ThreadProc methods for details on how each type of sink uses a thread to pull the notification out of the queue and pass the notification to its final destination within an Observer.

Return Values

HRESULT	S_OK on success
	E_FAIL on error.

See Also

CCxSink class, CCxQueuedSink class, CCxRoutedSink class, CCxWinMsgSink class, CCxThreadMsgSink class

CCxQueuedSink::OnNotifyNOTIFY

Prototype

```
virtual void OnNotifyNOTIFY(CCxNotificationInfo* pNotification);
```

Parameters

CCxNotificationInfo* pNotification	[in] pointer to an actual notification object.

Scope

Protected

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxSink.h>
```

Description

This virtual method is the method that gets called for final delivery of the CCxNotificationInfo object. Each derived class will have it's own implementation of this method. For example, a Windows message sink will implement this method with a PostMessage call. The Thread message sink will implement it using the PostThreadMessage call.

The default implementation in this class essentially does nothing because an Observer is responsible for explicitly pulling a notification out of the queue by calling WaitForNotify.

Return Values

None	

See Also

CCxSink class, CCxQueuedSink class, CCxRoutedSink class, CCxWinMsgSink class, CCxThreadMsgSink class

CCxQueuedSink::OnNotifySubjectBroken

Prototype

```
HRESULT OnNotifySubjectBroken(long lSubjectID);
```

Parameters

long lSubjectID	[in] ID of the Subject who this Observer is no longer subscribed to.

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxSink.h>
```

Description

This method provides the one and only implementation of the ICxObserverNotification::OnNotifySubjectBroken method. ALL derived sink classes use this implementation.

This method is called by the Notifications Package Management System to notify an Observer that it is no longer subscribed to the Subject whose ID is passed as lSubjectID.

This method puts the "subject broken" notification object into the m_qNotifications member variable.

Note that this is only the input side of the subject broken notification. The Management System uses this method to put the notification object into our queue member variable. In order to deliver the notification to the Observer, a derived class' ThreadProc method will pull the data from the queue and forward it on to the Observer. The details of how the notification is forwarded on to the Observer is different for each derived sink type. For these details refer to the individual sink object's ThreadProc documentation.

Return Values

HRESULT	S_OK on success
	E_FAIL on error.

See Also

CCxSink class, CCxQueuedSink class, CCxRoutedSink class, CCxWinMsgSink class, CCxThreadMsgSink class

CCxQueuedSink::OnNotifyNOTIFY_SUBJECTBROKEN

Prototype

```
virtual void OnNotifyNOTIFY_SUBJECTBROKEN(long lSubjectID);
```

Parameters

long lSubjectID	[in] ID of the Subject whose subscription is being destroyed in the Management System.

Scope

```
Protected
```

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxSink.h>
```

Description

This virtual method is the method that gets called for final delivery of a subject broken notification object. Each derived class will have it's own implementation of this method. For example, a Windows message sink will implement this method with a PostMessage call. The Thread message sink will implement it using the PostThreadMessage call.

The default implementation in this class essentially does nothing because an Observer is responsible for explicitly pulling a notification out of the queue by calling WaitForNotify.

Return Values

None	

See Also

CCxSink class, CCxQueuedSink class, CCxRoutedSink class, CCxWinMsgSink class, CCxThreadMsgSink class

CCxQueuedSink::PeekNotification

Prototype

```
int PeekNotification()
```

Parameters

None	

Scope

Public

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxSink.h>
```

Description

This method can be used to determine if there are any notifications waiting in the queue. If there are notifications, this method returns a non-zero value, otherwise, it returns 0.

Return Values

int	Number of notifications currently in the queue.

See Also

WaitForNotify

CCxQueuedSink::ThreadProc

Prototype

```
virtual DWORD ThreadProc();
```

Parameters

None	

Scope

Protected

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxSink.h>
```

Description

This virtual method is implemented because our base class method is defined as pure virtual. This method essentially does nothing except wait for an event to be told to shutdown the thread. There is no need to loop and watch for notifications be delivered into our queue because the Observer is required to manually pull notifications from the queue using the WaitForNotify method.

Return Values

DWORD	Always returns 0.

See Also

CCxSink::ThreadProc, CCxQueuedSink::ThreadProc, CCxWinMsgSink::ThreadProc, CCxThreadMsgSink::ThreadProc

CCxQueuedSink::WaitForNotify

Prototype

```
DWORD WaitForNotify(CCxNotificationInfo *&pData,
                        DWORD dwMilliseconds);
```

Parameters

CCxNotificationInfo *&pData	[out] pointer to a CCxNotificationInfo object that will receive the notification object.
DWORD dwMilliseconds	[in] Number of milliseconds to wait for a notification. If this amount of time expires, the method

Scope

Public

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxSink.h>
```

Description

This method can be used by Observers who wish to block their thread of execution until a notification is placed in the queue. When a notification is placed in the queue, the `WaitForNotify` method will return with the notification data in the `pData` parameter.

There are three scenarios involving the kind of notification and the return values this method sends back. The scenarios and return values are as follows:

4. The m_qNotifications member variable holds a CCxNotificationInfo notification: In this scenario, a pointer to a CCxNotificationInfo object is returned in the pData parameter. If a pointer is returned in pData, then the return value will be set to the ID of the Subject who sent the Notification.
5. The m_qNotifications member variable holds a "subject broken" notification. If a Subject has been removed from the notifications package, a "subject broken" notification is automatically sent to all subscribed Observers. If the notification queue holds one of these notifications, pData will be set to NULL, and the return value will be set to the ID of the Subject who was destroyed.
6. The timeout value expires before receiving a notification: In this scenario, pData is set to NULL and the return value is WAIT_TIMEOUT.

Return Values

`long`	If pData is Non-NULL, then the return value is the Subject ID who sent the notification. If dwMilliseconds expires, then we return WAIT_TIMEOUT from the WaitForMultipleObjects call.

See Also

PeekNotification

CCxQueuedSubjectObserver Class

The CCxQueuedSubjectObserver class implements an Observer with a CCxQueuedSink. CCxQueuedSubjectObserver inherits from CCxObserver. To view the UML class hierarchy diagram, refer to the CCxSubjectOnly section in this document.

The CreateObserver method in this class instantiates a CCxQueuedSink object.

CCxQueuedSubjectObserver is the base class for a number of other "helper" Observer classes in the Notifications Package. These helper classes provide a pure C++ implementation of an Observer. Absolutely no COM knowledge is needed in order to create and use these derived Observer classes.

The CCxQueuedSubjectObserver class defines the following methods:

Method Name	Description
`~CCxQueuedSubjectObserver`	Cleans up the Observer object
`CCxQueuedSubjectObserver`	Handles all initialization and creation for this object.
`CreateObserver`	Instantiates a CCxQueuedSink object and then connects the object to our ICxSubjectObserver COM object.

CCxQueuedSubjectObserver::CCxQueuedSubjectObserver

Prototype

```
CCxQueuedSubjectObserver(LPCTSTR pszName = NULL,
            long nPackageID = CX_PKGID_USER,
            LPCTSTR pszServer = NULL,
            long lFlags = CX_NOTIFICATION_INPROC);
```

Parameters

`LPCTSTR pszName`	[in] Name of the Observer. Default value is NULL if the default constructor is used. If NULL is passed, then the SetMyName method MUST be used to set the name of this Observer.
`long nPackageID`	[in] Package ID of this Observer. Default value is CX_PKGID_USER if the default constructor is used. If no value is passed, the SetMyName method must be used to set the Package ID.
`LPCTSTR pszServer`	[in] Used for DCOM creation of the ICxSubjectObserver COM object. If non-NULL, the name of the server computer where the CxNotify COM server DLL is registered and available for use.
`long lFlags`	[in] Creation flags. Currently this parameter is used only to determine whether to instantiate the ICxSubjectObserver object in process or out of process. The in process flag is CX_NOTIFICATION_INPROC, the out of process

	flag is CX_NOTIFICATION_OUTOFPROC.

Scope

Public

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxQueuedSO.h>
```

Description

The constructor creates a Notification Package Observer and automatically instantiates the ICxSubjectObserver COM object by calling the base class CCxSubjectOnly constructor. Constructor parameters can be used to efficiently initialize the object. The name and package ID parameters replace calling the ICxSubjectObserver::SetMyName method. The lFlags parameter handles creation of the object in process or out of process as opposed to using the CoCreateInstanceEx WIN32 API. Note that by default, the COM object is created in process.

After creating the ICxSubjectObserver object in our CCxSubjectOnly base class, this method calls the CreateObserver method. CreateObserver instantiates a CCxQueuedSink object.

Return Values

None	

See Also

ICxSubjectObserver COM interface
ICxSubjectObserver::SetMyName

CCxQueuedSubjectObserver::~CCxQueuedSubjectObserver

Prototype

```
virtual ~CCxQueuedSubjectObserver ();
```

Parameters

None	

Scope

Public

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxQueuedSO.h>
```

Description

This method handles all cleanup for this queued Subject-Observer object. The destructor will destroy the CCxQueuedSink object and then relies on the base class destructor(s) for their cleanup.

Return Values

None	

See Also

ICxSubjectObserver COM interface
CCxQueuedSink class

CCxQueuedSubjectObserver::CreateObserver

Prototype

```
virtual HRESULT CreateObserver();
```

Parameters

None	

Scope

Protected

Definition

Dynamic Link Library	CxNotifyClient.DLL
Header Files	#include <CxQueuedSO.h>

Description

This method instantiates a CCxQueuedSink object and sets the m_pSink member variable to the address of the sink.

The method also "connects" the sink object to the ICxSubjectObserver COM object created by our CCxSubjectOnly base class.

Return Values

HRESULT	S_OK if successful, else E_FAIL

See Also

ICxSubjectObserver COM interface
CCxQueuedSink class

CCxRoutedSink Class

The CCxRoutedSink class inherits from the CCxQueuedSink class. The CCxQueuedSink class inherits from the CCxSink class. Refer to the CCxSink class for a class hierarchy diagram of all the sink classes.

The CCxRoutedSink class relies on a "callback" mechanism in order to forward notifications back to an Observer. Since this class inherits from the CCxQueuedSink class, we get all of the built-in queuing capabilities provided by that class.

In order to use the CCxRoutedSink, an Observer is responsible for providing a notification route callback to this sink. To do this, the Observer must create an object that inherits from the CCxNotificationRoute class. The CCxNotificationRoute class provides a number of virtual method that can be implemented by the derived class. The CCxRoutedSink class knows exactly how to interact with another class that inherits from the CCxNotificationRoute.

The following methods are defined by the CCxRoutedSink class:

Method Name	Description
`CreateThreadCleanup`	Helper function that creates a manual event that is used for thread synchronization between the client's thread and the sink thread.
`CreateThreadCleanupDone`	Helper function that creates a manual event that is used for thread synchronization between the client's thread and the sink thread.
`CreateThreadInit`	Helper function that creates a manual event that is used for thread synchronization between the client's thread and the sink thread.
`CreateThreadInitDone`	Helper function that creates a manual event that is used for thread synchronization between the client's thread and the sink thread.
`NotificationMessage`	Can be used by a derived class' ThreadProc method to dispatch notifications from the internal queue member variable to the final destination in an Observer.
`OnNotifyNOTIFY`	Member function that is called by our NotificationMessage method. The implementation of OnNotifyNOTIFY delivers the final CCxNotificationInfo object to the Observer.
`OnNotifyNOTIFY_SUBJECTBROKEN`	Member function that is called by our NotificationMessage method. The implementation of OnNotifyNOTIFY_SUBJECTBROKEN delivers the final subject broken

	notification object to the Observer.
SetNotificationRoute	Called by an Observer to provide the sink with our callback pointer. The pointer passed to this method MUST inherit from the CCxNotificationRoute class.
ThreadCleanup	Called by the client to force the sink object's thread to callback to the CCxNotificationRoute::OnThreadCleanup method.
ThreadInit	Called by the client to force the sink object's thread to callback to the CCxNotificationRoute::OnThreadInit method.
ThreadProc	Virtual function that implements any special threading needs for this sink object. Derived classes implement their own version of ThreadProc according to the special needs of the sink object.

The following member variables are exposed by the CCxQueuedSink class:

Name	Data Type	Scope	Description
m_pRouter	CCxNotificationRoute *	Protected	Holds a pointer to our "callback" class. The callbacks occur by calling public functions of the CCxNotificationRoute class.

CCxRoutedSink::NotificationMessage

Prototype

```
virtual void NotificationMessage(
                  CCxNotificationFromServer * pNotification);
```

Parameters

CCxNotificationFromServer * pNotification	[in] Pointer to an object that comes directly from the Management System. The CCxNotificationFromServer object can wrap either a CCxNotificationInfo or simply a "Subject Broken" notification consisting of the Subject's ID.

Scope

```
Protected
```

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxSink.h>
```

Description

This method determines the type of notification object that is being received by the Management System. Classes can use this method after they take a notification object out of their queue. This method will determine the type of notification (CCxNotificationInfo or "Subject broken") and call the appropriate OnNotifyNOTIFY or OnNotifyNOTIFY_SUBJECTBROKEN method. These two derived methods implement the final delivery of the notification object to the Observer.

Return Values

None	

See Also

CCxRoutedSink::OnNotifyNOTIFY

Prototype

```
virtual void OnNotifyNOTIFY(CCxNotificationInfo* pNotification);
```

Parameters

CCxNotificationInfo* pNotification	[in] pointer to an actual notification object.

Scope

Protected

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxSink.h>
```

Description

This virtual method is the method that gets called for final delivery of the CCxNotificationInfo object back to the Observer. The implementation provided in the CCxRoutedSink object calls back into the CCxNotificationRoute object passed in our SetNotificationRoute method. It is the responsibility of the Observer to "wire" the CCxNotificationRoute and the CCxRoutedSink object's together by calling the SetNotificationRoute method.

Return Values

None	

See Also

SetNotificationRoute

CCxRoutedSink::OnNotifyNOTIFY_SUBJECTBROKEN

Prototype

```
virtual void OnNotifyNOTIFY_SUBJECTBROKEN(long lSubjectID);
```

Parameters

long lSubjectID	[in] ID of the Subject whose subscription is being destroyed in the Management System.

Scope

Protected

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxSink.h>
```

Description

This method gets called for final delivery of a subject broken notification object. The implementation provided in the CCxRoutedSink object calls back into the CCxNotificationRoute object passed in our SetNotificationRoute method. It is the responsibility of the Observer to "wire" the CCxNotificationRoute and the CCxRoutedSink object's together by calling the SetNotificationRoute method.

Return Values

None	

See Also

SetNotificationRoute

CCxRoutedSink::SetNotificationRoute

Prototype

```
CCxNotificationRoute *SetNotificationRoute(
                  CCxNotificationRoute *pRoute)
```

Parameters

CCxNotificationRoute *pRoute	Pointer to an object that inherits from CCxNotificationRoute

Scope

Public

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxSink.h>
```

Description

Call this member function and pass a pointer to an object that the sink object can call back into. The pointer passed to this method MUST inherit from the CCxNotificationRoute class. One or more of the virtual method defined in the CCxNotificationRoute class should be implemented by the derived class.

Refer to the CCxNotificationRoute documentation for details and capabilities of this class.

Return Values

`CCxNotificationRoute *`	Previous notification route pointer, or NULL if there is no previous route

See Also

CCxNotificationRoute class

CCxRoutedSink::ThreadCleanup

Prototype

```
virtual BOOL ThreadCleanup(DWORD dwWait = INFINITE);
```

Parameters

DWORD dwWait /*= INFINITE*/	[in] Timeout value in milliseconds to wait for the sink thread to callback into the CCxNotificationRoute::OnThreadCleanup method.

Scope

`Public`

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxSink.h>
```

Description

This member function sets an event telling the sink thread to callback into the CCxNotificationRoute::OnThreadCleanup method. The implementation of the CCxNotificationRoute::OnThreadCleanup method is completely dependent upon the requirements of the caller. After signaling the event, the caller's thread of execution is blocked dwWait milliseconds.

Users should be careful calling this method passing the 'INIFINITE' value in dwWait. This could potentially block their thread of execution forever.

Return Values

`BOOL`	TRUE if successful, else FALSE if a timeout occurred

See Also

CCxNotificationRoute::OnThreadCleanup

CCxRoutedSink::ThreadInit

Prototype

```
virtual BOOL ThreadInit(DWORD dwWait = INFINITE);
```

Parameters

DWORD dwWait /*= INFINITE*/	[in] Timeout value in milliseconds to wait for the sink thread to callback into the CCxNotificationRoute::OnThreadInit method.

Scope

Public

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxSink.h>
```

Description

This member function sets an event telling the sink thread to callback into the CCxNotificationRoute::OnThreadInit method. The implementation of the CCxNotificationRoute::OnThreadInit method is completely dependent upon the requirements of the caller. After signaling the event, the caller's thread of execution is blocked dwWait milliseconds.

Users should be careful calling this method passing the 'INIFINITE' value in dwWait. This could potentially block their thread of execution forever.

Return Values

BOOL	TRUE if successful, else FALSE if a timeout occurred

See Also

CCxNotificationRoute::OnThreadInit

CCxRoutedSink::ThreadProc

Prototype

```
virtual DWORD ThreadProc();
```

Parameters

None	

Scope

Protected

Definition

Dynamic Link Library	CxNotifyClient.DLL
Header Files	#include <CxSink.h>

Description

This member function does our thread work. The thread has a queue (CCxQueuedSink::m_qNotifications member variable) where ALL notifications from the Management System are gathered. The thread simply pops a notification off its queue, and then calls the NotificationMessage method. The NotificationMessage method determines the type of notification then calls the corresponding CCxNotificationRoute "callback" method.

Return Values

DWORD	Always returns 0.

See Also

CCxRoutedSubjectObserver Class

The CCxRoutedSubjectObserver class implements an Observer with a CCxRoutedSink. CCxRoutedSubjectObserver inherits from CCxObserver. To view the UML class hierarchy diagram, refer to the CCxSubjectOnly section in this document.

The CreateObserver method in this class instantiates a CCxRoutedSink object.

The CCxRoutedSubjectObserver class defines one of the "helper" Observer classes in the Notifications Package. These helper classes provide a pure C++ implementation of an Observer. Absolutely no COM knowledge is needed in order to create and use these derived Observer classes.

The CCxRoutedSubjectObserver class defines the following methods:

Method Name	Description
`~CCxRoutedSubjectObserver`	Cleans up the Observer object
`CCxRoutedSubjectObserver`	Handles all initialization and creation for this object.
`CreateObserver`	Creates a CCxRoutedSink object and automatically connects the object to our ICxSubjectObserver COM object.
`ThreadCleanup`	Called by the client to force the sink object's thread to callback to the CCxNotificationRoute::OnThreadCleanup method.
`ThreadInit`	Called by the client to force the sink object's thread to callback to the CCxNotificationRoute::OnThreadInit method.

The following member variables are exposed by the CCxRoutedSubjectObserver class:

Name	Data Type	Scope	Description
m_pRoute	CCxNotificationRoute *	Protected	Holds a pointer to our "callback" class. The callbacks occur by calling public functions of the CCxNotificationRoute class.

CCxRoutedSubjectObserver::CCxRoutedSubjectObserver

Prototype

```
CCxRoutedSubjectObserver(CCxNotificationRoute *pRoute,
            LPCTSTR pszName = NULL,
            long nPackageID = CX_PKGID_USER,
            LPCTSTR pszServer = NULL,
            long lFlags = CX_NOTIFICATION_INPROC);
```

Parameters

`CCxNotificationRoute *pRoute`	[in] Pointer to an object that inherits from CCxNotificationRoute. This parameter must NOT be NULL.
`LPCTSTR pszName`	[in] Name of the Observer. Default value is NULL if the default constructor is used. If NULL is passed, then the SetMyName method MUST be used to set the name of this Observer.
`long nPackageID`	[in] Package ID of this Observer. Default value is CX_PKGID_USER if the default constructor is used. If no value is passed, the SetMyName method must be used to set the Package ID.
`LPCTSTR pszServer`	[in] Used for DCOM creation of the ICxSubjectObserver COM object. If non-NULL, the name of the server computer where the CxNotify COM server DLL is registered and available for use.
`long lFlags`	[in] Creation flags. Currently this parameter is used only to determine whether to instantiate the ICxSubjectObserver object in process or out of process. The in process flag is CX_NOTIFICATION_INPROC, the out of process flag is CX_NOTIFICATION_OUTOFPROC.

Scope

`Public`

Definition

```
Dynamic Link Library        CxNotifyClient.DLL
Header Files                #include <CxRoutedSO.h>
```

Description

The constructor creates a Notification Package Observer and automatically instantiates the ICxSubjectObserver COM object by calling the base class CCxSubjectOnly constructor. Constructor parameters can be used to efficiently initialize the object. The name and package ID parameters replace calling the ICxSubjectObserver::SetMyName method. The lFlags parameter handles creation of the object in process or out of process as opposed to using the CoCreateInstanceEx WIN32 API. Note that by default, the COM object is created in process.

After creating the ICxSubjectObserver object in our CCxSubjectOnly base class, this method calls the CreateObserver method. CreateObserver instantiates a CCxRoutedSink object and "connects" the sink to the ICxSubjectObserver COM object.

Return Values

None	

See Also

ICxSubjectObserver COM interface
ICxSubjectObserver::SetMyName

CCxRoutedSubjectObserver::~CCxRoutedSubjectObserver

Prototype

```
virtual ~CCxRoutedSubjectObserver ();
```

Parameters

None	

Scope

Public

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxRoutedSO.h>
```

Description

This method handles all cleanup for this routed Subject-Observer object. The destructor will destroy the CCxRoutedSink object and then relies on the base class destructor(s) for their cleanup.

Return Values

None	

See Also

ICxSubjectObserver COM interface

CCxRoutedSubjectObserver::CreateObserver

Prototype

```
virtual HRESULT CreateObserver();
```

Parameters

None	

Scope

Protected

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxRoutedSO.h>
```

Description

This method instantiates a CCxRoutedSink object and sets the m_pSink member variable to the address of the sink.

The method also "connects" the sink object to the ICxSubjectObserver COM object created by our CCxSubjectOnly base class.

Return Values

HRESULT	S_OK if successful, else E_FAIL

See Also

ICxSubjectObserver COM interface
CCxQueuedSink class

CCxRoutedSubjectObserver::ThreadCleanup

Prototype

```
virtual BOOL ThreadCleanup(DWORD dwWait = INFINITE);
```

Parameters

DWORD dwWait /*= INFINITE*/	[in] Timeout value in milliseconds to wait for the sink thread to callback into the CCxNotificationRoute::OnThreadCleanup method.

Scope

```
Public
```

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxRoutedSO.h>
```

Description

This member function sets an event telling the sink thread to callback into the CCxNotificationRoute::OnThreadCleanup method. The implementation of the CCxNotificationRoute::OnThreadCleanup method is completely dependent upon the requirements of the caller. After signaling the event, the caller's thread of execution is blocked dwWait milliseconds.

Users should be careful calling this method passing the 'INIFINITE' value in dwWait. This could potentially block their thread of execution forever.

Return Values

BOOL	TRUE if successful, else FALSE if a timeout occurred

See Also

CCxNotificationRoute::OnThreadCleanup

CCxRoutedSubjectObserver::ThreadInit

Prototype

```
virtual BOOL ThreadInit(DWORD dwWait = INFINITE);
```

Parameters

DWORD dwWait /*= INFINITE*/	[in] Timeout value in milliseconds to wait for the sink thread to callback into the CCxNotificationRoute::OnThreadInit method.

Scope

```
Public
```

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxRoutedSO.h>
```

Description

This member function sets an event telling the sink thread to callback into the CCxNotificationRoute::OnThreadInit method. The implementation of the CCxNotificationRoute::OnThreadInit method is completely dependent upon the requirements of the caller. After signaling the event, the caller's thread of execution is blocked dwWait milliseconds.

Users should be careful calling this method passing the 'INIFINITE' value in dwWait. This could potentially block their thread of execution forever.

Return Values

BOOL	TRUE if successful, else FALSE if a timeout occurred

See Also

CCxNotificationRoute::OnThreadInit

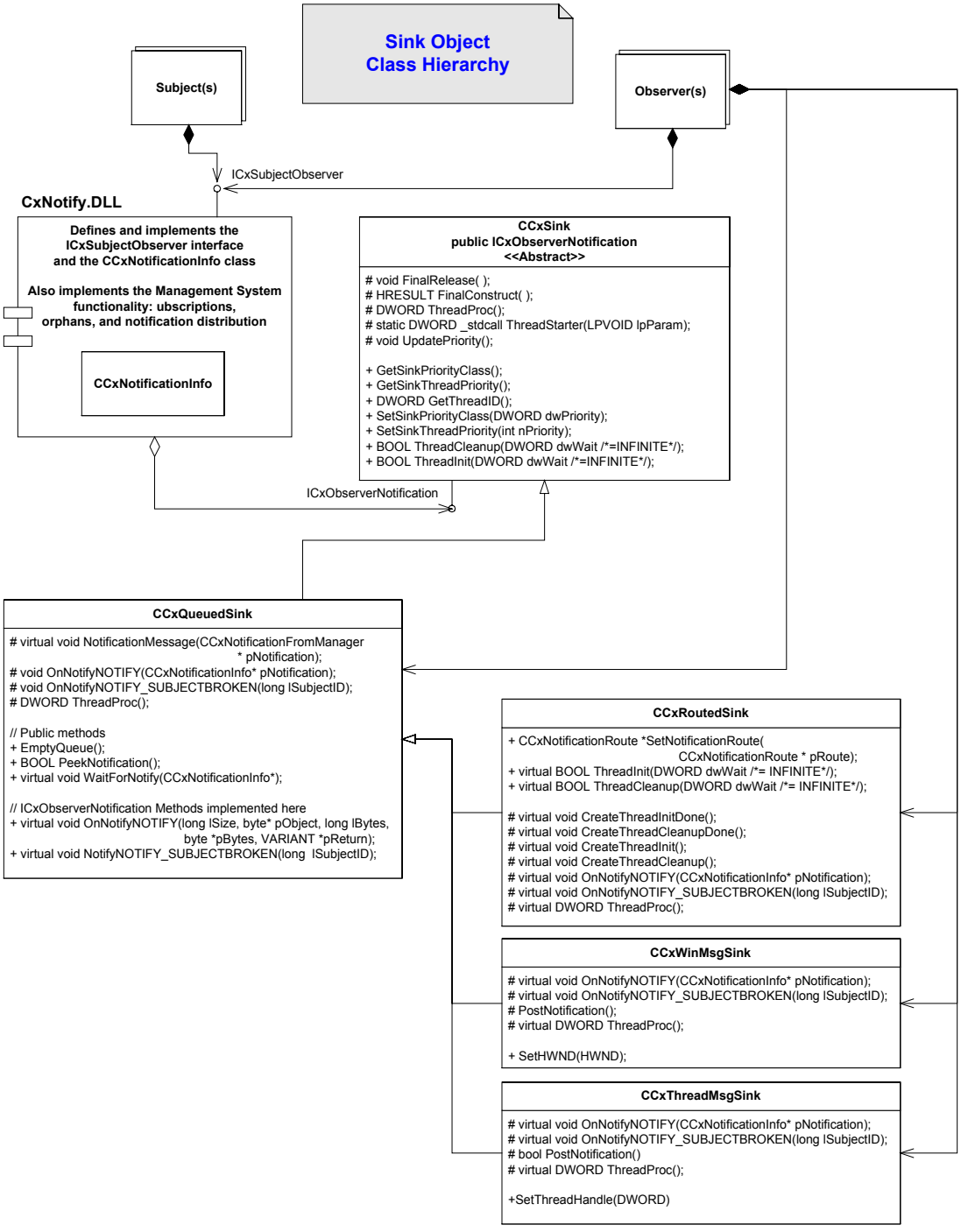
CCxSink Class

The CCxSink class is the base class for all sink objects provided by the Notifications Package. Sink objects are used by the Notifications Package Management System to deliver a notification from a Subject to the Observer. An Observer MUST instantiate a sink object in order to receive a notification.

By definition: If an entity creates an ICxSubjectObserver object and then instantiates a sink object, it is an Observer. In other words, to be an Observer in the Notifications Package framework, the Observer MUST have a sink object. The needs of the Observer will determine which particular type of sink object should be instantiated. For details in individual sink objects refer to their respective documentation.

The CCxSink class can NOT be instantiated by an Observer. This class is an abstract base class because it defines a number of pure virtual methods. The class inherits from the ICxObserverNotification COM interface, but does NOT implement its methods. The derived CCxQueuedSink class actually implements the COM interface methods for all classes that derive from CCxQueuedSink. The CCxSink::ThreadProc method is also a pure virtual function. All derived classes implement their own ThreadProc method according to their own needs. For detailed descriptions of the derived method implementation, refer to the documentation for the derived sink classes.

The entire sink object class hierarchy is shown in the following diagram:



All sink objects are implemented and exposed by the CxNotifyClient.DLL. For details on each of these classes, refer to the documentation on: CCxQueuedSink, CCxRoutedSink, CCxWinMsgSink, or CCxThreadMsgSink.

CCxSink::FinalConstruct

Prototype

```
HRESULT FinalConstruct()
```

Parameters

None	

Scope

```
Protected
```

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxSink.h>
```

Description

This member function starts our sink's worker thread. This thread watches for incoming notifications.

This member function calls the WIN32 CreateThread function and passes our ThreadStarter static function pointer as a parameter. ThreadStarter in turn calls a virtual ThreadProc method. The ThreadProc method is pure virtual in this class and MUST be implemented by a derived class. In short this means that we actually start the thread here in a base class, but the work performed by a derived class' ThreadProc method determines what work is done by the thread.

Return Values

HRESULT	S_OK if a thread was successfully created, else E_FAIL

See Also

CCxSink::FinalRelease

Prototype

```
void FinalRelease( )
```

Parameters

None	

Scope

```
Protected
```

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxSink.h>
```

Description

This method signals an event to tell our worker thread to terminate. This method also performs all cleanup.

Return Values

None	

See Also

CCxSink::GetSinkPriorityClass

Prototype

```
DWORD GetSinkPriorityClass()
```

Parameters

None	

Scope

Public

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxSink.h>
```

Description

This method returns the value of our m_dwPriorityClass member variable. The Priority class is one of the values defined with the WIN32 SetPriorityClass function.

For valid priority class values refer to either the WIN32 SetPriorityClass function or to our CCxSink::SetSinkPriorityClass method.

Return Values

DWORD	Returns the value of our m_dwPriorityClass member variable. The m_dwPriorityClass member variable will hold one of the value listed in the table above.

See Also

WIN32 SetPriorityClass function

CCxSink::GetSinkThreadPriority

Prototype

```
int GetSinkThreadPriority()
```

Parameters

None	

Scope

Public

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxSink.h>
```

Description

This method returns the value of our m_nThreadPriority member variable. The Priority class is one of the values defined with the WIN32 SetThreadPriority function.

For valid priority class values refer to either the WIN32 SetThreadPriority function or to our CCxSink::SetSinkThreadPriority method.

Return Values

int	Returns the value of our m_nThreadPriority member variable. The m_nThreadPriority member variable will hold one of the value listed in the table above.

See Also

CCxSink::GetThreadID

Prototype

```
DWORD GetThreadID()
```

Parameters

None	

Scope

Public

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxSink.h>
```

Description

This method returns the value of our m_dwThreadID member variable. This value gets set during the creation of our Sink's worker thread. It is returned as one of the parameters to the WIN32 CreateThread function.

Return Values

DWORD	Returns the value of our m_dwThreadID member variable.

See Also

CCxSink::SetSinkPriorityClass

Prototype

```
BOOL SetSinkPriorityClass(DWORD dwPriorityClass)
```

Parameters

DWORD dwPriorityClass	[in] Valid values are defined by the WIN32 SetPriorityClass function. See below

Scope

```
Public
```

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxSink.h>
```

Description

This member function allows the caller to set the priority class of the sink's worker thread. The implementation of this method actually sets one or more internal member variables and then sets an NT event. The NT event tells our worker thread to call the WIN32 SetPriorityClass function and use the new priority class value.

The priority class can be one of the following values.

Priority	Meaning
ABOVE_NORMAL_PRIORITY_CLASS	**Windows 2000:** Indicates a process that has priority above NORMAL_PRIORITY_CLASS but below HIGH_PRIORITY_CLASS.
BELOW_NORMAL_PRIORITY_CLASS	**Windows 2000:** Indicates a process that has priority above IDLE_PRIORITY_CLASS but below NORMAL_PRIORITY_CLASS .
HIGH_PRIORITY_CLASS	Specify this class for a process that performs time-critical tasks that must be executed immediately. The threads of the process preempt the threads of normal or idle priority class processes. An example is the Task List, which must respond

	quickly when called by the user, regardless of the load on the operating system. Use extreme care when using the high-priority class, because a high-priority class application can use nearly all available CPU time.
IDLE_PRIORITY_CLASS	Specify this class for a process whose threads run only when the system is idle. The threads of the process are preempted by the threads of any process running in a higher priority class. An example is a screen saver. The idle-priority class is inherited by child processes.
NORMAL_PRIORITY_CLASS	Specify this class for a process with no special scheduling needs.
REALTIME_PRIORITY_CLASS	Specify this class for a process that has the highest possible priority. The threads of the process preempt the threads of all other processes, including operating system processes performing important tasks. For example, a real-time process that executes for more than a very brief interval can cause disk caches not to flush or cause the mouse to be unresponsive.

Return Values

BOOL	TRUE if successful, else FALSE

See Also

CCxSink::SetSinkThreadPriority

Prototype

```
BOOL SetSinkThreadPriority(int nPriority)
```

Parameters

`int nPriority`	[in] Valid values are defined by the WIN32 SetThreadPriority function. See below.

Scope

```
Public
```

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxSink.h>
```

Description

This member function allows the caller to set the thread priority of the sink's worker thread. The implementation of this method actually sets one or more internal member variables and then sets an NT event. The NT event tells our worker thread to call the WIN32 SetThreadPriority function and use the new thread priority value. The call to SetThreadPriority MUST be made by the thread requesting the new setting.

The thread priority can be one of the following values.

Priority	Meaning
THREAD_PRIORITY_ABOVE_NORMAL	Indicates 1 point above normal priority for the priority class.
THREAD_PRIORITY_BELOW_NORMAL	Indicates 1 point below normal priority for the priority class.
THREAD_PRIORITY_HIGHEST	Indicates 2 points above normal priority for the priority class.
THREAD_PRIORITY_IDLE	Indicates a base priority level of 1 for IDLE_PRIORITY_CLASS, BELOW_NORMAL_PRIORITY_CLASS, NORMAL_PRIORITY_CLASS, ABOVE_NORMAL_PRIORITY_CLASS, or HIGH_PRIORITY_CLASS processes, and a base priority level of 16 for REALTIME_PRIORITY_CLASS processes.
THREAD_PRIORITY_LOWEST	Indicates 2 points below normal priority for the priority class.

THREAD_PRIORITY_NORMAL	Indicates normal priority for the priority class.
THREAD_PRIORITY_TIME_CRITICAL	Indicates a base priority level of 15 for IDLE_PRIORITY_CLASS, BELOW_NORMAL_PRIORITY_CLASS, NORMAL_PRIORITY_CLASS, ABOVE_NORMAL_PRIORITY_CLASS, or HIGH_PRIORITY_CLASS processes, and a base priority level of 31 for REALTIME_PRIORITY_CLASS processes.

Return Values

BOOL	TRUE if successful, else FALSE

See Also

CCxSink::ThreadCleanup

Prototype

```
virtual BOOL ThreadCleanup(DWORD dwWait /*=INFINITE*/)
```

Parameters

DWORD dwWait	[in] A timeout value that is passed to WaitForSingleObject or WaitForMultipleObjects.

Scope

```
Public
```

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxSink.h>
```

Description

This is a virtual function that should be implemented by a derived class. By default, this method does nothing. The following source code shows exactly what the default implementation does:

```
BOOL CCxSink::ThreadCleanup(DWORD dwWait /*=INFINITE*/)
{
      TRACE("CCxSink::ThreadCleanup() - dwWait = %ld\n", dwWait);
      ASSERT(FALSE);
      return FALSE;
}
```

In DEBUG mode, an assertion will be triggered. This should tell the developer that they either shouldn't be calling this method or that they need to implement this method in their derived class.

For information on a derived class implementation, refer to the CCxRoutedSink::ThreadCleanup method.

Return Values

DWORD	The value is determined by a derived class implementation.

See Also

CCxRoutedSink::ThreadCleanup

CCxSink::ThreadInit

Prototype

```
BOOL ThreadInit(DWORD dwWait /*=INFINITE*/)
```

Parameters

DWORD dwWait	[in] A timeout value that is passed to WaitForSingleObject or WaitForMultipleObjects.

Scope

```
Public
```

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxSink.h>
```

Description

This is a virtual function that should be implemented by a derived class. By default, this method does nothing. The following source code shows exactly what the default implementation does:

```
BOOL CCxSink::ThreadInit(DWORD dwWait /*=INFINITE*/)
{
      TRACE("CCxSink::ThreadInit() - dwWait = %ld\n", dwWait);
      ASSERT(FALSE);
      return FALSE;

}
```

In DEBUG mode, an assertion will be triggered. This should tell the developer that they either shouldn't be calling this method or that they need to implement this method in their derived class.

For information on a derived class implementation, refer to the CCxRoutedSink::ThreadInit method.

Return Values

DWORD	The value is determined by a derived class implementation.

See Also

CCxRoutedSink::ThreadInit

CCxSink::ThreadProc

Prototype

```
virtual DWORD ThreadProc() = 0;
```

Parameters

None	

Scope

Protected

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxSink.h>
```

Description

This method is pure virtual and MUST be implemented by a derived class.

Return Values

DWORD	Anything a derived class wants.

See Also

CCxRoutedSink::ThreadProc, CCxQueuedSink::ThreadProc, CCxWinMsgSink::ThreadProc, CCxThreadMsgSink::ThreadProc

CCxSink::ThreadStarter

Prototype

```
DWORD _stdcall ThreadStarter( LPVOID lpParam )
```

Parameters

LPVOID lpParam	[in] This parameter is cast to the 'this' pointer in our call to the WIN32 CreateThread function.

Scope

Protected

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxSink.h>
```

Description

This static function is the main entry point to our thread procedure. There is a single line of code in our implementation:

```
return ((CCxSink*)lpParam)->ThreadProc();
```

This line of code casts the lpParam parameter to a CCxSink pointer and then calls the polymorphic ThreadProc method. Each derived class MUST implement the ThreadProc method.

Return Values

DWORD	Returns the value from a derived class' implementation of ThreadProc

See Also

ThreadProc

CCxSink::UpdatePriority

Prototype

```
void UpdatePriority()
```

Parameters

None	

Scope

```
Protected
```

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxSink.h>
```

Description

This member function is called by a ThreadProc function any time that the SetSinkThreadPriority or SetSinkPriorityClass method is called.

It is not necessary for a client to call this method directly.

Return Values

None	

See Also

CCxSinkTypes Class

The CCxSinkTypes class is a helper class that provides sink type flags and strings that could be used to populate a GUI. This class is completely optional and provided only as a reference to programmers using the Notifications Package.

The class defines enumerated values and string names for all supported sink types exposed by the Notifications Package. For a detailed description of sinks, refer to the CCxSink class documentation.

The following table summarizes the available methods

Method Name	Description
`FindSinkTypeID`	Given a string find its associated sink type id.
`FindSinkTypeString`	Given a sink type id find its associated string representation.

The following table summarizes the member variables

Name	Data Type	Scope	Description
m_aSinkTypes	structSinkTypes *	Protected	An array of structures of sink types. The structure is defined as: typedef struct structSinkTypes { int m_id; LPCTSTR m_str; } structSinkTypes;

This class "wraps" the following enumerated sink types

```
enum
{
      CX_SINKTYPE_WINMSG = 0,
      CX_SINKTYPE_QUEUED,
      CX_SINKTYPE_ROUTED,
      CX_SINKTYPE_THREADMSG,
      CX_SINKTYPE_NONE,
      CX_SINKTYPE_COUNT       // CX_SINKTYPE_END_VALUE
                              // must be the last value
};
```

The structure defined in the member variable table holds both the enumerated value of a sink type and a corresponding string representation of the sink type. The string representation can be used to populate a list box in a GUI for example. The MultiNotify sample application provided with the Notifications Package uses the CCxSinkTypes class to populate a list box control.

The following code is used in our constructor to load the m_aSinkTypes member variable:

```
CCxSinkTypes::CCxSinkTypes()
{
```

```
m_aSinkTypes = new structSinkTypes[CX_SINKTYPE_COUNT + 1];

static structSinkTypes tmpSinkTypes [] =
{
{ CX_SINKTYPE_WINMSG,    _T( "Windows Message Sink" )},
{ CX_SINKTYPE_QUEUED,    _T( "Queued Sink" )},
{ CX_SINKTYPE_ROUTED,    _T( "Routed Sink" )},
{ CX_SINKTYPE_THREADMSG, _T( "Thread Message Sink" ) },
{ CX_SINKTYPE_NONE,      _T( "No Sink" )    },
{ CX_SINKTYPE_COUNT,     _T( "" ) }  // must be last value
                                     // and must be empty
};

// Fill the structure
for (int i = CX_SINKTYPE_WINMSG; i<=CX_SINKTYPE_COUNT; i++)
{
      m_aSinkTypes[i].m_id  = tmpSinkTypes[i].m_id;
      m_aSinkTypes[i].m_str = tmpSinkTypes[i].m_str;
}}
```

CCxSinkTypes::FindSinkTypeID

Prototype

```
int FindSinkTypeID(CString strSType);
```

Parameters

CString strSType	[in] Sink type string to search for. This string must be one of the values that was loaded into our structSinkTypes array member variable.

Scope

Public

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxSinkTypes.h>
```

Description

This method searches for the string passed in strSType. If it finds the string, the method returns the ID of the string. The ID is one of the enumerated sink type values defined above.

For example, if the string "Queued Sink" is passed into this method, we return the CX_SINKTYPE_QUEUED enumerated value.

Return Values

int	Enumerated value found in the m_aSinkTypes member variable, else CX_SINKTYPE_COUNT if the value is not found

See Also

FindSinkTypeString

CCxSinkTypes::FindSinkTypeString

Prototype

```
LPCTSTR FindSinkTypeString(int id);
```

Parameters

int id	[in] ID to search for. The ID number must be in our m_aSinkTypes member variable.

Scope

```
Public
```

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxSinkTypes.h>
```

Description

This method searches for the id passed in. If it finds the id, the method returns the string representation of the id.

For example, if the id CX_SINKTYPE_QUEUED is passed in, then we return "Queued Sink".

Return Values

LPCTSTR	Enumerated value found in the m_aSinkTypes member variable, else "" if the value is not found

See Also

FindSinkTypeID

CCxSubjectOnly Class

The CCxSubjectOnly class is a helper class that hides all COM details from the user of the Notifications Package. Users who want to create a Subject, but do not want to deal with the complexities of the COM CoCreateInstance API's can instantiate this class.

CCxSubjectOnly is the base class for a number of "helper" classes in the Notifications Package. These helper classes provide a pure C++ implementation of the Subject, and various kinds of Observers. Absolutely no COM knowledge is needed in order to create and use these classes. To see an overview of the class hierarchy of all the Subject and Observer "helper" classes, refer to the C++ Subjects and Observers section.

The CCxSubjectOnly class defines the following methods:

Method Name	Description
`CCxSubjectOnly`	Handles all initialization and creation for this object. Of particular importance is the creation of the ICxSubjectObserver COM object.
`~CCxSubjectOnly`	Cleans up this Subject object
`CreateICxSubjectObserver`	Creates the ICxSubjectObserver COM object
`GetCountMyObservers`	Returns a count of the number of Observers of this Subject
`GetError`	Returns a string description of the last error that occurred
`GetICxSubjectObserver`	Returns a pointer to the ICxSubjectObserver object stored in our internal member variable
`GetIDFromName`	Returns the ID of an object based on a name.
`GetMyName`	Returns the string name of this Subject
`GetMyObjectID`	Returns the object ID of this Subject
`GetMyPackageID`	Returns the package ID of this Subject
`GetNameFromID`	Returns a string name based on an ID
`GetSubscribedFilterMask`	Returns a mask of all subscriptions requested by all of this Subject's Observers.
`IsSubscribed`	Returns true if a particular Observer is subscribed to this Subject
`PostNotify`	Post an asynchronous notification
`SendNotify`	Sends a synchronous notification
`SetMyName`	Sets the name and package ID of this Subject.

The CCxSubjectOnly class exposes the following member variables:

Name	Data Type	Scope	Description
m_lFlags	long	Protected	Holds the flag values this Subject was created with.

m_lPackageID	long	Protected	Holds this Subject's package ID
m_pSubjectObserver	ICxSubjectObserver *	Private	Holds a pointer to the ICxSubjectObserver object
m_strName	CString	Protected	Holds this Subject's name
m_strServer	CString	Protected	Holds the name of the server where the ICxSubjectObserver object was instantiated from (DCOM).

CCxSubjectOnly::CCxSubjectOnly

Prototype

```
CCxSubjectOnly(LPCTSTR pszName = NULL,
            long nPackageID = CX_PKGID_USER,
            LPCTSTR pszServer = NULL,
            long lFlags = CX_NOTIFICATION_INPROC);
```

Parameters

`LPCTSTR pszName`	[in] Name of the Subject. Default value is NULL if the default constructor is used. If NULL is passed, then the SetMyName method MUST be used to set the name of this Subject.
`long nPackageID`	[in] Package ID of this Subject. Default value is CX_PKGID_USER if the default constructor is used. If no value is passed, the SetMyName method must be used to set the Package ID.
`LPCTSTR pszServer`	[in] Used for DCOM creation of the ICxSubjectObserver COM object. If non-NULL, the name of the server computer where the CxNotify COM server DLL is registered and available for use.
`long lFlags`	[in] Creation flags. Currently this parameter is used only to determine whether to instantiate the ICxSubjectObserver object in process or out of process. The in process flag is CX_NOTIFICATION_INPROC, the out of process flag is CX_NOTIFICATION_OUTOFPROC.

Scope

```
Public
```

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxSubject.h>
```

Description

The constructor creates a Notification Package Subject and automatically instantiates the ICxSubjectObserver COM object. Constructor parameters can be used to efficiently initialize the object. The name and package ID parameters replace calling the ICxSubjectObserver::SetMyName method. The lFlags parameter handles creation of the object in process or out of process as opposed to using the CoCreateInstanceEx WIN32 API. Note that by default, the COM object is created in process.

The constructor calls the CreateICxSubjectObserver method to create the COM object and assign it to our m_pSubjectObserver member variable. Note that since m_pSubjectObserver is a private member variable, the only access to it is via our GetICxSubjectObserver method.

Return Values

None	

See Also

ICxSubjectObserver COM interface
ICxSubjectObserver::SetMyName
GetICxSubjectObserver

CCxSubjectOnly::~CCxSubjectOnly

Prototype

```
virtual ~CCxSubjectOnly();
```

Parameters

None	

Scope

Public

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxSubject.h>
```

Description

This method handles all cleanup for this object. Of particular importance is the destruction of the ICxSubjectObserver COM object we created in our constructor.

Return Values

None	

See Also

ICxSubjectObserver COM interface

CCxSubjectOnly::CreateICxSubjectObserver

Prototype

```
HRESULT CreateICxSubjectObserver();
```

Parameters

None	

Scope

Protected

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxSubject.h>
```

Description

This method is automatically called by our constructor in order to create the ICxSubjectObserver COM object. The pointer to the object is stored in our m_pSubjectObserver member variable.

Return Values

HRESULT	S_OK if the COM object was successfully created. Otherwise we return the HRESULT value from CoCreateInstanceEx.

See Also

ICxSubjectObserver COM interface

CCxSubjectOnly::GetCountMyObservers

Prototype

```
HRESULT GetCountMyObservers(long &lCount);
```

Parameters

Long lCount	[out] pointer to a long that receives the count of Observers of this Subject

Scope

Public

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxSubject.h>
```

Description

This method returns the number of Observers of this Subject. For more information refer to the documentation of ICxSubjectObserver::GetCountMyObservers.

Return Values

HRESULT	Returns the value from ICxSubjectObserver::GetCountMyObservers.

See Also

ICxSubjectObserver COM interface
ICxSubjectObserver::GetCountMyObservers

CCxSubjectOnly::GetError

Prototype

```
HRESULT GetError(BSTR &bstrError);
```

Parameters

BSTR &bstrError	[out] Reference to a BSTR that receives the error string from the last method call on ICxSubjectObserver

Scope

Public

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxSubject.h>
```

Description

This method delegates to the ICxSubjectObserver::GetError method. For more details refer to the documentation of the ICxSubjectObserver::GetError method.

Return Values

HRESULT	Returns the HRESULT value from ICxSubjectObserver::GetError

See Also

ICxSubjectObserver COM interface
ICxSubjectObserver::GetError

CCxSubjectOnly::GetICxSubjectObserver

Prototype

```
ICxSubjectObserver *GetICxSubjectObserver(bool bAddRef = true);
```

Parameters

bool bAddRef	[in] If true, then addref the m_pSubjectObserver pointer before returning the pointer. Default value is 'true'.

Scope

Public

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxSubject.h>
```

Description

This method returns a pointer to the ICxSubjectObserver COM object we hold in our m_pSubjectObserver member variable.

Return Values

ICxSubjectObserver *	Returns the address of the ICxSubjectObserver COM object, else NULL if an error occurred.

See Also

ICxSubjectObserver COM interface

CCxSubjectOnly::GetIDFromName

Prototype

```
HRESULT GetIDFromName(BSTR bstrName, long &lID);
```

Parameters

BSTR bstrName	[in] Name to search for. Note that wildcards are NOT accepted in this parameter. The passed in name MUST match an existing Subject or Observer, if not, then lID will be set to -1.
long &lId	[out] Reference to a long that receives the ID of the named Subject or Observer in bstrName. If no Subject or Observer exists, lId is set to -1.

Scope

Public

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxSubject.h>
```

Description

This method returns the ID number of the Subject or Observer whose name is passed in bstrName.

Wildcard names are NOT supported by this method. The name passed is bstrName MUST match exactly to an existing Subject or Observer. Note that names are case sensitive.

Return Values

HRESULT	Returns the HRESULT from m_pSubjectObserver->GetIDFromName(…);

See Also

ICxSubjectObserver COM interface
ICxSubjectObserver::GetIDFromName

CCxSubjectOnly::GetMyName

Prototype

```
HRESULT GetMyName(BSTR &bstrName);
```

Parameters

BSTR &bstrName	[out] Reference to a BSTR that receives the name of this Subject

Scope

Public

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxSubject.h>
```

Description

This method returns the name of this Subject in the bstrName parameter. For additional details on this method refer to ICxSubjectObserver::GetMyName.

Return Values

HRESULT	Returns the HRESULT from m_pSubjectObserver->GetMyName(…);

See Also

ICxSubjectObserver COM interface
ICxSubjectObserver::GetMyName

CCxSubjectOnly::GetMyObjectID

Prototype

```
HRESULT GetMyObjectID(long &lID);
```

Parameters

long &lID	[out] A reference to a long that receives the object ID of this Subject.

Scope

Public

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxSubject.h>
```

Description

This method returns the object ID of this Subject in the lID parameter. For additional details on this method refer to ICxSubjectObserver::GetMyObjectID.

Return Values

HRESULT	Returns the HRESULT from m_pSubjectObserver->GetMyObjectID(…);

See Also

ICxSubjectObserver COM interface
ICxSubjectObserver::GetMyObjectID

CCxSubjectOnly::GetMyPackageID

Prototype

```
HRESULT GetMyPackageID(long &lPackageID);
```

Parameters

long &lPackageID	[out] Reference to a long that receives the package ID of this Subject.

Scope

Public

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxSubject.h>
```

Description

This method returns the package ID of this Subject in the lPackageID parameter. For additional details on this method refer to ICxSubjectObserver::GetMyPackageID.

Return Values

HRESULT	Returns the HRESULT from m_pSubjectObserver->GetMyPackageID(…);

See Also

ICxSubjectObserver COM interface
ICxSubjectObserver::GetMyPackageID

CCxSubjectOnly::GetNameFromID

Prototype

```
HRESULT GetNameFromID(long lID, BSTR &bstrName);
```

Parameters

long lID	[in] ID to search for
BSTR &bstrName	[out] Reference to a BSTR that receives the name of the Subject or Observer whose ID matches lID. NULL if not found.

Scope

```
Public
```

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxSubject.h>
```

Description

This method returns the name of the Subject or Observer whose ID matches lID. For additional information refer to the ICxSubjectObserver::GetNameFromID method.

Return Values

HRESULT	Returns the HRESULT from m_pSubjectObserver->GetNameFromID(…);

See Also

ICxSubjectObserver COM interface
ICxSubjectObserver::GetNameFromID

CCxSubjectOnly::GetSubscribedFilterMask

Prototype

```
HRESULT GetSubscribedFilterMask(long &lMask);
```

Parameters

long &lMask	[out] Reference to a long that receives a bitmask with ALL subscribed Observer subscription requests.

Scope

Public

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxSubject.h>
```

Description

This method returns a bit mask will all Observers' subscription request notification types AND'd together. For details on subscription notification types, refer to the ICxSubjectObserver::SubscribeByName and the ICxSubjectObserver::SubscribeByID methods.

Each Observer who subscribes to this Subject MUST pass in a notification type as part of the subscription request. Each of these notification types will be AND'd together and returned in the lMask parameter.

Return Values

HRESULT	Returns the HRESULT from m_pSubjectObserver->GetSubscribedFilterMask(…);

See Also

ICxSubjectObserver COM interface
ICxSubjectObserver::SubscribeByName
ICxSubjectObserver::SubscribeByID

CCxSubjectOnly::IsSubscribed

Prototype

```
HRESULT IsSubscribed(long lID);
```

Parameters

long lID	[in] ID of an Observer who may or may not be subscribed to this Subject

Scope

Public

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
```

Header Files #include <CxSubject.h>

Description

This method determines if the ObserverID passed by lID is subscribed to this Subject. This method wraps the ICxSubjectObserver::IsSubscribed method. For additional documentation, refer to the documentation of that method.

Return Values

HRESULT	S_OK if the Observer ID specified by lID is subscribed to this Subject, else E_FAIL if not.

See Also

ICxSubjectObserver COM interface
ICxSubjectObserver::IsSubscribed

CCxSubjectOnly::PostNotify

Prototype

```
HRESULT PostNotify(long lSizeNotification,
            byte* pNotification,
            long lDataType,
            long lNotificationType,
            long lPriority,
            long lExtra);
```

Parameters

long lSizeNotification	[in] The number of bytes of the pNotification buffer
byte *pNotification	[in] The notification data buffer. Anything can be embedded in this buffer EXCEPT embedded pointers. In other words, this must be a single contiguous piece of memory.
long lDataType	[in]Flag parameter specifying the type of data. The Notification Package provides some optional predefined data types, but there is no built-in dependency to the pre-defined types. Users are free to implement and use their own data types.
long lNotificationType	[in] Bit-mask used by the Management system. This value is bit-wise AND'd with the lNotificationFilter value that was passed during a subscription request from an Observer. If the bit-wise '&' returns TRUE, the Management System forwards the notification on to the Observer(s). Otherwise, the Management System stops the notification.
long lPriority	[in] The notification priority. The higher the number, the higher the priority.
long lExtra	[in] Extra data parameter that the Subject can use any way they want. Note: Pointer data must

	NOT be passed in this parameter.

Scope

Public

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxSubject.h>
```

Description

This method wraps the ICxSubjectObserver::PostNotify method. For a detailed description of the capabilities of this method refer to its documentation.

Return Values

HRESULT	HRESULT from the m_pSubjectObserver->PostNotify method

See Also

ICxSubjectObserver COM interface
ICxSubjectObserver::PostNotify

CCxSubjectOnly::SendNotify

Prototype

```
HRESULT SendNotify(long lSizeNotification,
                   byte* pNotification,
                   long lDataType,
                   long lNotificationType,
                   long lPriority,
                   long lExtra,
                   long lTimeout);
```

Parameters

lSizeNotification	[in] Size (in bytes) of the data parameter pNotification
byte *pNotification	[in] Data buffer
long lDataType	[in] Flag parameter specifying the type of data. The Notification Package provides optional predefined data types, but there is no built-in dependency to the pre-defined types. Users are free to implement and use their own data types.
lNotificationType	[in] Bit-mask used by the Management system. This value is bit-wise AND'd with the lNotificationFilter value that was passed during a subscription request from an Observer. If the bit-wise '&' returns TRUE, the Management System forwards the notification on to the Observer(s). Otherwise, the Management System stops the

	notification.
`long lPriority`	[in] The notification priority. The higher the number, the higher the priority.
`long lExtra`	[in] Extra data parameter that the user can use any way they want. Note: Pointer data must NOT be passed in this parameter.
`long lTimeout`	Timeout value in milliseconds to wait for observer to handle notifications before returning control. To wait forever, pass the value INFINITE.

Scope

`Public`

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxSubject.h>
```

Description

This method wraps the ICxSubjectObserver::SendNotify method. For a detailed description of the capabilities of this method refer to its documentation.

Return Values

`HRESULT`	HRESULT from the m_pSubjectObserver->SendNotify method

See Also

ICxSubjectObserver COM interface
ICxSubjectObserver::SendNotify

CCxSubjectOnly::SetMyName

Prototype

```
HRESULT SetMyName(BSTR bstrName, long lPackageID);
```

Parameters

`BSTR bstrName`	[in] New name of this object. Note that an objects name can change any number of times during its lifetime. Valid characters include all ASCII characters except: "^$|*+?()[]\" Special Note: As an object changes its name, it carries forward any previous Observer subscriptions that it may have had.
long lPackageID	[in] Package ID number. The package ID number must be a value of CX_PKGID_USER + n. CX_PKGID_USER is

	defined in CxNotifyServer.h

Scope

`Public`

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxSubject.h>
```

Description

This method wraps the ICxSubjectObserver::SetMyName method. For a detailed description of the capabilities of this method refer to its documentation.

Return Values

`HRESULT`	HRESULT from the m_pSubjectObserver->SetMyName method

See Also

ICxSubjectObserver COM interface
ICxSubjectObserver::SetMyName

CCxThreadMsgSink Class

Clients who wish to receive notifications via a worker thread should use the CCxThreadMsgSink class. The CCxThreadMsgSink inherits from a CCxQueuedSink, which in turn inherits from the CCxSink class. Refer to the CCxSink class to view the class hierarchy relationships between the sink classes.

Incoming notifications from the Management System arrive via the ICxObserverNotification COM interface and are placed in our queue member variable. Outgoing notifications to an Observer object are eventually sent via a call to PostThreadMessage using the thread ID passed in our SetThreadHandle method. The outgoing notification is taken out of the queue and forwarded on to a worker thread.

The CCxThreadMsgSink class only needs a valid thread ID in order to post notifications to a client. For an example of a worker thread class that can be used to receive and handle these notifications, refer to the CCxNotificationThread class documentation.

The CCxThreadMsgSink class defines the following methods:

Method Name	Description
`OnNotifyNOTIFY`	Member function that is called by our NotificationMessage method. The implementation of OnNotifyNOTIFY delivers the final CCxNotificationInfo object to the Observer.
`OnNotifyNOTIFY_SUBJECTBROKEN`	Member function that is called by our NotificationMessage method. The implementation of OnNotifyNOTIFY_SUBJECTBROKEN delivers the final subject broken notification object to the Observer.
`SetThreadHandle`	Called by an Observer to provide the sink with the thread ID where notifications get posted to via the PostThreadMessage WIN32 API.
`ThreadProc`	Virtual function that implements any special threading needs for this sink object. Derived classes implement their own version of ThreadProc according to the special needs of the sink object.

The CCxThreadMsgSink class exposes the following member variables:

Name	Data Type	Scope	Description
m_dwPostThreadMessageThreadID	DWORD	Protected	Holds the thread ID that notifications are posted to.

CCxThreadMsgSink::OnNotifyNOTIFY

Prototype

```
virtual void OnNotifyNOTIFY(CCxNotificationInfo* pNotification);
```

Parameters

CCxNotificationInfo* pNotification	[in] pointer to an actual notification object.

Scope

Protected

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxSink.h>
```

Description

This virtual method is the method that gets called for final delivery of the CCxNotificationInfo object back to the Observer. The implementation provided in the CCxThreadMsgSink object posts the notification to a worker thread by calling the WIN32 PostThreadMessage API. It is the responsibility of the Observer to create the worker thread and then call the SetThreadHandle method of our class.

Return Values

None	

See Also

SetThreadHandle

CCxThreadMsgSink::OnNotifyNOTIFY_SUBJECTBROKEN

Prototype

```
virtual void OnNotifyNOTIFY_SUBJECTBROKEN(long lSubjectID);
```

Parameters

long lSubjectID	[in] ID of the Subject whose subscription is being destroyed in the Management System.

Scope

Protected

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxSink.h>
```

Description

This method gets called for final delivery of a subject broken notification object. The implementation provided in the CCxThreadMsgSink object posts the notification to a worker thread by calling the WIN32 PostThreadMessage API. It is the responsibility of the Observer to create the worker thread and then call the SetThreadHandle method of our class.

Return Values

None	

See Also

SetThreadHandle

CCxThreadMsgSink::SetThreadHandle

Prototype

```
BOOL SetThreadHandle(DWORD dwThreadID)
```

Scope

Public

Parameters

DWORD dwThreadID	Thread handle where notifications will be sent

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxSink.h>
```

Description

Call this member function to set the thread handle where notifications will be forwarded. This thread handle MUST be a valid thread ID. Thread ID's come from the WIN32 API CreateThread as well as other thread creation API's.

The CCxNotificationThread class is provided as a helper class for users who would like to receive notification on a worker thread. Refer to the documentation on the CCxNotificationThread class for more information.

Return Values

BOOL	TRUE if the Thread handle is valid and messages can be posted to the thread, else FALSE

See Also

CCxNotificationThread class

CCxThreadMsgSink::ThreadProc

Prototype

```
virtual DWORD ThreadProc();
```

Parameters

None	

Scope

Protected

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxSink.h>
```

Description

This member function does our thread work. The thread has a queue (CCxQueuedSink::m_qNotifications member variable) where ALL notifications from the Management System are stored. The thread simply pops a notification off its queue, and then calls the CCxQueuedSink::NotificationMessage method. The CCxQueuedSink::NotificationMessage method determines the type of notification then calls the corresponding CCxNotificationThread "OnNotify" method.

Return Values

DWORD	Always returns 0.

See Also

CCxThreadMsgSubjectObserver Class

The CCxThreadMsgSubjectObserver class implements an Observer with a CCxThreadMsgSink. CCxThreadMsgSubjectObserver inherits from CCxQueuedSubjectObserver. To view the UML class hierarchy diagram, refer to the CCxSubjectOnly section in this document.

The CreateObserver method in this class instantiates a CCxThreadMsgSink object.

The CCxThreadMsgSubjectObserver class defines one of the "helper" Observer classes in the Notifications Package. These helper classes provide a pure C++ implementation of an Observer. Absolutely no COM knowledge is needed in order to create and use these derived Observer classes.

The CCxThreadMsgSubjectObserver class defines the following methods:

Method Name	Description
~CCxThreadMsgSubjectObserver	Cleans up the Observer object
CCxThreadMsgSubjectObserver	Handles all initialization and creation for this object.
CreateObserver	Creates a CCxThreadMsgSink object and automatically connects the object to our ICxSubjectObserver COM object.

The following member variables are exposed by the CCxThreadMsgSubjectObserver class:

Name	Data Type	Scope	Description
m_dwThreadID	DWORD	Protected	Holds the thread ID passed to us in the constructor

CCxThreadMsgSubjectObserver::CCxThreadMsgSubjectObserver

Prototype

```
CCxThreadMsgSubjectObserver(DWORD dwThreadID,
            LPCTSTR pszName = NULL,
            long nPackageID = CX_PKGID_USER,
            LPCTSTR pszServer = NULL,
            long lFlags = CX_NOTIFICATION_INPROC);
```

Parameters

DWORD dwThreadID	[in] Thread ID where notifications are posted to.
LPCTSTR pszName	[in] Name of the Observer. Default value is NULL if the default constructor is used. If NULL is passed, then the SetMyName method MUST be used to set the name of this Observer.
long nPackageID	[in] Package ID of this Observer. Default value is CX_PKGID_USER if the default constructor is used. If no value is passed, the SetMyName method must be used to set the Package ID.
LPCTSTR pszServer	[in] Used for DCOM creation of the

	ICxSubjectObserver COM object. If non-NULL, the name of the server computer where the CxNotify COM server DLL is registered and available for use.
long lFlags	[in] Creation flags. Currently this parameter is used only to determine whether to instantiate the ICxSubjectObserver object in process or out of process. The in process flag is CX_NOTIFICATION_INPROC, the out of process flag is CX_NOTIFICATION_OUTOFPROC.

Scope

Public

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxThreadMsgSO.h>
```

Description

The constructor creates a Notification Package Observer and automatically instantiates the ICxSubjectObserver COM object by calling the base class CCxSubjectOnly constructor. Constructor parameters can be used to efficiently initialize the object. The name and package ID parameters replace calling the ICxSubjectObserver::SetMyName method. The lFlags parameter handles creation of the object in process or out of process as opposed to using the CoCreateInstanceEx WIN32 API. Note that by default, the COM object is created in process.

After creating the ICxSubjectObserver object in our CCxSubjectOnly base class, this method calls the CreateObserver method. CreateObserver instantiates a CCxThreadMsgSink object and "connects" the sink to the ICxSubjectObserver COM object.

All notifications are sent to the thread specified in the dwThreadID parameter. dwThreadID MUST specify a valid thread ID. We use the PostThreadMessage to send notifications to this thread. To create a worker thread that can accept and handle notifications refer to the CCxNotificationThread documentation.

Return Values

None	

See Also

ICxSubjectObserver COM interface
CCxNotificationThread class
ICxSubjectObserver::SetMyName

CCxThreadMsgSubjectObserver::~CCxThreadMsgSubjectObserver

Prototype

```
virtual ~CCxThreadMsgSubjectObserver ();
```

Parameters

None	

Scope

Public

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxThreadMsgSO.h>
```

Description

This method handles all cleanup for this routed Subject-Observer object. The destructor will destroy the CCxThreadMsgSink object and then relies on the base class destructor(s) for their cleanup.

Return Values

None	

See Also

ICxSubjectObserver COM interface

CCxThreadMsgSubjectObserver::CreateObserver

Prototype

```
virtual HRESULT CreateObserver();
```

Parameters

None	

Scope

Protected

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxThreadMsgSO.h>
```

Description

This method instantiates a CCxThreadMsgSink object and sets the m_pSink member variable to the address of the sink.

The method also "connects" the sink object to the ICxSubjectObserver COM object created by our CCxSubjectOnly base class.

Return Values

HRESULT	S_OK if successful, else E_FAIL

See Also

ICxSubjectObserver COM interface
CCxThreadMsgSink class

CCxWinMsgSink Class

Clients who wish to receive notifications via a Windows handle should use the CCxWinMsgSink class. For an example of a Window class that can be used to receive and handle these notifications, refer to the CCxNotificationWnd class documentation.

The CCxWinMsgSink inherits from a CCxQueuedSink, which in turn inherits from the CCxSink class. Refer to the CCxSink class to view the class hierarchy relationships between the sink classes.

Incoming notifications from the Management System arrive via the ICxObserverNotification COM interface and are placed in our queue member variable. Outgoing notifications to an Observer object are eventually sent via a call to PostMessage using the Window handle passed in our SetWindowHandle method. The outgoing notification is taken out of the queue and forwarded on to a worker thread.

The CCxWinMsgSink class only needs a valid window handle in order to post notifications to a client. For an example of a hidden window class that can be used to receive and handle these notifications, refer to the CCxNotificationWnd class documentation.

The CCxWinMsgSink class defines the following methods:

Method Name	Description
`OnNotifyNOTIFY`	Member function that is called by our base class NotificationMessage method. The implementation of OnNotifyNOTIFY delivers the final CCxNotificationInfo object to the Observer.
`OnNotifyNOTIFY_SUBJECTBROKEN`	Member function that is called by our base class NotificationMessage method. The implementation of OnNotifyNOTIFY_SUBJECTBROKEN delivers the final subject broken notification object to the Observer.
`SetWindowHandle`	Called by an Observer to provide the sink with the window handle where notifications get posted to via the PostMessage WIN32 API.
`ThreadProc`	Virtual function that implements our thread procedure. This method takes notifications out of our queue and posts them to a window handle via the PostMessage WIN32 API.

The CCxWinMsgSink class exposes the following member variables:

Name	Data Type	Scope	Description
m_hWnd	HWND	Protected	Holds the window handle that notifications are posted

			to.

CCxWinMsgSink::OnNotifyNOTIFY

Prototype

```
virtual void OnNotifyNOTIFY(CCxNotificationInfo* pNotification);
```

Parameters

CCxNotificationInfo* pNotification	[in] pointer to an actual notification object.

Scope

```
Protected
```

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxSink.h>
```

Description

This virtual method is the method that gets called for final delivery of the CCxNotificationInfo object back to the Observer. The implementation provided in the CCxWinMsgSink object posts the notification to a window handle by calling the WIN32 PostMessage API. It is the responsibility of the Observer to create the window and then call the SetWindowHandle method of our class.

Return Values

None	

See Also

SetWindowHandle

CCxWinMsgSink::OnNotifyNOTIFY_SUBJECTBROKEN

Prototype

```
virtual void OnNotifyNOTIFY_SUBJECTBROKEN(long lSubjectID);
```

Parameters

long lSubjectID	[in] ID of the Subject whose subscription is being destroyed in the Management System.

Scope

```
Protected
```

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxSink.h>
```

Description

This method gets called for final delivery of a subject broken notification object. The implementation provided in the CCxWinMsgSink object posts the notification to a window by calling the WIN32 PostMessage API. It is the responsibility of the Observer to create the window and then call the SetWindowHandle method of our class.

Return Values

None	

See Also

SetWindowHandle

CCxWinMsgSink::ThreadProc

Prototype

```
virtual DWORD ThreadProc();
```

Parameters

None	

Scope

Protected

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxSink.h>
```

Description

This member function does our thread work. The thread has a queue (CCxQueuedSink::m_qNotifications member variable) where ALL notifications from the Management System are stored. The thread simply pops a notification off its queue, and then calls the CCxQueuedSink::NotificationMessage method. The CCxQueuedSink::NotificationMessage method determines the type of notification then calls the corresponding CCxNotificationThread "OnNotify" method. This class' implementation of the "OnNotify" method simply posts the notification object be calling the PostMessage WIN32 API.

Return Values

DWORD	Always returns 0.

See Also

CCxWinMsgSink::SetWindowHandle

Prototype

```
BOOL SetWindowHandle(HWND hWnd)
```

Scope

```
Public
```

Parameters

`HWND hWnd`	Window handle containing a Windows Message handler that will handle notifications

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxSink.h>
```

Description

Call this member function to set the window handle where notifications will be forwarded. This window handle MUST be a valid HWND.

The CCxNotificationWnd class is provided as a helper class for users who would like to receive notifications in a window. Refer to the documentation on the CCxNotificationWnd class for more information.

Return Values

`BOOL`	TRUE if hWnd is a valid window handle, else FALSE

See Also

CCxNotificationWnd class

CCxWinMsgSubjectObserver Class

The CCxWinMsgSubjectObserver class…

The CCxWinMsgSubjectObserver class implements an Observer with a CCxWinMsgSink. CCxWinMsgSubjectObserver inherits from CCxQueuedSubjectObserver. To view the UML class hierarchy diagram, refer to the CCxSubjectOnly section in this document.

The CreateObserver method in this class instantiates a CCxWinMsgSink object.

The CCxWinMsgSubjectObserver class defines one of the "helper" Observer classes in the Notifications Package. These helper classes provide a pure C++ implementation of an Observer. Absolutely no COM knowledge is needed in order to create and use these derived Observer classes.

The CCxWinMsgSubjectObserver class defines the following methods:

Method Name	Description
`~CCxWinMsgSubjectObserver`	Cleans up the Observer object
`CCxWinMsgSubjectObserver`	Handles all initialization and creation for this object.
`CreateObserver`	Creates a CCxWinMsgSink object and automatically connects the object to our ICxSubjectObserver COM object.
`SetHWND`	Allows the caller to set the m_hWndDestination member variable. This HWND value is used to post all notifications.

The following member variables are exposed by the CCxWinMsgSubjectObserver class:

Name	Data Type	Scope	Description
m_hWndDestination	HWND	Protected	Holds the window handle passed to us in the constructor

CCxWinMsgSubjectObserver::CCxWinMsgSubjectObserver

Prototype

```
CCxWinMsgSubjectObserver (HWND hWnd,
            LPCTSTR pszName = NULL,
            long nPackageID = CX_PKGID_USER,
            LPCTSTR pszServer = NULL,
            long lFlags = CX_NOTIFICATION_INPROC);
```

Parameters

`HWND hWnd`	[in] Window handle where notifications are posted

	to.
`LPCTSTR pszName`	[in] Name of the Observer. Default value is NULL if the default constructor is used. If NULL is passed, then the SetMyName method MUST be used to set the name of this Observer.
`long nPackageID`	[in] Package ID of this Observer. Default value is CX_PKGID_USER if the default constructor is used. If no value is passed, the SetMyName method must be used to set the Package ID.
`LPCTSTR pszServer`	[in] Used for DCOM creation of the ICxSubjectObserver COM object. If non-NULL, the name of the server computer where the CxNotify COM server DLL is registered and available for use.
`long lFlags`	[in] Creation flags. Currently this parameter is used only to determine whether to instantiate the ICxSubjectObserver object in process or out of process. The in process flag is CX_NOTIFICATION_INPROC, the out of process flag is CX_NOTIFICATION_OUTOFPROC.

Scope

`Public`

Definition

```
Dynamic Link Library        CxNotifyClient.DLL
Header Files                #include <CxWinMsgSO.h>
```

Description

The constructor creates a Notification Package Observer and automatically instantiates the ICxSubjectObserver COM object by calling the base class CCxSubjectOnly constructor. Constructor parameters can be used to efficiently initialize the object. The name and package ID parameters replace calling the ICxSubjectObserver::SetMyName method. The lFlags parameter handles creation of the object in process or out of process as opposed to using the CoCreateInstanceEx WIN32 API. Note that by default, the COM object is created in process.

After creating the ICxSubjectObserver object in our CCxSubjectOnly base class, this method calls the CreateObserver method. CreateObserver instantiates a CCxWinMsgSink object and "connects" the sink to the ICxSubjectObserver COM object.

All notifications are sent to the window handle specified in the hWnd parameter. hWnd MUST specify a valid window handle. We use the PostMessage to send notifications to this window. To create a hidden window that can accept and handle notifications refer to the CCxNotificationWnd documentation.

Return Values

None	

See Also

ICxSubjectObserver COM interface
CCxNotificationWnd class
CCxWinMsgSink class
ICxSubjectObserver::SetMyName

CCxWinMsgSubjectObserver::~CCxWinMsgSubjectObserver

Prototype

```
virtual ~ CCxWinMsgSubjectObserver ();
```

Parameters

None	

Scope

Public

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxWinMsgSO.h>
```

Description

This method handles all cleanup for this routed Subject-Observer object. The destructor will destroy the CCxWinMsgSink object and then relies on the base class destructor(s) for their cleanup.

Return Values

None	

See Also

ICxSubjectObserver COM interface
CCxWinMsgSink class

CCxWinMsgSubjectObserver::CreateObserver

Prototype

```
virtual HRESULT CreateObserver();
```

Parameters

None	

Scope

Protected

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxWinMsgSO.h>
```

Description

This method instantiates a CCxWinMsgSink object and sets the m_pSink member variable to the address of the sink.

The method also "connects" the sink object to the ICxSubjectObserver COM object created by our CCxSubjectOnly base class.

Return Values

HRESULT	S_OK if successful, else E_FAIL

See Also

ICxSubjectObserver COM interface
CCxWinMsgSink class

CCxWinMsgSubjectObserver::SetHWND

Prototype

```
virtual void SetHWND(HWND hWnd);
```

Parameters

HWND hWnd	[in] Handle to a window that will accept and process notifications.

Scope

```
Public
```

Definition

```
Dynamic Link Library          CxNotifyClient.DLL
Header Files                  #include <CxWinMsgSO.h>
```

Description

This method sets the value of our m_hWndDestination member variable. Note that our constructor also allows the creator of this object to pass in a window handle. This method will typically be used by clients that do not have a window handle available during the creation of this object, but have a handle available at a later time.

Return Values

Void	

See Also

ICxSubjectObserver COM interface

CCxWinMsgSink class

ICxSubjectObserver COM Interface

The ICxSubjectObserver interface is the interface used by both Subjects and Observers to communicate with the Notification Management System.

The following COM Interface methods are available:

Method Name	Description
GetCountMyObservers	Gets a count of the number of Observers of this object.
GetCountMySubscriptions	Gets a count of the number of subjects this object has subscribed to.
GetError	Gets a string description of the last error to occur.
GetIDFromName	Given a name, returns the ID of the notification client.
GetMyName	Gets the name of 'this' object.
GetMyObjectID	Gets the ID of 'this' object.
GetMyPackageID	Gets the Package ID of 'this' object.
GetNameFromID	Given a client ID, gets its notification subject name.
GetSubscribedFilterMask	Retrieves a mask of all Notification types that Observers of this subject have entered.
IsSubscribed	Determines whether the object identified by dwObserverID is subscribed to 'this' object as an observer.
PostNotify	Used by subjects that send notifications and don't care when the notifications arrive.
SendNotify	Used by subjects that send notifications and don't want the call to return control to their thread until the notifications have been delivered and handled.
SetMyName	Allows a notification client to set its own name and package ID.
SubscribeByID	Notification clients who know the name of a subject or wish to use wildcard subscriptions can call this method to subscribe to one or more subjects.
SubscribeByName	Notification clients who know the name of a subject or wish to use wildcard subscriptions can call this method to subscribe to one or more subjects.
UnsubscribeByID	Notification clients who have the ID of a subject can call this method to unsubscribe to the subject.
UnsubscribeByName	Notification clients who have the name of a subject or who wish to use wildcard names can call this method to unsubscribe from one or more subjects.

The following source code example illustrates how to create an ICxSubjectObserver COM object using either In-process or Out-of-process notifications. It also illustrates the use of CoCreateInstanceEx. For information on CoCreateInstance or CoCreateInstanceEx, refer to the WIN32 Platform SDK or MSDN documentation.

```
#include <CxNotify.h>

// Create the ICxSubjectObserver object
// using CoCreateInstanceEx.  If you want to use DCOM, provide
// a valid server name
ICxSubjectObserver *pSubjectObserver = NULL;
HRESULT hr;
COSERVERINFO serverinfo;
COSERVERINFO *pServerInfo     = NULL;
CString strServer             = "";
bool bInProcess               = true;

// If using DCOM…
if( !strserverName.IsEmpty() )
{
      // Initialize the serverinfo structure
      serverinfo.dwReserved1  = 0;
      serverinfo.dwReserved2  = 0;
      serverinfo.pwszName     = strServer.AllocSysString();
      serverinfo.pAuthInfo    = NULL;
      pServerInfo             = &serverinfo;
}

// CoCreateInstanceEx allows the creation of multiple
// COM objects simultaneously.  A MULTI_QI structure is
// used to do this.  We are only creating a single object
// in this example.
MULTI_QI qi= {&IID_ICxSubjectObserver,NULL,0};

// Create the object
hr = CoCreateInstanceEx(
            CLSID_CCxSubjectObserver,
            NULL,
            bInProcess? CLSCTX_INPROC_SERVER:CLSCTX_LOCAL_SERVER,
            pServerInfo,
            1,
            &qi);

if( SUCCEEDED(qi.hr) )
{
      pSubjectObserver = (ICxSubjectObserver*) qi.pItf;
      ASSERT(pSubjectObserver);
}
else
{
      pSubjectObserver = NULL;
      ASSERT(FALSE);
      AfxMessageBox("Error creating COM object");
      return;
}
```

Note that the source code example above provides only "Subject" capabilities to the `pSubjectObserver` object. The following source code makes the object a complete Subject-Observer by creating a CCxRoutedSink object and connecting this object to pSubjectObserver. Note one very important assumption in this code: The call to SetNotificationRoute passes the 'this' pointer. This is possible because the class making this call inherits from the CCxNotificationRoute object. For more information, refer to the CCxNotificationRoute documentation.

```
// Define our own package id
#define MY_PERSONAL_PKGID (CX_PKGID_USER + 1)

// Create the sink and connect it to the subject-observer
if( pSubjectObserver )
{
      CComObject<CCxRoutedSink> *pNotifier;

      // This is the same as a CoCreateInstance.
      // This CreateInstance eventually calls
      // CxRoutedSink::FinalConstruct() which spawns
      // our watcher thread.
      hr = CComObject<CCxRoutedSink>::CreateInstance(&pNotifier);
      if (FAILED(hr))
      {
            ASSERT(FALSE);
            return;
      }

      // We can make the following call if this object
      // inherits from CxNotificationRoute.  This is what tells
      // the notification system how to get back to us
      // and call our OnNotifyNOTIFY() and
      // OnNotifyNOTIFY_SUBJECTBROKEN() methods
      pNotifier->SetNotificationRoute(this);

      // We use the ATL call to connect the ICxSubjectObserver
      // to the sink
      DWORD dwCookie = 99;
      hr = AtlAdvise(pSubjectObserver,
                  pNotifier->GetUnknown(),
                  IID_ICxObserverNotification,
                  &dwCookie);

      if (FAILED(hr))
      {
            ASSERT(FALSE);
            return;
      }

      // Finally, set the name of this object
      hr = pSubjectObserver->SetMyName(CComBSTR("WhateverName"),
                  MY_PERSONAL_PKGID);

      if (FAILED(hr))
```

```
        {
                BSTR bstrError;
                CString strMsg;

                pSubjectObserver ->GetError(&bstrError);
                strMsg.Format("Error calling SetMyName \
                            from: '%s'\n\nHRESULT: \
                            0x%x\nDescription: %s", \
                            "WhateverName",
                            hr,
                            CString(bstrError));

                AfxMessageBox(msg);
                return;
        }

}
```

ICxSubjectObserver::GetCountMyObservers

Prototype

```
HRESULT GetCountMyObservers(/*[out]*/ long *lCount);
```

Parameters

`long *lCount`	[out] Pointer to a long that receives the number of observers of this subject.

Definition

```
Dynamic Link Library          CxNotify.DLL
Header Files                  #include <Notify.h>
```

Description

Gets a count of the number of Observers of this object.

Return Values

`HRESULT`	S_OK on success
	E_FAIL if an unknown error occurs. For information about the error, call ICxSubjectObserver::GetError
	E_POINTER if the pointer passed in is invalid

Source Code Example:

```
// Assume that the ICxSubjectObserver object has been created
// and its SetMyName method has been called…

// Call the method we are illustrating
if (pSubjectObserver)
{
      long lCount = 0;

      hr = pSubjectObserver->GetCountMyObservers(&lCount);

      // Check for errors
```

```
        if (FAILED(hr))
        {
              BSTR bstrName;
              BSTR bstrError;
              CString strMsg;

              // Note that we skip the error checking on
              // the GetMyName call
              hr = pSubjectObserver->GetMyName(&bstrName);

              pSubjectObserver ->GetError(&bstrError);
              strMsg.Format("Error calling GetCountMyObservers \
                          from: '%s'\n\nHRESULT: \
                          0x%x\nDescription: %s", \
                          CString(bstrName),
                          hr, CString(bstrError));

              AfxMessageBox(msg);
              return;
        }
}
```

See Also

ICxSubjectObserver::GetError
ICxSubjectObserver::SetMyName

ICxSubjectObserver::GetCountMySubscriptions

Prototype

```
HRESULT GetCountMySubscriptions(/*[out]*/ long *lCount);
```

Parameters

long *lCount	[out] Pointer to a long that receives the number of subscriptions this object has to other subjects.

Definition

```
Dynamic Link Library          CxNotify.DLL
Header Files                  #include <Notify.h>
```

Description

Gets a count of the number of subjects this object has subscribed to.

Return Values

HRESULT	S_OK on success
	E_FAIL if an unknown error occurs. For information about the error, call ICxSubjectObserver::GetError
	E_POINTER if the pointer passed in is invalid

Source Code Example:

```
// Assume that the ICxSubjectObserver object has been created
// and its SetMyName method has been called…
```

```
// Call the method we are illustrating
if (pSubjectObserver)
{
      long lCount = 0;

      hr = pSubjectObserver->GetCountMySubscriptions(&lCount);

      // Check for errors
      if (FAILED(hr))
      {
            BSTR bstrError;
            CString strMsg;
            BSTR bstrName;

            // Note that we skip the error checking on
            // the GetMyName call
            hr = pSubjectObserver->GetMyName(&bstrName);

            pSubjectObserver ->GetError(&bstrError);
            strMsg.Format("Error calling GetCountMySubscriptions\
                        from: '%s'\n\nHRESULT: \
                        0x%x\nDescription: %s", \
                        CString(bstrName),
                        hr, CString(bstrError));

            AfxMessageBox(msg);
            return;
      }
}
```

See Also

ICxSubjectObserver::GetError

ICxSubjectObserver::GetError

Prototype

```
HRESULT GetError(/*[out, retval]*/BSTR *pbstrError);
```

Parameters

`BSTR *pbstrError`	[out] String description of the last error to occur

Definition

```
Dynamic Link Library          CxNotify.DLL
Header Files                  #include <Notify.h>
```

Description

Gets a string description of the last error to occur.

Return Values

`HRESULT`	S_OK if pbstrError was successfully filled in

	E_FAIL on error. For information about the error, call ICxSubjectObserver::GetError

Source Code Example:

```
// Assume some ICxSubjectObserver call has resulted in a
// failed HRESULT

// Check for errors
if (FAILED(hr))
{
      BSTR bstrError;
      CString strMsg;

      pSubjectObserver->GetError(&bstrError);
      strMsg.Format("Error calling an ICxSubjectObserver\
                        method\nHRESULT: \
                        0x%x\nDescription: %s", \
                        hr, CString(bstrError));

      AfxMessageBox(msg);
      return;
}
```

See Also

ICxSubjectObserver::GetIDFromName

Prototype

```
HRESULT GetIDFromName (/*[in]*/ BSTR strname,
                       /*[out]*/ long * lObjectID);
```

Parameters

`BSTR bstrName`	[in] Name of the object to search for.
`long *lObjectID`	[out] ID of the found object, or -1 if the object does not exist

Definition

```
Dynamic Link Library          CxNotify.DLL
Header Files                  #include <Notify.h>
```

Description

Given a name, return the ID of the notification client. Each Subject and Observer in the Notifications Package is automatically assigned a unique ID. The Notifications Package itself provides and maintains this unique ID.

If the passed in string name is not found, then -1 is returned in lObjectID and E_FAIL is returned. All valid ID's start at 0 and are incremented by 1 for each successfully create ICxSubjectObserver object.

Warning: There is no guarantee that the same ID will be used for an object each time your application is executed!

Return Values

HRESULT	S_OK if the name was found
	E_FAIL on error or if the name was not found. For information about the error, call ICxSubjectObserver::GetError
	E_POINTER if the lObjectID pointer passed is invalid

Source Code Example:

```
// Assume that the ICxSubjectObserver object has been created
// and its SetMyName method has been called…

// Call the method we are illustrating
if (pSubjectObserver)
{
      long lID = 0;

      hr = pSubjectObserver->GetIDFromName(CComBSTR("Joe"),
                                           &lID);

      // Check for errors
      if (FAILED(hr))
      {
            BSTR bstrError;
            CString strMsg;
            BSTR bstrName;

            // Note that we skip the error checking on
            // the GetMyName call
            hr = pSubjectObserver->GetMyName(&bstrName);

            pSubjectObserver ->GetError(&bstrError);
            strMsg.Format("Error calling GetIDFromName \
                        from: '%s'\n\nHRESULT: \
                        0x%x\nDescription: %s", \
                        CString(bstrName),
                        hr, CString(bstrError));

            AfxMessageBox(msg);
            return;
      }

      // Remember that if no ID is found for "Joe", a -1 is
      // returned in the lID parameter
      if (lID != -1)
      {
            // Do something with the ID
      }
}
```

See Also

ICxSubjectObserver::GetError

ICxSubjectObserver::GetMyName

Prototype

```
HRESULT GetMyName(/*[out, retval]*/BSTR *strName);
```

Parameters

`BSTR *pbstrName`	[out] Current name of the calling object

Definition

```
Dynamic Link Library          CxNotify.DLL
Header Files                  #include <Notify.h>
```

Description

Gets the name of 'this' object. By default every Subject and Observer in the Notifications Package is given a name based on their unique ID. So, for Subject with an ID of 100, it's name is "100". Subjects and Observers set their name by calling the ICxSubjectObserver::SetMyName method.

Return Values

`HRESULT`	S_OK on success
	E_FAIL on error. For information about the error, call ICxSubjectObserver::GetError
	E_POINTER if the pbstrName pointer passed is NULL

Source Code Example:

```
// Assume that the ICxSubjectObserver object has been created
// and its SetMyName method has been called…

// Call the method we are illustrating
if (pSubjectObserver)
{
      BSTR bstrMyName;

      hr = pSubjectObserver->GetMyName(&bstrMyName);

      // Check for errors
      if (FAILED(hr))
      {
            BSTR bstrError;
            CString strMsg;

            pSubjectObserver ->GetError(&bstrError);
            strMsg.Format("Error calling GetMyName \
                        from: '%s'\n\nHRESULT: \
                        0x%x\nDescription: %s", \
                        "Unknown name",
                        hr, CString(bstrError));

            AfxMessageBox(msg);
            return;
      }
```

```
}
```

See Also

ICxSubjectObserver::GetError
ICxSubjectObserver::SetMyName

ICxSubjectObserver::GetMyObjectID

Prototype

```
HRESULT GetMyObjectID(/*[out]*/long * lID);
```

Parameters

long *lID	[out] Gets the internal Object ID.

Definition

```
Dynamic Link Library          CxNotify.DLL
Header Files                  #include <Notify.h>
```

Description

Gets the ID of 'this' object. By default every Subject and Observer is given a unique ID by the Notifications Package. Subjects and Observers do NOT have the ability to change their ID.

Return Values

HRESULT	S_OK on success
	E_FAIL on error. For information about the error, call ICxSubjectObserver::GetError
	E_POINTER if the lID pointer passed is NULL

Source Code Example:

```
// Assume that the ICxSubjectObserver object has been created
// and its SetMyName method has been called…

// Call the method we are illustrating
if (pSubjectObserver)
{
      long lMyID;

      hr = pSubjectObserver->GetMyObjectID(&lMyID);

      // Check for errors
      if (FAILED(hr))
      {
            BSTR bstrError;
            CString strMsg;
            BSTR bstrName;

            // Note that we skip the error checking on
            // the GetMyName call
```

```
            hr = pSubjectObserver->GetMyName(&bstrName);

            pSubjectObserver ->GetError(&bstrError);
            strMsg.Format("Error calling GetMyObjectID \
                        from: '%s'\n\nHRESULT: \
                        0x%x\nDescription: %s", \
                        CString(bstrName),
                        hr, CString(bstrError));

            AfxMessageBox(msg);
            return;
      }

      // Now do something with lMyID
}
```

See Also

ICxSubjectObserver::GetError

ICxSubjectObserver::GetMyPackageID

Prototype

```
HRESULT GetMyPackageID(/*[out]*/long * lPackageID);
```

Parameters

`long *lPackageID`	[out] Gets the package ID for this object.

Definition

```
Dynamic Link Library          CxNotify.DLL
Header Files                  #include <Notify.h>
```

Description

Gets the package ID of 'this' object. The package ID was set in the call to ICxSubjectObserver::SetMyName. All package ID's must be a value of CX_PKGID_USER + n. CX_PKGID_USER is defined in the header file CxNotifyServer.h.

Return Values

`HRESULT`	S_OK on success
	E_FAIL on error. For information about the error, call ICxSubjectObserver::GetError
	E_POINTER if the lPackageID pointer passed is NULL

Source Code Example:

```
// Assume that the ICxSubjectObserver object has been created
// and its SetMyName method has been called…

// Call the method we are illustrating
if (pSubjectObserver)
```

```
{
      long lMyPackageID;

      hr = pSubjectObserver->GetMyPackageID(&lMyPackageID);

      // Check for errors
      if (FAILED(hr))
      {
            BSTR bstrError;
            CString strMsg;
            BSTR bstrName;

            // Note that we skip the error checking on
            // the GetMyName call
            hr = pSubjectObserver->GetMyName(&bstrName);

            pSubjectObserver ->GetError(&bstrError);
            strMsg.Format("Error calling GetMyPackageID \
                        from: '%s'\n\nHRESULT: \
                        0x%x\nDescription: %s", \
                        CString(bstrName),
                        hr, CString(bstrError));

            AfxMessageBox(msg);
            return;
      }

      // Now, do something with the Package ID

}
```

See Also

ICxSubjectObserver::GetError

ICxSubjectObserver::GetNameFromID

Prototype

```
HRESULT GetNameFromID(/*[in]*/ long lID,
                  /*[out, retval]*/BSTR *pbstrName)
```

Parameters

`long *lID`	[in] ID to search for
`BSTR *pbstrName`	[out] Name found for lID. NULL if the ID was not found

Definition

```
Dynamic Link Library          CxNotify.DLL
Header Files                  #include <Notify.h>
```

Description

Given an ID, gets the object's name. A Subject or Observer sets its name by calling ICxSubjectObserver::SetMyName.

Return Values

HRESULT	S_OK on success
	E_FAIL on error. For information about the error, call ICxSubjectObserver::GetError
	E_POINTER if the pbstrName pointer passed is NULL

Source Code Example:

```
// Assume that the ICxSubjectObserver object has been created
// and its SetMyName method has been called…

// Call the method we are illustrating
if (pSubjectObserver)
{
      BSTR bstrSomeName;

      // Get the name of the object with ID 5
      hr = pSubjectObserver->GetNameFromID(5,
                              &bstrSomeName);

      // Check for errors
      if (FAILED(hr))
      {
            BSTR bstrError;
            CString strMsg;
            BSTR bstrName;

            // Note that we skip the error checking on
            // the GetMyName call
            hr = pSubjectObserver->GetMyName(&bstrName);

            pSubjectObserver ->GetError(&bstrError);
            strMsg.Format("Error calling GetNameFromID \
                        from: '%s'\n\nHRESULT: \
                        0x%x\nDescription: %s", \
                        CString(bstrName),
                        hr, CString(bstrError));

            AfxMessageBox(msg);
            return;
      }

      // Now, do something with the name we found

}
```

See Also:
ICxSubjectObserver::GetError
ICxSubjectObserver::SetMyName

ICxSubjectObserver::GetSubscribedFilterMask

Prototype

```
HRESULT GetSubscribedFilterMask(/*[out]*/ long *lMask);
```

Parameters

`long *lMask`	Bit-mask return value. All filters from all subscribed Observers are AND'd (&) together and returned in lMask

Definition

```
Dynamic Link Library          CxNotify.DLL
Header Files                  #include <Notify.h>
```

Description

Retrieves a mask of all Notification types that Observers have subscribed to. The Notification types from all Observers are Bit-wise AND'd together and then returned in lMask.

Observers subscribe to a Subject by using the ICxSubjectObserver::SubscribeByName or ICxSubjectObserver::SubscribeByID methods. One of the parameters passed in these methods is the Notification type. It is the notification type value that is returned in lMask.

Return Values

`HRESULT`	S_OK on success
	E_FAIL on error. For information about the error, call ICxSubjectObserver::GetError
	E_POINTER if the lMask pointer passed is NULL

Source Code Example:

```
// Assume that the ICxSubjectObserver object has been created
// and its SetMyName method has been called…

// Call the method we are illustrating
if (pSubjectObserver)
{
      long lMask;

      // Get the name of the object with ID 5
      hr = pSubjectObserver->GetSubscribedFilterMask(&lMask);

      // Check for errors
      if (FAILED(hr))
      {
            BSTR bstrError;
            CString strMsg;
            BSTR bstrName;

            // Note that we skip the error checking on
            // the GetMyName call
            hr = pSubjectObserver->GetMyName(&bstrName);

            pSubjectObserver->GetError(&bstrError);
            strMsg.Format("Error in GetSubscribedFilterMask \
                        called from: '%s'\n\nHRESULT: \
                        0x%x\nDescription: %s", \
```

```
                        CString(bstrName),
                        hr, CString(bstrError));

            AfxMessageBox(msg);
            return;
      }

      // Now, do something with the mask we got back

}
```

See Also

ICxSubjectObserver::GetError
ICxSubjectObserver::SubscribeByName
ICxSubjectObserver::SubscribeByID

ICxSubjectObserver::IsSubscribed

Prototype

```
HRESULT IsSubscribed(long lObserverID);
```

Parameters

long lObserverID	[in] ID of a Notification package object

Definition

```
Dynamic Link Library          CxNotify.DLL
Header Files                  #include <Notify.h>
```

Description

Determines whether the object identified by lObserverID is subscribed to 'this' object as an observer.

The ICxSubjectObserver::GetIDFromName method can be used to get an Object's ID.

Return Values

HRESULT	S_OK if the Observer is subscribed
	E_FAIL if the object is NOT subscribed

Source Code Example:

```
// Assume that the ICxSubjectObserver object has been created
// and its SetMyName method has been called…

// Call the method we are illustrating
if (pSubjectObserver)
{
      long lID = 0;
```

```
        // Get the ID number of "Joe".  Note that we skip
        // error checking after this call
        hr = pSubjectObserver->GetIDFromName(CComBSTR("Joe"),
                                             &lID);

        // Get the name of the object with ID 5
        hr = pSubjectObserver->IsSubscribed(lID);

        if (hr == S_OK)
        {
              // Yes, Joe is subscribed to us
        }
        else
        {
              // No, Joe is not subscribed to us
        }

}
```

See Also

ICxSubjectObserver::PostNotify

Prototype

```
HRESULT PostNotify(/*[in]*/ long lSizeNotification,
                   /*[in]*/ byte* pNotification,
                   /*[in]*/ long lDataType,
                   /*[in]*/ long lNotificationType,
                   /*[in]*/ long lPriority,
                   /*[in]*/ long lExtra);
```

Parameters

`long lSizeNotification`	[in] The number of bytes of the pNotification buffer
`byte *pNotification`	[in] The notification data buffer. Anything can be embedded in this buffer EXCEPT embedded pointers. In other words, this must be a single contiguous piece of memory.
`long lDataType`	[in]Flag parameter specifying the type of data. The Notification Package provides some optional predefined data types, but there is no built-in dependency to the pre-defined types. Users are free to implement and use their own data types.
`long lNotificationType`	[in] Bit-mask used by the Management system. This value is bit-wise AND'd with the lNotificationFilter value that was passed during a subscription request from an Observer. If the bit-wise '&' returns TRUE, the Management System forwards the notification on to the Observer(s). Otherwise, the Management System stops the notification.
`long lPriority`	[in] The notification priority. The higher the

	number, the higher the priority.
`long lExtra`	[in] Extra data parameter that the Subject can use any way they want. Note: Pointer data must NOT be passed in this parameter.

Definition

```
Dynamic Link Library          CxNotify.DLL
Header Files                  #include <Notify.h>
```

Description

PostNotify is an asynchronous notification. PostNotify is used by Subjects that are not required to wait until a notification has been delivered to an Observer.

When a Subject calls the PostNotify method, the Subject's thread of execution is used to create a CCxNotificationInfo object and place it into the Management System's queue. Once the notification object is in the queue, the thread of the calling Subject returns immediately.

Special Note on Memory Allocation: The Notifications Package uses the C++ 'new' and 'delete' methods for all memory allocation. The creation of the CCxNotificationInfo object copies data from the 'pNotification' byte buffer and places a copy of that data in the CCxNotificationInfo object. This allows the calling process to modify or delete the pNotification buffer once it returns. If the Subject deletes or changes the pNotification buffer after returning from PostNotify, any changes made to the buffer will not affect the CCxNotificationInfo object that is passed to Observer(s). C++ Observers that receive a CCxNotificationObject are responsible for deleting that object by calling the C++ 'delete' method. An optional mechanism to delete the CCxNotificationInfo object is to call the CCxNotificationInfo::Release method.

Return Values

`HRESULT`	S_OK on success
	E_FAIL on error. For information about the error, call ICxSubjectObserver::GetError

Source Code Example:

```
// Assume that the ICxSubjectObserver object has been created
// and its SetMyName method has been called…

// Call the method we are illustrating
if (pSubjectObserver)
{
      CString strData("some text");
      byte *pData;
      pData = (byte *)strData.GetBuffer(strData.GetLength());

      // Post an asynchronous notification
      hr = pSubjectObserver->PostNotify(strData.GetLength(),
                                        pData,
                                        CX_DTYPE_CHARSTAR,
                                        -1,   // All types
```

```
                                                0,    // Priority
                                                0);   // Extra


        // Check for errors
        if (FAILED(hr))
        {
              BSTR bstrError;
              CString strMsg;
              BSTR bstrName;

              // Note that we skip the error checking on
              // the GetMyName call
              hr = pSubjectObserver->GetMyName(&bstrName);

              pSubjectObserver ->GetError(&bstrError);
              strMsg.Format("Error calling PostNotify \
                          from: '%s'\n\nHRESULT: \
                          0x%x\nDescription: %s", \
                          CString(bstrName),
                          hr, CString(bstrError));

              AfxMessageBox(msg);
              return;
        }


}
```

See Also

ICxSubjectObserver::GetError
CCxNotificationInfo::CCxNotificationInfo
CCxNotificationInfo::Release

ICxSubjectObserver::SendNotify

Prototype

```
HRESULT SendNotify(/*[in]*/ long lSizeNotification,
                   /*[in]*/ byte *pNotification,
                   /*[in]*/ long lDataType,
                   /*[in]*/ long lNotificationType,
                   /*[in]*/ long lPriority,
                   /*[in]*/ long lExtra,
                   /*[in]*/ long lTimeout);
```

Parameters

`lSizeNotification`	[in] Size (in bytes) of the data parameter pNotification
`byte *pNotification`	[in] Data buffer
`long lDataType`	[in] Flag parameter specifying the type of data. The Notification Package provides optional predefined data types, but there is no built-in

	dependency to the pre-defined types. Users are free to implement and use their own data types.
`lNotificationType`	[in] Bit-mask used by the Management system. This value is bit-wise AND'd with the lNotificationFilter value that was passed during a subscription request from an Observer. If the bit-wise '&' returns TRUE, the Management System forwards the notification on to the Observer(s). Otherwise, the Management System stops the notification.
`long lPriority`	[in] The notification priority. The higher the number, the higher the priority.
`long lExtra`	[in] Extra data parameter that the user can use any way they want. Note: Pointer data must NOT be passed in this parameter.
`long lTimeout`	Timeout value in milliseconds to wait for observer to handle notifications before returning control. To wait forever, pass the value INFINITE.

Definition

```
Dynamic Link Library        CxNotify.DLL
Header Files                #include <Notify.h>
```

Description

SendNotify is a synchronous notification. Subjects that MUST wait until a notification has been delivered to an Observer use SendNotify.

When a Subject calls the SendNotify method, the Subject's thread of execution is used to create a CCxNotificationInfo object and place it into the Management System's queue. Once the notification object is in the Management System's queue, the thread of the calling Subject is blocked until either:

1. The Management System forwards the notification to all interested Observers and all Observers either delete the notification or call the CCxNotificationInfo::SetProcessed() method.
2. The timeout value passed in lTimeout expires.

Special Note on Memory Allocation: The Notifications Package uses the C++ 'new' and 'delete' methods for all memory allocation. The creation of the CCxNotificationInfo object copies data from the 'pNotification' byte buffer and places a copy of that data in the CCxNotificationInfo object. This allows the calling process to modify or delete the pNotification buffer once it returns. If the Subject deletes or changes the pNotification buffer after returning from PostNotify, any changes made to the buffer will not affect the CCxNotificationInfo object that is passed to Observer(s). C++ Observers that receive a CCxNotificationObject are responsible for deleting that object by calling the C++ 'delete' method. An optional mechanism to delete the CCxNotificationInfo object is to call the CCxNotificationInfo::Release method.

Return Values

HRESULT	S_OK on success
	E_FAIL on error. For information about the error, call ICxSubjectObserver::GetError

Source Code Example:

```
// Assume that the ICxSubjectObserver object has been created
// and its SetMyName method has been called…

// Call the method we are illustrating
if (pSubjectObserver)
{
      CString strData("some text");
      byte *pData;
      pData = (byte *)strData.GetBuffer(strData.GetLength());

      // Send the synchronous notification
      hr = pSubjectObserver->SendNotify(strData.GetLength(),
                                        pData,
                                        CX_DTYPE_CHARSTAR,
                                        -1,   // All types
                                        0,    // Priority
                                        0,    // Extra
                                        INFINITE);


      // Check for errors
      if (FAILED(hr))
      {
            BSTR bstrError;
            CString strMsg;
            BSTR bstrName;

            // Note that we skip the error checking on
            // the GetMyName call
            hr = pSubjectObserver->GetMyName(&bstrName);

            pSubjectObserver ->GetError(&bstrError);
            strMsg.Format("Error calling SendNotify \
                        from: '%s'\n\nHRESULT: \
                        0x%x\nDescription: %s", \
                        CString(bstrName),
                        hr, CString(bstrError));

            AfxMessageBox(msg);
            return;
      }


}
```

See Also

ICxSubjectObserver::GetError
CCxNotificationInfo::Release

ICxSubjectObserver::SetMyName

Prototype

```
HRESULT SetMyName(/*[in, string]*/BSTR strName,
                  /*[in]*/ long lPackageID);
```

Parameters

`BSTR bstrName`	[in] New name of this object. Note that an objects name can change any number of times during its lifetime. Valid characters include all ASCII characters except: "^$|*+?()[]\" Special Note: As an object changes its name, it carries forward any previous Observer subscriptions that it may have had.
long lPackageID	[in] Package ID number. The package ID number must be a value of CX_PKGID_USER + n. CX_PKGID_USER is defined in CxNotifyServer.h

Definition

```
Dynamic Link Library          CxNotify.DLL
Header Files                  #include <Notify.h>
```

Description

Allows a notification client to set its own name and package ID. By default the name of an object in the Notifications Package is the ASCII equivalent of its ID. When the ICxSubjectObserver object is first created, the Notifications Package assigns a unique ID to that object. The object can never change this unique ID. The string name of the object is the ASCII representation of this ID.

Notifications Package clients are free to change their name as many times as they want. Note that if the client is a Subject, then any previously created subscriptions will be carried forward with the new name. For example, if the Subject is initially named "Bill" and 2 Observers have subscribed to "Bill", when "Bill" changes its name to "Jim", the 2 Observer subscriptions are still valid. In other words, the 2 Observers are now subscribed to "Jim". This was a design decision that could have gone either way. The benefit of carrying subscriptions forward in this manner is that Subjects can evolve and change without their Observers knowing or caring that they have evolved.

The lPackageID parameter is used to specify which package this object belongs to. Cimetrix reserves a range of packages ID's (0 through CX_PKGID_USER - 1). Clients who use the Notifications Package must define package ID's above CX_PKGID_USER. CX_PKGID_USER is #defined in CxNotifyServer.h.

Return Values

HRESULT	S_OK on success
	E_FAIL on error. For information about the error, call ICxSubjectObserver::GetError

Source Code Example:

```
// Call the method we are illustrating
if (pSubjectObserver)
{
      // Set our name and package id
      hr = pSubjectObserver->SetMyName(CComBSTR("Joe",
                                    CX_PKGID_USER + 1);


      // Check for errors
      if (FAILED(hr))
      {
            BSTR bstrError;
            CString strMsg;
            BSTR bstrName;

            pSubjectObserver ->GetError(&bstrError);
            strMsg.Format("Error calling SetMyName \
                        \nHRESULT: \
                        0x%x\nDescription: %s", \
                        hr, CString(bstrError));

            AfxMessageBox(msg);
            return;
      }

}
```

See Also

ICxSubjectObserver::GetError

ICxSubjectObserver::SubscribeByID

Prototype

```
HRESULT SubscribeByID(/*[in]*/ long lSubjectID,
                      /*[in]*/ long lFlags,
                      /*[in]*/ long lNotificationFilter);
```

Parameters

long lSubjectID	[in] Subscribe to a specific subject ID
long lFlags	[in] Bit-mask field that can be used to set special handling and Orphan creation flags. The available flags are: • **CX_STYPE_ORPHANONFAIL** - Create an orphan only if the subscription request fails • **CX_STYPE_ORPHANONSUCCESS** - Create an orphan only if the subscription request succeeds • **CX_STYPE_ORPHANALWAYS** - Create an orphan if the

	subscription succeeds or fails • **CX_STYPE_ORPHAN4EVER** - Leave as an orphan even after the subject comes alive • **CX_STYPE_REPLACE** - Replace any existing subscriptions between the requesting Observer and the Subject specified by lSubjectID These flags are defined in CxNotifyServer.h. Any combination of these flags can be bit-wise OR'd '|' together. Pass a 0 if no special orphan handling is required.
`long lNotificationFilter`	[in] Bit-mask field that is used to filter notifications. Allowable values are anything that can be bitwise AND'd with the filter value passed during a notification call (see PostNotify or SendNotify). Pass -1 (0xffffffff) to subscribe to all notification types.

Definition

```
Dynamic Link Library          CxNotify.DLL
Header Files                  #include <Notify.h>
```

Description

Notification Package Observers who wish to use a Subject's ID to request a subscription should use this method. If an Observer knows the name of a Subject, the Observer can use the ICxSubjectObserver::GetIDFromName method to retrieve a Subject's ID.

To remove a subscription, use ICxSubjectObserver::UnsubscribeByID or ICxSubjectObserver::UnsubscribeByName.

Return Values

`HRESULT`	S_OK on creation of either a successful subscription or on creation of an orphan subscription
	E_FAIL on error. For information about the error, call ICxSubjectObserver::GetError

Source Code Example:

```
// Call the method we are illustrating
if (pSubjectObserver)
{
      long lJoesID;

      // Get the ID number of "Joe".  Note that we skip
      // error checking after this call
```

```
        hr = pSubjectObserver->GetIDFromName(CComBSTR("Joe"),
                                             & lJoesID);

        if (lJoesID == -1)
              return;

        // Subscribe by ID to Joe
        hr = pSubjectObserver->SubscribeByID(lJoesID,
                    CX_STYPE_ORPHAN4EVER | CX_STYPE_ORPHANONFAIL,
                    -1);  // Get all types of notifications

        // Check for errors
        if (FAILED(hr))
        {
              BSTR bstrError;
              CString strMsg;

              pSubjectObserver ->GetError(&bstrError);
              strMsg.Format("Error calling SubscribeByID \
                          \nHRESULT: \
                          0x%x\nDescription: %s", \
                          hr, CString(bstrError));

              AfxMessageBox(msg);
              return;
        }

}
```

See Also

ICxSubjectObserver::GetError
ICxSubjectObserver::UnsubscribeByID
ICxSubjectObserver::UnsubscribeByName

ICxSubjectObserver::SubscribeByName

Prototype

```
HRESULT SubscribeByName(/*[in]*/ BSTR bstrName,
                        /*[in]*/ long lFlags,
                        /*[in]*/ long lNotificationFilter,
                        /*[out]*/ long *plSubscriptions);
```

Parameters

`BSTR bstrName`	[in] Name of the subject you wish to subscribe to. This may also be a wildcard string. For example, "abc*" will subscribe to all subjects hose names begin with "abc". For a complete explanation of the wildcard capabilities of the Notifications Package, see the "Description" section below.
`long lFlags`	[in] Bit-mask field that can be used to set

	special handling and Orphan creation flags. The available flags are: • **CX_STYPE_ORPHANONFAIL** - Create an orphan only if the subscription request fails • **CX_STYPE_ORPHANONSUCCESS** - Create an orphan only if the subscription request succeeds • **CX_STYPE_ORPHANALWAYS** - Create an orphan if the subscription succeeds or fails • **CX_STYPE_ORPHAN4EVER** - Leave as an orphan even after the subject comes alive • **CX_STYPE_REPLACE** - Replace any existing subscriptions between the requesting Observer and the Subject named by bstrName These flags are defined in CxNotifyServer.h. Any combination of these flags can be bit-wise OR'd '|' together. Pass a 0 if no special orphan handling is required.
`long lNotificationFilter`	[in] Bit-mask field that is used to filter notifications. Allowable values are anything that can be bitwise AND'd with the filter value passed during a notification call (see PostNotify or SendNotify). Pass -1 (0xffffffff) to subscribe to all notification types.
`long *plSubscriptions`	[out] Pointer to a long that receives the number of successful subscriptions

Definition

```
Dynamic Link Library          CxNotify.DLL
Header Files                  #include <Notify.h>
```

Description

Notification clients who know the name of a subject or wish to use wildcard subscriptions can call this method to subscribe to one or more subjects.

The following examples illustrate the use of wildcard subscriptions:

- "*" - Subscribe to any Subject
- "Security*" - 'Securityabc', 'SecurityDEF', etc.
- "Security[1-5]" - Security1, Security2, … , Security5
- "Security?" - 'Security' followed any character
- "Security$" - 'MySecurity', 'YourSecurity', etc.
- "Security[a | f]" - 'Securitya' or 'Securityf'

Nested expressions can also be used. For example:

- "Security(([1-3])[a | b])" - Security1a, Security1b, … , Security3a, Security 3b would be matching names.

To remove a subscription, use ICxSubjectObserver::UnsubscribeByID or ICxSubjectObserver::UnsubscribeByName.

Return Values

HRESULT	S_OK on success
	E_FAIL on error. For information about the error, call ICxSubjectObserver::GetError

Source Code Example:

```
// Call the method we are illustrating
if (pSubjectObserver)
{
      long lCount = 0;

      // Subscribe by Name to everyone whose name
      // starts with "Joe"
      hr = pSubjectObserver->SubscribeByName(CComBSTR("Joe*"),
                  CX_STYPE_ORPHAN4EVER | CX_STYPE_ORPHANONFAIL,
                  -1,   // Get all types of notifications
                  &lCount);


      // Check for errors
      if (FAILED(hr))
      {
            BSTR bstrError;
            CString strMsg;

            pSubjectObserver ->GetError(&bstrError);
            strMsg.Format("Error calling SubscribeByName \
                        \nHRESULT: \
                        0x%x\nDescription: %s", \
                        hr, CString(bstrError));

            AfxMessageBox(msg);
            return;
      }

}
```

See Also

ICxSubjectObserver::GetError
ICxSubjectObserver::UnsubscribeByID
ICxSubjectObserver::UnsubscribeByName

ICxSubjectObserver::UnsubscribeByID

Prototype

```
HRESULT UnsubscribeByID(/*[in]*/long lSubjectID);
```

Parameters

long lSubjectID	[in] Unsubscribe from this subject only

Definition

```
Dynamic Link Library          CxNotify.DLL
Header Files                  #include <Notify.h>
```

Description

Notification clients who have the ID of a subject can call this method to unsubscribe to the subject.

Note that if multiple subscriptions have been created between the calling Observer and the Subject specified by lID, ALL subscriptions are removed.

Return Values

HRESULT	S_OK on success
	E_FAIL on error. For information about the error, call ICxSubjectObserver::GetError

Source Code Example:

```
// Call the method we are illustrating
if (pSubjectObserver)
{
      long lJoesID;

      // Get the ID number of "Joe".  Note that we skip
      // error checking after this call
      hr = pSubjectObserver->GetIDFromName(CComBSTR("Joe"),
                                           & lJoesID);

      if (lJoesID == -1)
            return;

      // Unsubscribe to Joe
      hr = pSubjectObserver->UnsubscribeByID(lJoesID);,


      // Check for errors
      if (FAILED(hr))
      {
            BSTR bstrError;
            CString strMsg;

            pSubjectObserver ->GetError(&bstrError);
            strMsg.Format("Error calling UnsubscribeByID \
                        \nHRESULT: \
                        0x%x\nDescription: %s", \
                        hr, CString(bstrError));

            AfxMessageBox(msg);
            return;
      }
```

```
}
```

See Also

ICxSubjectObserver::GetError
ICxSubjectObserver::SubscribeByID
ICxSubjectObserver::SubscribeByName

ICxSubjectObserver::UnsubscribeByName

Prototype

```
HRESULT UnsubscribeByName(/*[in]*/ BSTR bstrName);
```

Parameters

`BSTR bstrName`	[in] Subject name or wildcard.

Definition

```
Dynamic Link Library          CxNotify.DLL
Header Files                  #include <Notify.h>
```

Description

Notification clients who have the name of a subject or who wish to use wildcard names can call this method to unsubscribe from one or more subjects.

Note that if multiple subscriptions have been created between the calling Observer and the Subject specified by bstrName, ALL subscriptions are removed. When wildcards are used in bstrName, ALL subscriptions are removed for each matching name.

For a detailed description of wildcards, refer to ICxSubjectObserver::SubscribeByName.

Return Values

`HRESULT`	S_OK on success
	E_FAIL on error. For information about the error, call ICxSubjectObserver::GetError. Note that if there are no Subjects matching the passed in name, E_FAIL will be returned.

Source Code Example:

```
// Call the method we are illustrating
if (pSubjectObserver)
{
      long lCount = 0;

      // Unsubscribe from everyone whose name starts with "Joe"
      hr = pSubjectObserver->UnsubscribeByName(CComBSTR("Joe*"));
```

```
        // Check for errors
        if (FAILED(hr))
        {
              BSTR bstrError;
              CString strMsg;

              pSubjectObserver ->GetError(&bstrError);
              strMsg.Format("Error calling UnsubscribeByName \
                          \nHRESULT: \
                          0x%x\nDescription: %s", \
                          hr, CString(bstrError));

              AfxMessageBox(msg);
              return;
        }

}
```

See Also

ICxSubjectObserver::GetError
ICxSubjectObserver::SubscribeByID
ICxSubjectObserver::SubscribeByName

ICxObserverNotification COM Interface

The ICxObserverNotifcation COM interface is a source interface that must be implemented by a sink object. All sink objects provided by the Notifications Package expose an implementation of this COM interface.

The Notifications Package Management System uses the ICxObserverNotification COM interface to forward a notification from a Subject to the intended Observer(s). Each Observer has a sink that is used solely for the purpose of receiving these notifications.

The ICxObserverNotification interface is extremely simple. The following table defines the methods:

Method Name	Description
OnNotify	Called by the Management System to deliver a notification to an Observer.
OnNotifySubjectBroken	Called by the Management System when a subscription is being removed

ICxObserverNotification::OnNotify

Prototype

```
HRESULT OnNotify(long lSize,
                 byte* pObject,
                 long lBytes,
                 byte *pBytes,
                 VARIANT *pReturn);
```

Parameters

long lSize	[in] size of the entire original CCxNotificationInfo object.
byte* pObject	[in] a pointer to the original CCxNotificationInfo object.
long lBytes	[in] size of the byte buffer in the original CCxNotificationInfo object.
byte *pBytes,	[in] a pointer to the original CCxNotificationInfo notification data buffer.
VARIANT *pReturn	[out] - used to return the processed HANDLE back to the caller. This HANDLE is used solely for synchronous notifications.

Definition

```
Dynamic Link Library      CxNotify.DLL defines this interface
Header Files              #include <Notify.h> // Definition only
```

Description

This method is called by the Notifications Package Management System to deliver a notification to an Observer.

The implementation of this method actually "glues" a CCxNotificationInfo object back together. Because the Notifications Package support both in-process and out-of-process notifications, it was necessary to send the entire CCxNotificationInfo object separately from it's embedded byte buffer containing the notifications data. This is necessary because COM will not automatically pass an object across an application boundary if the object contains an embedded pointer to some other data.

In short, this method takes a CCxNotificationInfo object pointer and embeds a marshaled (by COM) pointer back into the CCxNotificationInfo object.

Return Values

HRESULT	S_OK on success
	E_FAIL on error.

See Also

CCxSink class, CCxQueuedSink class, CCxRoutedSink class, CCxWinMsgSink class, CCxThreadMsgSink class

ICxObserverNotification::OnNotifySubjectBroken

Prototype

```
HRESULT OnNotifySubjectBroken(long lSubjectID);
```

Parameters

long lSubjectID	[in] ID of the Subject who this Observer is no longer subscribed to.

Definition

```
Dynamic Link Library      CxNotify.DLL defines this interface
Header Files              #include <Notify.h> // Definition only
```

Description

This method is called by the Notifications Package Management System to notify an Observer that it is no longer subscribed to the Subject whose ID is passed as lSubjectID.

Return Values

HRESULT	S_OK on success
	E_FAIL on error.

See Also

CCxSink class, CCxQueuedSink class, CCxRoutedSink class, CCxWinMsgSink class, CCxThreadMsgSink class

Appendix 1: How to add notifications to an MFC application

The information in this article applies to:

- Cimetrix Notifications Package in CODE 99

Summary

The Cimetrix CIMAppObjects are installed as part of CODE 99. The Notifications Package is one of the packages that make up the CIMAppObjects.

The Notifications Package consists of three dynamic-link libraries, (DLLs), numerous header files, and complete sample applications with source code. Each deliverable is as follows:

- CxNotify.dll – COM server DLL that provides Notifications Package COM interfaces. This DLLs can be instantiated either as an in-process or out-of-process COM server (see below). Installed to %ROBTOP%\bin.
- CxNotifySupport.dll – A common DLL that is needed by both the CxNotify DLL and the CxNotifyClient.dll. This DLL exposes public classes and data structures that are needed and used by both DLLs. Installed to %ROBTOP%\bin.
- CxNotifyClient.dll – The notifications client DLL contains a number of helper classes that can be used by Notification Package clients. Installed to %ROBTOP%\bin.
- Header files – The CxNotifySupport and CxNotifyClient DLLs provide header files that expose all public helper classes that can be used by a client. Installed to %ROBTOP%\include.
- Sample Applications with source code – Currently there are two sample applications shipped with the Notifications Package: MultiNotify and MFCSample. MultiNotify is full blown comprehensive application illustrating ALL of the Notifications Package capabilities. MFCSample is a very simplistic sample application showing only the minimal features of the Notifications Package. The sample applications are installed to %ROBTOP%\examples\notifications.

Note that the %ROBTOP% environment variable is set during the installation of CODE 99. The default location is c:\Program Files\CODE99.

More Information

The Cimetrix Notifications Package can by used by any MFC client application. The client application may be a DLL or an executable. The following steps can be used by a client application to take advantage of the Notifications Package features and capabilities:

Create an MFC Application

To use the Notifications Package in an MFC application, you must first have an MFC application. To use the MFC AppWizard to generate your application from scratch, perform the following steps:

1. Start Microsoft Visual C++
2. Select **File** + **New** to create a new MFC application.
3. Select the **Projects** tab from the **New** dialog box.
4. Select the **MFC AppWizard (DLL)** or **MFC AppWizard (EXE)** project type.
5. Fill in all of the required project fields on the **Projects** tabbed dialog, then press **OK** to continue.
6. Select the appropriate application in the **MFC AppWizard – Step 1** dialog, then press **Next** to continue.
7. Watch for the **Automation** check box, as you proceed through each AppWizard dialog.

- If the new application will be single-threaded, make sure to check the **Automation** check box before proceeding. Selecting the **Automation** check box forces the wizard to add single-threaded COM initialization code to your application's CWinApp derived class's InitInstance() method.
- If you want your MFC application to be multi-threaded, follow the steps outlined in the *Initialize COM Libraries* section below.

You are now ready to begin adding code to the application for the Notifications Package.

Initialize COM libraries

For single-threaded applications created using the MFC AppWizard that selected the Automation checkbox:

Feel free to skip this step. The MFC AppWizard will have already added the correct single threaded COM initialization code to your CWinApp derived class's InitInstance() method.

For existing single-threaded applications that did not select the Automation checkbox during the AppWizard session:

Add the following code to your CWinApp derived class's InitInstance() method:

```
// Initialize single threaded COM libraries
if (!AfxOleInit())
{
      AfxMessageBox("COM initialization failed.");
      return FALSE;
}
```

For existing MFC applications that spawn and contain multiple threads:

Add the following code to your CWinApp derived class's InitInstance() method:

```
// Initialize COM for multiple threads
HRESULT hRes = CoInitializeEx(NULL,COINIT_MULTITHREADED);
if (FAILED(hRes))
{
      AfxMessageBox("CoInitializeEx() error" );
      return FALSE;
}
```

Because CoInitializeEx() is defined in objbase.h, you must add the following code to your stdafx.h file:

```
#include <objbase.h>
```

To successfully call CoInitializeEx, you must modify your MFC application's preprocessor build settings and define the following value for all build configurations:

```
_WIN32_DCOM
```

If you do not define `_WIN32_DCOM`, you will receive compiler errors for CoInitializeEx().

Now, compile your application to verify that all settings are correct.

Add Notifications Package Support

There are two options for adding Notifications Package support to an MFC application:

- Using the CIMAppObject Developer Studio Addin
- Manually editing source code.

Option 1: Using the CIMAppObject Developer Studio Addin – For a detailed explanation on the working of this Addin, refer to Article ID: QX10001, *HOWTO: Use the CIMAppObject Developer Studio Addin Toolbar*. By using the addin, you can be guaranteed that all necessary files will be automatically updated. The HOWTO article explains how to use the "Add Cimetrix CIMAppObject Support" button that is part of the Addin. This is the best option for adding Notifications Package support to your application.

Follow these steps to use the Addin:

1. Press the **Add Cimetrix CIMAppObject Support** button on your toolbar. Note that if you have not enabled the CIMAppObject Developer Studio Addin, you will need to follow the directions provided in article ID Q1000 "*HOWTO: Use the CIMAppObject Developer Studio Addin Toolbar*".
2. Select the **Notifications** checkbox in the popup dialog.
3. Verify that the **Add to Project**, **Header File**, and **C++ File** values are all correct in the popup dialog. If they are not correct, use the controls on the popup dialog to fill in the correct values.
4. Press the **OK** button to add Notifications Package support to your application.

Option 2: Manually editing source code – Two files must be modified for the MFC application to support the Notifications Package.
Follow these steps:

1. Open the **stdafx.h** header file for your MFC application. At the end of the file, add the following code:
```
#include <CxNotify.h>       // Notification Package
#include <ObjBase.h>        // COM Calls - CoCreateInstanceEx
```
2. Open the **stdafx.cpp** file for your MFC application. At the end of the file, add the following code
```
#include <Notify_i.c>       // Notification Package IID's/CLSID's
```
3. Save all files and re-compile your application to verify that everything compiles correctly. You are now ready to use the classes and COM objects provided by the Notifications Package.

Adding a Notifications Package Subject

According to the Notifications Package definitions, Subjects only send notifications. They do not receive notifications. For details on Notifications Package terminology, refer to the *Cimetrix Notifications Package White Paper*. This white paper is installed with the Cimetrix CIMAppObjects Notification Package.

There are two options for adding a Notifications Package Subject to your MFC application:

1. Using the CIMAppObject Developer Studio Addin
2. Manually modifying source code

Option 1: Using the CIMAppObject Developer Studio Addin
Follow these steps:

1. Open the header file that defines the C++ class that you want to add the Notifications Package Subject to.
2. Press the **Add a CIMAppObject Member Variable** button on the CIMAppObject Developer Studio Addin toolbar. A dialog will pop up allowing you to select the appropriate package and class.
3. Select **Notifications** on the popup dialog using the CIMAppObject Packages list box. The Select a class list box now contains a number of available Notifications Package classes.
4. Select the **CCxSubjectOnly** class on the popup dialog and from the Select a class list box.
5. Type a member variable name in the popup dialog and in the **Name** field. Note that you can member variable pointers by putting an asterisk before the member variable name. For example, typing *** m_pSubject** for the member variable name defines a pointer to a CcxSubjectOnly object.
6. Verify that the header file name and the class name is correct on the popup dialog. If not, select the appropriate filename and class using the controls on the popup dialog.
7. Select the appropriate access privilege (public, protected, or private) On the popup dialog.
8. Press the **OK** button to add the member variable.
9. In the constructor of the class where the member variable was declared (or any other appropriate place), you will need to allocate the object. Add the following code:

```
m_pSubject = new CCxSubjectOnly; // Uses the default constructor
```

Option 2: Manually modifying source code
Follow these steps:

1. Open the appropriate header file you wish to add the member variable to.
2. Locate a public, protected, or private section in the class definition.
3. Type the following in the header file

```
CcxSubjectOnly *m_pSubject;      // Declares a subject pointer
```

4. In the constructor of the class where the member variable was declared (or any other appropriate place), you will need to allocate the object. Add the following code:

```
m_pSubject = new CCxSubjectOnly; // Uses the default constructor
```

Optionally, the CCxSubjectOnly class can be passed the following parameters in its constructor:

LPCTSTR pszName – Name of this subject
Long lPackageID – PackageID for this subject
LPCTSTR pszServer – DCOM server used to instantiate object
Long lReserved – reserved parameter

For details on available constructor parameters and CCxSubjectOnly methods, refer to the on-line IDE Ready help.

Adding a Notifications Package Subject-Observer

According to the Notifications Package definitions, any Subject-Observer can send or receive notifications. When sending a notification, the object is using its "Subject" abilities. When receiving a notification, the object is using its "Observer" abilities. By definition, all Observers are also a Subject. In other words, to create an Observer, you must really create a Subject-Observer.

For details on Notifications Package terminology refer to the *Cimetrix Notifications Package White Paper*. This white paper is installed with the Cimetrix CIMAppObjects Notification Package and is installed as part of CODE99.

There are two options for adding a Subject-Observer to your MFC application:

1. Using the CIMAppObject Developer Studio Addin
2. Manually by modifying source code

Option 1: Using the CIMAppObject Developer Studio Addin
Follow these steps:

1. Open the header file that defines the C++ class that you want to add the Notifications Package Observer to.
2. Press the **Add a CIMAppObject Member Variable** button on the CIMAppObject Developer studio addin toolbar. A dialog will pop up allowing you to select the appropriate package and class.
3. Select **Notifications** on the popup dialog and using the **CIMAppObject Packages** list box. The **Select a class** list box now contains a number of available Notifications Package classes.
4. Select the **CCxRoutedSubjectObserver** class in the popup dialog using the **Select a class** list box.
5. Type a member variable name in the popup dialog and in the **Name** field. Note that you can create member variable pointers by putting an asterisk before the member variable name. For example, typing *** m_pRoutedSO** for the name defines a pointer to a CcxRoutedSubjectObserver object.
6. Verify that the header file name and the class name on the popup dialog is correct. If not, select the appropriate filename and class on the dialog.
7. Select the appropriate access privilege (public, protected, or private) on the popup dialog.
8. Press the **OK** button to add the member variable.
9. Next, skip to step #4, in Option 2, below. This explains how to correctly instantiate the object in your constructor.

Option 2: Manually modifying source code
Follow these steps:

1. Open the appropriate header file you to which you want to add the member variable.
2. Locate a public, protected, or private section in the class definition.
3. Type the following in the header file:
```
CcxRoutedSubjectObserver *m_pRoutedSO; // Declares a pointer
```
4. In the constructor of the class where the member variable is declared (or any other appropriate place), you will need to allocate the object. Add the following code:
```
m_pRoutedSO = new CcxRoutedSubjectObserver(this);
```

Optionally, the CCxRoutedSubjectObserver class can be passed the following parameters in its constructor:

LPCTSTR pszName – Name of this subject-observer
long lPackageID – PackageID for this subject-observer
LPCTSTR pszServer – DCOM server used to instantiate object
long lReserved – reserved parameter

Note that we are not passing in the name of the object. We can either pass it as a parameter in the constructor or we can use the object's SetMyName method later. The following section, *Send and Receive a Notification*, explains how to use the SetMyName method.

For details on available constructor parameters and CCxRoutedSubjectObserver methods, refer to the on-line IDE Ready help.

5. Note that the "this" pointer is passed into the CCxRoutedSubjectObserver object. For the allocation to work correctly, the "this" pointer **must** inherit from the CCxNotificationRoute class. Open the header file of the class responsible for allocating the CcxRoutedSubjectObserver and add the **public CcxNotificationRoute** code to your class declaration. For example:

```
class xxxx : public yyyy, public CcxNotificationRoute
{
      …
}
```

6. The CCxNotificationRoute class defines two virtual methods. Note that these are **not** pure virtual methods. There are default implementations that essentially do nothing. One method is for receiving notifications from a Subject and the other method is for receiving a notification when a Subject is destroyed. Currently, we are only interested in receiving notifications from a Subject. To receive the notification data, add the following declaration to the same class that inherits from CcxNotificationRoute

```
// CCxNotificationRoute virtual methods
void OnNotifyNOTIFY(CcxNotificationInfo* pNotification);
```

7. Add the following code in the CPP file for the class that inherits from CCxNotificationRoute:

```
void CYourClass::OnNotifyNOTIFY(CcxNotificationInfo* pNotification)
{
   CString s;
   if (pNotification)
   {
         // Assume string data – Note we could check the data type
         // if we wanted to
         for(int i = 0; i < pNotification->Size(); i++ )
               s += *(pNotification->Bytes() + i);

         // Now do something with the data
         // TODO
         MessageBeep(100);

         // The Observer is responsible for deleting this data now
         delete pNotification;
   }
}
```

8. Save all files and recompile the application.

The application now contains a fully functional Subject and a fully functional Subject-Observer. You are now ready to wire them together and then send and receive a notification.

Send and Receive a Notification

To send and receive a notification, the following must be true and available to clients of the Notifications Package:

- At least one subject must exist. The Subject may optionally define its own name to make it easier for Observers to "connect" or Subscribe to it.
- At least one Observer must exist.
- The Observer must create a Subscription to a Subject.

In the preceding steps, we created a Subject and a routed Subject-Observer. In our example we didn't have the Subject set its own name, although we could have provided a name string in the constructor of the CCxSubjectOnly object. Because we didn't define the Subject name, let's do so now. Add the following code after the statements where we allocated the CCxSubjectOnly object as follows:

```
// Have the Subject set its name for potential Observers
m_pSubject->SetMyName(CComBSTR("test"), CX_PKGID_USER + 1);
```

This code sets the subject's name to **test**. Observers can now use the name, **test,** to create a subscription. The CX_PKGID_USER value is provided by the Notifications Package. It defines a number that is the lowest possible package ID value. Cimetrix reserves all package IDs up to the CX_PKGID_USER value.

The package ID parameter allows Subjects to be grouped by package. For more details on the package ID parameter and its capabilities, refer to the *Notifications Package White Paper* or the Notifications Package reference documentation.

Now that we have a named subject, we need our Observer to subscribe to this subject. In the code following our allocation of the m_pRoutedSO member variable, add the following:

```
// Create a subscription to "test"
HRESULT hr;
long lCount = 0;
hr = m_pRoutedSO->SubscribeByName(CcomBSTR("test"),
            CX_STYPE_ORPHANONFAIL | CX_STYPE_ORPHAN4EVER,
            -1,
            &lCount);

if (FAILED(hr))
{
     BSTR bstrError;
     char msg[1024];
     m_pSubject->GetError(&bstrError);
     sprintf(msg, "Subscribe Error: Unable to subscribe to
'test'.\n\nHRESULT: 0x%x\nDescription: %s", hr, CString(bstrError));
     AfxMessageBox(msg);
}
```

The parameters passed to SubscribeByName are as follows:

- **test** – this is the name of the Subject we want to subscribe to.
- **CX_STYPE_ORPHANONFAIL** – this is a special orphan handling flags. Orphans are important because what if the m_pRoutedSO object has been instantiated, but the test Subject has not yet been instantiated? If the test Subject has not been created at the time the subscription request is made, then the subscription request would fail. Orphan flags allow the Observer to say, "I don't care if the Subject is alive now. When the Subject becomes alive, just hook us up," (i.e., "I want to subscribe to the Subject)).
- **-1** – The minus one signifies that this Observer is interested in any kind of notification that is sent by the "test" Subject.
- **lCount** – lCount will contain the number of successful subscriptions that were created. Note that this count could be zero if the "test" Subject is instantiated sometime after the Observer is instantiated.

Sending a Notification

Now, we are ready to send a notification. To make sending a notification easy and since we have assumed that the notification is a text string (see the preceding *Adding a Notifications Package Subject-Observer* section where we provided the OnNotifyNOTIFY method), add a button and an OnClick method handler to your MFC application's main window. How and where you create your button depends on the kind of MFC application you have created. For details on how to create buttons and add button click handlers, refer to the MSDN on-line documentation.

Once you have created the button and the button handler's OnClick method, add the following code to the OnClick method:

```
      // Let's use some 'hard-coded' text for now.  Later, you
      // could add an edit control that accepted text.  Then, you
      // could send the edit control's text as the notification
      CString s("some text");
      HRESULT hr = m_pSubject->PostNotify(s.GetLength(),// Size
                  (unsigned char*)s.GetBuffer(0),       // Data
                  CX_DATA_STRING,// Data type
                  CX_NTYPE_ALL, // Notification Filter
                  0,            // priority
                  TEST_EXTRA_DATA);// long lExtra parameter

      if (FAILED(hr))
      {
            char msg[1024];
            BSTR bstrError;
            m_pSubject->GetError(&bstrError);
            sprintf(msg, "Notification Error: HRESULT =
0x%x\nDescription: '%s'", hr, CString(bstrError));
            AfxMessageBox(msg);
      }
```

Save all files and rebuild the entire application. Now you have a fully functional MFC application that can send and receive notifications.

Testing Notifications Package Functionality

To test the Notifications Package functionality, place a breakpoint in the `OnNotifyNOTIFY()` method we created in the previous section. Each time you press the notification button in your MFC application, the breakpoint should be triggered. You should be able to use the debugger's watch window on the pNotification object that is passed into `OnNotifyNOTIFY()` and verify that you are receiving the `"some text"` string. Also, because we added the MessageBeep() function in our OnNotifyNOTIFY method, you will hear a beep each time the application receives the notification.

For additional information on the Notifications Package, refer to the following articles:

```
ARTICLE-ID: QX10001
TITLE     : HOWTO: Use the CIMAppObject Developer Studio Addin
            Toolbar

ARTICLE-ID: WP10004
TITLE     : Cimetrix Notifications Package White Paper
```

For source code sample applications, refer to the MultiNotify, SingleNotify, and StressNotifications example applications provided in the %ROBTOP%/examples/Notifications directory.

Index