OPC Common Definitions and Interfaces

Version 1.0

October 27, 1998

Specification Type	Industry Standard Specification		
Title:	**OPC Common Definitions**	Date:	October 27, 1998
Version:	1.0	Soft Source:	MS-Word OpcComn.doc
Author:	Opc Task Force	Status:	**Release**

Synopsis:

This is the specification of rules, design criteria and interfaces that are common to 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

NON-EXCLUSIVE LICENSE AGREEMENT

The OPC Foundation, a non-profit corporation (the "OPC Foundation"), has established a set of standard OLE/COM interface protocols intended to foster greater interoperability between automation/control applications, field systems/devices, and business/office applications in the process control industry.

The current OPC specifications, prototype software examples and related documentation (collectively, the "OPC Materials"), form a set of standard OLE/COM interface protocols based upon the functional requirements of Microsoft's OLE/COM technology. Such technology defines standard objects, methods, and properties for servers of real-time information like distributed process systems, programmable logic controllers, smart field devices and analyzers in order to communicate the information that such servers contain to standard OLE/COM compliant technologies enabled devices (e.g., servers, applications, etc.).

The OPC Foundation will grant to you (the "User"), whether an individual or legal entity, a license to use, and provide User with a copy of, the current version of the OPC Materials so long as User abides by the terms contained in this Non-Exclusive License Agreement ("Agreement"). If User does not agree to the terms and conditions contained in this Agreement, the OPC Materials may not be used, and all copies (in all formats) of such materials in User's possession must either be destroyed or returned to the OPC Foundation. By using the OPC Materials, User (including any employees and agents of User) agrees to be bound by the terms of this Agreement.

LICENSE GRANT:

Subject to the terms and conditions of this Agreement, the OPC Foundation hereby grants to User a non-exclusive, royalty-free, limited license to use, copy, display and distribute the OPC Materials in order to make, use, sell or otherwise distribute any products and/or product literature that are compliant with the standards included in the OPC Materials.

All copies of the OPC Materials made and/or distributed by User must include all copyright and other proprietary rights notices include on or in the copy of such materials provided to User by the OPC Foundation.

The OPC Foundation shall retain all right, title and interest (including, without limitation, the copyrights) in the OPC Materials, subject to the limited license granted to User under this Agreement.

WARRANTY AND LIABILITY DISCLAIMERS:

User acknowledges that the OPC Foundation has provided the OPC Materials for informational purposes only in order to help User understand Microsoft's OLE/COM technology. THE OPC MATERIALS ARE PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, WARRANTIES OF PERFORMANCE, MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT. USER BEARS ALL RISK RELATING TO QUALITY, DESIGN, USE AND PERFORMANCE OF THE OPC MATERIALS. The OPC Foundation and its members do not warrant that the OPC Materials, their design or their use will meet User's requirements, operate without interruption or be error free.

IN NO EVENT SHALL THE OPC FOUNDATION, ITS MEMBERS, OR ANY THIRD PARTY BE LIABLE FOR ANY COSTS, EXPENSES, LOSSES, DAMAGES (INCLUDING, BUT NOT LIMITED TO, DIRECT, INDIRECT, CONSEQUENTIAL, INCIDENTAL, SPECIAL OR PUNITIVE DAMAGES) OR INJURIES INCURRED BY USER OR ANY THIRD PARTY AS A RESULT OF THIS AGREEMENT OR ANY USE OF THE OPC MATERIALS.

GENERAL PROVISIONS:

This Agreement and User's license to the OPC Materials shall be terminated (a) by User ceasing all use of the OPC Materials, (b) by User obtaining a superseding version of the OPC Materials, or (c) by the OPC Foundation, at its option, if User commits a material breach hereof. Upon any termination of this Agreement, User shall immediately cease all use of the OPC Materials, destroy all copies thereof then in its possession and take such other actions as the OPC Foundation may reasonably request to ensure that no copies of the OPC Materials licensed under this Agreement remain in its possession.

User shall not export or re-export the OPC Materials or any product produced directly by the use thereof to any person or destination that is not authorized to receive them under the export control laws and regulations of the United States.

The Software and Documentation are provided with Restricted Rights. Use, duplication or disclosure by the U.S. government is subject to restrictions as set forth in (a) this Agreement pursuant to DFARs 227.7202-3(a); (b) subparagraph (c)(1)(i) of the Rights in Technical Data and Computer Software clause at DFARs 252.227-7013; or (c) the Commercial Computer Software Restricted Rights clause at FAR 52.227-19 subdivision (c)(1) and (2), as applicable. Contractor/ manufacturer is the OPC Foundation, P.O. Box 140524, Austin, Texas 78714-0524.

Should any provision of this Agreement be held to be void, invalid, unenforceable or illegal by a court, the validity and enforceability of the other provisions shall not be affected thereby.

This Agreement shall be governed by and construed under the laws of the State of Minnesota, excluding its choice or law rules.

This Agreement embodies the entire understanding between the parties with respect to, and supersedes any prior understanding or agreement (oral or written) relating to, the OPC Materials.

Table of Contents

1. Introduction

1.1 Readers Guide

This document contains common rules and design criteria and the specification of interfaces which are common for several topics.

Specific interface specifications to develop OPC clients and/or OPC Servers (e.g., for DataAccess, Alarm&Event Handling or Historical DataAccess) are available as separate documents.

Chapter 1 is this Readers Guide.

Chapter 2 describes the fundamentals of the design and characteristics of OPC components.

Chapter 3 describes issues that are common to all OPC interfaces.

Chapter 4 specifies the shutdown capability of OPC Servers.

Chapter 5 specifies IOPCCommon, an interface that is also “common” to all types of OPC Servers.

Chapter 6 gives general information about OPC Server registration.

Chapter 7 specifies the interface for OPC Server Browsing.

Appendix A contains the IDL of the common interfaces.

Finally, Appendix B specifies a sample string filter function. It defines the minimum filtering required on various methods of the OPC Server Interfaces.

2. OPC Design Fundamentals

OPC is based on Microsoft's OLE/COM technology.

2.1 Interface Definitions

OPC specifications always contain two sets of interfaces; Custom Interfaces and Automation interfaces. This is shown in Figure 2-1.

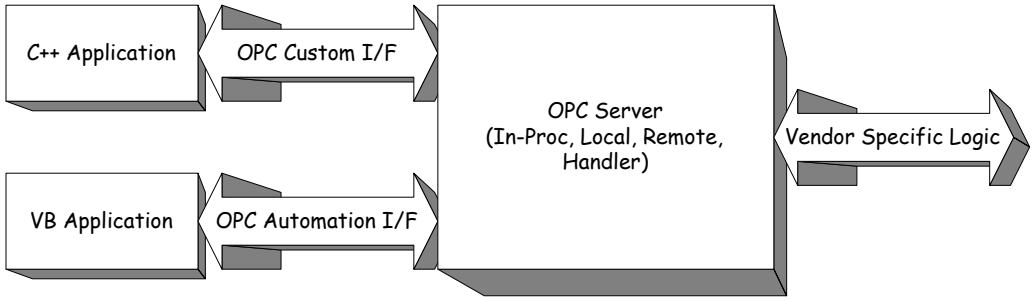

Figure 2-1 - The OPC Interfaces

An OPC client application communicates to an OPC server through the specified custom and automation interfaces. OPC servers must implement the custom interface, and optionally may implement the automation interface. In some cases the OPC Foundation provides a standard automation interface wrapper. This "wrapperDLL" can be used for any vendor-specific custom-server.

2.1.1 Required Interface Definition

OPC server developers must implement all functionality of required interfaces. An OPC client communicates to an OPC server by calling functions from the OPC required interfaces.

2.1.2 Optional Interface Definition

OPC server developers may implement the functionality of the optional interfaces

An optional interface is one that the server developer may elect to implement. When an OPC Server supports an optional interface, all functions within that optional interface must be implemented, even if the function just returns E_NOTIMPL. An OPC client that wishes to use the functionality of an optional interface will query the OPC server for the optional interface. The client must be designed to not require that this optional interface exist.

2.1.3 Which interface should the client application use.

In general, client programs which are created using scripting languages will use the automation interface. Client programs which are created in C++ will find it easiest to use the custom interface for maximum performance.

2.2 UNICODE, NT and WIN95

All string parameters to the OPC Interfaces are UNICODE, because the native OLE APIs are all UNICODE. Microsoft Visual Basic 4.0 and higher is UNICODE internally and, while it normally converts strings to ANSI when calling a DLL, it will pass strings directly as UNICODE where a corresponding TYPELIB indicates this should be done (as it will for OPC).

At the time of this writing, MIDL 3.0 or later is required in order to correctly compile the IDL code and generate proxy/stub software. Microsoft Windows NT 4.0 (or later), or Windows 95 with DCOM support is required to properly handle the marshaling of OPC parameters.

Note that in order to implement OPC servers which will run on both Microsoft Windows NT and Microsoft Windows 95 it is necessary for these servers to test the platform at runtime. In the case of Microsoft Windows 95, conversion of any strings to be passed to Win32 from UNICODE to ANSI needs to be done.

2.3 Threads and Multitasking

This specification does NOT require any particular threading model for the server.

The topic of multiple threads and their relationship to OLE is important. While these issues are also difficult to summarize, the performance gains for a medium to large scale server are worth the investment.

For OPC Servers

For servers, the default handling of threads by OLE is very simplistic. OLE will use one thread per local or remote server to handle all requests for all clients. An alternate approach is referred to 'Apartment Model Threading' where all OLE calls into an OLE server are guaranteed to be serialized. The apartment model simplifies the issues surrounding. multiple client access.

An advantage to this single threaded approach is that it simplifies implementation of servers with respect to reentrancy issues. Since all method calls are serialized automatically by the message loop, methods are never reentered or interrupted by other methods. Another advantage is that it insures (as required by COM) that all access to an object is done by the thread that created the object.

The major disadvantage of this single threaded approach is that all method calls must run to completion without significant delay. Any delay by a call prevents execution of the message loop and dispatch of additional requests, thus blocking all clients of the server. This means that a data read or write will need to be buffered so as not to seriously compromise speed. In particular, this means that physical communications (unless they are very fast) should be handled by a separate thread within the server (clearly logic related to data handling by this thread would need to be thread safe). This in turn makes write verification and error handling for writes more difficult. These issues are reflected in the design of the interfaces, particularly in the areas of 'allowed behavior'. It will be noted later that the design allows for optional Read and Write modes where the data is read or written directly to the device.

For OPC Clients

It is currently a requirement of COM that an object be accessed only by the thread that created it. This applies both to the actual objects in the server and to any 'proxy' objects represented by a marshaling stub or handler. Note that there are ways to partially relax this constraint (e.g. through the use of CoMarshallInterThreadInterfaceInStream()) however,this simply routes all method calls back through the thread that created the object and this involves considerable overhead. In addition, no matter how many threads attempt to access the objects in parallel, they will all be gated by the operation of the dispatch loop in the thread owning the object which will tend to negate any performance improvement.

Note the general OLE rule that code within asynchronous OLE methods (e.g. OnDataChange) cannot make synchronous or asynchronous OLE calls.

3. OPC Common Interface Issues

3.1 Common Interface Issues

This section describes issues which are common to all interfaces, and some background information about how the designers of OPC expected these interfaces to be implemented and used.

3.1.1 Custom vs. Automation Interface

OPC specifications always contain two sets of interfaces; Custom Interfaces and Automation Interfaces. It has been found that it is not possible to define a single (dual-automation) interface which is both highly efficient and provides the look-and-feel of typical automation servers, like Excel.

In general, client programs which are created using scripting languages, like Visual Basic (or VBA) will use the automation interface. Client programs which are created in C++ will find it easiest to use the custom interface for maximum performance.

OPC servers must implement the custom interface, and optionally may implement the automation interface. The OPC Foundation provides a standard automation interface wrapper. This "wrapperDLL" can be used for any vendor-specific custom-server.

3.1.2 Required vs Optional Interface Definition

OPC server developers must implement all functionality of required interfaces. An OPC client communicates to an OPC server by calling functions from the OPC required interfaces.

OPC server developers may implement the functionality of the optional interfaces.

An optional interface is one that the server developer may elect to implement. When an OPC Server supports an optional interface, all functions within that optional interface must be implemented, even if the function just returns E_NOTIMPL. An OPC client that wishes to use the functionality of an optional interface will query the OPC server for the optional interface. The client must be designed to not require that this optional interface exist.

3.1.3 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, otherwise they will experience memory leaks that are difficult to find. See the IDL files to determine which parameters are out parameters. The recommended approach is for a client to create a subroutine that is used for freeing each type of structure properly.

Independent of success/failure, the server must always return well defined values for 'out' parameters. Releasing the allocated resources is the client's responsibility.

Note: If the error result is any FAILED error such as E_OUTOFMEMORY , the OPC server should return NULL for all `out' pointers (this is standard COM behavior). This rule also applies to the error arrays (ppErrors) returned by many of the functions below. In general, a robust OPC client should check each out or in/out pointer for NULL prior to freeing it.

3.1.4 Null Strings and Null Pointers

Both of these terms are used. They are NOT the same thing. A NULL Pointer is an invalid pointer (0) which will cause an exception if used. A NUL String is a valid (non zero) pointer to a 1 character array where that character is a NUL (i.e. 0). If a NUL string is returned from a method as an [out] parameter (or as an element of a structure) it must be freed, otherwise the memory containing the NUL will be lost. Also note that a NULL pointer cannot be passed for an [in,string] argument due to COM

marshalling restrictions. In this case a pointer to a NUL string should be passed to indicate an omitted parameter.

3.1.5 Returned Arrays

You will note the syntax **size_is(,dwCount)** in the IDL of several interfaces 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 , creation, and access of the data by the server and client.

3.1.6 Errors and return codes

The OPC specifications describe interfaces and corresponding behavior that an OPC server implements, and an OPC client application depends on. A list of errors and return codes is contained in each specification. For each method described a list of all possible OPC error codes as well as the **most common** OLE error codes is included. It is likely that clients will encounter additional error codes such as RPC and Security related codes in practice and they should be prepared to deal with them.

In all cases 'E' error codes will indicate FAILED type errors and 'S' error codes will indicate at least partial success.

4. Shutdown of OPCServers

The shutdown capability allows an OPC Server to request that all clients disconnect from the server. It is provided for all types of OPC Servers (DataAccess, Alarm&Event, ...).

The functionality is available via a Connection point on the Server object and a corresponding Client side IOPCShutdown interface. Clients should make use of this feature to support graceful shutdown.

4.1 IConnectionPointContainer (on OPCServer)

This interface provides access to the connection point for IOPCShutdown.

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 are well defined by Microsoft and are not discussed here.

Note: OPC Compliant servers are not required to support more than one connection between each Server and the Client. Given that servers are client specific entities it is expected that a single connection will be sufficient for virtually all applications. For this reason (as per the COM Specification) the EnumConnections method for IConnectionPoint interface for the IOPCShutdown is allowed to return E_NOTIMPL.

4.1.1 IConnectionPointContainer::EnumConnectionPoints

```
HRESULT         EnumConnectionPoints(
                IEnumConnectionPoints **ppEnum
                );
```

Description

Create an enumerator for the Connection Points supported between the OPC Group and the Client.

Parameters	Description
ppEnum	Where to save the pointer to the connection point enumerator. See the Microsoft documentation for a discussion of IEnumConnectionPoints.

HRESULT Return Codes

Return Code	Description
S_OK	The function was successful.
For other codes see the OLE programmers reference	

Comments

OPCServers must return an enumerator that includes IOPCShutdown. Additional vendor specific callbacks are also allowed.

4.1.2 IConnectionPointContainer:: FindConnectionPoint

```
HRESULT FindConnectionPoint(
        REFIID riid,
        IConnectionPoint **ppCP
        );
```

Description

Find a particular connection point between the OPC Server and the Client.

Parameters	Description
ppCP	Where to store the Connection Point. See the Microsoft documentation for a discussion of IConnectionPoint.
riid	The IID of the Connection Point. (e.g. IID_IOPCShutdown)

HRESULT Return Codes

Return Code	Description
S_OK	The function was successful.
For other codes see the OLE programmers reference	

Comments

OPCServers must support IID_IOPCShutdown. Additional vendor specific callbacks are also allowed.

4.2 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 server needs to shutdown. The client should release all connections and interfaces for this server.

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

4.2.1 IOPCShutdown::ShutdownRequest

```
HRESULT ShutdownRequest (
   [in] LPWSTR  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, Remove all groups and release all interfaces.

Parameters	Description
szReason	An optional text string provided by the server indicating the reason for the shutdown. The server may pass a pointer to a NUL string if no reason is provided.

HRESULT Return Codes

Return Code	Description
S_OK	The client must always return S_OK.

Comments

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

5. IOPCCommon

This interface is used by all OPC Server types (DataAccess, Alarm&Event, Historical Data). It provides the ability to set and query a LocaleID which would be in effect for the particular client/server session. That is, the actions of one client do not affect any other clients.

As with other interfaces such as IUnknown, the instance of this interface for each server is unique. That is, an OPC Data Access server object and and OPC Alarms and Events server object might both provide an implementation of IOPCCommon. A client which is maintaining connections to both servers would, as with any other interface, use the interfaces on these two objects independently.

5.1.1 IOPCCommon::SetLocaleID

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

Description

Set the default LocaleID for this server/client session. This localeid will be used by the GetErrorString method on this interface. It should also be used as the 'default' localeid by any other server functions that are affected by localid. Other OPC interfaces may provide additional LocaleID capability by allowing this LocalID to be overridden either via a parameter to a method or via a property on a child object.

Parameters	Description
dwLcid	The default LocaleID for this server/client session

Return Codes

Return Code	Description
E_FAIL	The operation failed.
E_INVALIDARG	An argument to the function was invalid. (For example, the LocaleID specified is not valid.)
S_OK	The operation succeeded.

Comments

The default value for the server should be LOCALE_SYSTEM_DEFAULT.

5.1.2 IOPCCommon::GetLocaleID

```
HRESULT GetLocaleID (
        [out] LCID *pdwLcid
        );
```

Description

Return the default LocaleID for this server/client session.

Parameters	Description
pdwLcid	Where to return the default LocaleID for this server/client session

Return Codes

Return Code	Description
E_FAIL	The operation failed.
E_INVALIDARG	An argument to the function was invalid. (For example, the passed pointer is not valid.)
S_OK	The operation succeeded.

Comments

5.1.3 IOPCCommon::QueryAvailableLocaleIDs

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

Description

Return the available LocaleIDs for this server/client session.

Parameters	Description
pdwCount	Where to return the LocaleID count
pdwLcid	Where to return the LocaleID list.

Return Codes

Return Code	Description
E_FAIL	The operation failed.
E_INVALIDARG	An argument to the function was invalid. (For example, the passed pointer is not valid.)
S_OK	The operation succeeded.

Comments

5.1.4 IOPCCommon::GetErrorString

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

Description

Returns the error string for a server specific error code.

Parameters	Description
dwError	A server specific error code that the client application had returned from an interface function from the server, and for which the client application is requesting the server's textual representation.
ppString	Pointer to pointer where server supplied result will be saved

Return Codes

Return Code	Description
E_FAIL	The operation failed.
E_OUTOFMEMORY	Not enough memory
E_INVALIDARG	An argument to the function was invalid. (For example, the error code specified is not valid.)
S_OK	The operation succeeded.

Comments

The expected behavior is that this will include handling of Win32 errors as well (such as RPC errors).

Client must free the returned string.

It is recommended that the server put any OPC specific strings into an external resource to simplify translation.

Note that if this method is being called via DCOM then it is very possible that RPC or other network related errors will be returned. For this reason it is probably good practice for the client to attempt to call a local Win32 function such as FormatMessage if this function fails.

5.1.5 IOPCCommon::SetClientName

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

Description

Allows the client to optionally register a client name with the server. This is included primarily for debugging purposes. The recommended behavior is that the client set his Node name and EXE name here.

Parameters	Description
szName	An arbitrary string containing information about the client task.

Return Codes

Return Code	Description
E_FAIL	The operation failed.
E_INVALIDARG	An argument to the function was invalid. (For example, the pointer specified is not valid.)
S_OK	The operation succeeded.

Comments

6. Installation and Registration Issues

This section describes all installation issues which are common to all OPC Servers (no matter which interfaces they implement). Specific installation and registration issues will be described in the interface-specific documents.
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 Component Categories

With the possibly huge amount of available components on a single computer system, their management becomes increasingly difficult. OPC Clients often need to enumerate the OPC Servers that they want to use in a certain context. In its first version, OPC specified a sub-key called OPC to tag the OPC Server entries in the registry. Clients have to browse for this subkey. This method is inefficient as it requires browsing all CLSID entries in the registry. Name collisions may occur. And finally, access to remote registries will be restricted in NT5.0.

For all server specifications past DataAccess 1.0A, OPC uses Component Categories as a way to categorize OPC Servers by their implemented functionality. Clients can use the new interface IOPCServerList to obtain a list of servers with the required functionality. See the following chapter for the specification of this interface

OPC defines “implemented categories” for each version of each OPC Interface specification. Each category is identified by a globally unique identifier (GUID), the CATID. CATIDs are specified in the registry section of each specification.
It is expected that a server will first create any category it uses and then will register for that category. Unregistering a server should cause it to be removed from that category. See the ICatRegister documentation for additional information.
A single server may belong to more than one category. I.e., it may support DataAccess Versions 1.0A and 2.0 and in addition Alarm&Event Handling.

6.1.1 Component Categories Registration

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

6.1.1.1 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 from the Alarm & Events Sample Server:

```
#include <comcat.h>

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

The OPC Alarm & Events 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 );
}
```

6.1.1.2 Client Enumeration

Clients will use the Interface IOPCServerList to obtain a list of servers either locally or on a remote host. This interface basically provides the functionality of the Component Categories Manager. It has been defined by OPC, because access to the Component Categories Manager does not work for remote machines.

See the following chapter for the specification of IOPCServerList.

6.2 Registry Entries for the Proxy/Stub DLL

The proxy/stub DLLs are used for marshalling interfaces to LOCAL or REMOTE servers. It is generated directly from the IDL code and should be the same for every OPC Server. In general the Proxy/Stub will use self registration. (Define REGISTER_PROXY_DLL during the build). Since this is completely automatic and transparent it is not discussed further.
Also note that a prebuilt and tested proxy/stub DLL will be provided at the OPC Foundation Web site making it unnecessary for vendors to rebuild this DLL.

Although vendors are allowed to add their own interfaces to OPC objects (as with any COM object) they should NEVER modify the standard OPC IDL files or Proxy/Stub DLLs to include such interfaces. Such interfaces should ALWAYS be defined in a separate vendor specific IDL file and should be marshalled by a separate vendor specific Proxy/Stub DLL.

6.3 Creating the Registry Entries

COM defines a “self-registration” mechanism that enables you to encapsulate registry needs into a DLL or EXE, providing clients and servers an easy way to make sure that any given module is fully and accurately registered. In addition, COM also includes “unregistration” so that a server can remove all of its registry entries when the DLL or EXE is removed from the file system, thereby keeping the registry clean from useless entries.

When asked to self-register, a server must create *all* entries for *every* component that it supports, including any entries for type libraries. When asked to “un-register” the server must remove those entries that it created in its self-registration.

For a DLL server, these requests are made through calls to the exported functions *DllRegisterServer* and *DllUnregisterServer*, which must exist in the DLL under these exact names. Both functions take no arguments and return an HRESULT to indicate the result. The two applicable error codes are SELFREG_E_CLASS (failure to register/unregister CLSID information) and SELFREG_E_TYPELIB (failure to register/unregister TypeLib information).[1]

If the server is packaged in an EXE module, then the application wishing to register the server launches the EXE server with the command-line argument /RegServer or -RegServer (case-insensitive). If the application wishes to unregister the server, it launches the EXE with the command-line argument /UnregServer or -UnregServer. The self-registering EXE detects these command-line arguments and invokes the same operations as a DLL would within DllRegisterServer and DllUnregisterServer, respectively, registering its module path under LocalServer32 instead of InprocServer32 or InprocHandler32.

The server must register the full path to the installation location of the DLL or EXE module for their respective InprocServer32, InprocHandler32, and LocalServer32 keys in the registry. The module path is easily obtained through the Win32 API function GetModuleFileName.

NOTE: The server should NOT register the proxy/stub interfaces. They should be registered by the proxy/stub DLL as discussed earlier.

The registry entries for proxy interfaces can be easily generated when compiling the proxy dll. Simply define the constant REGISTER_PROXY_DLL during compilation, and export DllRegisterServer and DllUnregisterServer during the link. One can now populate the registry by executing regsvr32 and passing the proxy dll name as an argument.

The following are the Microsoft COM required registry entries for a local server (EXE) shown in Registry File (.reg) format:

```
REGEDIT

HKEY_CLASSES_ROOT\MyVendor.ServerName.1 = My OPC Server Description
HKEY_CLASSES_ROOT\MyVendor.ServerName.1\CLSID = {Your Server's unique CLSID}

HKEY_CLASSES_ROOT\CLSID\{Your Server's unique CLSID} = My OPC Server Description
HKEY_CLASSES_ROOT\CLSID\{Your Server's unique CLSID}\ProgID = MyVendor.ServerName.1
HKEY_CLASSES_ROOT\CLSID\{Your Server's unique CLSID}\LocalServer32 = c:\FULLPATH\MyOPCserver.exe
```

The following are the Microsoft COM required registry entries for an Inproc server (DLL) shown in Registry File (.reg) format:

```
REGEDIT

HKEY_CLASSES_ROOT\MyVendor.ServerName.1 = My OPC Server Description
HKEY_CLASSES_ROOT\MyVendor.ServerName.1\CLSID = {Your Server's unique CLSID}

HKEY_CLASSES_ROOT\CLSID\{Your Server's unique CLSID} = My OPC Server Description
HKEY_CLASSES_ROOT\CLSID\{Your Server's unique CLSID}\ProgID = MyVendor.ServerName.1
HKEY_CLASSES_ROOT\CLSID\{Your Handler's unique CLSID}\InprocServer32 = c:\FULLPATH\MyOPCserver.dll
```

The following are the OPC required registry entries for all Data Access 1.0 servers shown in Registry File (.reg) format. ***Only servers that support the Data Access 1.0 interface should make these entries***:

```
REGEDIT

HKEY_CLASSES_ROOT\MyVendor.ServerName.1\OPC
HKEY_CLASSES_ROOT\MyVendor.ServerName.1\OPC\Vendor = My Vendor Name
```

[1] SELFREG_E_CLASS and SELFREG_E_TYPELIB are defined in the OLE Control's header OLECTL.H.

6.4 Version Convention

All OPC provided runtime files (DLLs and EXEs) will contain version information embedded in the file's resource. By convention, the version number will use the following format:

```
MM.mm.bb
```

Where:

MM == Major Version
mm == Minor Version
bb == Build Number

The version resource provides two version numbers, one for file and one for product. The same version number will be used for both fields. In the resource, the version numbers are represented by four comma delimited integers. To represent our three-part version number, the third integer will always be zero. For example, if the version is 5.2.41 then the version resource (in the source .RC file) will look like this:

```
VS_VERSION_INFO VERSIONINFO
 FILEVERSION 5,2,0,41
 PRODUCTVERSION 5,2,0,41
 FILEFLAGSMASK 0x3fL
#ifdef _DEBUG
 FILEFLAGS 0x1L
#else
 FILEFLAGS 0x0L
#endif
 FILEOS 0x40004L
 FILETYPE 0x2L
 FILESUBTYPE 0x0L
BEGIN
    BLOCK "StringFileInfo"
    BEGIN
        BLOCK "040904b0"
        BEGIN
            VALUE "CompanyName", "OPC Foundation \0"
            VALUE "FileDescription", "OPC Alarm and Event Server Proxy/Stub\0"
            VALUE "FileVersion", "5.2.41\0"
            VALUE "InternalName", "opc_aeps\0"
            VALUE "LegalCopyright", "Copyright © 1997 OPC Foundation\0"
            VALUE "OriginalFilename", "opc_aeps.dll\0"
            VALUE "ProductName", "OPC Alarm and Event Server Proxy/Stub\0"
            VALUE "ProductVersion", "5.2.41\0"
        END
    END
    BLOCK "VarFileInfo"
    BEGIN
        VALUE "Translation", 0x409, 1200
    END
END
```

The version information will be used to insure that during installation, an older version of a file will not overwrite a newer version.

6.5 Installing OPC Binaries

All OPC vendors will need to install the appropriate OPC Foundation provided components (proxy/stub DLLs, Automation wrappers etc.) to work with their components.

> ***Since multiple vendors will be installing identical OPC Foundation components, it is imperative that all vendors follow these installation instructions exactly without deviation:***

All OPC Foundation binaries must be installed and registered in the Windows Systems directory. This is the directory returned by the WIN32 function GetSystemDirectory. If a given file already exists in this directory, the program should overwrite it with your application file only if your file is a more

recent version. The GetFileTime, GetFileVersionInfo, and GetFileInformationByHandle functions can be used to determine which file is more recent.

All OPC Foundation binaries must be installed/uninstalled with reference counting.
After copying a file, your installation program must make sure to increment the usage counter for that file in the registry. When removing an application, it should decrement the use counter. If the result is zero, the user should be given the option of unregistering and deleting the file. The user should be warned that other applications may actually use this file and will not work if it is missing. The registry key used for reference counting of all files is:

```
\HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\SharedDLLs
```

The following example shows a reference count of 5 for OPCPROXY.DLL and a reference count of 3 for OPCENUM.EXE:

```
\HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\SharedDLLs
       C:\WINNT\System32\OPCPROXY.DLL=5
       C:\WINNT\System32\OPCENUM.EXE=3
```

Most installation utilities like InstallShield handle the installation of shared, version checked files easily.

7. OPC Server Browser

7.1 Overview

The OPC Foundation supplied Server Browser OPCENUM.EXE can reside on any machine, will access the local Component Categories Manger and provides a new interface *IOPCServerList* which can be marshaled and used by remote clients. This server has a published classid (see below) and can be installed once on any machine which hosts OPC servers. The client still needs to know the nodename of the target machine however he can now create this object remotely and use it's *IOPCServerList* interface to determine what types and brands of servers are available on that machine.

7.2 Information for Users

The OPC Server Browser (OPCENUM.EXE) and the required proxy/stub (OPCCOMN_PS.DLL) can be obtained from the OPC Foundation Web Site. The EXE and DLL should be copied to the main WINDOWS directory (see the section “Installing OPC Binaries”, above).

The EXE is installed by running

```
OPCENUM /RegServer
```

or

```
OPCENUM /Service
```

to install the server as a service on Windows NT.

The DLL is installed by running

```
REGSVR32 OPCComn_ps.dll
```

No further user action is required. Doing the steps above will allow Client programs you have purchased which support this server browser capability to function properly. Note that the OPC Server Browser is designed to allow access by any user regardless of the DCOM security setup.

7.3 Information for Server Programmers

Note that the OPC Foundation provides the OPC Browser Object. OPC Servers should NOT implement this interface. OPC Servers should simply register themselves with the appropriate component category as described on the appropriate OPC Specification.

7.4 Information for Client Programmers

Client programmers should create the OPC Server Browser Object on the target machine by passing its class id (CLSID_OPCServerList as defined in **opc_cats.c**) to CoCreateInstanceEx. They should obtain the OPCServerList interface (IID_IOPCServerList as defined in **opccomn_i.c**). They can then use this interface to obtain lists of the available servers for particular component categories. The OPC Component categories for the various OPC Server types are defined in **opc_cats.c**. The marshalling for this interface is included in the OPCComn_ps.dll.

7.5 IOPCServerList Reference

The interface is designed to be as simple as possible to use. It is similar to the standard ICatInformation but has been simplified and also modified so that it can work remotely. It provides just the minimum functionality required for this particular application. It provides the methods which are described in more detail later.

7.5.1 IOPCServerList::EnumClassesofCategory

```
HRESULT EnumClassesOfCategories(
            [in] ULONG cImplemented,
            [in,size_is(cImplemented)] CATID rgcatidImpl[],
            [in] ULONG cRequired,
            [in,size_is(cRequired)] CATID rgcatidReq[],
            [out] IEnumGUID** ppenumClsid);
```

Description

Returns a standard EnumCLSID containing the CLSIDs of the servers that implement any of the listed categories on the target machine. This method is similar to the method of the same name provided in ICatInformation except that the caller should use a value of 0 instead of –1 for the cImplemented and cRequired arguments to include classes regardless of which classes they implement or require (respectively).

Note that the easiest way to use this method is to pass in a single CATID (such as an OPC Data Access 2.0 Server) and to pass a 0 for Required IDs. This will give you an enumeration of the CLSIDs of the servers that implement the specified category.

Parameters	Description
cImplemented	0 (see description, above) The number of category IDs in the *rgcatidImpl* array
rgcatidImpl	An array of category identifiers.
cRequired	0 (see description, above) The number of category IDs in the *rgcatidReq* array.
rgcatidReq	An array of category identifiers.
ppenumClsid	The location in which to return an **IEnumGUID** interface that can be used to enumerate the CLSIDs of the classes that implement category *rcatid*.

Return Codes

Return Code	Description
E_FAIL	The operation failed.
REGDB_E_CLASSNOTREG	Unable to create an instance of the Component Categories Manager on the remote machine.
E_INVALIDARG	One or more arguments are incorrect.
E_OUTOFMEMORY	Insufficient memory to create and return an enumerator object.
S_OK	The operation succeeded.

7.5.2 IOPCServerList::GetClassDetails

```
HRESULT GetClassDetails(
            [in] REFCLSID clsid,
            [out] LPOLESTR* ppszProgID,
            [out] LPOLESTR* ppszUserType);
```

Description

Given a class ID, obtain the ProgID and the User Readable Name of the associated server.

Parameters	Description
clsid	One of the CLSIDs returned by EnumClassesOfCategory (above).
ppszProgID	[out] ProgID for the specified CLSID.
ppszUserType	[out] User Readable Name for the specified CLSID.

Return Codes

Return Code	Description
E_FAIL	The operation failed.
REGDB_E_CLASSNOTREG	There is no CLSID registered for the class object.
REGDB_E_READREGDB	There was an error reading the registry.
OLE_E_REGDB_KEY	The *ProgID = MainUserTypeName* or *CLSID = MainUserTypeName* keys are missing from the registry.
E_INVALIDARG	One or more arguments are incorrect.
E_OUTOFMEMORY	Insufficient memory to create and return an enumerator object.
S_OK	The operation succeeded.

7.5.3 IOPCServerList::CLSIDFromProgID

```
HRESULT CLSIDFromProgID(
            [in] LPCOLESTR szProgId,
            [out] LPCLSID clsid);
```

Description

Given the ProgID which as a string, return the CLSID which is a GUID. This is useful when the client (e.g. an Automation Wrapper DLL) already knows the PROGID of the target server on a remote machine. ProgID is a string and thus easy to deal with however this needs to be translated to a CLSID to be passed to CoCreateInstanceEx.

Parameters	Description
szProgId	ProgID string for which to read the CLSID.
clsid	[out] CLSID which is registered for the given ProgID.

Return Codes

Return Code	Description
E_FAIL	The operation failed.
REGDB_E_CLASSNOTREG	There is no CLSID registered for the class object.
REGDB_E_READREGDB	There was an error reading the registry.
OLE_E_REGDB_KEY	The *ProgID = MainUserTypeName* or *CLSID = MainUserTypeName* keys are missing from the registry.
E_INVALIDARG	One or more arguments are incorrect.
E_OUTOFMEMORY	Insufficient memory to create and return an enumerator object.
S_OK	The operation succeeded.

8. Appendix A – OPC Common 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 opcda.idl.

The resulting **OPCCOMN.H** file should be **included** in all clients and servers.

The resulting **OPCCOMN_I.C** file defines the interface IDs and should be **linked** into all clients and servers.

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.

```
// OPCCOMN.IDL
// REVISION:  04/06/98 08:00 PM (EST)
// VERSIONINFO    1.0.0.0
//
// 04/09/98 acc import unknwn.idl rather than oaidl.idl
// 06/15/98 acc add 'library' object at end to allow typelib generation
// 06/19/98 acc change V2 uuids prior to final release
//              to avoid conflict with 'old' OPCDA Automation uuids
// 09/18/98 acc add OPCServerList IDL (with help from Gary Klassen)
//

import "unknwn.idl";
import "comcat.idl";


//*****************************************************
// All servers except OPCDA1.0 have the ability to
// make callbacks into the client on shutdown via
// IOPCShutdown
//*****************************************************
[
  object,
  uuid(F31DFDE1-07B6-11d2-B2D8-0060083BA1FB),
  pointer_default(unique)
]
interface IOPCShutdown : IUnknown
{
  HRESULT ShutdownRequest (
    [in, string] LPCWSTR szReason
    );
}

//*****************************************************
// All servers except OPCDA1.0 support IOPCCommon
//*****************************************************
[
  object,
  uuid(F31DFDE2-07B6-11d2-B2D8-0060083BA1FB),
  pointer_default(unique)
]
interface IOPCCommon : IUnknown
```

```
{

  HRESULT SetLocaleID (
    [in] LCID dwLcid
    );

  HRESULT GetLocaleID (
    [out] LCID *pdwLcid
    );

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

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

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

}


//*****************************************************
// The OPCEnum.EXE object provided by the OPC Foundation
// supports the IOPCServerList interface via DCOM
// to allow clients to determine available OPC servers
// on remote machines
//*****************************************************

[
      object,
      uuid(13486D50-4821-11D2-A494-3CB306C10000),
      pointer_default(unique)
]
interface IOPCServerList : IUnknown
{
      HRESULT EnumClassesOfCategories(
            [in] ULONG cImplemented,
            [in,size_is(cImplemented)] CATID rgcatidImpl[],
            [in] ULONG cRequired,
            [in,size_is(cRequired)] CATID rgcatidReq[],
            [out] IEnumGUID** ppenumClsid);

      HRESULT GetClassDetails(
            [in] REFCLSID clsid,
            [out] LPOLESTR* ppszProgID,
            [out] LPOLESTR* ppszUserType);

      HRESULT CLSIDFromProgID(
            [in] LPCOLESTR szProgId,
            [out] LPCLSID clsid);
};
```

```
//************************************************
// This TYPELIB is generated as a convenience to users of high level
tools
// which are capable of using or browsing TYPELIBs.
// 'Smart Pointers' in VC5 is one example.
//************************************************
[
    uuid(B28EEDB1-AC6F-11d1-84D5-00608CB8A7E9),
    version(1.0),
    helpstring("OPCCOMN 1.0 Type Library")
]
library OPCCOMN
{
    importlib("stdole32.tlb");
    importlib("stdole2.tlb");

    interface IOPCCommon;
    interface IOPCShutdown;
      interface IOpcServerList;
};
```

9. Appendix B – 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:

```
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( LPCTSTR String, LPCTSTR Pattern, BOOL bCaseSensitive )
{
    TCHAR   c, p, l;
    for (; ;)
    {
        switch (p = ConvertCase( *Pattern++, bCaseSensitive ) )
        {
        case 0:                                 // end of pattern
            return *String ? FALSE : TRUE;  // if end of string TRUE

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

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

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

                while( (p = ConvertCase( *Pattern++, bCaseSensitive) )
                         != _T('\0') )
```

```
                {
                    if (p == _T(']'))     // if end of char set, then
                        break;            // no match found

                    if (p == _T('-'))
                    {   // check a range of chars?
                        p = ConvertCase( *Pattern, bCaseSensitive );
                        // get high limit of range
                        if (p == 0  ||  p == _T(']'))
                            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) )
                         != _T('\0') )
                {
                    if (p == _T(']'))         // if end of char set, then
                        return FALSE;         // no match found

                    if (p == _T('-'))
                    {   // check a range of chars?
                        p = ConvertCase( *Pattern, bCaseSensitive );
                        // get high limit of range
                        if (p == 0  ||  p == _T(']'))
                            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 != _T(']'))     // got a match in char set
                    p = *Pattern++;             // skip to end of set
            }

            break;

        case _T('#'):
            c = *String++;
            if( !_istdigit( 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;
        }
    }
}
```