Using PLCopen OPC-UA Client Function Blocks to model MDIS Information

Tim Fortin, Senior Principal Engineer, Honeywell Process Solutions

At its core, OPC UA combines a robust, secure communication protocol with a flexible information modeling framework. A challenge for the PLC vendor deploying OPC UA in embedded devices is how to represent different customers' information modeling requirements in a common way. In particular, what does it mean for the embedded OPC UA client application to "understand" an information model? What follows is a description of the discovery process leading to one such solution.

The Oil and Gas Industry Challenge

A challenge in the upstream oil and gas industry is minimizing the engineering cost associated with integration of the various components which encompass subsea control with those components that comprise topside control. The MCS-DCS Interface Standardization (MDIS) network[1] was formed to address the desire for a standard communication interface between the subsea gateway, the MCS (Master Control System) and the DCS (Distributed Control system).

OPC UA was selected by MDIS as the protocol to provide the communication link between the MCS and the DCS. OPC UA offers a number of features which aligned with the requirements of the MDIS protocol selection process. Among these are communication integrity, redundancy, robustness and independent organization support. Additionally, and perhaps most importantly, OPC UA provides a rich information modeling framework.

The OPC UA Information Model

The OPC UA specifications define a base information model and an underlying infrastructure. This combination of information model and infrastructure enables vendors or organizations such as MDIS to define domain-specific information models. Key foundational principles include:

- Object-oriented techniques such as type hierarchies and inheritance ensures that client applications can understand and process instances of the same type in the same way.
- Instances include type information.
- Full exposure of type information as well as instance information in the address space. Both are accessed exactly the same way.
- Subtyping enables vendor specific type extensions.

When data items within the server's address space can be readily identified as having known semantics, the client application can expose this generically, irrespective of the server vendor. For MDIS, the OPC UA client that understands the semantics of an MDIS valve object can, for example, expose a suitable faceplate which represents a valve object. A single faceplate can be reused with any MDIS compliant OPC UA server, reducing the engineering effort when a system is deployed.

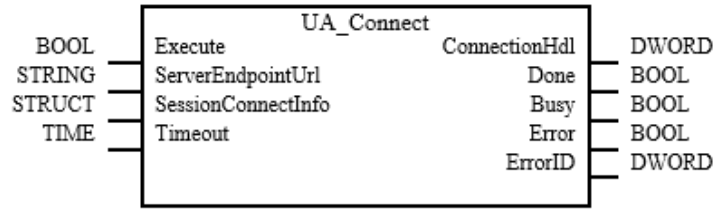
PLCopen and the Embedded OPC UA Client

While relatively easy to grasp the power of a published information model for the user of a GUI-based OPC UA client application, how can the user of an embedded OPC UA client achieve the same?

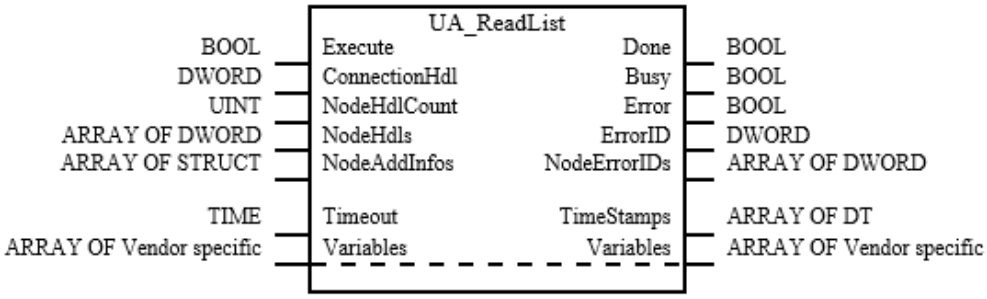
Enter the PLCopen, OPC Foundation Joint Technical Specification "PLCopen OPC-UA Client for IEC 61131-3". As stated by the PLCopen technical working group, "With this functionality implemented on a controller it becomes possible to initiate a communication session to any other available OPC UA Server. The controller can exchange complex data structures horizontally with other controllers independently from fieldbus system, or vertically with other devices using an OPC UA server call in an MES/ERP system in order to collect data or write new production orders to the cloud."

[1] http://www.mdis-network.com/

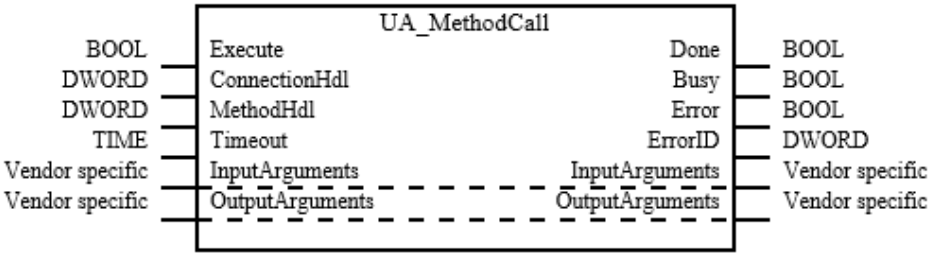
The PLCopen specification defines a number of IEC 61131-3 function blocks and associated data structures which collectively represent most OPC UA client behaviors. For example the UaConnect function block initiates a transport connection and OPC UA session:



The UA_ReadList collects the values from multiple nodes in the server's address space:



Ua_MethodCall is used to invoke a method object in the server's address space:



Circling back to the question of the embedded OPC UA client from above, it turns out that the confluence of the PLCopen OPC UA Client Function blocks and the flexibility of the 61131-3 programming model provides an answer.

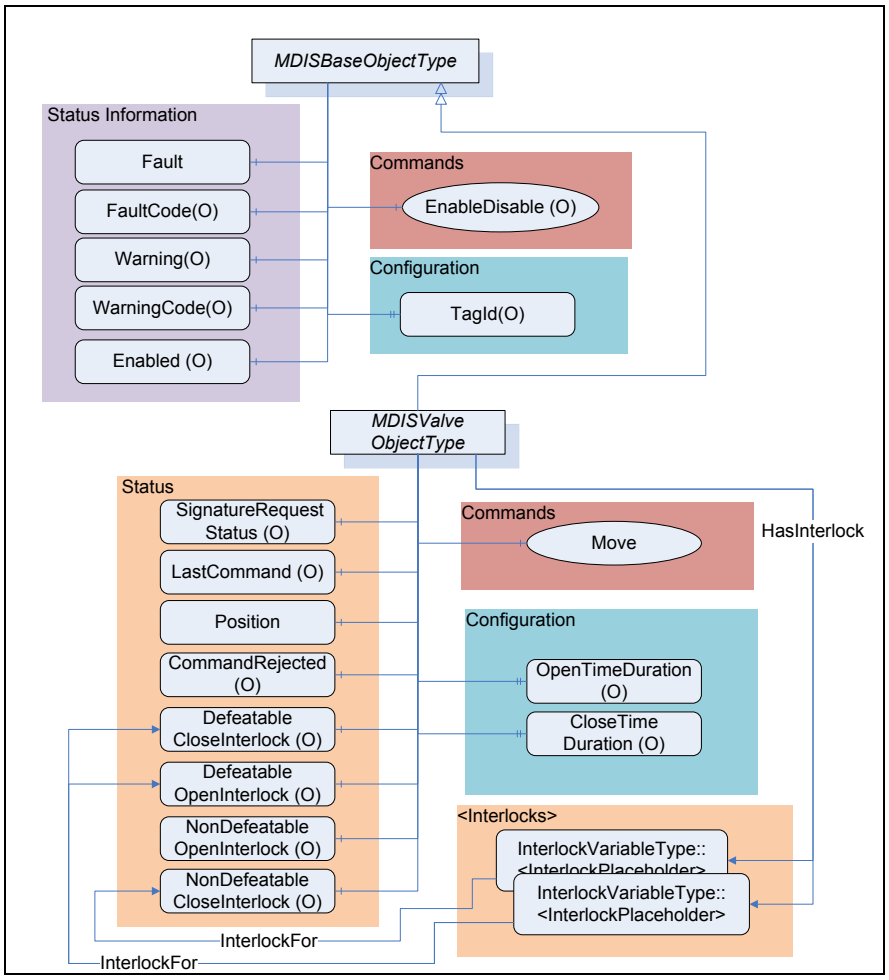
The MDIS Information Model

At its core, the MDIS information model consists of a set of object definitions representing components commonly associated with subsea control. This object set includes:

- MDISBaseObject
- MDISDiscreteInstrumentObject
- MDISDiscreteOutObject
- MDISDigitalInstrumentObject
- MDISDigitalOutObject
- MDISInstrumentObject
- MDISInstrumentOutObject
- MDISChokeObject
- MDISValveObject

Picking the MDISValveObject as representative, the “MDIS OPC UA Companion Specification” depicts the object as shown below:

MDISValveObjectType

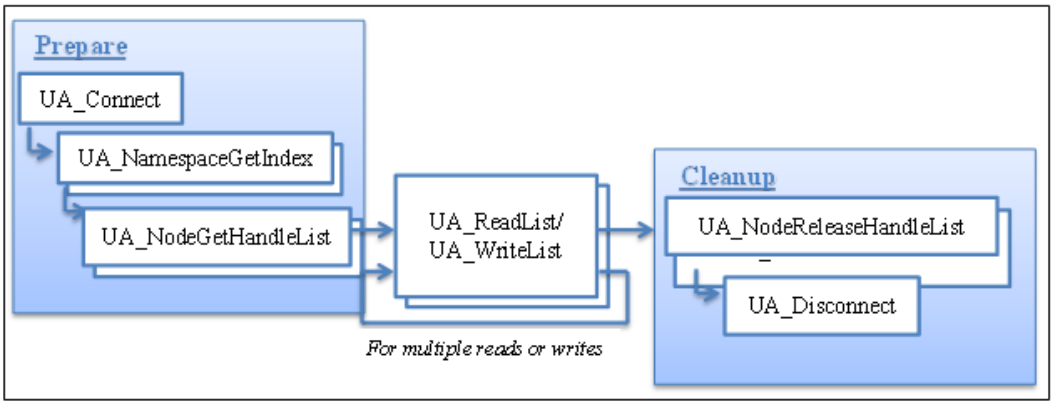


Note the hierarchical definition of MDISValveObjectType and the fact that it is a subtype of MDISBaseObjectType as are all of the MDIS object types listed earlier. This means that all MDIS objects share certain “base” features (e.g., “Fault” flag) and related semantics which are defined by the MDISBaseObjectType.

An MDIS compliant OPC UA server exposes the MDIS information model as additional OPC UA types within the server’s set of object type definitions. Instances of the MDISValveObject represent a physical subsea valve device.

Using the PLCopen OPC UA Client Function Blocks

Back to the 61131-3 environment, clearly a collection of several PLCopen function blocks can be “wired together” to enable the exchange data between the OPC UA client and a given MDIS server’s valve object. This workflow is shown conceptually in the PLCopen specification as follows:

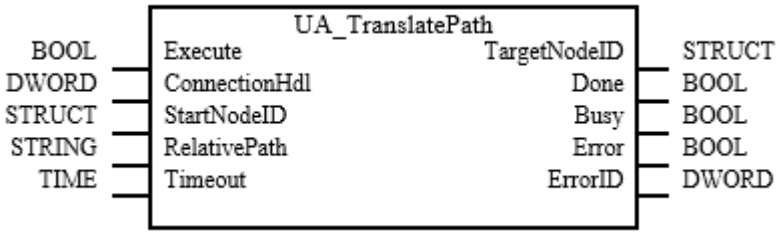


However, because each unique MDIS valve object has unique identifiers for each of its information model components, a custom configuration is required for each valve. That is, the OPC identifier for say, the "Fault" component is different for every valve, and function block configuration in the workflow depiction above needs to be adjusted for each valve accordingly in order to read the value of "Fault".

The Power of the Browse Name

When defining the model for an object – its type definition - as part of an OPC UA information model, one of the required elements is called the "Browse Name" which is a string representation assigned to each of the object's constituent components. OPC UA requires that the Browse Name associated with components of an object instance must be identical to the Browse name defined in the corresponding object type. In the case of the MDIS valve object and its Fault component, the browse name defined by the MDIS specification is "Fault" and every MDIS valve instance in any MDIS server will have a Fault component whose browse name is "Fault".

So why is this important? Among the many services supported by OPC UA servers and available to OPC UA clients is one that can provide or "translate" the unique internal identifier for a data item given its "Browse Path". A "Browse Path" is simply a Browse Name together with additional location and/or semantic context. A client interested in reading the value of the Fault component of a particular valve object simply invokes the service, requesting the unique identifier for the component whose browse name is "Fault" belonging to the specific valve object instance. Further, PLCopen defines a special function block, UA_TranslatePath, for just this purpose[2,3].



Revisiting the example workflow shown earlier, the (assumed, not shown) step in the flow between UA_NameSpaceGetIndex and UA_NodeGetHandleList which required the user to manually obtain the list of unique, server-specific identifiers for each of the data items intended as input to the UA_ReadList block can be replaced with a bit of clever structured text logic using the UA_TranslatePath block. That is, where before it was necessary to manually collect the list of unique IDs for each data item intended to be read/written either via outside browsing of the server's address space or server vendor documentation, when we know that these data items are constituent components of an MDIS valve then we can create a generic strategy which utilizes the known Browse Names for these components together with the UA_TranslatePath block. This generic strategy is able to access the components of any MDIS valve

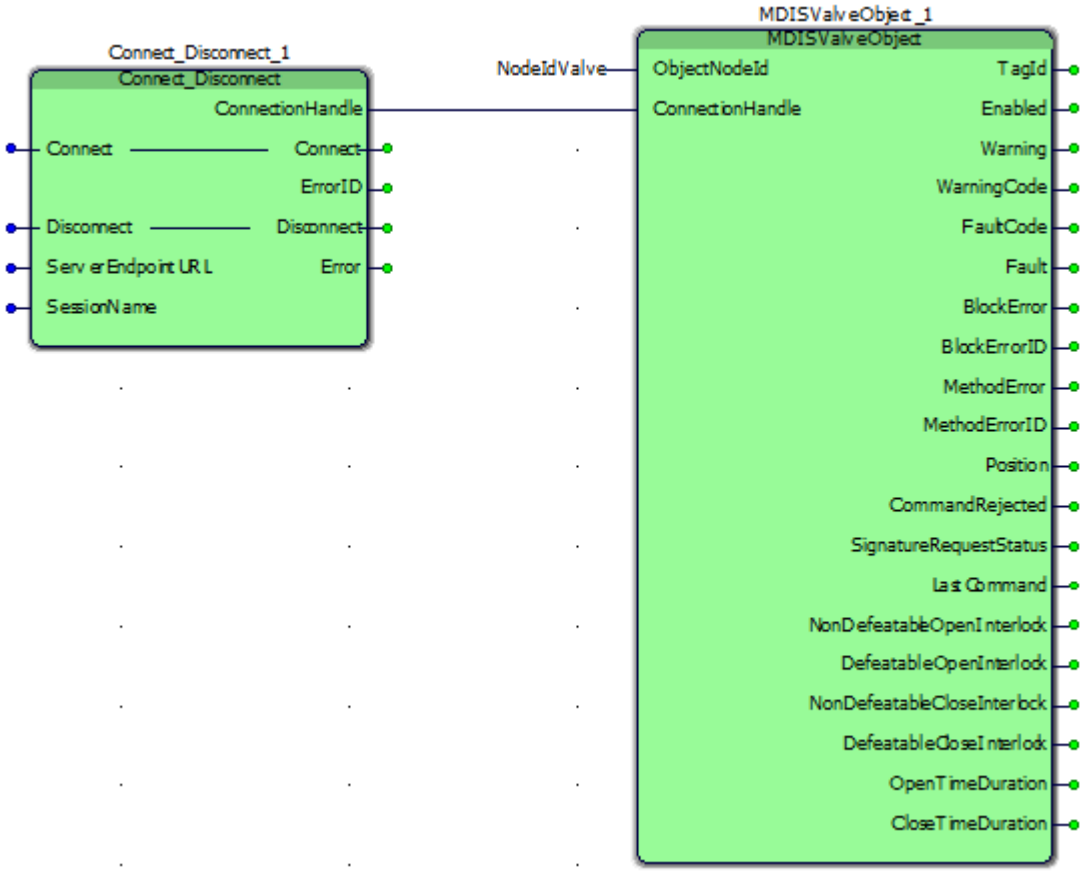
[2] In the block depicted, "Browse Path" is basically the combination of "StartNodeID" and "RelativePath"

[3] The function block depicted is from the PLCOpen specification "OPC-UA Client Function Blocks for IEC 61131-3" version 1.0. A new specification version 1.1 has recently been released which defines a new "UA_TranslatePathList" block offering greater flexibility.

object of any MDIS OPC UA server without any knowledge beforehand of the unique identifiers associated with these components.

Custom Function Blocks Incorporating the MDIS Information Model

What follows is an example from a set of custom 61131-3 function blocks designed to consume an MDIS server's information model and which utilize the translate path technique described above. It should be noted that the resultant blocks were in fact successfully tested at the MDIS Interoperability test event held this past September in Houston, TX.



The MDISValveObject custom function block shown was developed using the 61131-3 structured text programming language. Internally the block uses these PLCopen function blocks[4]:

- UA_TranslatePath
- UA_NodeGetHandleList
- UA_ReadList
- UA_MethodGetHandle
- UA_MethodCall

The programming logic that makes up the MDISValveObject is summarized below in pseudo code (PLCopen block calls shown in **bold**):

```
BEGIN PROGRAM

    (** Initialization **)
    IF (ConnectionHandle = 0) THEN
```

[4] The Connect_Disconnect block also shown is a custom "convenience" block designed to simplify connection management and which incorporates PLCOpen blocks UA_Connect and UA_Disconnect

```
        RelativePathListData[1] := 'TagId';
        RelativePathListData[2] := 'Enabled';
        RelativePathListData[3] := 'Warning';
        (** Additional paths ... **)
        RelativePathListData[16] := 'CloseTimeDuration';
        RelativePathListData[17] := 'Move';
        RelativePathListData[18] := 'EnableDisable';

        TranslatedNodeIds[];
        NodeHandleList[];
        MethodHandleList[];

        DoTranslatePaths := TRUE;
        DoGetNodeHandles := FALSE;
        DoGetMethodHandles := FALSE;
        DoReadList := FALSE;
        CallEnableDisable := FALSE;
        CallMove := FALSE;

END_IF;

(** Translate the browse paths to NodeIds **)
IF (ConnectionHandle <> 0) AND (DoTranslatePaths = TRUE) THEN
        Success := UA_TranslatePath (IN=>RelativePathList, OUT=>TranslatedNodeIds[], OUT=>ErrorID);
        DoTranslatePaths := FALSE;
        IF (Success = TRUE) THEN
                DoGetNodeHandles := TRUE;
                Error := FALSE;
        ELSE
                Error := TRUE;
        END_IF;
END_IF;

(** Obtain data node handles for the list of translated NodeIds **)
IF (ConnectionHandle <> 0) AND DoGetNodeHandles = TRUE) THEN
        Success := UA_NodeGetHandleList (IN=>TranslatedNodeIds[1..16], OUT=>NodeHandleList[], OUT=>ErrorID);
        DoGetNodeHandles := FALSE;
        IF (Success = TRUE) THEN
                DoGetMethodHandles := TRUE;
                Error := FALSE;
        ELSE
                Error := TRUE;
        END_IF;
END_IF;


(** Obtain method node handles for the list of translated NodeIds **)
IF (ConnectionHandle <> 0) AND DoGetMethodHandles = TRUE) THEN
        Success := UA_MethodGetHandle (IN=>TranslatedNodeIds[17..18], OUT=>MethodHandleList[],
OUT=>ErrorID);
        DoGetMethodHandles := FALSE;
        IF (Success = TRUE) THEN
                DoReadList := TRUE;
                Error := FALSE;
        ELSE
                Error := TRUE;
        END_IF;
END_IF;
```

```
        (** Read data values for data node handles **)
        IF (ConnectionHandle <> 0) AND DoReadList = TRUE) THEN
                (** etc. **)
        END_IF;

        (** Call the EnableDisable method **)
        IF (ConnectionHandle <> 0) AND (CallEnableDisable = TRUE) THEN
                (** etc. **)
        END_IF;

        (** Call the Move method **)
        IF (ConnectionHandle <> 0) AND (CallMove = TRUE) THEN
                (** etc. **)
        END_IF;

END PROGRAM;
```

Given the identifier of any MDIS valve object instance in any OPC UA server, this block will programmatically determine server specific identifiers for the each of the valve components, begin collecting current data values and be at the ready to invoke either of the two methods supported by this object.

In addition, once a library of custom function blocks representing all of the MDIS objects has been assembled, they themselves can be further organized into hierarchies representing aggregate device objects such as CIMVs[5] or MPFMs[6].

Summary Thoughts

What became obvious during this journey to understand how to incorporate one OPC UA information model into the world of the embedded OPC UA client is that pretty much any information model can be similarly represented. In fact, as more and more OPC UA companion specifications continue to be developed for diverse industry applications, ready-made function block libraries tailored for the various information models can easily be envisioned. To that end, it should be noted the PLCopen group is currently wrapping up work on a new specification "Creating PLCopen compliant Libraries" intended to provide consistency across different vendor libraries. Incorporating a given information model into the embedded client will eventually become a simple matter of downloading the appropriate library.

References

1. MDIS OPC UA Companion Specification, Release Candidate 1.0, June 20, 2016
2. PLCopen OPC UA Client for IEC61131-3 Version 1.1, September 5, 2016, PLCopen and OPC Foundation
3. Mahnke, W., Leitner, S., Damm, M., (2009). OPC Unified Architecture. Berlin: Springer

[5] Chemical Injection Metering Valves

[6] Multiphase Flow Meters