TM

OPC

OLE for Process Control

OPC Alarms and Events

JUNE 2, 1999

Version 1.01

Specification Type	Industry Standard Specification		
Title:	**OPC Alarms and Events**	Date:	June 2, 1999
Version:	1.01	Soft	MS-Word
		Source:	OPCAE101_cust .doc
Author:	Opc Foundation	Status:	**Release 1.01**

Synopsis:

This specification is the specification of the interface for developers of OPC clients and OPC servers.. The specification is a result of an analysis and design process to develop a standard interface to facilitate the development of servers and clients by multiple vendors that shall inter-operate seamlessly together.

Trademarks:

Required Runtime Environment:

This specification requires Windows 95 Windows NT 4.0 or later

Revision 1.01 Highlights

This revision includes minor additions to the IDL in the form of "reserved" words added to various structures. Adding these words to pad the structures insures that the structures will give the same result regardless of the 'packing' value used by the compiler and that clients and servers compiled with different packing values will be compatible. If you have used any packing value other than the default packing value of '8' then you should rebuild and relink your applications to insure they are compatible with other OPC Alarms and Events applications.

Table of Contents

1. Introduction

1.1 Background

Today with the level of automation that is being applied in manufacturing, operators are dealing with higher and higher amounts of information. Alarming and event subsystems have been used to indicate areas of the process that require immediate attention. Areas of interest include (but are not limited to); safety limits of equipment, event detection, abnormal situations. In addition to operators, other client applications may collect and record alarm and event information for later audit or correlation with other historical data.

Alarm and event engines today produce an added stream of information that must be distributed to users and software clients that are interested in this information. Currently most alarming/event systems use their own proprietary interfaces for dissemination and collection of data. There is no capability to augment existing alarm solutions with other capabilities in a plug-n-play environment. This requires the developer to recreate the same infrastructure for their products as all other vendors have had to develop independently with no interoperability with any other systems.

In keeping with the desire to integrate data at all levels of a business (as was stated in the OPC Data background information), alarm information can be considered to be another type of data. This information is a valuable component of the information architecture outlined in the OPC Data specification.

Manufacturers and consumers want to use off the shelf, open solutions from vendors that offer superior value that solves a specific need or problem.

1.2 Purpose

To identify interfaces used to pass alarm and event information between components which would be suitable to standardization. Additionally this document details the design of those interfaces in such a way as to compliment the existing OPC Data Access Interfaces.

1.3 Relationship to Other OPC Specifications

This specification complements but is separate from the OPC Data Access and the OPC Historical Data Access specifications. It references the OPC Common specification, in that OPC Event Servers support the interfaces specified there.

1.4 Scope

1.4.1 General

The scope of this document is to provide a specification for a software “conduit” for alarm and event information to be broadcast from servers to clients. “Conduit” refers to the notion that this document is not intended to specify solutions for alarming problems, but rather provide an enabling technology that will permit multi-vendor solutions to operate in a heterogeneous computing environment.

1.4.2 Multiple Levels of Capability

The OPC Alarms and Event specification accommodates a variety of applications that need to share alarm and event information. In particular, there are multiple levels of capability for handling alarm and event functionality, from the simple to the sophisticated.

1.4.2.1 Types of Alarm and Event Servers

There are several types of OPC Alarm and Event Servers. Some key types supported by this specification are:

- Components that can detect alarms and/or events and report them to one or more clients.
- Components that can collect alarm and event information from multiple sources (whether by subscribing to other OPC alarm and event servers or by detecting alarms and events on it's own) and report such information to one or more clients.

Distinctions are made between these two roles because this specification does not overburden simple alarm and event servers, but also facilitates more sophisticated servers. Simpler software components or devices that can detect and report alarms and events, should not have to also perform advanced sorting or filtering operations. In other words, the required server interface is kept simple. It supports the reporting of information but not much more.

Thus, simple event servers may choose to restrict the functionality of the event filtering they provide. Also, they may choose to not implement such functions as area browsing, enabling/disabling of conditions, and translation to itemIDs.

Optional objects and interfaces are noted in the reference portion of this specification. Similarly, methods which may return E_NOTIMPL, or which may have varying levels of functionality are also noted.

1.4.2.2 Types of Alarm and Event Clients

Clients for OPC alarm and event servers are typically components that subscribe to and display, process, collect and/or log alarm and event information. The clients of OPC alarms and events servers may include (but are not limited to) :

- operator stations
- event/alarm logging components
- event/alarm management subsystems

1.4.2.3 Client – Server Interactions

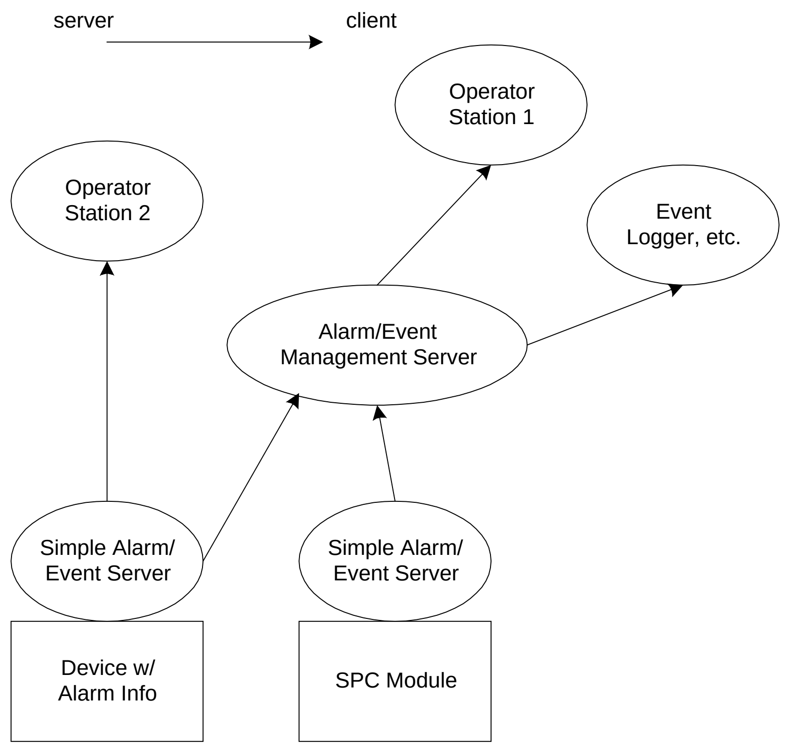


Figure 1-1. Interaction between several OPC Alarm and Event Servers and Clients

Figure 1-1 shows several types of OPC Alarm and Event clients and servers including a Device, SPC Module, Operator Stations, Event Logger, and an Alarm/Event Management subsystem. The arrowhead end of the lines connecting the components indicate the client side of the connection. Notice that there are multiple roles played by some components. The Alarm/Event Management server is also a client to more than one OPC Alarm and Event server. In this model, the Alarm/Event Management server is acting as kind of a collector or data concentrator, providing its clients with perhaps more organized information or a more advanced interface. Unlike the Alarm/Event Management server, the Device and SPC Modules implement the simplest Alarm/Event server interface.

1.5 References

- OPC Data Access Custom Interface Standard, Version 2.0 (Release Candidate 1), OPC Task force, January 8, 1998.
- The Component Object Model Specification, Version 0.9, Microsoft Corporation, (available from Microsoft's FTP site), October 24, 1995.

1.6 Audience

This document is intended to be used as reference material for developers of OPC compliant alarm clients and servers. It is assumed that the reader is familiar with Microsoft OLE/COM technology, the needs of the process control industry and the OPC Data Access 2.0 specification.

1.7 Deliverables

This document covers the analysis and design for a COM compliant custom interface. A separate document describes a related OLE Automation interface.

2. Fundamental Concepts

2.1 Overview

This specification describes objects and interfaces which are implemented by OPC Event Servers, and which provide the mechanisms for OPC Clients to be notified of the occurrence of specified events and alarm conditions. These interfaces also provide services which allow OPC Clients to determine the events and conditions supported by an OPC Event Server, and to obtain their current status.

This specification deals with entities commonly referred to in the process control industry as *alarms* and *events*. In informal conversation, the terms *alarm* and *event* are often used interchangeably and their meanings are not distinct.

Within this specification, an *alarm* is an abnormal *condition* and is thus a special case of a *condition*. A *condition* is a named state of the OPC Event Server, or of one of its contained objects, which is of interest to its OPC Clients. For example, the tag FIC101 may have the “LevelAlarm” or “DeviationAlarm” conditions associated with it.

Furthermore, a condition may be defined (optionally) to include multiple *sub-conditions*. For example, a LevelAlarm condition may include the “HighAlarm”, “HighHighAlarm”, “LowAlarm”, and “LowLowAlarm” sub-conditions[1].

On the other hand, an *event* is a detectable occurrence which is of significance to the OPC Event Server, the device it represents, and its OPC Clients. An event may or may not be associated with a condition. For example, the transitions into the LevelAlarm condition and the return to normal are events which are associated with conditions. However, operator actions, system configuration changes, and system errors are examples of events which are not related to specific conditions. OPC Clients may subscribe to be notified of the occurrence of specified events.

2.2 OPC Event Servers

Any COM object which implements the IOPCEventServer interface is an OPC Event Server.

The IOPCEventServer interface provides methods enabling the OPC Client to:

- Determine the types of events which the OPC Event Server supports.
- Enter subscriptions to specified events, so that OPC Clients can receive notifications of their occurrences.
- Specify a client callback interface to be invoked if the OPC Event Server is shutting down.

2.3 Areas

The expectation is that events and conditions available in the server are organized within one or more process areas. An area is a grouping of plant equipment configured by the user, typically according to areas of operator responsibility. The definition of the area configuration is outside the scope of this specification. Implementation of the area concept is optional.

If areas are available, an OPCEventAreaBrowser object may be created by the client to browse the process area organization. The client can filter event subscriptions by specifying the process areas to limit the event notifications sent by the server.

[1] Some servers may choose to represent these as *conditions*, rather than *sub-conditions*, as is shown in Appendix B.

2.4 Conditions

2.4.1 General

A *condition* is a named state of the OPC Event Server, or of one of its contained OPC Items (if it is also an OPC Data Access Server), which is of interest to its OPC Clients. An *alarm* is merely a special case of a condition, one which is deemed to be abnormal and requiring special attention. This specification deals with conditions in general, and does not treat alarms in any special way.

Within the OPC Event Server, conditions are represented by objects of type OPCCondition[2]. Each OPCCondition is associated with an OPCSource, as shown in figure 2-1. An OPCSource may be a process tag (e.g. FIC101) or possibly a device or subsystem. An OPCSource may be an OPCItem if the OPC Event Server is (or is associated with) an OPC Data Access Server.

Conditions may be *single state*, or *multi-state*. A multi-state condition is one whose state encompasses multiple “ranges” or sub-states which are of interest. For example, a “LevelAlarm” condition may have multiple sub-states including “HighAlarm” and “HighHighAlarm”. Each sub-state is represented by an object of the type OPCSubCondition (which again is not a COM object). Each OPCSubCondition is associated with an OPCCondition , as shown in figure 2-1. The sub-states of a multi-state condition must be mutually exclusive, e.g. a tag cannot be in both HighAlarm and HighHighAlarm at the same time.

The rationale for sub-conditions is to allow clients to more easily deal with closely related event notifications. For example, it is easier for an alarm display client to detect and correctly display the fact that FIC101 has moved from “HighAlarm” to “HighHighAlarm” if these states are modeled as sub-conditions of the same condition (“LevelAlarm”), than if they are modeled as independent conditions. The independent condition model makes it more difficult for the client to determine when conditions are mutually exclusive.

A single state condition has only one sub-state of interest, and thus has only one sub-condition associated with it. An example of a single state condition is a “hardware failure” condition, where a hardware device is either in the failed condition or not.

It is important to maintain a clear distinction between OPCCondition/OPCSubCondition classes and instances. When discussing a condition or sub-condition in isolation, we are likely dealing with a class of conditions or sub-conditions. However, when discussing a condition or sub-condition in conjunction with an OPCSource, we are dealing with a particular instance. For example, a “LevelAlarm” is a class of OPCConditions, which may be defined for many analog tags in the process control system. However, if we say that FIC101 is in “LevelAlarm”, we are dealing with the particular instance of “LevelAlarm” associated with FIC101.

[2] The OPCCondition discussed here is not a COM object, but is an abstract model of what we think will commonly be happening within the vendor specific server. It is not directly exposed through any of the interfaces defined in this specification. Strictly speaking, this specification defines the interfaces and their behaviors on a “black box” called an OPC Event Server, and says nothing about any internal details which might produce such behavior. However, the OPCCondition is a useful model to help explain and clarify the various behaviors.

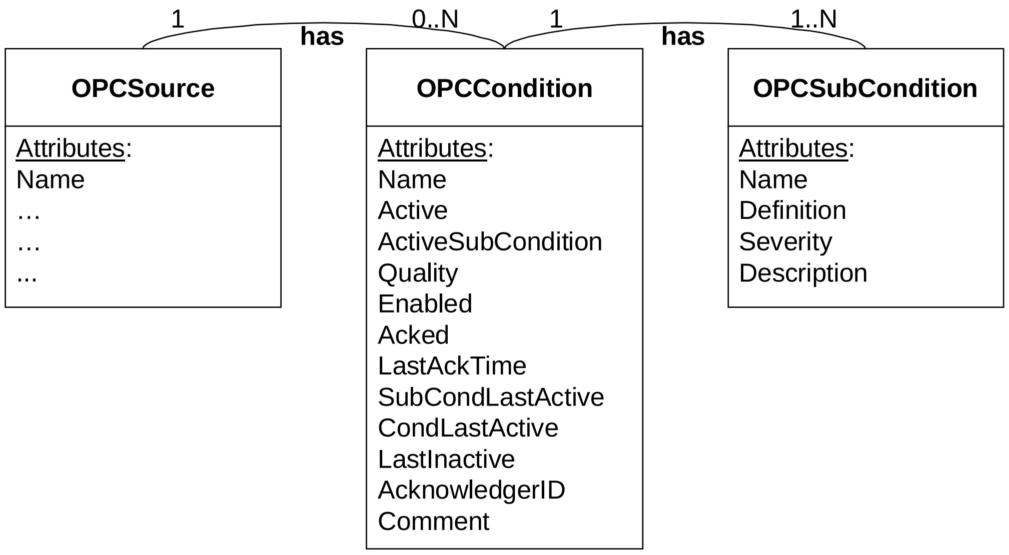


Figure 2-1. Relationship between Server Objects, OPCConditions, and OPCSubConditions.

OPCConditions and OPCSubConditions are defined by the implementer of the OPC Event Server, and the mechanisms for defining OPCConditions and OPCSubConditions are outside the scope of this specification.

2.4.2 Attributes of OPCConditions

Each OPCCondition has the following attributes:

Name	The name assigned to the condition, e.g. “LevelAlarm”. The name of a condition must be unique within the event server.
Active	The associated object is currently in the state represented by the condition.
ActiveSubCondition	If *Active*, this is the name of the SubCondition which is currently active. For example, if the LevelAlarm condition is active, the ActiveSubCondition value might be “HighAlarm”. For single-state conditions, the value would be the condition name.
Enabled	The condition is currently being checked by the OPC Event Server.
Quality	The current quality of the data value(s) upon which this condition is based. (see Condition Quality below)
Acked	If *Active*, the condition has been acknowledged.
LastAckTime	Time of the most recent acknowledgement (of any sub-condition).
SubCondLastActive	Time of the most recent transition into the currently active sub-condition. This is the time value which must be specified when acknowledging the condition.
CondLastActive	Time of most recent transition into this condition. There may be transitions among the sub-conditions which are more recent.
LastInactive	Time of most recent transition out of this condition.
AcknowledgerID	The ID of the client who last acknowledged this condition.
Comment	The comment string passed in by the client who last acknowledged this condition.

2.4.2.1 Condition Quality

Since a condition is usually based on one or more OPCItems which have a Quality attribute, the condition also has an associated quality. If the process value is "Uncertain", the "LevelAlarm" condition is also questionable. As with OPCItems, conditions will have a mandatory Quality attribute and when the quality changes, it will generate an event notification. The quality is not handled as another parameter since it is closely associated with the condition.

It is up to the server to determine how to derive the value of Quality. Servers may also wish to define a special EventCategory to report bad quality attributes for values.

Values for the Quality property conform to the OPC Quality Flags definition in the OPC Data Access server specification.

2.4.3 Attributes of OPCSubConditions

Each OPCSubCondition has the following attributes:

Name	The name assigned to the sub-condition, e.g. "HighAlarm" for a sub-condition of "LevelAlarm". In the case of a single-state alarm, the sub-condition name is the same as the associated condition name. The name of the sub-condition must be unique within its associated condition.
Definition	An expression which defines the sub-state represented by the sub-condition (see Condition Definitions below).
Severity	The severity of any event notifications generated on behalf of this sub-condition (see Severity below). Note that different sub-conditions of the same condition may have different severity levels.
Description	The text string to be included in any event notification generated on behalf of this sub-condition.

2.4.3.1 Condition Definitions

Condition definitions are server specific. Some examples are:

1. A boolean expression over one or more OPCItems, e.g. FIC101.PV > 100 & FIC101.PV < 150. This might be the definition for the HighAlarm sub-condition of the LevelAlarm condition.

2. A text string referring to a condition defined by the underlying system or device, e.g. "DeviceFailure".

3. A text string indicating a condition which is associated with the OPC Event Server. Examples of OPC Event Server conditions are:

- Shutting Down at specified time
- Server overloaded
- Underlying system/device is down
- Etc.

2.4.3.2 Severity

The severity value is an indication of the urgency of the sub-condition. This is also commonly called 'priority', especially in relation to process alarms. Values will range from 1 to 1000, with 1 being the lowest severity and 1000 being the highest. Typically, a severity of 1 would indicate in event which is informational in nature, while a value of 1000 would indicate an event of catastrophic nature which could potentially result in severe financial loss or loss of life.

It is expected that few server implementations will support 1000 distinct severity levels. Therefore, server developers are responsible for distributing their severity levels across the 1 – 1000 range in such a manner that clients can assume a linear distribution. For example, a client wishing to present five severity levels to a user should be able to do the following mapping:

Client Severity	**OPC Severity**
HIGH	801 – 1000
MEDIUM HIGH	601 – 800
MEDIUM	401 – 600
MEDIUM LOW	201 – 400
LOW	1 – 200

In many cases a strict linear mapping of underlying device severities to the OPC Severity range is not appropriate. The server developer will instead intelligently map the underlying device severities to the 1 – 1000 OPC Severity range in some other fashion. In particular, it is recommended that server developers map device events of high urgency into the OPC severity range of 667 – 1000, device events of medium urgency into the OPC severity range of 334 – 666, and low urgency device events into OPC severities of 1 – 333.

For example, if a device supports 16 severity levels, which are clustered such that severities 0, 1, and 2 are considered to be LOW, 3 – 7 are MEDIUM, and 8 – 15 are HIGH, then an appropriate mapping might be as follows:

OPC Range	**Device Severity**	**OPC Severity**
HIGH (667 – 1000)	15	1000
	14	955
	13	910
	12	865
	11	820
	10	775
	9	730
	8	685
MEDIUM (334 – 666)	7	650
	6	575
	5	500
	4	425
	3	350
LOW (1 – 333)	2	300
	1	150
	0	1

Some servers may not support any events which are catastrophic in nature, so they may choose to map all of their severities into a subset of the 1 – 1000 range (for example, 1 – 666). Other servers may not support any events which are merely informational, so they may choose to map all of their severities into a different subset of the 1 – 1000 range (for example, 334 – 1000).

The purpose of this approach is to allow clients to use severity values from multiple servers from different vendors in a consistent manner.

2.4.4 Enabling and Disabling

Clients may enable and disable conditions, and the resulting behavior is illustrated in the state diagram below. Additional behaviors are noted below:

- The server may choose to continue to test for a condition while it is disabled. However, no event notifications will be generated while the condition is disabled, nor can it be acknowledged while it is disabled.
- It is server-specific as to whether or not the following condition properties are defined while in the disabled state: Active, ActiveSubCondition, Quality, Acked, LastAckTime, SubCondLastActive, CondLastActive, LastInactive, AcknowledgerID, and Comment.
- On a refresh, no event notifications will be generated for disabled conditions.
- When enabled, the Time attribute associated with the "Condition Active" event notification will either be the time the condition is first discovered after enabling, or the time it became active (server-specific).

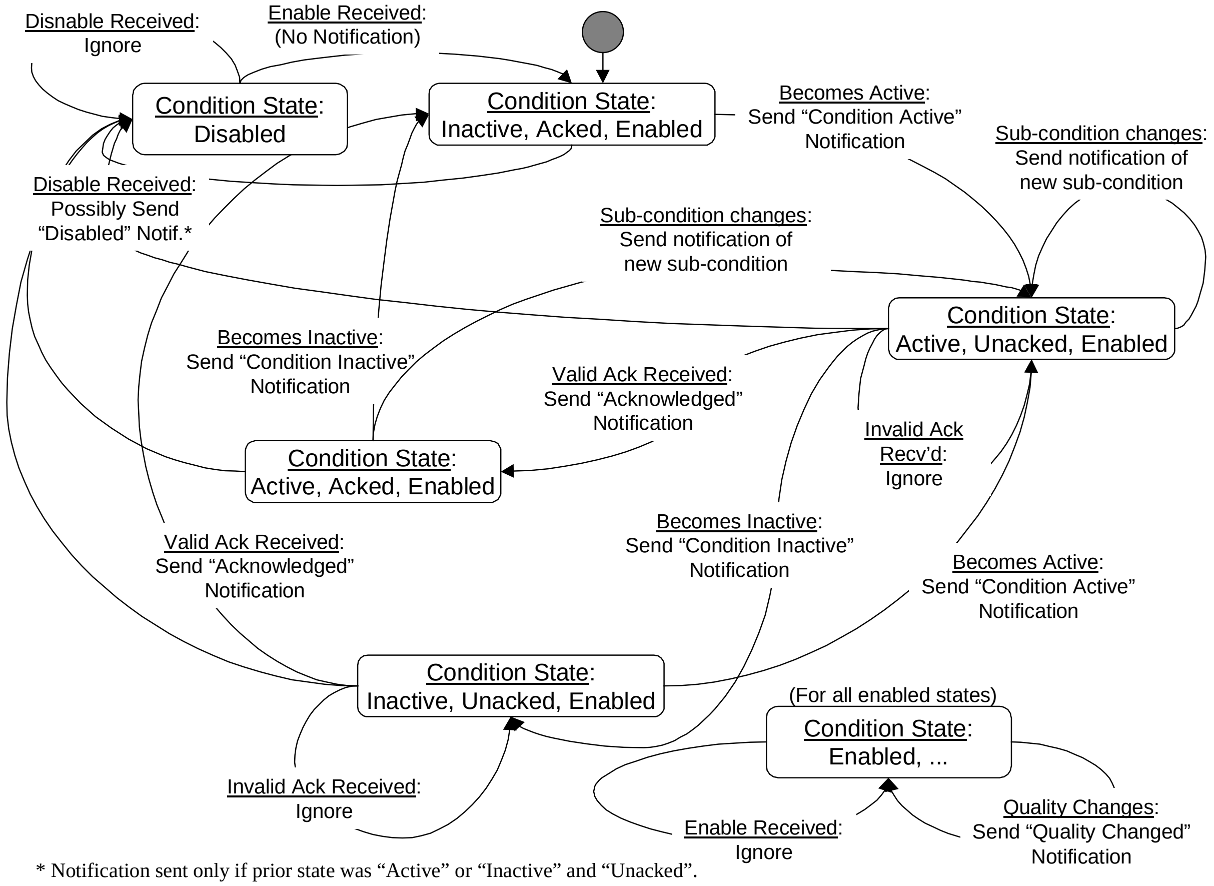
2.4.5 Interfaces

None. OPCConditions and OPCSubConditions are not COM objects. They are defined by the implementer of the OPC Event Server, and their definition is outside the scope of this specification. Methods to support client access to conditions are defined in the IOPCEventServer interface.

2.4.6 Condition States

Figure 2-2 shows a state machine for an OPCCondition which requires acknowledgement. Note that the intent of this diagram is to convey the expected behavior of conditions, as viewed by a client of the OPC Event Server. It is not intended to specify implementation, other than that the implementation must support the expected behavior.

Each state transition is an event. Event notification messages are sent at each state transition.


Disnable Received: Ignore
Enable Received: (No Notification)
Condition State: Disabled
Condition State: Inactive, Acked, Enabled
Becomes Active: Send "Condition Active" Notification
Sub-condition changes: Send notification of new sub-condition
Disable Received: Possibly Send "Disabled" Notif.*
Sub-condition changes: Send notification of new sub-condition
Condition State: Active, Unacked, Enabled
Becomes Inactive: Send "Condition Inactive" Notification
Valid Ack Received: Send "Acknowledged" Notification
Invalid Ack Recv'd: Ignore
Condition State: Active, Acked, Enabled
Becomes Inactive: Send "Condition Inactive" Notification
Valid Ack Received: Send "Acknowledged" Notification
Becomes Active: Send "Condition Active" Notification
Condition State: Inactive, Unacked, Enabled
(For all enabled states)
Condition State: Enabled, ...
Invalid Ack Received: Ignore
Enable Received: Ignore
Quality Changes: Send "Quality Changed" Notification

* Notification sent only if prior state was "Active" or "Inactive" and "Unacked".

Figure 2-2. OPCCondition State Machine

Every event notification which is condition-related (see the section below on Events and Event Notifications) and which requires acknowledgment includes the Name of the condition, the time that the condition most recently entered the active state or transitioned into a new sub-condition (SubCondLastActive property), and the Cookie which uniquely identifies the event notification. This information is specified by an OPC Client when acknowledging the condition. This information is used by the OPC Event Server to identify which specific event occurrence (state transition) is being acknowledged. If an acknowledgment is received with an out-of-date SubCondLastActive property (this can occur due to latency in the system), the condition state does not become acknowledged.

Note that an acknowledgement effects the condition state only if it (the condition) is currently active or it is currently inactive and the most recent active condition was unacknowledged. If an inactive, unacknowledged condition again becomes active, all subsequent acknowledgements will be validated against the newly active condition state attributes. The server may optionally use the Cookie attribute of the Event Notification to log acknowledgement of "old" condition activations, but such "late" acknowledgements have no affect on the current state of the condition.

Acknowledgment of the condition active state may come from the OPC client or may be due to some logic internal to the OPC Event Server. For example, acknowledgment of a related OPCCondition may result in this OPCCondition becoming acknowledged, or the OPCCondition may be set up to automatically acknowledge itself when the condition becomes inactive.

For conditions that do not track or require acknowledgement, the state transitions are simpler - just between enabled inactive, enabled-active, and disabled states.

Enabling a condition places it in the inactive-acked-enabled state. It is possible for the condition to become active very quickly after being enabled. No special scan/calculation are performed as part of the enabling action.

It is recommended that the event server generate tracking events for enable and disable operations, rather than generating an event notification for each condition instance being enabled or disabled. Enabling and disabling by area could result in a flood of event notifications if this recommendation is not followed.

2.5 Events and Event Notifications

2.5.1 General

An *event* is a detectable occurrence which is of significance to the OPC Event Server, the device it represents, and its OPC Clients. An event has no direct representation within the OPC model. Rather, its occurrence is made known via an *Event Notification*. Event Notifications are represented by objects of class OPCEventNotification[3], which are described in the following section. (OPCEventNotifications are not COM objects.)

There are three types of events:

1. *Condition-related* events are associated with OPCConditions, and represent transitions into or out of the states represented by OPCConditions and OPCSubConditions. An example is the tag FIC101 transitioning into the LevelAlarm condition and HighAlarm sub-condition.
2. *Tracking-related* events are not associated with conditions, but represent occurrences which involve the interaction of an OPC Client with a “target” object within the OPC Event Server. An example of such an event is a control change in which the operator, (the OPC Client), changes the set point of tag FIC101 (the “target”).
3. *Simple* events are all events other than the above. An example of a simple event is a component failure within the system/device represented by the OPC Event Server.

2.5.2 Event Notifications

OPCEventNotifications are sent to subscribing clients using the Connection Point callback interface supplied by the OPC Client in the event subscription (see Subscriptions to Event Notifications below).

The types of OPCEventNotifications form an inheritance hierarchy as shown in figure 2-3.

[3] The OPCEventNotification discussed here is not a COM object, but is an abstract model of what we think will commonly be happening within the vendor specific server. It is not directly exposed through any of the interfaces defined in this specification, although event notification attributes are provided to the client in the ONEVENTSTRUCT (see the description of the IOPCEventSink interface later in this document). Strictly speaking, this specification defines the interfaces and their behaviors on a “black box” called an OPC Event Server, and says nothing about any internal details which might produce such behavior. However, the OPCEventNotification is a useful model to help explain and clarify the various behaviors.

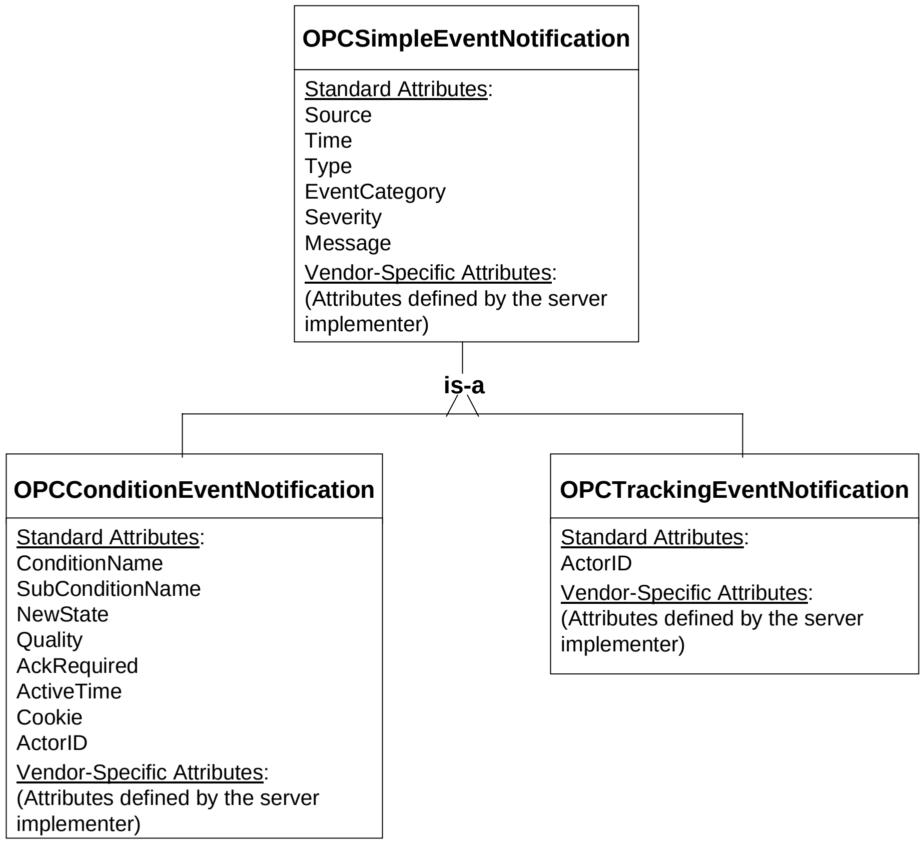

OPCSimpleEventNotification
Standard Attributes:
Source
Time
Type
EventCategory
Severity
Message
Vendor-Specific Attributes:
(Attributes defined by the server implementer)
is-a
OPCConditionEventNotification
Standard Attributes:
ConditionName
SubConditionName
NewState
Quality
AckRequired
ActiveTime
Cookie
ActorID
Vendor-Specific Attributes:
(Attributes defined by the server implementer)
OPCTrackingEventNotification
Standard Attributes:
ActorID
Vendor-Specific Attributes:
(Attributes defined by the server implementer)

Figure 2-3. OPCEventNotification Type Hierarchy

2.5.2.1 Standard Attributes

All OPCEventNotifications have standard attributes which are defined by this specification, and are included in the ONEVENTSTRUCT returned to clients with event notifications. See the discussion of the IOPCEventSink interface in Section 5.6.1.

2.5.2.1.1 OPCSimpleEventNotifications

OPCSimpleEventNotifications have the following standard attributes. Note that OPCConditionEventNotifications and OPCTrackingEventNotifications also include these standard attributes through inheritance.

Source	A reference to the object which generated the event notification. For example, this would be a tag name (e.g. FIC101) if the event pertains to a tag entering the LevelAlarm condition (condition-related event). It could also be a tag name for a tracking event such as the operator changing the set point value for FIC101. For a simple event such as a system error, the Source value might be “System”.
Time	The time that the event occurred.
Type	The type of the event, i.e. condition-related, tracking-related, or simple.
EventCategory	The category to which this event belongs (see Event Categories below).
Severity	The urgency of the event. This may be a value in the range of 1 – 1000, as described

	in Section 2.4.3.2.
Message	Message text which describes the event. For condition-related events, this will generally include the description property of the active sub-condition.

2.5.2.1.2 OPCTrackingEventNotifications

Tracking events have the attributes of a simple event plus the following:

ActorID	The identifier of the OPC Client which initiated the action resulting in the tracking-related event. For example, if the tracking-related event is a change in the set point of FIC101, the ActorID might be a reference to the client application which initiated the change or might be the userID of the operator who specified the change. This value is server specific, and its definition is outside the scope of this specification.

2.5.2.1.3 OPCConditionEventNotifications

Condition events have the attributes of a simple event plus the following:

ConditionName	The name of the associated OPCCondition.
SubConditionName	The name of the currently active OPCSubCondition.
ChangeMask	Indicates to the client which properties of the condition have changed, to have caused the server to send the event notification.
NewState	Indicates the new state of the condition. This indicates the new values for the *Enabled*, *Active*, and *Acked* properties of the condition.
ConditionQuality	Indicates the quality of the underlying data items upon which this condition is based.
AckRequired	An indicator as to whether or not an acknowledgement is required. Many event notifications related to conditions do not normally require an acknowledgment, e.g. the receipt of an acknowledgment or the transition to the inactive state. Furthermore, some conditions may be configured (using facilities outside the scope of this specification) to not require acknowledgment even for transitions into the condition, or for transitions among sub-conditions (e.g. transition into LevelAlarm or transition from HighAlarm to HighHighAlarm). In this case, it is the responsibility of the server to automatically place the condition into the Acknowledged state, since an acknowledgment will never be received.
ActiveTime	The time of the transition into the condition or sub-condition which is associated with this event notification. This corresponds to the SubCondLastActive property of the associated OPCCondition object and is used to correlate condition acknowledgements with a particular transition into the condition/sub-condition.
Cookie	Server defined cookie associated with the event notification. This value is used by the client when acknowledging the condition. This value is opaque to the client.
ActorID	The identifier of the OPC Client which acknowledged the condition, which is maintained as the AcknowledgerID property of the condition. This is included in event notifications generated by condition acknowledgments.

2.5.2.2 Vendor-Specific Attributes

In addition to the standard attributes described above, implementers of OPC Event Servers may choose to provide additional attributes with event notifications. In order to promote consistency among event server implementations, implementers are encouraged to select their attribute names from those listed in Appendix C where applicable.

2.5.3 Event Categories

EventCategories define groupings of events supported by an OPC Event server. Examples of event categories might include “Process Events”, “System Events”, or “Batch Events”. Event categories may be defined for all event types, i.e. Simple, Tracking, and Condition-Related. However, a particular event category can include events of only one type. A given Source (e.g. “System” or “FIC101”) may generate events for multiple event categories. Names of event categories must be unique within the event server. The definition of event categories is server specific and is outside the scope of this specification. A list of recommended event categories for each event type is provided in Appendix B.

The name of the event category is included in every event notification. Event subscriptions may be filtered based on event category.

2.5.4 Interfaces

OPC Event Servers provide interfaces to allow OPC Clients to determine the types of events which the OPC Event Server supports, and to enter subscriptions to specified events.

2.6 Subscriptions to Event Notifications

2.6.1 General

In order to receive event notifications, OPC Clients must subscribe to them. A subscription is entered with an OPC Event Server by requesting it to create an OPCEventSubscription object. An OPC Client may have one or more OPCEventSubscriptions active with a single OPC Event Server.

OPCEventSubscriptions are “connectable objects” in that they implement the DCOM Connection Point interfaces. This is the mechanism used to send event notifications to OPC Clients.

2.6.2 Properties of OPCEventSubscriptions

OPCEventSubscriptions have the following property:

Filter	A structure containing criteria for selecting events of interest to the client (see Filters below). A null Filter results in the OPC Client receiving all event notifications.

2.6.3 Filters

Events may be selected using the following criteria:

- Type of event, i.e. simple, condition, or tracking.
- Event categories
- Lowest severity, i.e. all events with a severity greater than or equal to the specified severity.
- Highest severity, i.e. all events with a severity less than or equal to the specified severity.
- Process areas
- Event sources

A list of values for a single criterion are logically ORed together (e.g. if two event categories are specified, event notifications for both categories will be received). If multiple criteria are specified, they will be logically ANDed together, i.e. only those events satisfying all criteria will be selected. An example is specifying both lowest priority and highest priority will result in the selection of events with priorities lying between the two values.

For example, the following filter:

```
Type = CONDITION
Category = PROCESS
LowSeverity = 600
Area = AREA1, AREA2
```

would result in the selection of condition-related events within the “Process” category in both AREA1 and AREA2 which are of high urgency (greater than or equal to 600).

An OPCEventSubscription has only one filter.

2.6.4 Interfaces

OPCEventSubscriptions provide an interface to allow the OPC Client to specify the Filter. In addition, they implement the standard DCOM Connection Point interfaces, to provide the mechanism for notifying OPC Clients of event occurrences.

2.7 Condition State Synchronization

OPC Clients can obtain the current state of all conditions which are active, or which are inactive but unacknowledged, by requesting a “refresh” from each active OPCEventSubscription object. The server will respond by sending the appropriate events to the client, via the event call back mechanism, for all conditions selected by the filter for each subscription. When invoking the client’s call back, the server will indicate whether the invocation is for a refresh or is an original notification. Refresh and original event notifications will not be mixed in the same call back invocation.

This design assumes that the client needs only the current state information for conditions, so only condition-releated event notifications are refreshed. It should be noted that “refresh” is not a general replay capability, since the server is not required to maintain an event history.

Refresh event notifications may be sent in an arbitrary order and may be out of sequence. Since conditions may change state while the server is replying to a refresh request, the refresh event notification may no longer reflect the current condition state by the time the client receives it. Similarly, a client may receive an original event notification after receiving a refresh event notification for the same event. Clients will need to compare timestamps to ensure that they have the correct state of the condition.

2.8 Error Handling

OPC Event Servers may report internal or source connection errors as standard events, which may be simple events or condition-related events. Events for server errors belong to the OPC_SERVER_ERROR event category. The specific events included in this category are vendor specific, but they should cover cases such as:

- Internal buffer overflow
- Event source communication problems
- Client communication problems

In the case of loss of communication from an event source, the currently active conditions from that source should have their quality attribute updated to signify the loss of communication. This can be accomplished by setting the quality to “Bad” with a substatus of “Comm Failure”. This change in quality must result in event notifications to all subscribers.

3. Architectural Overview

3.1 Relationship to OPC Data Access Server

Any COM object which supports the IOPCEventServer interface is an OPC Event Server. In many cases, an OPC Data Access Server will also expose an OPCEventServer object and will fill both the roles of data server and event server. However, there may be other situations where it is advantageous to have a dedicated OPC Event Server, i.e. one which is not also an OPC Data Access Server object.

3.2 Overview of Objects and Interfaces

3.2.1 General

This specification defines the following COM objects, which are briefly covered in the following sections: OPCEventServer, OPCEventSubscription, and OPCEventAreaBrowser.

Figure 3-1 shows the how these objects are related.

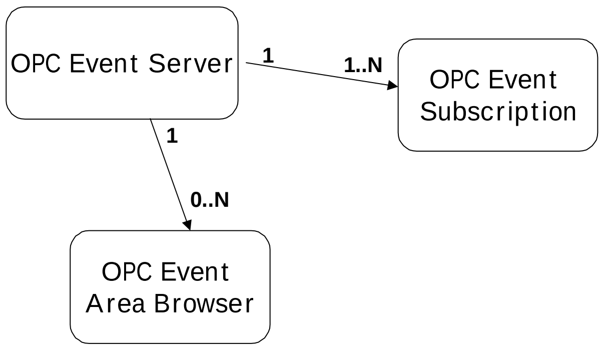


Figure 3-1 - Relationship of OPC Event Server Objects

3.2.2 OPCEventServer Object

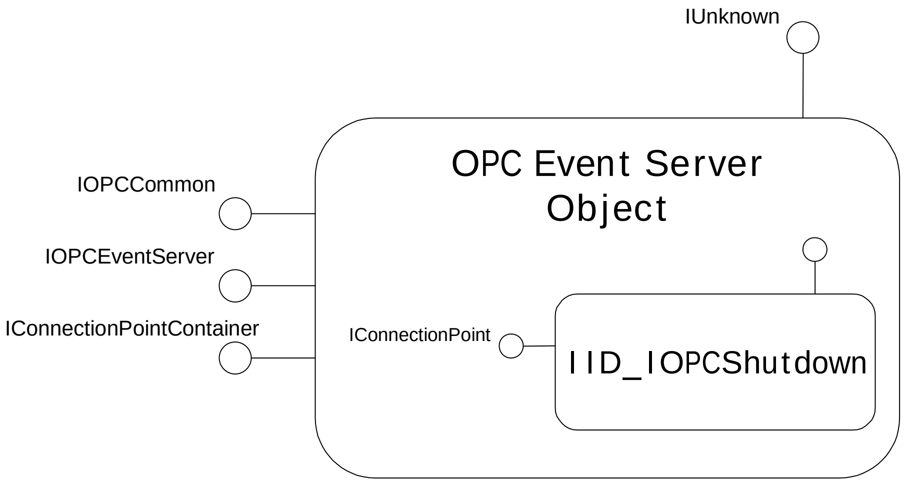


Figure 3-2 - OPC Event Server Object

Figure 3-2 is a view of an OPC Event Server and IID_IOPCShutdown objects. These objects are created (or connected to) using the DCOM facilities CoCreateInstance or CoCreateInstanceEx. As noted earlier, this may be an OPC (Data Access) Server object which also implements the IOPCEventServer interface, or may be a distinct COM object which implements this interface but not the data access interfaces.

The IOPCCommon interface is used to perform certain functions which are common to other OPC servers, e.g. Data Access. Examples of such common functions are the management of LocaleIDs and retrieval of error strings.

The IOPCEventServer interface is used to create OPC Event Subscription and OPC Event Area Browser objects, query vendor-specific event categories and event parameters, and manage conditions.

The IConnectionPointContainer and IConnectionPoint interfaces are the standard DCOM interfaces for connectable objects, and are used to handle the callbacks for server notifications to the client of impending shutdown.

3.2.3 OPCEventSubscription Object

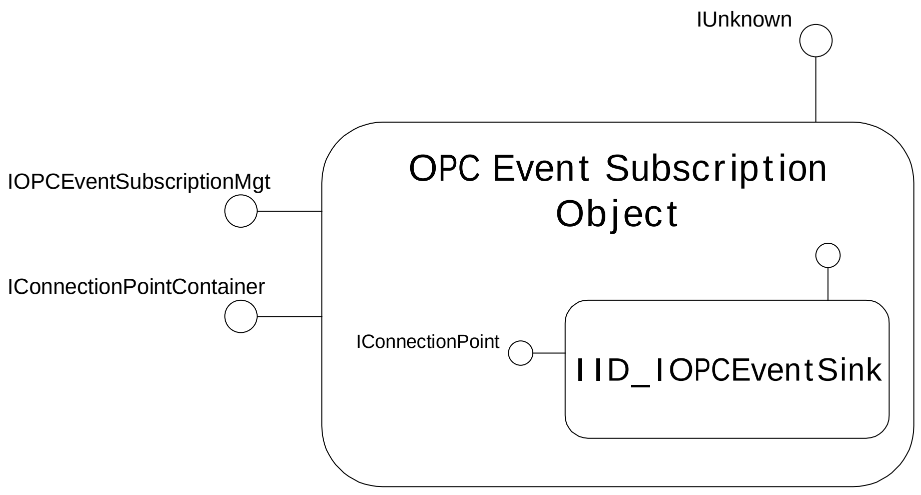


Figure 3-3 - OPC Event Subscription Object

Figure 3-3 is a view of the OPCEventSubscription and IID_IOPCEventSink objects, which are created by the OPC Event Server when the client subscribes to events using the `IOPCEventServer::CreateEventSubscription` method.

The IOPCEventSubscriptionMgt interface is used to configure filters and other attributes for OPC event reporting.

The IConnectionPointContainer and IConnectionPoint interfaces are the standard DCOM interfaces for connectable objects, and are used to handle the callbacks for event notifications.

3.2.4 OPCEventAreaBrowser Object (optional)

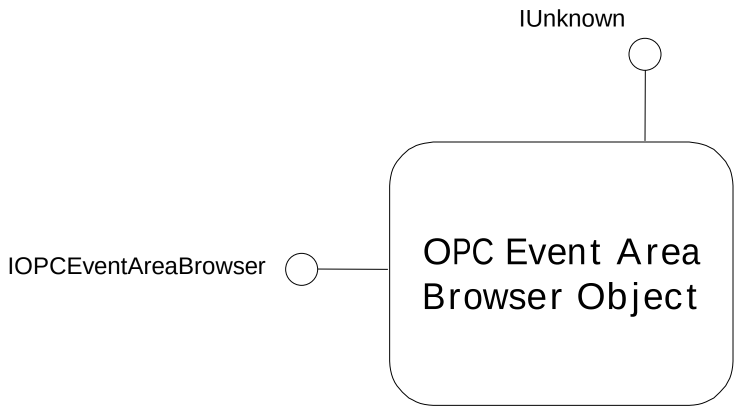


Figure 3-4 - OPC Event Area Browser Object

Figure 3-4 is a view of the OPCEventAreaBrowser object which is created by the OPC Event Server when the client invokes the `IOPCEventServer::CreateAreaBrowser` method.

The IOPCEventAreaBrowser interface provides a way for clients to browse the process area organization implemented by the server. The expectation is that events and conditions provided by the server are organized into one or more process areas, and that the client can filter event subscriptions according to specified process areas.

This object is optional, and may not be exposed by simple event servers.

4. OPC Event Server Quick Reference

This section includes a quick reference for the methods in the Custom Interface. These interfaces, their parameters, and behavior are defined in detail in section 5.

4.1 Custom Interface – Server Side

Note: this section does not show additional standard COM interfaces, such as IUnknown, which are also supported by the event server.

OPCEventServer
  IOPCCommon
  IOPCEventServer
  IConnectionPointContainer

OPCEventAreaBrowser (optional)
  IOPCEventAreaBrowser

OPCEventSubscription
  IOPCEventSubscriptionMgt
  IConnectionPointContainer

4.1.1 OPCEventServer Object

IOPCCommon

HRESULT	SetLocaleID (dwLcid)
HRESULT	GetLocaleID (pdwLcid)
HRESULT	QueryAvailableLocaleIDs (pdwCount, pdwLcid)
HRESULT	GetErrorString (dwError, ppString)
HRESULT	SetClientName (szName)

IOPCEventServer

HRESULT	GetStatus (ppEventServerStatus)
HRESULT	CreateEventSubscription (bActive, dwBufferTime, dwMaxSize, hClientSubscription, riid, ppUnk, pdwRevisedBufferTime, pdwRevisedMaxSize)
HRESULT	QueryAvailableFilters (pdwFilterMask)
HRESULT	QueryEventCategories (dwEventType, pdwCount, ppdwEventCategories, ppEventCategoryDescs)
HRESULT	QueryConditionNames (dwEventCategory, pdwCount, ppszConditionNames)
HRESULT	QuerySubConditionNames (szConditionName, pdwCount, ppszSubConditionNames)
HRESULT	QuerySourceConditions (szSource, pdwCount, ppszConditionNames)
HRESULT	QueryEventAttributes (dwEventCategory, pdwCount, ppdwAttrIDs, ppszAttrDescs, ppvtAttrTypes)
HRESULT	TranslateToItemIds (szSource, dwEventCategory, szConditionName, szSubConditionName, dwCount, pdwAssocAttrIDs, ppszAttrItemIDs, ppszNodeNames, pCLSIDs) – *See Note1*
HRESULT	GetConditionState (szSource, szConditionName, ppConditionState)
HRESULT	EnableConditionByArea (dwNumAreas, pszAreas) – *See Note1*
HRESULT	EnableConditionBySource (dwNumSources, pszSources) – *See Note1*
HRESULT	DisableConditionByArea (dwNumAreas, pszAreas) – *See Note1*
HRESULT	DisableConditionBySource (dwNumSources, pszSources) – *See Note1*
HRESULT	AckCondition (dwCount, szAcknowledgerID, szComment, pszSource, pszConditionName, pftActiveTime, pdwCookie, ppErrors)
HRESULT	CreateAreaBrowser (riid, ppUnk) – *See Note1*

IOPCConnectionPointContainer

HRESULT	EnumConnectionPoints (ppEnum)
HRESULT	FindConnectionPoint (riid, ppCP)

4.1.2 OPCEventAreaBrowser Object (optional)

IOPCEventAreaBrowser

HRESULT	ChangeBrowsePosition (dwBrowseDirection, szString)
HRESULT	BrowseOPCAreas (dwBrowseFilterType, szFilterCriteria, ppIEnumString)
HRESULT	GetQualifiedAreaName (szAreaName, pszQualifiedAreaName)
HRESULT	GetQualifiedSourceName (szSourceName, pszQualifiedSourceName)

4.1.3 OPCEventSubscription Object

IOPCEventSubscriptionMgt

HRESULT	SetFilter (dwEventType, dwNumCategories, pdwEventCategories, dwLowSeverity, dwHighSeverity, dwNumAreas, pszAreaList, dwNumSources, pszSourceList) – *See Note2*
HRESULT	GetFilter (pdwEventType, pdwNumCategories, ppdwEventCategories, pdwLowSeverity, pdwHighSeverity, pdwNumAreas, ppszAreaList, pdwNumSources, ppszSourceList)
HRESULT	SelectReturnedAttributes (dwEventCategory, dwCount, dwAttributeIDs)
HRESULT	GetReturnedAttributes (dwEventCategory, pdwCount, pdwAttributeIDs)
HRESULT	Refresh (dwConnection)
HRESULT	CancelRefresh (dwConnection)
HRESULT	GetState (pbActive, pdwBufferTime, pdwMaxSize, phClientSubscription)
HRESULT	SetState (bActive, dwBufferTime, dwMaxSize, hClientSubscription, pdwRevisedBufferTime, pdwRevisedMaxSize)

IOPCConnectionPointContainer

HRESULT	EnumConnectionPoints (ppEnum)
HRESULT	FindConnectionPoint (riid, ppCP)

Note1: These methods may not be supported by simple event servers, and may return E_NOTIMPL.

Note2: The functionality of this method may be restricted by simple event servers.

4.2 Custom Interface – Client Side

IOPCEventSink

HRESULT	OnEvent (hClientSubscription, bRefresh, bLastRefresh, dwCount, pEvents)

IOPCShutdown

HRESULT	ShutdownRequest (szReason)

5. OPC Event Server Custom Interfaces

5.1 Overview

The OPC Event Server Custom Interface objects include the following:

- OPCEventServer
- OPCEventSubscription
- OPCEventAreaBrowser

The interfaces and behaviors of these objects are described in detail in this chapter. Developers of OPC Event servers are required to implement the OPC objects by providing the functionality defined in this chapter.

This chapter also references and defines expected behavior for the standard OLE interfaces that an OPC Event server and an OPC Event client are required to implement to build and deliver OPC compliant components.

In addition, standard and custom enumerator objects are created and interfaces to these objects are returned in several cases. In general the enumerator objects and interfaces are described only briefly since their behavior is well defined by OLE.

Note that for proper operation, enumerators are created and returned from methods on objects rather than through QueryInterface. The enumerator defined in this specification is:

- Server process area enumerator - (see IOPCEventAreaBrowser::BrowseOPCAreas)

Additional enumerators may be created when dealing with connection points (see the IOPCEventSubscriptionMgt interface). However, they are created using standard COM interfaces defined for connectable objects.

5.2 General Information

Ownership of memory

Per the COM specification, clients must free all memory associated with ‘out’ or ‘in/out’ parameters. This includes memory that is pointed to by elements within any structures. This is very important for client writers to understand as problems will result in troublesome and difficult to locate memory leaks. See the IDL file to determine which parameters are out parameters. The recommended approach is for the client to create a subroutine to be used for properly freeing each type of structure.

Standard Interfaces

Note that (per the COM specification) all methods must be implemented on each interface. Methods which are not required can return E_NOTIMPL or occasionally S_OK depending on the situation.

Null Strings and Null Pointers

Both of these terms are used below. They are NOT the same thing. A NULL Pointer is an invalid pointer (0) which will cause an exception if used. A NULL String is a valid (non zero) pointer to a 1 character array where that character is a NULL (i.e. 0).

Note that COM does not allow NULL to be passed for *Out* or *In/Out* parameters.

Returned Arrays

You will note the syntax **“size_is(,dwCount)”** in the IDL used in combination with pointers to pointers. This indicates that the returned item is a pointer to an actual array of the indicated type rather than a pointer to an array of pointers to items of the indicated type. This simplifies marshaling as well as creation and access of the data by the server and client.

5.3 OPCEventServer Object

5.3.1 Overview

The OPCEventServer object is the primary object that an OPC Event Server exposes. The interfaces that this object provides include:

- IUnknown
- IOPCCommon
- IOPCEventServer
- IConnectionPointContainer

5.3.2 IUnknown

The server must provide a standard IUnknown Interface. Since this is a well defined interface it is not discussed in detail. See the OLE Programmer's reference for additional information. This interface must be provided, and all functions implemented as required by Microsoft..

5.3.3 IOPCCommon

Other OPC Servers such as Data Access share this interface design. It provides the ability to set and query a LocaleID which would be in effect for the particular client/server session. That is, as with a Group definition, the actions of one client do not affect any other clients.

A quick reference for this interface is provided below. A more detailed discussion can be found in the OPC Common specification.

```
HRESULT SetLocaleID (
    [in] LCID dwLcid
    );

HRESULT GetLocaleID (
    [out] LCID *pdwLcid
    );

HRESULT QueryAvailableLocaleIDs (
    [out] DWORD *pdwCount,
    [out, sizeis(dwCount)] LCID *pdwLcid
    );

HRESULT GetErrorString(
    [in] HRESULT dwError,
    [out, string] LPWSTR *ppString
    );

HRESULT SetClientName (
    [in, string] LPCWSTR szName
    );
```

5.3.4 IOPCEventServer

This is the main interface to the alarm and event capabilities of an OPC Event Server. This interface is used to create OPC Event Subscription objects, to create OPC Event Area Browser objects, to query event categories and associated event parameters, to manage conditions, and to perform miscellaneous operations such as getting the status of the event server.

5.3.4.1 IOPCEventServer::GetStatus

```
HRESULT GetStatus (
    [out] OPCEVENTSERVERSTATUS ** ppEventServerStatus
    );
```

Description

Returns current status information for the OPC Event server.

Parameters	**Description**
ppEventServerStatus	Pointer to where the OPCEVENTSERVERSTATUS structure pointer should be returned. The server allocates the structure.

Return Codes

Return Code	**Description**
E_FAIL	The operation failed.
E_OUTOFMEMORY	Not enough memory
E_INVALIDARG	An argument to the function was invalid.
S_OK	The operation succeeded.

Comments

The OPCEVENTSERVERSTATUS is described below.

Client must free the structure as well as the VendorInfo string within the structure.

Periodic calls to GetStatus would be a good way for the client to determine that the server is still connected and available.

5.3.4.1.1 OPCEVENTSERVERSTATUS

```
typedef struct {
    FILETIME            ftStartTime;
    FILETIME            ftCurrentTime;
    FILETIME            ftLastUpdateTime;
    OPCEVENTSERVERSTATE   dwServerState;
    WORD                wMajorVersion;
    WORD                wMinorVersion;
    WORD                wBuildNumber;
    [string]    LPWSTR      szVendorInfo;
} OPCEVENTSERVERSTATUS;
```

This structure is used to communicate the status of the server to the client. This information is provided by the server in the IOPCEventServer::GetStatus() call.

Member	**Description**
ftStartTime	Time (UTC) the event server was started. This is constant for the server instance and is not reset when the server changes states. Each instance of a server should keep the time when the process started.
ftCurrentTime	The current time (UTC) as known by the server.
ftLastUpdateTime	The time (UTC) the server sent an event notification (via the IOPCEventSink::OnEvent) to this client. This value is maintained on an instance basis.
dwServerState	The current status of the server. Refer to **OPC Event Server State values below.**
wMajorVersion	The major version of the server software
wMinorVersion	The minor version of the server software
wBuildNumber	The 'build number' of the server software
szVendorInfo	Vendor specific string providing additional information about the server. It is recommended that this mention the name of the company and the type of device(s) supported.

OPCEVENTSERVERSTATE Values	**Description**
OPC_STATUS_RUNNING	The server is running normally. This is the usual state for a server
OPC_STATUS_FAILED	A vendor specific fatal error has occurred within the server. The server is no longer functioning. The recovery procedure from this situation is vendor specific. An error code of E_FAIL should generally be returned from any other server method.
OPC_STATUS_NOCONFIG	The server is running but has no configuration information loaded and thus cannot function normally. Note this state implies that the server needs configuration information in order to function. Servers which do not require configuration information should not return this state.
OPC_STATUS_SUSPENDED	The server has been temporarily suspended via some vendor specific method and is not getting or sending data.
OPC_STATUS_TEST	The server is in Test Mode. Events may be generated in a simulation mode, this is server specific.

5.3.4.2 IOPCEventServer:: CreateEventSubscription

```
HRESULT CreateEventSubscription(
    [in] BOOL bActive,
    [in] DWORD dwBufferTime,
    [in] DWORD dwMaxSize,
    [in] OPCHANDLE hClientSubscription,
    [in] REFIID riid,
    [out, iid_is(riid)] LPUNKNOWN * ppUnk
    [out] DWORD * pdwRevisedBufferTime,
    [out] DWORD * pdwRevisedMaxSize,
    );
```

Description

Add an Event Subscription object to an Event Server.

Create an OPCEventSubcription object on behalf of this client and return an interface to the Client. This object will support at least IUnknown, IOPCEventSubscriptionMgt and IConnectionPointContainer. The client can manage the state of this interface including the filter and can create subscriptions to it via ConnectionPoints as described later.

The Event Subscription Object uses conventional reference counting and thus will be deleted with all interfaces to it are released.

Parameters	**Description**
bActive	FALSE if the Event Subscription is to be created inactive. TRUE if the Event Subscriptions is to be created as active. If the subscription is inactive, then the server will not send event notifications to the client based on the subscription, and has no responsibility to buffer or maintain the event notifications. Thus event notifications may be lost.
dwBufferTime	The requested buffer time. The buffer time is in milliseconds and tells the server how often to send event notifications. This is a minimum time - do not send event notifications any faster that this UNLESS dwMaxSize is greater than 0, in which case the server will send an event notification sooner to obey the dwMaxSize parameter. A value of 0 for dwBufferTime means that the server should send event notifications as soon as it gets them. This parameter along with the dwMaxSize parameter are used to improve communications efficiency between client and server. This parameter is a recommendation from the client, and the server is allowed to ignore the parameter. The server will return the buffer time it is actually providing in pdwRevisedBufferTime.

dwMaxSize	The requested maximum number of events that will be sent in a single IOPCEventSink::OnEvent callback. A value of 0 means that there is no limit to the number of events that will be sent in a single callback.. Note that a value of dwMaxSize greater than 0, may cause the server to call the OnEvent callback more frequently than specified in the dwBufferTime parameter when a large number of events are being generated in order to limit the number of events to the dwMaxSize. This parameter is a recommendation from the client and the server is allowed to ignore this parameter. The server will return the actual number of events it is actually providing in pdwRevisedMaxSize.
hClientSubscription	Client provided handle for this event subscription. This handle is passed back in the OnEvent callback to identify the subscription object that is calling back. The client should assign a unique value of hClientSubscription for each subscription object in order to detect the source of the callback information.
riid	The type of interface desired (e.g. IID_IOPCEventSubscriptionMgt)
ppUnk	Where to store the returned interface pointer. NULL is returned for any FAILED HRESULT.
pdwRevisedBufferTime	The buffer time that the server is actually providing, which may differ from dwBufferTime.
pdwRevisedMaxSize	The maximum number of events that the server will actually be sending in a single IOPCEventSink::OnEvent callback, which may differ from dwMaxSize.

Return Codes

Return Code	Description
E_FAIL	The operation failed.
E_OUTOFMEMORY	Not enough memory
E_INVALIDARG	Bad argument was passed.
OPC_S_INVALIDBUFFERTIME	The buffer time parameter was invalid .
OPC_S_INVALIDMAXSIZE	The max size parameter was invalid.
S_OK	The operation succeeded.

Comments

5.3.4.3 IOPCEventServer::QueryAvailableFilters

```
HRESULT QueryAvailableFilters(
    [out] DWORD * pdwFilterMask,
    );
```

Description

The QueryAvailableFilters method gives clients a means of finding out exactly which filter criteria are supported by a given event server. This method would typically be invoked before configuring the filter on an OPCEventSubscription object.

The client passes a pointer to where information is to be saved.

Parameters	**Description**
pdwFilterMask	This is a pointer to a bit mask which indicates which types of filtering are supported by the server. See below for mask values.

HRESULT Return Codes

Return Code	**Description**
S_OK	The function was successful.
E_FAIL	The function was unsuccessful.

Filter Mask Values

Filter Mask Item	**Value**	**Description**
OPC_FILTER_BY_EVENT	1	The server supports filtering by event type.
OPC_FILTER_BY_CATEGORY	2	The server supports filtering by event categories.
OPC_FILTER_BY_SEVERITY	4	The server supports filtering by severity levels.
OPC_FILTER_BY_AREA	8	The server supports filtering by process area.
OPC_FILTER_BY_SOURCE	16	The server supports filtering by event sources.

Comments

5.3.4.4 IOPCEventServer::QueryEventCategories

```
HRESULT QueryEventCategories(
    [in] DWORD dwEventType,
    [out] DWORD* pdwCount,
    [out, size_is(*pdwCount)] DWORD** ppdwEventCategories,
    [out, size_is(,*pdwCount)] LPWSTR** ppEventCategoryDescs
    );
```

Description

The QueryEventCategories method gives clients a means of finding out the specific categories of events supported by a given server. This method would typically be invoked prior to specifying an event filter. Servers will be able to define their own custom event categories, but a list of recommended categories is provided in Appendix B.

Parameters	**Description**
dwEventType	A DWORD bit mask specifying which event types are of interest; OPC_SIMPLE_EVENT, OPC_CONDITION_EVENT, OPC_TRACKING_EVENT, OPC_ALL_EVENTS, These types can be OR'ed together to select multiple event types. A value of 0 is an error and causes E_INVALIDARG to be returned.
pdwCount	The number of event categories (size of the EventCategoryID, and EventCategoryDesc arrays) returned by the function.
ppdwEventCategories	Array of DWORD codes for the vendor-specific event categories implemented by the server. These IDs can be used in the event subscription interface for specifying filters.
ppEventCategoryDescs	Array of strings for the text names or descriptions for each of the event category IDs. This array corresponds to the EventCategories array.

Return Codes

Return Code	**Description**
E_FAIL	The operation failed.
E_INVALIDARG	A bad parameter was passed.
E_OUTOFMEMORY	Not enough memory
S_OK	The operation succeeded.

Recommended Event Categories

Server implementers are encouraged to implement the event categories described in Appendix B, in order to provide a level of consistency among event server implementations.

Comments

The number of event categories returned will vary depending on the sophistication of the server, but is expected to be less than 30 for most servers, making this interface more appropriate than a custom enumerator.

5.3.4.5 IOPCEventServer::QueryConditionNames

```
HRESULT QueryConditionNames{
    [in] DWORD dwEventCategory,
    [out] DWORD* pdwCount,
    [out, size_is(,*pdwCount)] LPWSTR** ppszConditionNames
    );
```

Description

The QueryConditionNames method gives clients a means of finding out the specific condition names which the event server supports for the specified event category. This method would typically be invoked prior to specifying an event filter. Condition names are server specific.

Parameters	**Description**
dwEventCategory	A DWORD event category code, as returned by the QueryEventCategories method. Only the names of conditions within this event category are returned.
pdwCount	The number of condition names being returned.
ppszConditionNames	Array of strings containing the condition names for the specified event category.

Return Codes

Return Code	**Description**
E_FAIL	The operation failed.
E_INVALIDARG	A bad parameter was passed.
E_OUTOFMEMORY	Not enough memory
S_OK	The operation succeeded.

Comments

The number of condition names returned will vary depending on the sophistication of the server, but is expected to be less than 30 for most servers, making this interface more appropriate than a custom enumerator.

5.3.4.6 IOPCEventServer::QuerySubConditionNames

```
HRESULT QuerySubConditionNames{
    [in] LPWSTR szConditionName,
    [out] DWORD* pdwCount,
    [out, size_is(,*pdwCount)] LPWSTR** ppszSubConditionNames
    );
```

Description

The QuerySubConditionNames method gives clients a means of finding out the specific sub-condition names which are associated with the specified condition name. Condition names are server specific.

Parameters	**Description**
szConditionName	A condition name, as returned by the QueryConditionNames method. Only the names of sub-conditions associated with this condition are returned.
pdwCount	The number of sub-condition names being returned.
ppszSubConditionNames	Array of strings containing the sub-condition names associated with the specified condition.

Return Codes

Return Code	**Description**
E_FAIL	The operation failed.
E_INVALIDARG	A bad parameter was passed.
E_OUTOFMEMORY	Not enough memory
S_OK	The operation succeeded.

Comments

The number of sub-condition names returned will vary depending on the sophistication of the server, but is expected to be less than 10 for most servers, making this interface more appropriate than a custom enumerator.

5.3.4.7 IOPCEventServer::QuerySourceConditions

```
HRESULT QuerySourceConditions{
    [in] LPWSTR szSource,
    [out] DWORD* pdwCount,
    [out, size_is(,*pdwCount)] LPWSTR** ppszConditionNames
    );
```

Description

The QuerySourceConditions method gives clients a means of finding out the specific condition names associated with the specified source (e.g. FIC101).. Condition names are server specific.

Parameters	Description
szSource	A source name, as returned by the IOPCEventAreaBrower::GetQualifiedSourceName method. Only the names of conditions associated with this source are returned.
pdwCount	The number of condition names being returned.
ppszConditionNames	Array of strings containing the condition names for the specified source.

Return Codes

Return Code	Description
E_FAIL	The operation failed.
E_INVALIDARG	A bad parameter was passed.
E_OUTOFMEMORY	Not enough memory
S_OK	The operation succeeded.

Comments

The number of condition names returned will vary depending on the sophistication of the server, but is expected to be less than 10 for most servers, making this interface more appropriate than a custom enumerator.

5.3.4.8 IOPCEventServer::QueryEventAttributes

```
HRESULT QueryEventAttributes(
    [in] DWORD dwEventCategory,
    [out] DWORD* pdwCount,
    [out, size_is(,*pdwCount)] DWORD** ppdwAttrIDs,
    [out, size_is(,*pdwCount)] LPWSTR** ppszAttrDescs
    [out, size_is(,*pdwCount)] VARTYPE** ppvtAttrTypes
    );
```

Description

Using the EventCategories returned by the QueryEventCategories method, client application can invoke the QueryEventAttributes method to get information about the vendor-specific attributes the server can provide as part of an event notification for an event within the specified event category. Simple servers may not support *any* vendor-specific attributes for some or even all EventCategories.

Attributes of event notifications are described in Section 2.5.2. Some possible vendor-specific attributes are included in Appendix C.

Parameters	**Description**
dwEventCategory	One of the Event Category codes returned from the QueryEventCategories function.
pdwCount	The number of event attributes (size of the AttrID, and AttrDescs, and AttrTypes arrays) returned by the function.
ppdwAttrIDs	Array of DWORD codes for vendor-specific event attributes associated with the event category and available from the server. These attribute IDs can be used in the event subscription interface to specify the information to be returned with an event notification.
ppszAttrDescs	Array of strings for the text names or descriptions for each of the event attribute IDs. This array corresponds to the AttrIDs array.
ppvtAttrTypes	Array of VARTYPES identifying the data type of each of the event attributes. This array corresponds to the AttrIDs array.

Return Codes

Return Code	**Description**
E_FAIL	The operation failed.
E_INVALIDARG	A bad parameter was passed.
E_OUTOFMEMORY	Not enough memory
S_OK	The operation succeeded.

Comments

All events of a particular event category have the potential of supporting the same attribute information. For event categories, where different instances of that category in the same server have different attributes, the server should return the union of all attributes and the client must allow for some attributes in event notifications to be null.

5.3.4.9 IOPCEventServer::TranslateToItemIDs

```
HRESULT TranslateToItemIDs(
    [in] LPWSTR szSource,
    [in] DWORD dwEventCategory
    [in] LPWSTR szConditionName,
    [in] LPWSTR szSubconditionName,
    [in] DWORD dwCount,
    [in, size_is(dwCount)] DWORD* pdwAssocAttrIDs,
    [out, size_is(,dwCount)] LPWSTR** ppszAttrItemIDs,
    [out, size_is(,dwCount)] LPWSTR** ppszNodeNames,
    [out, size_is(,dwCount)] CLSID** ppCLSIDs
    );
```

Description

Many OPC Alarm & Event servers are associated with OPC Data Access servers. Since these servers may provide a Data Access interface to some or all of the attributes associated with events, applications need the ability to determine the specific ItemID for one or more specific attribute ID codes given an associated source ID in order to be able to access the attribute via the Data Access interface. TranslateToItemIDs performs the required translation. This function will be useful for the case where the client wishes to use the OPC Data Access interface to subscribe to real-time data associated with a given event or alarm.

Given an event source, and an array of associated attribute ID codes, return an array of the item ID strings corresponding to each attribute ID. The event source, along with the associated attribute IDs are returned as part of the IOPCEventSink::OnEvent callback mechanism. Attribute ID codes and descriptions for a given event category can also be queried via the IOPCEventServer::QueryEventAttributes function. The server must return a NULL string for those attribute IDs that do not have a corresponding item ID.

Parameters	**Description**
szSource	An event source for which to return the item IDs corresponding to each of an array of attribute IDs if they exist. (From OnEvent or from IOPCEventAreaBrowser)
dwEventCategory	A DWORD event category code indicating the category of events for which item IDs are to be returned. (From OnEvent or from QueryEventCategories)
szConditionName	The name of a condition within the event category for which item IDs are to be returned. (From OnEvent or from QueryConditionNames)
szSubconditionName	The name of a sub-condition within a multi-state condition. (From OnEvent or from QuerySubconditionNames) This should be a NULL string for a single state condition.
dwCount	The number of event attribute IDs (size of the AssocAttrIDs array) passed into the function.
ppdwAssocAttrIDs	Array of DWORD IDs of vendor-specific event attributes associated with the generator ID and available from the server for which to return ItemIDs. Note: these attribute IDs are returned by the IOPCEventSink::OnEvent callback, and are selected via the IOPCEventSubscriptionMgt::SelectReturnedAttributes method.

ppszAttrItemIDs	Array of item ID strings corresponding to each event attribute ID associated with the generator ID. This array is the same length as the AssocAttrIDs array passed into the function. A Null string is returned if no item ID is available for this attribute.
ppszNodeNames	Array of network node names of the associated OPC Data Access Servers. A Null string is returned if the OPC Data Access Server is running on the local node.
ppCLSIDs	Array of class IDs for the associated OPC Data Access Servers.

Return Codes

Return Code	**Description**
E_FAIL	The operation failed.
E_NOTIMPL	This capability not implemented by this server.
E_INVALIDARG	A bad parameter was passed.
E_OUTOFMEMORY	Not enough memory
S_OK	The operation succeeded.

Comments

5.3.4.10 IOPCEventServer::GetConditionState

```
HRESULT GetConditionState (
    [in]  LPWSTR szSource,
    [in]  LPWSTR szConditionName,
    [in]  DWORD dwNumEventAttrs,
    [in, size_is(dwNumEventAttrs)] DWORD* pdwAttributeIDs,
    [out] OPCCONDITIONSTATE ** ppConditionState
    );
```

Description

Returns the current state information for the condition instance corresponding to the szSource and szConditionName. The OPCCONDITIONSTATE structure is defined below. See section 2.4 for a discussion of conditions and their states.

Parameters	**Description**
szSource	A source name, as returned by the IOPCEventAreaBrower::GetQualifiedSourceName method. The state of the condition instance associated with this source is returned.
szConditionName	A condition name, as returned by the QueryConditionNames method. The state of this condition is returned.
dwNumEventAttrs	The requested number of event attributes to be returned in the OPCCONDITIONSTATE structure. Can be zero if no attributes are desired
pdwAttributeIDs	The array of Attribute IDs indicating which event attributes should be returned in the OPCCONDITIONSTATE structure.
ppConditionState	Pointer to where the OPCCONDITIONSTATE structure pointer should be returned. The server allocates the structure.

Return Codes

Return Code	Description
E_FAIL	The operation failed.
E_OUTOFMEMORY	Not enough memory
E_INVALIDARG	An argument to the function was invalid.
E_NOTIMPL	This method is not implemented by this server.
OPC_E_NOINFO	Although this server implements this method and the specified condition name is valid, no information is currently available for this condition. Such a situation may arise for servers which maintain condition state information only for active or unacknowledged conditions.
S_OK	The operation succeeded.

Comments

Client must free the structure.

Some servers may not maintain sufficient condition state information to fully implement this method. In this case, the server should return E_NOTIMPL. If a server chooses to implement this method, it must return valid information for every member of OPCCONDITIONSTATE.

5.3.4.10.1 OPCCONDITIONSTATE

```
typedef  struct {
        WORD        wState;
        LPWSTR      szActiveSubCondition;
        LPWSTR      szASCDefinition;
        DWORD       dwASCSeverity;
        LPWSTR      szASCDescription;
        WORD        wQuality;
        FILETIME    ftLastAckTime;
        FILETIME    ftSubCondLastActive;
        FILETIME    ftCondLastActive;
        FILETIME    ftCondLastInactive;
        LPWSTR      szAcknowledgerID;
        LPWSTR      szComment;
        DWORD       dwNumSCs;
        [size_is (dwNumSCs)] LPWSTR * pszSCNames;
        [size_is (dwNumSCs)] LPWSTR * pszSCDefinitions;
        [size_is (dwNumSCs)] DWORD  * pdwSCSeverities;
        [size_is (dwNumSCs)] LPWSTR * pszSCDescriptions;
        DWORD       dwNumEventAttrs;
        [size_is(dwNumEventAttrs)] VARIANT*      pEventAttributes;
        [size_is(dwNumEventAttrs)] HRESULT*      pErrors;
} OPCCONDITIONSTATE;
```

Member	Description
wState	A WORD bit mask of three bits specifying the new state of the condition: OPC_CONDITION_ACTIVE, OPC_CONDITION_ENABLED,

	OPC_CONDITION_ACKED.
szActiveSubCondition	The name of the currently active sub-condition, for multi-state conditions which are active. For a single-state condition, this contains the condition name. For inactive conditions, this value is NULL.
szASCDefinition	An expression which defines the sub-state represented by the szActiveSubCondition, for multi-state conditions. For a single state condition, the expression defines the state represented by the condition. For inactive conditions, this value is NULL.
dwASCSeverity	The severity of any event notification generated on behalf of the szActiveSubCondition (0..1000). See section 2.4.3.2. For inactive conditions, this value is 0.
szASCDescription	The text string to be included in any event notification generated on behalf of the szActiveSubCondition. For inactive conditions, this value is NULL.
wQuality	Quality associated with the condition state. See Section 2.4.2.1. Values are as defined for the OPC Quality Flags in the OPC Data Access Server specification.
ftLastAckTime	The time of the most recent acknowledgment of this condition (of any sub-condition). Contains 0 if the condition has never been acknowledged.
ftSubCondLastActive	Time of the most recent transition into szActiveSubCondition. This is the time value which must be specified when acknowledging the condition. Contains 0 if the condition has never been active.
ftCondLastActive	Time of the most recent transition into the condition. There may be transitions among the sub-conditions which are more recent. Contains 0 if the condition has never been active.
ftCondLastInactive	Time of the most recent transition out of this condition. Contains 0 if the condition has never been active, or if it is currently active for the first time and has never been exited.
szAcknowledgerID	This is the ID of the client who last acknowledged this condition. Contains NULL if the condition has never been acknowledged.
szComment	The comment string passed in by the client who last acknowledged this condition. Contains NULL if the condition has never been acknowledged.
dwNumSCs	The number of sub-conditions defined for this condition. For multi-state conditions, this value will be

	greater than one. For single-state conditions, this value will be 1.
pszSCNames	Pointer to an array of sub-condition names defined for this condition. For single-state conditions, the array will contain one element, the value of which is the condition name (see Section 2.4.3).
pszSCDefinitions	Pointer to an array of sub-condition definitions (see Section 2.4.3).
pdwSCSeverities	Pointer to an array of sub-condition severities (see Section 2.4.3).
pszSCDefinitions	Pointer to an array of sub-condition definitions (see Section 2.4.3).
dwNumEventAttrs	The length of the arrays **pEventAttributes** and **pErrors**. Must be equal to **dwNumEventAttrs** passed into function **GetConditionState().**
pEventAttributes	Pointer to an array of vendor specific attributes associated with that latest event notification for this condition. The order of the items returned matches the order that was specified by **pdwAttributeIDs**. If a server cannot provide reasonable data for an attribute, the returned VARIANT should be set to VT_EMPTY.
pErrors	Pointer to an array of HRESULT values for each requested attribute ID specified by **pdwAttributeIDs**. Servers should return S_OK if the Attribute ID is valid or E_FAIL if not.

State Values

State	**Value**	**Description**
OPC_CONDITION_ACTIVE	1	The condition has become active.
OPC_CONDITION_ENABLED	2	The condition has been enabled.
OPC_CONDITION_ACKED	4	The condition has been acknowledged.

5.3.4.11 IOPCEventServer::EnableConditionByArea

```
HRESULT EnableConditionByArea(
    [in] DWORD dwNumAreas,
    [in, size_is(dwNumAreas)] LPWSTR* pszAreas
    );
```

Description

Places all conditions for all sources within the specified process areas into the enabled state. Therefore, the server will now generate condition-related events for these conditions.

The effect of this method is global within the scope of the event server. Therefore, if the server is supporting multiple clients, the conditions are enabled for all clients, and they will begin receiving the associated condition-related events.

Parameters	Description
dwNumAreas	The number of process areas for which conditions are to be enabled.
pszAreas	An array of area names, as returned by IOPCEventAreaBrowser::GetQualifiedAreaName.

Return Codes

Return Code	Description
E_FAIL	The operation failed.
E_INVALIDARG	One or more of the specified arguments is not valid.
E_NOTIMPL	The server does not support this method.
S_OK	The operation succeeded.

Comments

Because of the global effect of this method, some event server implementers may choose not to implement it. In this case, the server should return E_NOTIMPL.

A condition may be associated with multiple sources (see Section 2.4). These sources may be distributed among multiple areas. Enabling the conditions in one area does not change the enabled/disabled state of conditions of the same name, which are associated with sources in other areas. For example, the “LevelAlarm” condition may be enabled for sources in “Area1” and disabled for sources in “Area2”.

5.3.4.12 IOPCEventServer::EnableConditionBySource

```
HRESULT EnableConditionBySource(
    [in] DWORD dwNumSources,
    [in, size_is(dwNumSources)] LPWSTR* pszSources
    );
```

Description

Places all conditions for the specified event sources into the enabled state. Therefore, the server will now generate condition-related events for these conditions.

The effect of this method is global within the scope of the event server. Therefore, if the server is supporting multiple clients, the conditions are enabled for all clients, and they will begin receiving the associated condition-related events.

Parameters	**Description**
dwNumSources	The number of event sources for which conditions are to be enabled.
pszSources	An array of source names, as returned by IOPCEventAreaBrowser::GetQualifiedSourceName

Return Codes

Return Code	**Description**
E_FAIL	The operation failed.
E_INVALIDARG	One or more of the specified arguments is not valid.
E_NOTIMPL	The server does not support this method.
S_OK	The operation succeeded.

Comments

Because of the global effect of this method, some event server implementers may choose not to implement it. In this case, the server should return E_NOTIMPL.

A condition may be associated with multiple sources (see Section 2.4). Enabling conditions associated with one source does not change the enabled/disabled state of conditions of the same name, which are associated with other sources. For example, the “LevelAlarm” condition may be enabled for “A100” and disabled for “FIC101”.

5.3.4.13 IOPCEventServer::DisableConditionByArea

```
HRESULT DisableConditionByArea(
    [in] DWORD dwNumAreas,
    [in, size_is(dwNumAreas)] LPWSTR* pszAreas
    );
```

Description

Places all conditions for all sources within the specified process areas into the disabled state. Therefore, the server will now cease generating condition-related events for these conditions.

The effect of this method is global within the scope of the event server. Therefore, if the server is supporting multiple clients, the conditions are disabled for all clients, and they will stop receiving the associated condition-related events.

Parameters	**Description**
dwNumAreas	The number of process areas for which conditions are to be disabled.
pszAreas	An array of area names, as returned by IOPCEventAreaBrowser::GetQualifiedAreaName

Return Codes

Return Code	**Description**
E_FAIL	The operation failed.
E_INVALIDARG	One or more of the specified arguments is not valid.
E_NOTIMPL	The server does not support this method.
S_OK	The operation succeeded.

Comments

Because of the global effect of this method, some event server implementers may choose not to implement it. In this case, the server should return E_NOTIMPL.

A condition may be associated with multiple sources (see Section 2.4). These sources may be distributed among multiple areas. Disabling the conditions in one area does not change the enabled/disabled state of conditions of the same name, which are associated with sources in other areas. For example, the “LevelAlarm” condition may be enabled for sources in “Area1” and disabled for sources in “Area2”.

5.3.4.14 IOPCEventServer::DisableConditionBySource

```
HRESULT DisableConditionBySource(
    [in] DWORD dwNumSources,
    [in, size_is(dwNumSources)] LPWSTR* pszSources
    );
```

Description

Places all conditions for the specified event sources into the disabled state. Therefore, the server will no longer generate condition-related events for these conditions.

The effect of this method is global within the scope of the event server. Therefore, if the server is supporting multiple clients, the conditions are disabled for all clients, and they will stop receiving the associated condition-related events.

Parameters	**Description**
dwNumSources	The number of event sources for which conditions are to be disabled.
pszSources	An array of source names, as returned by IOPCEventAreaBrowser::GetQualifiedSourceName

Return Codes

Return Code	**Description**
E_FAIL	The operation failed.
E_INVALIDARG	One or more of the specified arguments is not valid.
E_NOTIMPL	The server does not support this method.
S_OK	The operation succeeded.

Comments

Because of the global effect of this method, some event server implementers may choose not to implement it. In this case, the server should return E_NOTIMPL.

A condition may be associated with multiple sources (see Section 2.4). Disabling conditions associated with one source does not change the enabled/disabled state of conditions of the same name, which are associated with other sources. For example, the “LevelAlarm” condition may be enabled for “A100” and disabled for “FIC101”.

5.3.4.15 IOPCEventServer::AckCondition

```
HRESULT AckCondition(
    [in] DWORD dwCount
    [in, string] LPWSTR szAcknowledgerID,
    [in, string] LPWSTR szComment,
    [in, size_is(dwCount)] LPWSTR* pszSource,
    [in, size_is(dwCount)] LPWSTR* pszConditionName,
    [in, size_is(dwCount)] FILETIME* pftActiveTime,
    [in, size_is(dwCount)] DWORD* pdwCookie,
    [out, size_is(,dwCount)] HRESULT **ppErrors
    );
```

Description

The client uses the AckCondition method to acknowledge one or more conditions in the Event Server. The client receives event notifications from conditions via the IOPCEventSink::OnEvent callback. This AckCondition method specifically acknowledges the condition becoming active or transitioning into a different sub-condition (and no other state transition of the condition). One or more conditions belong to a specific event source – the source of the event notification. For each condition-related event notification, the corresponding Source, Condition Name, Active Time and Cookie is received by the client as part of the OnEvent callback parameters.

Parameters	**Description**
dwCount	The number of acknowledgments passed with this function.
szAcknowledgerID	A string passed in by the client, identifying who is acknowledging the conditions. This is an attribute (AcknowledgerID) of the condition that identifies who acknowledged the condition. This is just a string generated by the client. This is also also included as the ActorID in the acknowledgment event notification sent to all subscribing clients. A NULL string is not allowed, since a NULL AcknowledgerID indicates that the event was automatically acknowledged by the server.
szComment	Comment string passed in by the client associated with acknowledging the conditions. A NULL string indicating no comment is allowed.
pszSource	Array of event source strings identifying the source (or owner) of each condition that is being acknowledged, e.g. FIC101. Sources are passed to the client in the szSource member of the ONEVENTSTRUCT by the IOPCEventSink::OnEvent callback.
pszConditionName	Array of Condition Name strings identifying each condition that is being acknowledged. Condition Names are unique within the scope of the event server. Examples of Condition Names might be “LevelAlarm” or “Deviation”. Condition Names are passed to the client in the szConditionName member of the ONEVENTSTRUCT by the IOPCEventSink::OnEvent callback.

pftActiveTime	Array of active times corresponding to each Source and ConditionName pair. This parameter uniquely identifies a specific transition of the condition to the active state or into a different sub-condition and is the same as the SubCondLastActive condition attribute. Active Times are passed to the client in the ftActiveTime member of the ONEVENTSTRUCT by the IOPCEventSink::OnEvent callback. If the condition has become active again or transitioned into a different sub-condition at a later time, this acknowledgment will be ignored.
pdwCookie	Array of server supplied "cookies" corresponding to each Source and Condition Name pair, that in addition to the Active Time, uniquely identifies a specific event notification. Cookies are passed to the client in the dwCookie member of the ONEVENTSTRUCT by the IOPCEventSink::OnEvent callback. The client is responsible for returning the same cookie parameter, received in the event notification, back to the server in the condition acknowledgment.
ppErrors	Array of HRESULTS indicating the success of the individual acknowledgments. The errors correspond to the Source and ConditionName pairs passed in to the method.

Return Codes

Return Code	**Description**
E_FAIL	The operation failed.
E_INVALIDARG	A bad parameter was passed. (szAcknowledgerID is a NULL string)
E_OUTOFMEMORY	Not enough memory.
S_OK	The operation succeeded.
S_FALSE	One or more of ppErrors in not S_OK.

ppError Codes

Return Code	**Description**
S_OK	The acknowledgment succeeded for the corresponding Source and ConditionName pair.
OPC_S_ALREADYACKED	The condition has already been acknowledged.
OPC_E_INVALIDTIME	Time does not match latest active time. The pftActiveTime did not match the current SubCondLastActive attribute of the condition.
E_INVALIDARG	A bad parameter was passed. (source, condition name or cookie)

Comments

The client is required to pass the ftActiveTime and dwCookie received from the IOPCEventSink::OnEvent callback to the AckCondition method without modification.

5.3.4.16 IOPCEventServer::CreateAreaBrowser

```
HRESULT CreateAreaBrowser(
    [in] REFIID riid,
    [out, iid_is(riid) LPUNKNOWN* ppUnk
    );
```

Description

Create an OPCEventAreaBrowser object on behalf of this client and return the interface to the Client. This object will support the IUnknown and IOPCEventAreaBrowser interfaces. The client can use this interface to browse the process areas available from the server as described in the IOPCEventAreaBrowser interface shown below.

If the OPC Event Server does not support browsing of the process area space, then this method will fail.

The client may create multiple OPCEventAreaBrowser objects in order to support concurrent access to multiple levels, in the case of a hierarchical area name space.

The OPCEventAreaBrowser uses conventional reference counting and thus will be deleted with all interfaces to it are released.

Parameters	**Description**
riid	The type of interface desired (e.g. IID_IOPCEventAreaBrowser)
ppUnk	Where to store the returned interface pointer. NULL is returned for any HRESULT other than S_OK.

Return Codes

Return Code	**Description**
E_FAIL	The operation failed.
E_OUTOFMEMORY	Not enough memory
E_INVALIDARG	Bad argument was passed.
E_NOTIMPL	The server does not support area browsing.
S_OK	The operation succeeded.

Comments

5.3.5 IConnectionPointContainer

The general principles of ConnectionPoints are not discussed here as they are covered very clearly in the Microsoft Documentation. The reader is assumed to be familiar with this technology.

Likewise the details of the IEnumConnectionPoints, IConnectionPoint and IEnumConnections interfaces and their proper use in this context are well defined by Microsoft and are not discussed here.

The IConnectionPointContainer interface discussed here is implemented on the OPCEventServer object. In theory, the Advise and Unadvise methods of the connection points could be implemented within the IOPCEventServer interface. However use of a separate ConnectionPoint implementation is more in keeping with state of the art Microsoft implementations.

The IOPCShutdown callback object implemented by the client application is assumed to service a single Event Server, since no identification information is passed to the client.

Note: OPC Compliant servers are not required to support more than one connection between each Event Server Object. Given this, it is expected that a single connection will be sufficient for virtually all applications. For this reason (as per Microsoft Recommendations) the EnumConnections method for the IConnectionPoint interface for IOPCShutdown::ShutdownRequest callback is allowed to return E_NOTIMPL.

EnumConnectionPoints

See the Microsoft documentation for a description of this method.

OPC Event Servers must return an enumerator that includes IOPCShutdown. Additional vendor specific callbacks are also allowed.

FindConnectionPoint

See the Microsoft documentation for a description of this method.

OPC Event Servers must support IID_ IOPCShutdown. Additional vendor specific callbacks are also allowed.

5.3.6 IConnectionPoint

An IConnectionPoint for IOPCShutdown is returned from the Event Server's ConnectionPointContainer. Refer to the Microsoft documentation of this interface for additional information on its methods, which included Advise and Unadvise.

5.4 OPCEventAreaBrowser Object (optional)

The OPCEventAreaBrowser is the object that an OPC Event server supplies to manage browsing the process area space of the server. The interfaces that this object provides include:

- IUnknown
- IOPCEventAreaBrowser

This object is optional, and may not be supported by simple event servers.

5.4.1 IOPCEventAreaBrowser

This interface provides a way for clients to browse the process area organization implemented by the server. The expectation is that events and conditions available in the server are organized in one or more process areas, and the client can filter event subscriptions by specifying the process areas to limit the event notifications sent by the server. These areas are for use in specifying event filters (see the IOPCEventSubscriptionMgt interface below). They are logically independent of the IOPCBrowseServerAddressSpace of the OPC Data Access interfaces and associated ItemIDs. The relationship between the Server Address Space and the Server process area space is completely up to the server implementation.

Note that the reason for making this a set of methods rather than an ActiveX control is to allow it to more easily be integrated with other browsing methods and address spaces that the Client may already be dealing with.

Note that this interface behaves very much like an Enumerator in that it creates an object 'behind the scenes' and maintains state information (the current position in the address hierarchy) on behalf of the client.

Here is an overview of how this interface is used:

The browse position is initially set to the 'root' of the area space. The client can optionally choose a starting point by calling ChangeBrowsePosition. For a HIERARCHICAL space the client may pass any partial path (although the client will typically pass a NULL string to indicate the root). This sets an initial position from which to browse up or down.

The Client can browse the items below (contained in) the current position via BrowseOPCAreas. For a hierarchical space you can specify AREA (which returns only areas on that level) or SOURCE (which returns only sources on that level). A String enumerator is returned.

This browse can also be filtered by a vendor specific filter string.

Note that in a hierarchy, the enumerator will return 'short' strings; the name of the 'child'. These short strings will generally not be sufficient for the Area List array of the event subscription filter. The client should always convert this short string to a 'fully qualified' string via GetQualifiedAreaName or GetQualifiedSourceName. For example the short string might be REACTOR5; the fully qualified string might be AREA1.REACTOR5.

If the client browsed for AREAs then the result (short string) may be passed to ChangeBrowsePosition to move 'down'. This method can also move 'up' in which case the short string is not used.

5.4.1.1 IOPCEventAreaBrowser::ChangeBrowsePosition

```
HRESULT ChangeBrowsePosition(
    [in]  OPCAEBROWSEDIRECTION dwBrowseDirection,
    [in, string] LPCWSTR  szString
     );
```

Description

Provides a way to move ‘up’ or ‘down’ in a hierarchical space from the current position, or a way to move to a specific position in the area space tree. The target szString must represent an area, rather than a source.

Parameters	**Description**
dwBrowseDirection	OPCAE_BROWSE_UP,OPCAE_BROWSE_DOWN or OPCAE_BROWSE_TO
szString	For DOWN, the partial area name of the area to move into. This would be one of the strings returned from BrowseOPCAreas. For UP this parameter is ignored and should point to a NULL string. For BROWSE_TO, the fully qualified area name (as obtained from GetQualifiedAreaName method) or NULL to go to the root.

Return Codes

Return Code	**Description**
E_FAIL	The function failed
E_INVALIDARG	Bad Direction or String.
OPC_E_INVALIDBRANCHNAME	szString is not a recognized area name.
S_OK	The function was successful

Comments

An error is returned if the passed string does not represent an area.

5.4.1.2 IOPCEventAreaBrowser::BrowseOPCAreas

```
HRESULT BrowseOPCAreas(
    [in] OPCAEBROWSETYPE   dwBrowseFilterType,
    [in, string] LPCWSTR  szFilterCriteria,
    [out] LPENUMSTRING  * ppIEnumString
     );
```

Description

Return an IEnumString for a list of Areas as determined by the passed parameters. The position from which the browse is done can be set via the ChangeBrowsePosition.

Parameters	**Description**
dwBrowseFilterType	OPC_AREA - returns only areas. OPC_SOURCE - returns only sources.
szFilterCriteria	A server specific filter string. See Appendix A for the definition of the syntax which must be supported by all servers. The implementer may extend this syntax to provide additional capabilities. A NULL string indicates no filtering.
ppIEnumString	Where to save the returned interface pointer. NULL if the HRESULT is other than S_OK

Return Codes

Return Code	**Description**
E_FAIL	The function failed
S_FALSE	There is nothing to enumerate
E_INVALIDARG	A bad parameter was passed.
E_OUTOFMEMORY	Not enough memory
S_OK	The function was successful

Comments

The returned enumerator may be empty if no Areas or Sources satisfied the filter constraints. The strings returned by the enumerator represent the Areas or Sources contained in the current level. They do not include ?? and delimiter or “parent” names.

Clients are allowed to create and hold multiple enumerators in order to maintain more than one “browse position” at a time. Changing the browse position in one enumerator will not affect any other enumerator the client has created. The client must release each enumerator when finished with it.

5.4.1.3 IOPCEventAreaBrowser::GetQualifiedAreaName

```
HRESULT GetQualifiedAreaName(
    [in] LPCWSTR szAreaName,
    [out , string] LPWSTR *pszQualifiedAreaName
    );
```

Description

Provides a mechanism to assemble a fully qualified Area name in a hierarchical space. This is required since at each point one is browsing just the names below the current node.

Parameters	Description
szAreaName	The name of an Area at the current level, obtained from the string enumerator returned by BrowseOPCAreas with a BrowseFilterType of OPC_AREA.
pszQualifiedAreaName	Where to return the resulting fully qualified area name.

Return Codes

Return Code	Description
E_FAIL	The function failed
E_INVALIDARG	A bad parameter was passed.
E_OUTOFMEMORY	Not enough memory
S_OK	The function was successful

Comments

The server must return strings that can be added to the pszAreaList for the IOPCEventSubscriptionMgt::SetFilter method, and can be used in the IOPCEventAreaBrowser::ChangeBrowsePosition method to move to a specific place in the process area space tree.

5.4.1.4 IOPCEventAreaBrowser::GetQualifiedSourceName

```
HRESULT GetQualifiedSourceName(
    [in] LPCWSTR szSourceName,
    [out , string] LPWSTR *pszQualifiedSourceName
    );
```

Description

Provides a mechanism to assemble a fully qualified Source name in a hierarchical space. This is required since at each point one is browsing just the names below the current node.

Parameters	**Description**
szSourceName	The name of a Source at the current level, obtained from the string enumerator returned by BrowseOPCAreas with a BrowseFilterType of OPC_SOURCE.
pszQualifiedSourceName	Where to return the resulting fully qualified source name.

Return Codes

Return Code	**Description**
E_FAIL	The function failed
E_INVALIDARG	A bad parameter was passed.
E_OUTOFMEMORY	Not enough memory
S_OK	The function was successful

Comments

The server must return strings that can be added to pszSources for the IOPCEventServer::EnableConditionBySource method.

5.5 OPCEventSubscription Object

The OPCEventSubscription object is the object that an OPC Event server delivers to manage a single event subscription. It is created by invoking IOPCEventServer::CreateEventSubscription. This object provides the following interfaces:

- IUnknown
- IOPCEventSubscriptionMgt
- IConnectionPointContainer

In addition, OPCEventSubscription contains an IID_IOPCEventSink object which supports the IConnectionPoint interface.

Each subscription between a client and server will have only one filter, though that filter can include several criteria. Clients can implement multiple filters using multiple subscriptions, each with their own filter. When the subscription is established, a default filter is created that is equivalent to “no filtering” i.e. send all event notifications.

The criteria for defining the scope of the filter is to eliminate the majority of events a client is not interested in, without having to be exhaustive. The primary reason for the filter is to reduce unnecessary communication overhead and to improve performance. The most important filtering criteria then are severity and process area. This filter mechanism provides a set of filter criteria that are simple yet powerful - but do not cover every possible specific type of filter the client may wish for. The client can do additional filtering on received event notifications, further customizing exactly which event notifications are displayed or stored.

The functionality provided by each of these interfaces is defined in this section.

5.5.1 IOPCEventSubscriptionMgt

This interface specifies how to manage a particular subscription to OPC event information. It is used to specify criteria for selecting events of interest, to specify vendor-specific information to be returned in event notifications, and to request a refresh of selected conditions.

5.5.1.1 IOPCEventSubscriptionMgt::SetFilter

```
HRESULT SetFilter(
    [in] DWORD  dwEventType,
    [in] DWORD dwNumCategories,
    [in, size_is(dwNumCategories)] DWORD* pdwEventCategories,
    [in] DWORD dwLowSeverity,
    [in] DWORD dwHighSeverity,
    [in] DWORD dwNumAreas,
    [in, size_is(dwNumAreas)] LPWSTR* pszAreaList,
    [in] DWORD dwNumSources,
    [in, size_is(dwNumSources] LPWSTR* pszSourceList
    );
```

Description

Sets the filtering criteria to be used for the event subscription.

Events may be selected using the following criteria:

- Type of event, i.e. simple, condition, or tracking.
- Event categories
- Lowest severity, i.e. all events with a severity greater than or equal to the specified severity.
- Highest severity, i.e. all events with a severity less than or equal to the specified severity.
- Process areas
- Event Sources

A list of values for a single criterion are logically ORed together (e.g. if two event categories are specified, event notifications for both categories will be received). If multiple criteria are specified, they will be logically ANDed together, i.e. only those events satisfying all criteria will be selected. An example is specifying both lowest severity and highest severity will result in the selection of events with severities lying between the two values.

An OPCEventSubscription object has only one filter.

Parameters	**Description**
dwEventType	A DWORD bit mask specifying which event types are of interest; OPC_SIMPLE_EVENT, OPC_CONDITION_EVENT, OPC_TRACKING_EVENT, OPC_ALL_EVENTS. These types can be OR'ed together to filter multiple types. A value of 0 is an error and E_INVALIDARG will be returned.
dwNumCategories	Length of array of event categories. A length of 0 indicates all categories should be included in the filter.
pdwEventCategories	Array of event categories of interest. These are DWORD event category codes returned by IOPCEventServer::QueryEventCategories. A NULL pointer should be specified if dwNumCategories is 0.
dwLowSeverity	Lowest severity of interest. To receive events of all severities, set dwLowSeverity to 0.

dwHighSeverity	Highest severity of interest. To receive events of all severities, set dwHighSeverity to 1000. The server is responsible for mapping its internal severity levels to evenly span the 0 to 1000 range.
dwNumAreas	Length of array of areas. A length of 0 indicates all areas should be included in the filter.
pszAreaList	Array of process area strings of interest - only events or conditions in these areas will be reported. Area strings can be obtained using IOPCEventAreaBrowser::GetArea. A NULL pointer should be specified if dwNumAreas is 0.
dwNumSources	Length of array of event sources. A length of 0 indicates all sources should be included in the filter.
pszSourceList	Array of event sources of interest - only events from these sources will be reported. It is possible to specify sources using the wildcard syntax described in Appendix A. A NULL pointer should be specified if dwNumSources is 0.

Return Codes

Return Code	Description
E_FAIL	The operation failed.
E_INVALIDARG	A bad parameter was passed.
E_OUTOFMEMORY	Not enough memory
OPC_E_BUSY	A refresh operation is currently in progress on this event subscription object.
S_OK	The operation succeeded.
S_FALSE	One or more of the specified filter criteria were ignored.

Comments

Servers may not support all the various filter criteria. The specific filter criteria supported by a given server can be determined via the IOPCEventServer::QueryAvailableFilters method. If a filter criterion is specified that is not supported by the server, it will ignore that filter criterion and return S_FALSE.

Note that for a given condition, if the event notifications corresponding to acknowledge or return to normal have different severity levels than the event notification for the condition becoming active, it is possible that the client may receive one set of notifications but not the others due to filtering by severity.

5.5.1.2 IOPCEventSubscriptionMgt::GetFilter

```
HRESULT GetFilter(
    [out] DWORD* pdwEventType,
    [out] DWORD* pdwNumCategories,
    [out, size_is(,*pdwNumCategories)] DWORD** ppdwEventCategories,
    [out] DWORD* pdwLowSeverity,
    [out] DWORD* pdwHighSeverity,
    [out] DWORD* pdwNumAreas,
    [out, size_is(,*pdwNumAreas)] LPWSTR** ppszAreaList
    [out] DWORD* pdwNumSources,
    [out, size_is(,*pdwNumSources)] LPWSTR** ppszSourceList
     );
```

Description

Returns the filter currently in use for event subscriptions.

Parameters	**Description**
pdwEventType	A DWORD bit map specifying which event types are of allowed through the filter; OPC_SIMPLE_EVENT, OPC_CONDITION_EVENT, OPC_TRACKING_EVENT, OPC_ALL_EVENTS. These types can be OR'ed together to filter multiple types.
pdwNumCategories	Length of the event category array returned. A length of 0 indicates an empty array.
ppdwEventCategories	Array of event categories for the filter.
pdwLowSeverity	Lowest severity allowed through filter. If the server does not support filtering on severity, the returned value will be 0.
pdwHighSeverity	Highest severity allowed through filter. If the server does not support filtering on severity, the returned value will be 1000.
pdwNumAreas	Length of the area list array returned. A length of 0 indicates an empty array.
ppszAreaList	List of process areas for the filter.
pdwNumSources	Length of the event source list returned. A length of 0 indicates an empty array.
ppszSourceList	List of sources for the filter.

Return Codes

Return Code	**Description**
E_FAIL	The operation failed.
E_INVALIDARG	A bad parameter was passed.
E_OUTOFMEMORY	Not enough memory
S_OK	The operation succeeded.

Comments

If a server does not support one or more of the filter criteria requested in SetFilter, it returns empty arrays for lists, and values which indicate no filtering is taking place for non-list items. In these cases, it does not return any filters which may have been requested in SetFilter, but which were ignored.

5.5.1.3 IOPCEventSubscriptionMgt::SelectReturnedAttributes

```
HRESULT SelectReturnedAttributes(
    [in] DWORD dwEventCategory,
    [in] DWORD dwCount,
    [in, size_is(dwCount)] DWORD* dwAttributeIDs,
    );
```

Description

For each Event Category, SelectReturnedAttributes sets the attributes to be returned with event notifications in the IOPCEventSink::OnEvent callback.

This method can be called multiple times in order to specify the attributes to return for each unique event type and event category pair. For a given event type and event category pair, the attributes returned can be "cleared" by setting the dwCount parameter to zero. If this is called multiple times for the same event type and event category pair, then the latest call will be in effect.

Parameters	**Description**
dwEventCategory	The specific event category for which the list of attributes applies. These are returned from the IOPCEventServer::QueryEventCategories method.
dwCount	The size of the attribute IDs array.
dwAttributeIDs	The list IDs of the attributes to return with event notifications for the event type and event category specified. These are returned from the IOPCEventServer::QueryEventAttributes method.

Return Codes

Return Code	**Description**
E_FAIL	The operation failed.
E_INVALIDARG	A bad parameter was passed.
S_OK	The operation succeeded.

Comments

5.5.1.4 IOPCEventSubscriptionMgt::GetReturnedAttributes

```
HRESULT GetReturnedAttributes(
    [in] DWORD dwEventCategory,
    [out] DWORD * pdwCount,
    [out, size_is(,pdwCount)] DWORD* pdwAttributeIDs,
    );
```

Description

For each Event Category, GetReturnedAttributes retrieves the attributes which are currently specified to be returned with event notifications in the IOPCEventSink::OnEvent callback. All retrieved attributes have been specified by previous calls to SelectReturnedAttributes.

Parameters	Description
dwEventCategory	The specific event category for which to retrieve the list of attributes.
pdwCount	The size of the attribute IDs array which is being returned. Is set to zero if no attributes are currently specified.
dwAttributeIDs	The list IDs of the attributes which are currently specified to be returned with event notifications for the event type and event category specified.

Return Codes

Return Code	Description
E_FAIL	The operation failed.
E_INVALIDARG	A bad parameter was passed.
S_OK	The operation succeeded.

Comments

5.5.1.5 IOPCEventSubscriptionMgt::Refresh

```
HRESULT Refresh(
    [in] DWORD dwConnection,
    );
```

Description

Force a refresh for all active conditions and inactive, unacknowledged conditions whose event notifications match the filter of the event subscription.

Clients will often need to get the current condition information from the server, particularly at client startup, for things such as a current alarm summary. The OPC Event Server supports this requirement by resending the most recent event notifications which satisfy the filter in the event subscription and which are related to active and/or unacknowledged conditions. The client can then derive the current condition status from the "refreshed" event notifications.

Parameters	**Description**
dwConnection	The OLE Connection number returned from IConnectionPoint::Advise. This is passed to help the server determine which OPC event sink to call when the request completes.

HRESULT Return Codes

Return Code	**Description**
S_OK	The function was successful.
OPC_E_BUSY	There is currently another refresh in progress on this event subscription.
E_FAIL	The function was unsuccessful.

Comments

When the client needs a refreshed list of active conditions, it will request a "refresh" from the server. The server will send event notifications to that specific client indicating that they are "refresh" instead of "original" event notifications. Since the client only needs to get the current state information for conditions, only condition events will be refreshed. ***Note: "Refresh" is not a general "replay" capability since the server is not required to maintain an event history. Refresh is only for updating the client's state information for active or unacknowledged conditions***. See section 2.6, *Subscriptions to Event Notifications*.

In addition to the refresh indicator, there may be other differences between original and refresh event notifications. Specifically, since some attribute information available at the time of the original event notification may be unavailable at the time of the refresh, some attributes in the refresh may be null.

Refresh event notifications and original event notifications will not be mixed in the same invocation of the event callback, though refresh and original event callback invocations may be interleaved. Thus, it is the responsibility of the client to check time stamps on the event notifications and put them into the correct order, to ensure correct condition status is obtained.

The client will receive the maximum number of event notifications per single callback, according to the specification in the IOPCEventServer::CreateEventSubscription method. When sending refresh event notifications, the server will indicate if there are more refresh event notifications to send (see the bLastRefresh parameter of IOPCEventSink::OnEvent).

This method is applicable to condition-related events only. Notifications for simple events and tracking events are not returned, even if they would satisfy the filter of the event subscription.

This method is applicable both when the subscription is active and when it is inactive (see the discussion of the pbActive flag for the SetState method).

5.5.1.6 IOPCEventSubscriptionMgt::CancelRefresh

```
HRESULT CancelRefresh(
    [in] DWORD dwConnection,
    );
```

Description

Cancels a refresh in progress for the event subscription.

If a refresh is in progress, the server should send one final callback with the last refresh flag set and the number of events equal to zero.

Parameters	**Description**
dwConnection	The OLE Connection number returned from IConnectionPoint::Advise. This is passed to help the server determine which OPC event sink to call when the request completes.

HRESULT Return Codes

Return Code	**Description**
S_OK	The function was successful.
E_FAIL	The function was unsuccessful.

Comments

5.5.1.7 IOPCEventSubscriptionMgt::GetState

```
HRESULT GetState(
    [out] BOOL * pbActive,
    [out] DWORD * pdwBufferTime,
    [out] DWORD * pdwMaxSize,
    [out] OPCHANDLE * phClientSubscription,
    );
```

Description

Get the current state of the subscription. Client passes pointers to where information is to be saved.

Parameters	**Description**
pbActive	The current active state of the subscription.
pdwBufferTime	The current buffer time configured for event notification. See the discussion in IOPCEventServer::CreateEventSubscription.
pdwMaxSize	The current max number of events that will be sent in a single IOPCEventSink::OnEvent callback. See the discussion in IOPCEventServer::CreateEventSubscription.
phClientSubscription	The client supplied subscription handle

HRESULT Return Codes

Return Code	**Description**
S_OK	The function was successful.
E_FAIL	The function was unsuccessful.

Comments

This function is typically called to obtain the current values of this information prior to calling SetState. This information was all supplied by the client when the subscription was created. This function is also useful for debugging.

5.5.1.8 IOPCEventSubscriptionMgt::SetState

```
HRESULT SetState(
    [unique, in] BOOL * pbActive,
    [unique, in] DWORD * pdwBufferTime,
    [unique, in] DWORD * pdwMaxSize,
    [in] OPCHANDLE hClientSubscription
    [out] DWORD * pdwRevisedBufferTime,
    [out] DWORD * pdwRevisedMaxSize,
    );
```

Description

Client can set various properties of the event subscription. Pointers to items are used so that the client can omit properties he does not want to change by passing a null pointer.

Parameters	**Description**
pbActive	TRUE (non-zero) to activate the subscription. FALSE (0) to deactivate the subscription. If the client deactivates the subscription, then the server will no longer send event notifications to the client based on that subscription, and has no responsibility to buffer or maintain the event notifications. Thus event notifications may be lost. Even if the subscription is inactive, the Refresh method will still function. In effect, this allows a client to obtain current condition states from time to time (by invoking Refresh) without the need to process event notifications in “real time”.
pdwBufferTime	New buffer time requested for the subscription by the client. See the discussion in IOPCEventServer::CreateEventSubscription.
pdwMaxSize	New maximum number of event notifications to send with a single IOPCEventSink::OnEvent callback. See the discussion in IOPCEventServer::CreateEventSubscription.
phClientSubscription	New client supplied handle for the subscription. This handle is returned in the data stream provided to the client’s IOPCEventSink by the subscription’s IConnectionPoint.
pdwRevisedBufferTime	The buffer time that the server is actually providing, which may differ from dwBufferTime.
pdwRevisedMaxSize	The maximum number of events that the server will actually be sending in a single IOPCEventSink::OnEvent callback, which may differ from dwMaxSize.

HRESULT Return Codes

Return Code	Description
S_OK	The function was successful.
E_INVALIDARG	A bad parameter was passed.
E_FAIL	The function was unsuccessful.
OPC_S_INVALIDBUFFERTIME	The buffer time parameter was invalid .
OPC_S_INVALIDMAXSIZE	The max size parameter was invalid.

Comments

5.5.2 IConnectionPointContainer

The general principles of ConnectionPoints are not discussed here as they are covered very clearly in the Microsoft Documentation. The reader is assumed to be familiar with this technology.

Likewise the details of the IEnumConnectionPoints, IConnectionPoint and IEnumConnections interfaces and their proper use in this context are well defined by Microsoft and are not discussed here.

The IConnectionPointContainer interface discussed here is implemented on an OPCEventSubscription object as obtained from IOPCEventServer::CreateEventSubscription(). This EventSubscription object will support at least the IOPCEventSubscriptionMgt and IConnectionPointContainer. Note that in theory, the Advise and Unadvise methods of the connection points could be implemented within the IOPCEventSubscriptionMgt interface however use of a separate ConnectionPoint implementation is more in keeping with state of the art Microsoft implementations.

One callback object implemented by the client application can be used to service multiple Alarm Servers. Therefore, information about the server must be provided to the client application for it to be able to successfully interpret the items that are contained in the callback. Each callback will contain only items from within the specified Server.

Note: OPC Compliant servers are not required to support more than one connection between each Subscription Object and the Client (although they do need to support creation of multiple Subscription Objects by a client in case the client wants to monitor them based on more than one set of filter criteria). Given this and the fact that Subscription Objects are client specific entities it is expected that a single connection will be sufficient for virtually all applications. For this reason (as per Microsoft Recommendations) the EnumConnections method for the IConnectionPoint interface for IOPCEventSink::OnEvent callback is allowed to return E_NOTIMPL.

IEnumConnectionPoints

See the Microsoft documentation for a description of this method.

OPC Event Subscriptions must return an enumerator that includes IOPCEventSink. Additional vendor specific callbacks are also allowed.

FindConnectionPoint

See the Microsoft documentation for a description of this method.

OPC Event Subscriptions must support IID_ IOPCEventSink. Additional vendor specific callbacks are also allowed.

5.5.3 IConnectionPoint

An IConnectionPoint for IOPCEventSink is returned from the Event Subscription’s ConnectionPointContainer. Refer to the Microsoft documentation of this interface for additional information on its methods, which include Advise and Unadvise.

The data returned to the Advise connection is returned via IOPCEventSink, which receives both new and refresh event notifications.

The registered callback function may be specified by the client application such that it spans multiple event subscriptions. Therefore, information about the event subscription must be provided to the client application to be able to successfully interpret the items that are contained in the event stream. Each event stream must only contain the items defined within the specified event subscription.

5.6 Client Side Interfaces

5.6.1 IOPCEventSink

In order to use connection points, the client must create an object which supports both the IUnknown and IOPCEventSink interfaces. The client would pass a pointer to the IUnknown interface (NOT the IOPCEventSink) to the Advise method of the proper IConnectionPoint in the event subscription (as obtained from IConnectionPointContainer:: FindConnectionPoint or EnumConnectionPoints). The event server will call QueryInterface on the client object to obtain the IOPCEventSink interface. Note that the transaction must be performed in this way in order for the interface marshalling to work properly for Local or Remote servers.

The event server invokes the OnEvent method to notify the client of events which satisfy the filter criteria for the particular event subscription.

The client need only provide a full implementation of OnEvent. There are no other methods of IOPCEventSink.

Note that callbacks can occur for two reasons: event notification or refresh. A server can be written such that it performs several of these operations in parallel. In this case the client can determine the 'cause' of a particular callback by examining the bRefresh parameter in the OnEvent callback.

5.6.1.1 IOPCEventSink::OnEvent

```
HRESULT OnEvent(
    [in] OPCHANDLE hClientSubscription,
    [in] BOOL bRefresh,
    [in] BOOL bLastRefresh,
    [in] DWORD dwCount,
    [in, size_is(dwCount)] ONEVENTSTRUCT* pEvents,
    );
```

Description

This method is provided by the client to handle notifications from the OPCEventSubscription for events. This method can be called whether this is a refresh or standard event notification.

Parameters	**Description**
hClientSubscription	the client handle for the subscription object sending the event notifications.
bRefresh	TRUE if this is a subscription refresh
bLastRefresh	TRUE if this is the last subscription refresh in response to a specific invocation of the IOPCEventSubscriptionMgt::Refresh method.
dwCount	number of event notifications
pEvents	array of event notifications

HRESULT Return Codes

Return Code	**Description**
S_OK	The client must always return S_OK. The server will get an error following the call if the client or the connection has failed.

Comments

The server needs to free pEvents after the client returns from this function.

Also – as per the COM specification, the client is restricted in what functions are allowed within the callback. For example, no blocking function may be called.

Callbacks can occur for one of the following reasons:

- One or more new events have occurred.
- This is a response to a Refresh.

5.6.1.1.1 ONEVENTSTRUCT

```
typedef  struct {
        WORD        wChangeMask,
        WORD        wNewState,
        LPWSTR      szSource,
        FILETIME    ftTime,
        LPWSTR      szMessage,
        DWORD       dwEventType,
        DWORD     dwEventCategory,
        DWORD     dwSeverity,
        LPWSTR    szConditionName,
        LPWSTR    szSubConditionName,
        WORD      wQuality,
        BOOL      bAckRequired,
        FILETIME  ftActiveTime,
        DWORD     dwCookie,
        DWORD     dwNumEventAttrs,
        [size_is (dwNumEventAttrs)] VARIANT* pEventAttributes,
        LPWSTR    szActorID,
} ONEVENTSTRUCT;
```

Member	Description
	The following items are present for all event types.
szSource	The source of event notification. This Source can be used in the IOPCEventServer::TranslateToItemIDs method to determine any related OPC Data Access itemIDs.
ftTime	Time of the event occurrence - for conditions, time that the condition transitioned into the new state or sub-condition. For example, if the event notification is for acknowledgment of a condition, this would be the time that the condition became acknowledged.
szMessage	Event notification message describing the event.
dwEventType	OPC_SIMPLE_EVENT, OPC_CONDITION_EVENT, or OPC_TRACKING_EVENT for Simple, Condition-Related, or Tracking events, respectively.
dwEventCategory	Standard and Vendor-specific event category codes. See section 2.5.3
dwSeverity	Event severity (0..1000). See section 2.4.3.2.
dwNumEventAttrs	The length of the vendor specific event attribute array.
pEventAttributes	Pointer to an array of vendor specific event attributes returned for this event notification. See the IOPCEventSubscriptionMgt::SelectReturnedAttributes method. The order of the items returned matches the order that was specified by the select.
	The following items are present only for Condition-Related Events (see dwEventType)
szConditionName	The name of the condition related to this event notification.
szSubConditionName	The name of the current sub-condition, for multi-state conditions. For a single-state condition, this contains the condition name.

wChangeMask	Indicates to the client which properties of the condition have changed, to have caused the server to send the event notification. It may have one or more of the following values: OPC_CHANGE_ACTIVE_STATE OPC_CHANGE_ACK_STATE OPC_CHANGE_ENABLE_STATE OPC_CHANGE_QUALITY OPC_CHANGE_SEVERITY OPC_CHANGE_SUBCONDITION OPC_CHANGE_MESSAGE OPC_CHANGE_ATTRIBUTE If the event notification is the result of a Refresh, these bits are to be ignored. For a “new event”, OPC_CHANGE_ACTIVE_STATE is the only bit which will always be set. Other values are server specific. (A “new event” is any event resulting from the related condition leaving the Inactive and Acknowledged state.)
wNewState	A WORD bit mask of three bits specifying the new state of the condition: OPC_CONDITION_ACTIVE, OPC_CONDITION_ENABLED, OPC_CONDITION_ACKED. See section 2.4.9 and Figure 2-2 for exactly which state transitions generate event notifications.
wQuality	Quality associated with the condition state. See Section 2.4.2.1. Values are as defined for the OPC Quality Flags in the OPC Data Access Server specification.
bAckRequired	This flag indicates that the related condition requires acknowledgment of this event. The determination of those events which require acknowledgment is server specific. For example, transition into a LimitAlarm condition would likely require an acknowledgment, while the event notification of the resulting acknowledgment would likely not require an acknowledgment.
ftActiveTime	Time that the condition became active (for single-state conditions), or the time of the transition into the current sub-condition (for multi-state conditions). This time is used by the client when acknowledging the condition (see IOPCEventServer::AckCondition method).
dwCookie	Server defined cookie associated with the event notification. This value is used by the client when acknowledging the condition (see IOPCEventServer::AckCondition method). This value is opaque to the client.
	The following is used only for Tracking Events and for Condition-Related Events which are acknowledgment notifications (see dwEventType).

szActorID	For tracking events, this is the actor ID for the event notification. For condition-related events, this is the AcknowledgerID when OPC_CONDITION_ACKED is set in wNewState. If the AcknowledgerID is a NULL string, the event was automatically acknowledged by the server. For other events, the value is a pointer to a NULL string.

Event Type Values

Event Type	Value	Description
OPC_SIMPLE_EVENT	1	Simple event.
OPC_TRACKING_EVENT	2	Tracking event.
OPC_CONDITION_EVENT	4	Condition-Related event.

Change Mask Values

Change Mask Item	Value	Description
OPC_CHANGE_ACTIVE_STATE	1	The condition's active state has changed.
OPC_CHANGE_ACK_STATE	2	The condition's acknowledgment state has changed.
OPC_CHANGE_ENABLE_STATE	4	The condition's enabled state has changed.
OPC_CHANGE_QUALITY	8	The ConditionQuality has changed.
OPC_CHANGE_SEVERITY	16	The severity level has changed.
OPC_CHANGE_SUBCONDITION	32	The condition has transitioned into a new sub-condition.
OPC_CHANGE_MESSAGE	64	The event message has changed (compared to prior event notifications related to this condition).
OPC_CHANGE_ATTRIBUTE	128	One or more event attributes have changed (compared to prior event notifications related to this condition).

New State Values

New State	Value	Description
OPC_CONDITION_ENABLED	1	The condition has been enabled.
OPC_CONDITION_ACTIVE	2	The condition has become active.
OPC_CONDITION_ACKED	4	The condition has been acknowledged.

5.6.2 IOPCShutdown

In order to use this connection point, the client must create an object that supports both the IUnknown and IOPCShutdown Interface. The client would pass a pointer to the IUnknown interface (NOT the IOPCShutdown) to the Advise method of the proper IConnectionPoint in the server (as obtained from IConnectionPointContainer:: FindConnectionPoint or EnumConnectionPoints). The Server will call QueryInterface on the client object to obtain the IOPCShutdown interface. Note that the transaction must be performed in this way in order for the interface marshalling to work properly for Local or Remote servers.

The ShutdownRequest method on this interface will be called when the event server needs to shutdown. The client should release all connections and interfaces for this event server.

A client which is connected to multiple servers (for example event servers and/or other servers such as data access servers from one or more vendors) should maintain separate shutdown callbacks for each object since any server can shut down independently of the others.

5.6.2.1 IOPCShutdown::ShutdownRequest

```
HRESULT ShutdownRequest (
    [in, string] LCPWSTR szReason
    );
```

Description

This method is provided by the client so that the server can request that the client disconnect from the server. The client should UnAdvise all connections and release all interfaces.

Parameter	Description
szReason	A text string indicating the reason for the shutdown request.

HRESULT Return Codes

Return Code	Description
S_OK	The client must always return S_OK.

Comments

The shutdown connection point is on a ‘per server object’ basis. That is, it relates to the object created by CoCreate… If a client connects to multiple server objects then it should monitor each one separately (using separate callbacks) for shutdown requests.

6. Installation Issues

It is assumed that the server vendor will provide a SETUP.EXE to install the needed components for their server. This will not be discussed further. Other than the actual components, the main issue affecting OLE software is management of the Windows Registry and Component Catagories. The issues here are (a) what entries need to be made and (b) how they can be made.

6.1 Common Topics

Certain installation and registry topics are common to all of the OPC Servers. These include self registration, automatic proxy/stub registration, and registry reference counting. These topics are discussed in the OPC Common Specification and are not repeated here. Instead, the server developer should refer to the OPC Common Specification for guidelines in these areas.

6.2 Component Categories Registration

During the registration process, each OPC Alarm and Events Server must register itself with the Component Categories Manager, a Microsoft supplied system COM object. OPC Alarm and Events Clients will query the Components Category Manager to enumerate the CLSIDs of all registered OPC Alarm and Events Servers.

Note: At this time the Component Categories Manager stores its information in the registry, however this will change in the near future. Please use the Component Categories Manager API to access this information rather than using the registry directly.

6.2.1 Server Registration

To Register with the Component Categories Manager, a server should first register the OPC defined Category ID (CATID) and the OPC defined Category Description by calling ICatRegister:: RegisterCategories(), and then register its own CLSID as an implementation of the CATID with a call to ICatRegister:: RegisterClassImplCategories().

To get an interface pointer to ICatRegister, call CoCreateInstance() as in this example:

```
#include <comcat.h>

CoCreateInstance(CLSID_StdComponentCategoriesMgr, NULL, CLSCTX_INPROC_SERVER,
IID_ICatRegister, (void**)&pcr);
```

The sample server code uses helper functions defined in CATHELP.CPP to make the actual calls to ICatRegister. Here is how the sample server registers and un-registers the component categories:

```
#include "cathelp.h"
#include "opc_ae.h"
#include "opcaedef.h"

void RegisterServer()
{
   // register component categories
   HRESULT hr;

   // IID_OPCEventServerCATID is the Category ID (a GUID) defined in opc_ae.idl.
   // OPC_EVENTSERVER_CAT_DESC is the category description defined in opcaedef.h
   // All servers should register the categogy this way

   hr = CreateComponentCategory( IID_OPCEventServerCATID,
OPC_EVENTSERVER_CAT_DESC);

   // CLSID_OPCEventServer is the CLSID for this sample server.  Each server
   // will need to register its own unique CLSID here with the component manager.

   hr = RegisterCLSIDInCategory( CLSID_OPCEventServer, IID_OPCEventServerCATID );
}


void UnregisterServer()
{
   UnRegisterCLSIDInCategory( CLSID_OPCEventServer, IID_OPCEventServerCATID );
}
```

6.2.2 Client Enumeration

Editor's Note: This section will change if the TSC adopts the proposed DCOM aware remote OPC browse server.

To get a list of CLSIDs of all OPC Alarm and Event Servers registered with the Component Categories Manager, the client calls ICatInformation::EnumClassesOfCategories() to return an enumerator interface, IEnumCLSID as in this code snippet:

```
ICatInformation* pcr = NULL ;
HRESULT hr = S_OK ;

hr = CoCreateInstance(CLSID_StdComponentCategoriesMgr,
         NULL, CLSCTX_INPROC_SERVER, IID_ICatInformation, (void**)&pcr);

IEnumCLSID* pEnumCLSID;

CLSID catid = IID_OPCEventServerCATID;
pcr->EnumClassesOfCategories(1, &catid, 1, &catid, &pEnumCLSID);

// get 10 at a time for efficiency
unsigned long c;
CLSID clsids[10];

while (SUCCEEDED(hr = pEnumCLSID->Next(10, clsids, &c)))
{
   for( unsigned long i = 0; i < c; i++ )
  {
   // clsid[i] is a CLSID that implements the component category ...
      .
      .
      .
  }
}
```

7. Summary of OPC Error Codes

We have attempted to minimize the number of unique errors by identifying common generic problems and defining error codes that can be reused in many contexts. An OPC server should only return those OPC errors that are listed for the various methods in this specification or are standard Microsoft errors. Note that OLE itself will frequently return errors (such as RPC errors) in addition to those listed in this specification.

The most important thing for a client is to check FAILED for any error return. Other than that, (the statements above not withstanding) a robust, user friendly client should assume that the server may return any error code and should call the GetErrorString function to provide user readable information about those errors.

Standard COM errors that are commonly used by OPC Servers	**Description**
E_FAIL	Unspecified error
E_INVALIDARG	The value of one or more parameters was not valid. This is generally used in place of a more specific error where it is expected that problems are unlikely or will be easy to identify (for example when there is only one parameter).
E_NOINTERFACE	No such interface supported
E_NOTIMPL	Not implemented
E_OUTOFMEMORY	Not enough memory to complete the requested operation. This can happen any time the server needs to allocate memory to complete the requested operation.

OPC Specific Errors	**Description**
OPC_E_BUSY	A refresh operation is currently in progress on the event subscription object.
OPC_E_INVALIDBRANCHNAME	The string was not recognized as an area name
OPC_S_INVALIDBUFFERTIME	The specified buffer time parameter was invalid.
OPC_S_INVALIDMAXSIZE	The specified max size parameter was invalid.
OPC_E_INVALIDTIME	The specified time does not match the latest sub-condition active time for the condition being acknowledged.
OPC_E_NOINFO	No information is currently available for the specified condition.
OPC_S_ALREADYACKED	The condition has already been acknowledged.

You will see in the appendix that these error codes use ITF_FACILITY. This means that they are context specific (i.e. OPC specific). The calling application should check first with the server providing the error (i.e. call GetErrorString).

The OPC Specific error codes and their associated strings (English) are embedded in the resource of the proxy/stub (opc_aeps.dll) so FormatMessage() can be called to retrieve the strings:

```
rtn = FormatMessage(
               FORMAT_MESSAGE_ALLOCATE_BUFFER |
               FORMAT_MESSAGE_IGNORE_INSERTS |FORMAT_MESSAGE_FROM_HMODULE,
   GetModuleHandle(_T("opc_aeps")),
   GetScode( dwError ),
   MAKELANGID(LANG_NEUTRAL, SUBLANG_NEUTRAL),
   (LPTSTR) &lpMsgBuf, 0, NULL );
```

Error codes (the low order word of the HRESULT) from 0000 to 0200 are reserved for Microsoft use (although some were inadverdantly used for OPC 1.0 errors). Codes from 0200 through 8000 are reserved for future OPC use. Codes from 8000 through FFFF can be vendor specific.

Appendix A – Sample String Filter Function

This function provides essentially the same functionality as the LIKE operator in Visual Basic.

MatchPattern

Syntax

BOOL MatchPattern(LPCTSTR *string***, LPCTSTR** *pattern***, BOOL** *bCaseSensitive* **)**

Return Value

If *string* matches *pattern*, return is **TRUE**; if there is no match, return is **FALSE**. If either *string* or *pattern* is Null, return is **FALSE;**

Parameters

string	String to be compared with pattern.
pattern	Any string conforming to the pattern-matching conventions described in Remarks.
bCaseSensitive	**TRUE** if comparison should be case sensitive.

Remarks

A versatile tool used to compare two strings. The pattern-matching features allow you to use wildcard characters, character lists, or character ranges, in any combination, to match strings. The following table shows the characters allowed in *pattern* and what they match:

Characters in *pattern*	**Matches in *string***
?	Any single character.
*****	Zero or more characters.
#	Any single digit (0-9).
[*charlist*]	Any single character in *charlist*.
[**!***charlist*]	Any single character not in *charlist*.

A group of one or more characters (*charlist*) enclosed in brackets (**[ ]**) can be used to match any single character in *string* and can include almost any charcter code, including digits.

Note To match the special characters left bracket (**[**), question mark (**?**), number sign (**#**), and asterisk (*****), enclose them in brackets. The right bracket (**]**) can't be used within a group to match itself, but it can be used outside a group as an individual character.

By using a hyphen (**-**) to separate the upper and lower bounds of the range, *charlist* can specify a range of characters. For example, `[A-Z]` results in a match if the corresponding character position in *string* contains any uppercase letters in the range A-Z. Multiple ranges are included within the brackets without delimiters.

Other important rules for pattern matching include the following:

- An exclamation point (!) at the beginning of charlist means that a match is made if any character except the characters in charlist is found in string. When used outside brackets, the exclamation point matches itself.

- A hyphen (-) can appear either at the beginning (after an exclamation point if one is used) or at the end of charlist to match itself. In any other location, the hyphen is used to identify a range of characters.
- When a range of characters is specified, they must appear in ascending sort order (from lowest to highest). [A-Z] is a valid pattern, but [Z-A] is not.
- The character sequence `[]` is considered a zero-length string ("").

Here is the code:

```
// matchpattern.h

#ifndef __MATCHPATTERN_H
#define __MATCHPATTERN_H

// By redefining MCHAR, _M and _ismdigit you may alter the type
// of string MatchPattern() works with. For example to operate on
// wide strings, make the following definitions:
// #define MCHAR     WCHAR
// #define _M(x)     L ## x
// #define _ismdigit   iswdigit



#ifndef MCHAR

#define MCHAR   TCHAR
#define _M(a)   _T(a)
#define _ismdigit   _istdigit

#endif



extern BOOL  MatchPattern( const MCHAR* String, const MCHAR * Pattern, BOOL
bCaseSensitive = FALSE );


#endif





// matchpattern.cpp
#include "MatchPattern.h"



inline int ConvertCase( int c, BOOL bCaseSensitive )
{
   return bCaseSensitive ? c : toupper(c);
}
```

```
//***********************************************************************
// return TRUE if String Matches Pattern --
// -- uses Visual Basic LIKE operator syntax
// CAUTION: Function is recursive
//***********************************************************************
BOOL MatchPattern( const MCHAR *String, const MCHAR *Pattern, BOOL
bCaseSensitive )
{
   if( !String )
      return FALSE;
   if( !Pattern )
      return TRUE;
   MCHAR   c, p, l;
   for (; ;)
   {
      switch (p = ConvertCase( *Pattern++, bCaseSensitive ) )
      {
      case 0:                              // end of pattern
         return *String ? FALSE : TRUE;  // if end of string TRUE

      case _M('*'):
         while (*String)
         {               // match zero or more char
            if (MatchPattern (String++, Pattern, bCaseSensitive))
               return TRUE;
         }
         return MatchPattern (String, Pattern, bCaseSensitive );

      case _M('?'):
         if (*String++ == 0)             // match any one char
            return FALSE;                   // not end of string
         break;

      case _M('['):
         if ( (c = ConvertCase( *String++, bCaseSensitive) ) == 0)      // match
char set
            return FALSE;                   // syntax
         l = 0;
         if( *Pattern == _M('!') )  // match a char if NOT in set []
         {
            ++Pattern;

            while( (p = ConvertCase( *Pattern++, bCaseSensitive) ) != _M('\0') )
            {
               if (p == _M(']'))               // if end of char set, then
                  break;           // no match found

               if (p == _M('-'))
               {            // check a range of chars?
                  p = ConvertCase( *Pattern, bCaseSensitive );   // get high
limit of range
                  if (p == 0  ||  p == _M(']'))
                     return FALSE;           // syntax

                  if (c >= l  &&  c <= p)
                     return FALSE;              // if in range, return FALSE
               }
               l = p;
               if (c == p)                 // if char matches this element
                  return FALSE;                 // return false
            }
         }
         else // match if char is in set []
```

```
        {
           while( (p = ConvertCase( *Pattern++, bCaseSensitive) ) != _M('\0') )
           {
              if (p == _M(']'))                // if end of char set, then
                 return FALSE;           // no match found

              if (p == _M('-'))
              {            // check a range of chars?
                 p = ConvertCase( *Pattern, bCaseSensitive );   // get high
limit of range
                 if (p == 0  ||  p == _M(']'))
                    return FALSE;           // syntax

                 if (c >= l  &&  c <= p)
                    break;              // if in range, move on
              }
              l = p;
              if (c == p)                 // if char matches this element
                 break;                  // move on
           }

           while (p  &&  p != _M(']'))          // got a match in char set
              p = *Pattern++;             // skip to end of set
        }

        break;

     case _M('#'):
        c = *String++;
        if( !_ismdigit( c ) )
           return FALSE;   // not a digit

        break;

     default:
        c = ConvertCase( *String++, bCaseSensitive );
        if( c != p )            // check for exact char
           return FALSE;                    // not a match

        break;
     }
  }
}
```

Appendix B – Event Types, Event Categories, and Conditions

The following table shows recommended event categories for each event type, and recommended conditions corresponding to each event category. It is recommended that OPC condition names leverage Foundation Fieldbus naming as appropriate. As an example, the condition indicating a PV has entered into a High High Alarm condition is named HI_HI which then matches the Foundation Fieldbus HI_HI Alarm Type.

Event Type	**Event Category**	**CONDITION**
Condition Related	Level	PVLEVEL (Multi State)
		SPLEVEL (Multi State)
		LO_LO (Single State)
		LO (Single State)
		HI (Single State)
		HI_HI (Single State)
	Deviation	DV_LO (Single State)
	Deviation	DV_HI (Single State)
	Discrete	CFN
		TRIP
		COS
	Statistical	
	System Failure	SYSTEM_FAILURE
Simple	Device Failure	
	Batch Status	
	System Message	
Tracking	Operator Process Change	
	System Configuration	
	Advanced Control	

Appendix C – Event Attributes

The following are recommended attributes for the event categories listed in Appendix B.

Event Type.Category	**ATTRIBUTE**	**NOTES**
ALL	ACK COMMENT	Latest comment from IOPCEventServer::AckCondition()
	AREAS	SAFEARRAY of BSTRS. Each string is a Qualified Area Name to which this Source belongs.
Condition.Level	CV	Current Value
	LIMIT VALUE EXCEEDED	
	NEXT LIM	
	PREV LIM	
	DEADBAND	
	LOOP DESC	
Condition.Discrete	NORMAL STATE	
	CV	Current Value
	LOOP DESC	
Condition.Deviation	CV	Current Value
	LIMIT EXCEEDED	
	NEXT LIM	
	PREV LIM	
	LOOP DESC	
Condition.System	HELPFILE	
Simple.Devicefailed	DEVICE NAME	
	ERROR CODE/STRING	
Simple.Batch	BATCHID	
Simple.System	?	
Tracking. Operator Process Change	PREV VALUE	
	NEW VALUE	
	NAME OF PARAMETER	
	COMMENT	
Tracking.Advanced	PREV VALUE	
	NEW VALUE	
	NAME OF PARAMETER	
Tracking.Sysconfig	PREV VALUE	
	NEW VALUE	
	NAME OF PARAMETER	

Appendix D – Event Server IDL Specification

The current files require MIDL compiler 3.00.15 or later and the WIN NT 4.0 release SDK.

Use the command line MIDL /ms_ext /c_ext /app_config opc_ae.idl.

The resulting **OPC_AE.H** file can be **included** in clients and servers. The resulting **OPC_AE_I.C** file defines the interface IDs and can be **linked** into clients and servers that include **OPC_AE.H**.

Alternatively, clients and servers may choose to use the Type Library that is embedded in the resource of the proxy/stub DLL (**OPC_AEPS.DLL**). In Visual C++ this is accomplished with the #import statement:

```
#import "opc_aeps.dll" exclude("_FILETIME")
using namespace OPC_AE;
```

NOTE: This IDL file and the Proxy/Stub generated from it should NEVER be modified in any way. If you add vendor specific interfaces to your server (which is allowed) you must generate a SEPARATE vendor specific IDL file to describe only those interfaces and a separate vendor specific ProxyStub DLL to marshall only those interfaces.

```
// opc_ae.idl : IDL source for opc_aeps.dll
//
// REVISION:  05/25/99 09:32 AM (GMT)
// VERSIONINFO  1.0.5.0
//
// This file will be processed by the MIDL tool to
// produce the type library (opc_ae.tlb) and marshalling code (opc_aeps.dll).
// The type library is embedded in the resource of opc_aeps.dll

import "oaidl.idl";
import "ocidl.idl";


// define OPC Alarm & Events Component Categories


   [
     uuid(58E13251-AC87-11d1-84D5-00608CB8A7E9),
     helpstring("OPC Event Server Category ID (CATID)"),
     pointer_default(unique)
   ]
   interface OPCEventServerCATID
   {
     // This empty interface is here so that
     // IID_OPCEventServerCATID will be defined
   };



typedef DWORD OPCHANDLE;

typedef enum { OPCAE_BROWSE_UP = 1,
        OPCAE_BROWSE_DOWN,
        OPCAE_BROWSE_TO
} OPCAEBROWSEDIRECTION;

typedef enum { OPC_AREA = 1,
        OPC_SOURCE
} OPCAEBROWSETYPE;


typedef enum { OPCAE_STATUS_RUNNING = 1,
        OPCAE_STATUS_FAILED,
        OPCAE_STATUS_NOCONFIG,
        OPCAE_STATUS_SUSPENDED,
        OPCAE_STATUS_TEST
} OPCEVENTSERVERSTATE;





typedef  struct {
          WORD       wChangeMask;
          WORD       wNewState;
   [string]   LPWSTR       szSource;
          FILETIME      ftTime;
   [string]   LPWSTR       szMessage;
          DWORD      dwEventType;
          DWORD      dwEventCategory;
          DWORD      dwSeverity;
   [string]   LPWSTR       szConditionName;
   [string]   LPWSTR       szSubconditionName;
          WORD       wQuality;
          WORD       wReserved;   // added for natural alignment
          BOOL       bAckRequired;
          FILETIME      ftActiveTime;
          DWORD      dwCookie;
          DWORD      dwNumEventAttrs;
   [size_is(dwNumEventAttrs)] VARIANT*      pEventAttributes;
   [string]   LPWSTR       szActorID;
} ONEVENTSTRUCT;


typedef struct {
   FILETIME        ftStartTime;
   FILETIME        ftCurrentTime;
   FILETIME        ftLastUpdateTime;
   OPCEVENTSERVERSTATE  dwServerState;
   WORD         wMajorVersion;
   WORD         wMinorVersion;
   WORD         wBuildNumber;
   WORD         wReserved;   // added for natural alignment
   [string]   LPWSTR     szVendorInfo;
} OPCEVENTSERVERSTATUS;


typedef  struct {
   WORD  wState;
   WORD  wReserved1;     // added for natural alignment
   LPWSTR  szActiveSubCondition;
   LPWSTR  szASCDefinition;
   DWORD dwASCSeverity;
   LPWSTR  szASCDescription;
   WORD  wQuality;
   WORD  wReserved2;     // added for natural alignment
   FILETIME   ftLastAckTime;
   FILETIME   ftSubCondLastActive;
   FILETIME   ftCondLastActive;
   FILETIME   ftCondLastInactive;
   LPWSTR  szAcknowledgerID;
   LPWSTR  szComment;
   DWORD dwNumSCs;
   [size_is (dwNumSCs)] LPWSTR * pszSCNames;
   [size_is (dwNumSCs)] LPWSTR * pszSCDefinitions;
   [size_is (dwNumSCs)] DWORD  * pdwSCSeverities;
   [size_is (dwNumSCs)] LPWSTR * pszSCDescriptions;
   DWORD dwNumEventAttrs;
   [size_is(dwNumEventAttrs)] VARIANT*     pEventAttributes;
   [size_is(dwNumEventAttrs)] HRESULT*     pErrors;
} OPCCONDITIONSTATE;



   [
      uuid(65168851-5783-11D1-84A0-00608CB8A7E9),

      helpstring("IOPCEventServer Interface"),
      pointer_default(unique)
   ]
   interface IOPCEventServer : IUnknown
   {
      HRESULT GetStatus(
         [out] OPCEVENTSERVERSTATUS **ppEventServerStatus
         );


      HRESULT CreateEventSubscription(
         [in] BOOL bActive,
         [in] DWORD dwBufferTime,
         [in] DWORD dwMaxSize,
         [in] OPCHANDLE hClientSubscription,
         [in] REFIID riid,
         [out, iid_is(riid)] LPUNKNOWN * ppUnk,
         [out] DWORD *pdwRevisedBufferTime,
         [out] DWORD *pdwRevisedMaxSize
         );

      HRESULT QueryAvailableFilters(
         [out] DWORD* pdwFilterMask
         );


      HRESULT QueryEventCategories(
         [in]  DWORD  dwEventType,
         [out] DWORD* pdwCount,
         [out, size_is(,*pdwCount)] DWORD** ppdwEventCategories,
         [out, size_is(,*pdwCount)] LPWSTR** ppszEventCategoryDescs
         );

      HRESULT QueryConditionNames(
         [in]  DWORD  dwEventCategory,
         [out] DWORD* pdwCount,
         [out, size_is(,*pdwCount)] LPWSTR** ppszConditionNames
         );


      HRESULT QuerySubConditionNames(
         [in]  LPWSTR szConditionName,
         [out] DWORD* pdwCount,
         [out, size_is(,*pdwCount)] LPWSTR** ppszSubConditionNames
         );
```

```
HRESULT QuerySourceConditions(
   [in]  LPWSTR szSource,
   [out] DWORD* pdwCount,
   [out, size_is(,*pdwCount)] LPWSTR** ppszConditionNames
   );


HRESULT QueryEventAttributes(
   [in]  DWORD dwEventCategory,
   [out] DWORD* pdwCount,
   [out, size_is(,*pdwCount)] DWORD** ppdwAttrIDs,
   [out, size_is(,*pdwCount)] LPWSTR** ppszAttrDescs,
   [out, size_is(,*pdwCount)] VARTYPE** ppvtAttrTypes
   );

HRESULT TranslateToItemIDs(
   [in] LPWSTR szSource,
   [in] DWORD dwEventCategory,
   [in] LPWSTR szConditionName,
   [in] LPWSTR szSubconditionName,
   [in] DWORD  dwCount,
   [in, size_is(dwCount)] DWORD* pdwAssocAttrIDs,
   [out, size_is(,dwCount)] LPWSTR** ppszAttrItemIDs,
   [out, size_is(,dwCount)] LPWSTR** ppszNodeNames,
   [out, size_is(,dwCount)] CLSID** ppCLSIDs
   );

HRESULT GetConditionState (
   [in]  LPWSTR szSource,
   [in]  LPWSTR szConditionName,
   [in]  DWORD dwNumEventAttrs,
   [in, size_is(dwNumEventAttrs)] DWORD* pdwAttributeIDs,
   [out] OPCCONDITIONSTATE ** ppConditionState
   );

HRESULT EnableConditionByArea(
   [in] DWORD dwNumAreas,
   [in, size_is(dwNumAreas)] LPWSTR* pszAreas
   );


HRESULT EnableConditionBySource(
   [in] DWORD dwNumSources,
   [in, size_is(dwNumSources)] LPWSTR* pszSources
   );

HRESULT DisableConditionByArea(
   [in] DWORD dwNumAreas,
   [in, size_is(dwNumAreas)] LPWSTR* pszAreas
   );

HRESULT DisableConditionBySource(
   [in] DWORD dwNumSources,
   [in, size_is(dwNumSources)] LPWSTR* pszSources
   );

HRESULT AckCondition(
   [in] DWORD dwCount,
   [in, string] LPWSTR szAcknowledgerID ,
   [in, string] LPWSTR szComment ,
   [in, size_is(dwCount)] LPWSTR* pszSource,
   [in, size_is(dwCount)] LPWSTR* pszConditionName,
        [in, size_is(dwCount)] FILETIME* pftActiveTime,
        [in, size_is(dwCount)] DWORD* pdwCookie,
        [out, size_is(,dwCount)] HRESULT **ppErrors
        );

     HRESULT CreateAreaBrowser(
        [in] REFIID riid,
        [out, iid_is(riid)] LPUNKNOWN* ppUnk
        );

  };
  [

     uuid(65168855-5783-11D1-84A0-00608CB8A7E9),

     helpstring("IOPCEventSubscriptionMgt Interface"),
     pointer_default(unique)
  ]
  interface IOPCEventSubscriptionMgt : IUnknown
  {
     HRESULT SetFilter(
        [in] DWORD  dwEventType,
        [in] DWORD dwNumCategories,
        [in, size_is(dwNumCategories)] DWORD* pdwEventCategories,
        [in] DWORD dwLowSeverity,
        [in] DWORD dwHighSeverity,
        [in] DWORD dwNumAreas,
        [in, size_is(dwNumAreas)] LPWSTR* pszAreaList,
        [in] DWORD dwNumSources,
        [in, size_is(dwNumSources)] LPWSTR* pszSourceList
        );

     HRESULT GetFilter(
        [out] DWORD* pdwEventType,
        [out] DWORD* pdwNumCategories,
        [out, size_is(,*pdwNumCategories)] DWORD** ppdwEventCategories,
        [out] DWORD* pdwLowSeverity,
        [out] DWORD* pdwHighSeverity,
        [out] DWORD* pdwNumAreas,
        [out, size_is(,*pdwNumAreas)] LPWSTR** ppszAreaList,
        [out] DWORD* pdwNumSources,
        [out, size_is(,*pdwNumSources)] LPWSTR** ppszSourceList
        );

     HRESULT SelectReturnedAttributes(
        [in] DWORD dwEventCategory,
        [in] DWORD dwCount,
        [in, size_is(dwCount)] DWORD* dwAttributeIDs
        );

     HRESULT GetReturnedAttributes(
        [in]  DWORD dwEventCategory,
        [out] DWORD * pdwCount,
        [out, size_is(,*pdwCount)] DWORD** ppdwAttributeIDs
     );


     HRESULT Refresh(
        [in] DWORD dwConnection
        );

     HRESULT CancelRefresh(
        [in] DWORD dwConnection
        );

     HRESULT GetState(
        [out] BOOL * pbActive,
        [out] DWORD * pdwBufferTime,
        [out] DWORD * pdwMaxSize,
        [out] OPCHANDLE * phClientSubscription
        );

     HRESULT SetState(
        [unique, in] BOOL *  pbActive,
        [unique, in] DWORD * pdwBufferTime,
        [unique, in] DWORD * pdwMaxSize,
        [in] OPCHANDLE hClientSubscription,
        [out] DWORD * pdwRevisedBufferTime,
        [out] DWORD * pdwRevisedMaxSize
        );
  };
  [

     uuid(65168857-5783-11D1-84A0-00608CB8A7E9),

     helpstring("IOPCEventAreaBrowser Interface"),
     pointer_default(unique)
  ]
  interface IOPCEventAreaBrowser : IUnknown
  {
     HRESULT ChangeBrowsePosition(
        [in]  OPCAEBROWSEDIRECTION dwBrowseDirection,
        [in, string] LPCWSTR  szString
        );

     HRESULT BrowseOPCAreas(
        [in] OPCAEBROWSETYPE   dwBrowseFilterType,
        [in, string] LPCWSTR  szFilterCriteria,
        [out] LPENUMSTRING  * ppIEnumString
        );

     HRESULT GetQualifiedAreaName(
        [in] LPCWSTR szAreaName,
        [out, string] LPWSTR *pszQualifiedAreaName
        );

     HRESULT GetQualifiedSourceName(
        [in] LPCWSTR szSourceName,
        [out, string] LPWSTR *pszQualifiedSourceName
        );

  };
  [

     uuid(6516885F-5783-11D1-84A0-00608CB8A7E9),

     helpstring("IOPCEventSink Interface"),
     pointer_default(unique)
  ]
  interface IOPCEventSink : IUnknown
  {
     HRESULT OnEvent(
        [in] OPCHANDLE hClientSubscription,
        [in] BOOL bRefresh,
        [in] BOOL bLastRefresh,
        [in] DWORD dwCount,
        [in, size_is(dwCount)] ONEVENTSTRUCT* pEvents
        );

   };



[
   uuid(65168844-5783-11D1-84A0-00608CB8A7E9),
   version(1.0),
   helpstring("opc_ae 1.0 Type Library")
]
library OPC_AE
{
   importlib("stdole32.tlb");
   importlib("stdole2.tlb");

   interface IOPCEventServer;
   interface IOPCEventSubscriptionMgt;
   interface IOPCEventAreaBrowser;
   interface IOPCEventSink;
   interface OPCEventServerCATID;

};
```

Appendix E – OPCAEDEF.H

```
/*++

Module Name:

    opcaedef.h

Abstract:

   Macros defined for OPC Alarm & Events Clients and Servers

Author:

    Jim Luth - OPC Alarm & Events Committee

Revision History:

--*/

#ifndef __OPCAEDEF_H
#define __OPCAEDEF_H



// OPC Alarm & Event Component Category Description
#define OPC_EVENTSERVER_CAT_DESC L"OPC Alarm & Event Server Version 1.0"


//******************************************************
// OPC Quality flags
//
// Masks for extracting quality subfields
// (note 'status' mask also includes 'Quality' bits)
//
#define    OPC_QUALITY_MASK            0xC0
#define    OPC_STATUS_MASK             0xFC
#define    OPC_LIMIT_MASK              0x03

// Values for QUALITY_MASK bit field
//
#define    OPC_QUALITY_BAD             0x00
#define    OPC_QUALITY_UNCERTAIN       0x40
#define    OPC_QUALITY_GOOD            0xC0

// STATUS_MASK Values for Quality = BAD
//
#define    OPC_QUALITY_CONFIG_ERROR    0x04
#define    OPC_QUALITY_NOT_CONNECTED   0x08
#define    OPC_QUALITY_DEVICE_FAILURE  0x0c
#define    OPC_QUALITY_SENSOR_FAILURE  0x10
#define    OPC_QUALITY_LAST_KNOWN      0x14
#define    OPC_QUALITY_COMM_FAILURE    0x18
#define    OPC_QUALITY_OUT_OF_SERVICE  0x1C

// STATUS_MASK Values for Quality = UNCERTAIN
//
#define    OPC_QUALITY_LAST_USABLE     0x44
#define    OPC_QUALITY_SENSOR_CAL      0x50
#define    OPC_QUALITY_EGU_EXCEEDED    0x54
```

```
#define    OPC_QUALITY_SUB_NORMAL      0x58

// STATUS_MASK Values for Quality = GOOD
//
#define    OPC_QUALITY_LOCAL_OVERRIDE  0xD8


// State bit masks
#define OPC_CONDITION_ENABLED      0x0001
#define OPC_CONDITION_ACTIVE       0x0002
#define OPC_CONDITION_ACKED        0x0004


// bit masks for m_wChangeMask
#define OPC_CHANGE_ACTIVE_STATE    0x0001
#define OPC_CHANGE_ACK_STATE       0x0002
#define OPC_CHANGE_ENABLE_STATE    0x0004
#define OPC_CHANGE_QUALITY         0x0008
#define OPC_CHANGE_SEVERITY        0x0010
#define OPC_CHANGE_SUBCONDITION    0x0020
#define OPC_CHANGE_MESSAGE         0x0040
#define OPC_CHANGE_ATTRIBUTE       0x0080


// dwEventType
#define OPC_SIMPLE_EVENT           0x0001
#define OPC_TRACKING_EVENT         0x0002
#define OPC_CONDITION_EVENT        0x0004

#define OPC_ALL_EVENTS  (OPC_SIMPLE_EVENT | OPC_TRACKING_EVENT |
OPC_CONDITION_EVENT )


// QueryAvailableFilters() bit masks
#define OPC_FILTER_BY_EVENT        0x0001
#define OPC_FILTER_BY_CATEGORY     0x0002
#define OPC_FILTER_BY_SEVERITY     0x0004
#define OPC_FILTER_BY_AREA         0x0008
#define OPC_FILTER_BY_SOURCE       0x0010


#endif
```

Appendix F – OPCAE_ER.H

```
/*++

Module Name:

    opcae_er.h

Abstract:

    This file is generated by the MC tool from the opcae_er.mc message
    file.

Author:

    Jim Luth - OPC Alarm & Events Committee

Revision History:

--*/
/*
Code Assignements:
  0000 to 0200 are reserved for Microsoft use
  (although some were inadverdantly used for OPC Data Access 1.0 errors).
  0200 to 8000 are reserved for future OPC use.
  8000 to FFFF can be vendor specific.

*/


#ifndef __OPCAE_ER_H
#define __OPCAE_ER_H

// Since we use FACILITY_ITF our codes must be in the range 0x200 - 0xFFFF
// success codes
//
//  Values are 32 bit values layed out as follows:
//
//   3 3 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1
//   1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
//  +---+-+-+-----------------------+-------------------------------+
//  |Sev|C|R|     Facility          |               Code            |
//  +---+-+-+-----------------------+-------------------------------+
//
//  where
//
//      Sev - is the severity code
//
//          00 - Success
//          01 - Informational
//          10 - Warning
//          11 - Error
//
//      C - is the Customer code flag
//
//      R - is a reserved bit
//
//      Facility - is the facility code
//
//      Code - is the facility's status code
//
```

```
//
// Define the facility codes
//


//
// Define the severity codes
//


//
// MessageId: OPC_S_ALREADYACKED
//
// MessageText:
//
//  The condition has already been acknowleged
//
#define OPC_S_ALREADYACKED               ((HRESULT)0x00040200L)

//
// MessageId: OPC_S_INVALIDBUFFERTIME
//
// MessageText:
//
//  The buffer time parameter was invalid
//
#define OPC_S_INVALIDBUFFERTIME          ((HRESULT)0x00040201L)

//
// MessageId: OPC_S_INVALIDMAXSIZE
//
// MessageText:
//
//  The max size parameter was invalid
//
#define OPC_S_INVALIDMAXSIZE             ((HRESULT)0x00040202L)

// error codes
//
// MessageId: OPC_E_INVALIDBRANCHNAME
//
// MessageText:
//
//  The string was not recognized as an area name
//
#define OPC_E_INVALIDBRANCHNAME          ((HRESULT)0xC0040203L)

//
// MessageId: OPC_E_INVALIDTIME
//
// MessageText:
//
//  The time does not match the latest active time
//
#define OPC_E_INVALIDTIME                ((HRESULT)0xC0040204L)

//
// MessageId: OPC_E_BUSY
//
// MessageText:
//
//  A refresh is currently in progress
//
```

```
#define OPC_E_BUSY                       ((HRESULT)0xC0040205L)

//
// MessageId: OPC_E_NOINFO
//
// MessageText:
//
//  Information is not available
//
#define OPC_E_NOINFO                     ((HRESULT)0xC0040206L)

#endif
```