USE OF SPECIFICATION - TERMS, CONDITIONS & NOTICES

The material in this document details an Object Management Group specification in accordance with the terms, conditions and notices set forth below. This document does not represent a commitment to implement any portion of this specification in any company's products. The information contained in this document is subject to change without notice.

LICENSES

The companies listed above have granted to the Object Management Group, Inc. (OMG) a nonexclusive, royalty-free, paid up, worldwide license to copy and distribute this document and to modify this document and distribute copies of the modified version. Each of the copyright holders listed above has agreed that no person shall be deemed to have infringed the copyright in the included material of any such copyright holder by reason of having used the specification set forth herein or having conformed any computer software to the specification.

Subject to all of the terms and conditions below, the owners of the copyright in this specification hereby grant you a fully-paid up, non-exclusive, nontransferable, perpetual, worldwide license (without the right to sublicense), to use this specification to create and distribute software and special purpose specifications that are based upon this specification, and to use, copy, and distribute this specification as provided under the Copyright Act; provided that: (1) both the copyright notice identified above and this permission notice appear on any copies of this specification; (2) the use of the specifications is for informational purposes and will not be copied or posted on any network computer or broadcast in any media and will not be otherwise resold or transferred for commercial purposes; and (3) no modifications are made to this specification. This limited permission automatically terminates without notice if you breach any of these terms or conditions. Upon termination, you will destroy immediately any copies of the specifications in your possession or control.

PATENTS

The attention of adopters is directed to the possibility that compliance with or adoption of OMG specifications may require use of an invention covered by patent rights. OMG shall not be responsible for identifying patents for which a license may be required by any OMG specification, or for conducting legal inquiries into the legal validity or scope of those patents that are brought to its attention. OMG specifications are prospective and advisory only. Prospective users are responsible for protecting themselves against liability for infringement of patents.

GENERAL USE RESTRICTIONS

Any unauthorized use of this specification may violate copyright laws, trademark laws, and communications regulations and statutes. This document contains information which is protected by copyright. All Rights Reserved. No part of this work covered by copyright herein may be reproduced or used in any form or by any means--graphic, electronic, or mechanical, including photocopying, recording, taping, or information storage and retrieval systems--without permission of the copyright owner.

DISCLAIMER OF WARRANTY

WHILE THIS PUBLICATION IS BELIEVED TO BE ACCURATE, IT IS PROVIDED "AS IS" AND MAY CONTAIN
ERRORS OR MISPRINTS. THE OBJECT MANAGEMENT GROUP AND THE COMPANIES LISTED ABOVE
MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS PUBLICATION,
INCLUDING BUT NOT LIMITED TO ANY WARRANTY OF TITLE OR OWNERSHIP, IMPLIED WARRANTY OF
MERCHANTABILITY OR WARRANTY OF FITNESS FOR A PARTICULAR PURPOSE OR USE.
IN NO EVENT SHALL THE OBJECT MANAGEMENT GROUP OR ANY OF THE COMPANIES LISTED ABOVE
BE LIABLE FOR ERRORS CONTAINED HEREIN OR FOR DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
CONSEQUENTIAL, RELIANCE OR COVER DAMAGES, INCLUDING LOSS OF PROFITS, REVENUE, DATA OR
USE, INCURRED BY ANY USER OR ANY THIRD PARTY IN CONNECTION WITH THE FURNISHING,
PERFORMANCE, OR USE OF THIS MATERIAL, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH
DAMAGES.

The entire risk as to the quality and performance of software developed using this specification is borne by you. This
disclaimer of warranty constitutes an essential part of the license granted to you to use this specification.

RESTRICTED RIGHTS LEGEND

Use, duplication or disclosure by the U.S. Government is subject to the restrictions set forth in subparagraph (c) (1) (ii) of
The Rights in Technical Data and Computer Software Clause at DFARS 252.227-7013 or in subparagraph (c)(1) and (2)
of the Commercial Computer Software - Restricted Rights clauses at 48 C.F.R. 52.227-19 or as specified in 48 C.F.R. 227-
Regulations and its successors, as applicable. The specification copyright owners are as indicated above and may be
contacted through the Object Management Group, 250 First Avenue, Needham, MA 02494, U.S.A.

TRADEMARKS

The OMG Object Management Group Logo®, CORBA®, CORBA Academy®, The Information Brokerage®, XMI®
and IIOP® are registered trademarks of the Object Management Group. OMG™, Object Management GroupTM, CORBA
logos™, OMG Interface Definition Language (IDL)™, The Architecture of Choice for a Changing World™,
CORBA services™, CORBA facilities™, CORBAmed™, CORBA-net™, Integrate 2002™, Middleware That's
Everywhere™, UML™, Unified Modeling Language™, The UML Cube logo™, MOF™, CWM™, The CWM Logo™,
Model Driven Architecture™, Model Driven Architecture Logos™, MDA™, OMG Model Driven Architecture™, OMG
MDA™ and the XMI Logo™ are trademarks of the Object Management Group. All other products or company names
mentioned are used for identification purposes only, and may be trademarks of their respective owners.

COMPLIANCE

The copyright holders listed above acknowledge that the Object Management Group (acting itself or through its
designees) is and shall at all times be the sole entity that may authorize developers, suppliers and sellers of computer
software to use certification marks, trademarks or other special designations to indicate compliance with these materials.

Software developed under the terms of this license may claim compliance or conformance with this specification if and
only if the software compliance is of a nature fully matching the applicable compliance points as stated in the
specification. Software developed only partially matching the applicable compliance points may claim only that the
software was based on this specification, but may not claim compliance or conformance with this specification. In the
event that testing suites are implemented or approved by Object Management Group, Inc., software developed using this
specification may claim compliance or conformance with the specification only if the software satisfactorily completes
the testing suites.
OMG’s Issue Reporting Procedure

All OMG specifications are subject to continuous review and improvement. As part of this process we encourage readers to report any ambiguities, inconsistencies, or inaccuracies they may find by completing the Issue Reporting Form listed on the main web page http://www.omg.org, under Documents, Report a Bug/Issue (http://www.omg.org/technology/agreement.htm).
Contents

1. Overview ................................................. 1-1
   1.1 Introduction ........................................ 1-1
   1.2 Problems Being Addressed ............................ 1-3
      1.2.1 Data Access ...................................... 1-4
      1.2.2 Concurrency Control .............................. 1-5
      1.2.3 Data Semantics ................................. 1-5
   1.3 Problems Not Being Addressed ....................... 1-5
   1.4 Design Rationale ..................................... 1-5
      1.4.1 Adherence to OPC ............................... 1-5
      1.4.2 Simplicity and Uniformity ...................... 1-5
      1.4.3 High Performance Implementations ............. 1-6
   1.5 Conformance to the DAIS ............................. 1-7
      1.5.1 Conformance to the Server ...................... 1-7
      1.5.2 Conformance to Data Access .................... 1-7
      1.5.3 Conformance to Alarms and Events ............. 1-8

2. Relations to Other Standards ......................... 2-1
   2.1 OLE for Process Control (OPC) ..................... 2-1
      2.1.1 Objects ......................................... 2-1
   2.2 Data Access Facility (DAF) ........................ 2-4
      2.2.1 Resources and Properties ...................... 2-4
      2.2.2 Information model/schema ...................... 2-5
      2.2.3 Data Types ..................................... 2-6
   2.3 COM and CORBA IDL Differences .................... 2-6
      2.3.1 Object Referencing ............................. 2-6
2.3.2 Interface Management .......................... 2-7
2.3.3 Error Management .......................... 2-7
2.3.4 IDL ........................................ 2-7
2.4 IEC 1346-1, Structuring and Naming .................. 2-7
2.5 IEC 61970 EMS API ................................ 2-7
2.6 XPath ........................................... 2-8

3. **DAIS Server ........................................ 3-1**
   3.1 Common Declarations .......................... 3-1
      3.1.1 Character Encoding ...................... 3-1
      3.1.2 Common IDL Overview .................. 3-1
      3.1.3 DAFIdentifiers IDL .................... 3-2
      3.1.4 DAFDescriptions IDL .................. 3-2
      3.1.5 DAISCommon IDL ....................... 3-2
      3.1.6 Iterator Methods IDL .................. 3-10
      3.1.7 DAISNode IDL ............................ 3-11
      3.1.8 DAISType IDL ............................ 3-16
      3.1.9 DAISProperty IDL ..................... 3-19
      3.1.10 DAISSession IDL ...................... 3-22
      3.1.11 Filter Definitions .................... 3-25
      3.1.12 Logical Expressions and Navigation ...... 3-25
      3.1.13 Authorization .......................... 3-28
      3.1.14 Requirement Levels .................... 3-29
   3.2 Server .......................................... 3-30
      3.2.1 DAISServer IDL Overview .............. 3-30
      3.2.2 DAIS Server IDL ........................ 3-30

4. **DAIS Data Access ................................ 4-1**
   4.1 Information Model ............................... 4-1
      4.1.1 Nodes, Items, Types, and Properties ...... 4-2
      4.1.2 Naming .................................. 4-3
      4.1.3 Item Values ................................ 4-4
      4.1.4 OPC Recommended Properties ............... 4-5
      4.1.5 Utility SCADA/EMS Measurement Model ...... 4-5
   4.2 API .............................................. 4-8
      4.2.1 Data Access IDL Overview ............... 4-8
      4.2.2 DAISDASession IDL ...................... 4-8
      4.2.3 DAISDANode IDL ........................... 4-12
      4.2.4 DAISItem IDL ............................. 4-15
      4.2.5 DAISDAIO IDL ............................. 4-23
      4.2.6 DAISGroupEntry IDL ...................... 4-32
Preface

About the Object Management Group

The Object Management Group, Inc. (OMG) is an international organization supported by over 600 members, including information system vendors, software developers and users. Founded in 1989, the OMG promotes the theory and practice of object-oriented technology in software development. The organization’s charter includes the establishment of industry guidelines and object management specifications to provide a common framework for application development. Primary goals are the reusability, portability, and interoperability of object-based software in distributed, heterogeneous environments. Conformance to these specifications will make it possible to develop a heterogeneous applications environment across all major hardware platforms and operating systems.

OMG’s objectives are to foster the growth of object technology and influence its direction by establishing the Object Management Architecture (OMA). The OMA provides the conceptual infrastructure upon which all OMG specifications are based. More information is available at http://www.omg.org/.

OMG Documents

The OMG Specifications Catalog is available from the OMG website at:


The OMG documentation is organized as follows:

OMG Modeling Specifications
Includes the UML, MOF, XMI, and CWM specifications.
**OMG Middleware Specifications**
Includes CORBA/IIOP, IDL/Language Mappings, Specialized CORBA specifications, and CORBA Component Model (CCM).

**Platform Specific Model and Interface Specifications**
Includes CORBA services, CORBA facilities, OMG Domain specifications, OMG Embedded Intelligence specifications, and OMG Security specifications.

**Obtaining OMG Documents**
The OMG collects information for each book in the documentation set by issuing Requests for Information, Requests for Proposals, and Requests for Comment and, with its membership, evaluating the responses. Specifications are adopted as standards only when representatives of the OMG membership accept them as such by vote. (The policies and procedures of the OMG are described in detail in the *Object Management Architecture Guide*.)

OMG formal documents are available from our web site in PostScript and PDF format. Contact the Object Management Group, Inc. at:

OMG Headquarters  
250 First Avenue  
Needham, MA 02494  
USA  
Tel: +1-781-444-0404  
Fax: +1-781-444-0320  
pubs@omg.org  
http://www.omg.org

**Typographical Conventions**
The type styles shown below are used in this document to distinguish programming statements from ordinary English. However, these conventions are not used in tables or section headings where no distinction is necessary.

**Helvetica bold** - OMG Interface Definition Language (OMG IDL) and syntax elements.

**Courier bold** - Programming language elements.

**Helvetica** - Exceptions

Terms that appear in *italics* are defined in the glossary. Italic text also represents the name of a document, specification, or other publication.
Acknowledgments

The following companies submitted parts of this specification:
- ABB Automation Systems
- Alstom ESCA
- Langdale Consultants

Supporting Organizations

The following organizations have been involved in the process of contributing to, and reviewing this specification. These companies have indicated their support for the specification. We thank them for participating and giving their valuable input.
- Hitachi Ltd.
- IBM Corporation
- NIIP Project Office
- SISCO Inc.
Overview

Contents

This chapter contains the following sections.

<table>
<thead>
<tr>
<th>Section Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Introduction”</td>
<td>1-1</td>
</tr>
<tr>
<td>“Problems Being Addressed”</td>
<td>1-3</td>
</tr>
<tr>
<td>“Problems Not Being Addressed”</td>
<td>1-5</td>
</tr>
<tr>
<td>“Design Rationale”</td>
<td>1-5</td>
</tr>
<tr>
<td>“Conformance to the DAIS”</td>
<td>1-7</td>
</tr>
</tbody>
</table>

1.1 Introduction

The purpose of the DAIS Application Program Interface (API) is to support efficient real time transfer of large amounts of data from an industrial process to a wide range of clients. It supports discovery of parameters and update of parameter values. The DAIS is intended for on-line data transfer and cannot be used to configure servers implementing the API.

Control systems used to monitor and control industrial processes consist of the following major pieces:

- Process instrumentation making sensor data and actuation capabilities available.
- Process stations or remote terminal units (RTUs) reading sensor data and controlling actuators.
• Supervisory Control and Data Acquisition (SCADA) system making processed sensor data and control capabilities available to operators, applications, or other systems.

• Management systems using SCADA data to make further processing and control.

SCADA and management systems can be regarded having a server part where data processing is performed and a Human Machine Interface (HMI) part where visualization and command dialogs are made. The SCADA and management system may have common or different HMI.

This results in a hierarchical structure shown below.

![Control system structure](image)

*Figure 1-1  Control system structure*

The solid arrows in Figure 1-1 correspond to transfer of data from servers to clients. The data can be state variables and parameters (for example, measured values, calculated values, limit values, dead bands, scan rates, engineering units).

The dashed arrows correspond to data written to servers by clients. The data can be control variables and parameters. The API support simple writes operations. More complex controls like select and execute (for example, breaker on/off) or raise lower commands can be implemented combining multiple write and read operations.

As indicated in Figure 1-1 the DAIS API can be used at several levels in a system. For example, a DAIS server can be an RTU/Process station communication unit, a SCADA server, or even a Management system.

DAIS support both subscription and read/write operations. A subscription means server has no *a priori* knowledge of clients and it is clients that establish connection with servers. Once connection is established a server calls the clients back when data becomes available or updated. The callbacks mean the DAIS API also defines an interface that the client has to implement.
Figure 1-2 shows the bi-directional subscription with connection establishment and callback interfaces.

For historical reasons SCADA systems for different industrial processes has evolved along different lines. SCADA for power systems has evolved onto a UNIX base and SCADA for most other industrial processes has evolved onto a Windows NT base. For UNIX based systems APIs formulated in CORBA (Common Object Request Broker Architecture) IDL (Interface Definition Language) are now emerging (for example, DAIS RFP [1] and UMS DAF [2]). For Windows NT based systems, such as OLE for Process Control (OPC) [3], has become the dominating standard. OPC defines three different APIs; measurement data access [4], alarms & events [5], and history [6]. OPC is based on Microsoft COM.

OPC is focused on the interfaces and does not explicitly describe the information model behind the interface. The information model is however implicit and can be derived from the OPC specifications. This specification describes both the API and information model expressed in UML.

Within the CCAPI project an information model Common Information Model (CIM) [7] has been developed. This model is now evolving into the IEC 61970-30x draft standard [8]. The CIM contains a SCADA information model (61970-303) with its roots in power transmission and generation. The DAIS API supports the CIM.

1.2 Problems Being Addressed

The DAIS API is intended for transfer of process data on subscription basis as indicated in Figure 1-1 and Figure 1-2. Process data consists of quality tagged and time stamped scalar values. The API is intended to efficiently transfer large amounts of data simultaneously to many clients (subscribers). Clients and servers involved in data exchange can be of many kinds (for example, HMI or management systems as indicated in Figure 1-1). A client may also appear as a server (for example, aggregating data from other servers or performing calculations as indicated in Figure 1-3). This creates hierarchical structures of DAIS servers.
Figure 1-3  Using DAIS as interface between multiple servers

As an example the servers in the leftmost layer in Figure 1-3 might be OPC compliant RTUs (or IEDs), next right might be communication front ends, and the rightmost server may provide both telemetered and estimated data.

The DAIS API is intended to be used for a wide range of industrial processes. For example:

- power transmission
- power generation
- power distribution
- water and sewage management
- oil and gas
- district heating
- pulp and paper
- food manufacturing

The kinds of data that can be reached through the DAIS API are:

- measurement data access
- alarms & events access

This data is typically available from hardware units in the process (for example, RTUs, PLCs, distributed controllers) or other control centers (for example, SCADA systems). Refined or calculated data, parameters, and alarms & events might also come from applications (for example, custom calculations, state estimation, optimization) in SCADA or Management systems. These data might be provided through the DAIS API as well.

1.2.1 Data Access

This specification provides interfaces for data access including:

- Discovery of data available in a server.
• Discovery of the information model supported by a server (for example, available types and their properties).
• Synchronous and asynchronous read or write of server data.
• Creation and maintenance of subscriptions at the server.
• Client side subscription callback interfaces for event driven data transfer.

1.2.2 Concurrency Control

There are no explicit means to synchronize clients. Time stamping of data is provided so that clients can judge the age.

1.2.3 Data Semantics

Data is hierarchically organized in trees of nodes and items where the items are the leafs. The nodes in the hierarchy have a type (for example, substation, pump, breaker, or any collection of items). The type is transparent to the interface. An item is an instance of a property belonging to a type. Properties can describe any kind of state variables, control variables, or parameters existing in a control system.

Data transferred by reads or callbacks are time stamped and quality coded.

1.3 Problems Not Being Addressed

The problem of configuring a server or control system with nodes, items, areas, sources, reasons, condition spaces, and source conditions is outside the scope of this specification.

The problem of supervising the control system equipment (for example, communication lines, computers, hard disks) and provide specific interfaces for this is outside the scope of this specification. It is however possible for a DAIS server to provide control system statuses as measurements.

1.4 Design Rationale

Besides meeting the requirements spelled out in the RFP there are a number of design goals that have shaped solutions.

1.4.1 Adherence to OPC

OPC has been in use for a number of years and this specification leverages on the experience gained by OPC. There are a large number of OPC based products in the market place and cases where DAIS and OPC will be bridged are likely. Adherence to OPC is important to facilitate simple bridging and porting DAIS software to/from OPC.

1.4.2 Simplicity and Uniformity

Some design principles used when creating OPC were:
• Method behavior is sometimes controlled by an input parameter.
• Related data is transferred in multiple parallel vectors.
• Outputs are always returned in one or more output parameters.

To simplify and get a more uniform interface these principles have been replaced by the following:

• A method has one single behavior resulting in some OPC methods being replaced by more than one DAIS method.
• Related data is kept together in structs resulting in reduction of the number of parameters compared to OPC.
• Outputs are returned as method return results resulting in the OPC HRESULT parameter being replaced by exceptions and reduced number of output parameters compared to OPC.

1.4.3 High Performance Implementations

A DAIS server is a real-time system required to deliver data in high rates and volumes. The performance requirements mean that a typical DAIS server does not use a relational database management system for on-line operation but some kind of real-time database. The DAIS API efficiently encapsulates such real-time databases from clients.

To be effectively delivering data DAIS must not introduce performance bottlenecks of its own. This has influenced the design in several ways, listed below.

1.4.3.1 Subscription

The subscription mechanism consists of two phases. In the first the client negotiates with the server on what data items to subscribe for and in the second the actual data transfer takes place. This minimizes the amount of transferred data between the server and the client during on-line operation.

1.4.3.2 Sequences

DAIS support to use sequences of data in calls rather than having calls requiring single valued parameters. This allows clients to ask for processing of multiple data in a single call rather than making multiple calls thus reducing the number of LAN round trips.

1.4.3.3 Iterators

Large volumes of data are not efficiently transferred in one method call. For this reason many methods return an iterator that is used to transfer optimal volumes of data in each call.
1.4.3.4 Data Value Representation

The basic unit of data is a union type: SimpleValue. SimpleValue exploits our knowledge of the basic data types needed and eliminates CORBA any from the highest bandwidth part of the interface. This can make a significant impact on performance when accumulated across large amounts of data.

1.5 Conformance to the DAIS

The DAIS has three major conformance points:

1. The DAIS Server
2. The DAIS Data Access
3. The DAIS Alarms and Events.

An implementation

• shall conform to point 1.
• shall conform to either point 2 or point 3.
• may conform to both point 2 and point 3.

1.5.1 Conformance to the Server

The DAIS interface as defined in Section 3.2, “Server,” on page 3-30 is a mandatory conformance point for a DAIS Server with the exception of the following optional methods:

• DAIS::Server::find_views()
• DAIS::Server::create_data_access_session_for_view()
• DAIS::Server::create_alarms_and_events_session_for_view()
• DAIS::Server::inspect()

A server shall also conform to

• Section 3.1.1, “Character Encoding,” on page 3-1.

The use of XPath as described in Section 3.1.12, “Logical Expressions and Navigation,” on page 3-25, Section 5.2.4.2, “IDL,” on page 5-14, and Section 5.2.7.3, “Condition Logic,” on page 5-38 is an optional conformance point.

1.5.2 Conformance to Data Access

The DAIS interface as defined in Section 3.1, “Common Declarations,” on page 3-1 and Section 4.2, “API,” on page 4-8 is a mandatory conformance point for Data Access.
1.5.3 Conformance to Alarms and Events

The DAIS interface as defined in Section 3.1, “Common Declarations,” on page 3-1 and Section 5.2, “API,” on page 5-6 is a mandatory conformance point for Alarms and Events.
Relations to Other Standards

Contents

This chapter contains the following sections.

<table>
<thead>
<tr>
<th>Section Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>“OLE for Process Control (OPC)”</td>
<td>2-1</td>
</tr>
<tr>
<td>“Data Access Facility (DAF)”</td>
<td>2-4</td>
</tr>
<tr>
<td>“COM and CORBA IDL Differences”</td>
<td>2-6</td>
</tr>
<tr>
<td>“IEC 1346-1, Structuring and Naming”</td>
<td>2-7</td>
</tr>
<tr>
<td>“IEC 61970 EMS API”</td>
<td>2-7</td>
</tr>
<tr>
<td>“XPath”</td>
<td>2-8</td>
</tr>
</tbody>
</table>

2.1 OLE for Process Control (OPC)

Differences and similarities between OPC and DAIS are described in this chapter.

2.1.1 Objects

OPC is a service API providing access to data managed by the server. The data (for example, nodes, items, areas, sources, conditions, reasons) are not instantiated as objects at the client. This means that OPC does not define any particular APIs for the data instances that a client can deal with directly. OPC has a few coarse objects like OPCGroup and OPCEventSubscription supporting the data access. DAIS has adopted this principle and is identical to OPC in this respect.
In OPC each client has its own OPC server object. In DAIS there is only one DAIS server object shared by all clients. To support individual client sessions a new session interface is defined. There is one interface for data access session objects and one for alarms & events session objects. The session objects correspond to the OPC server object.

The OPC server object has interfaces for browsing server data (for example, IOPCBrowseServerAddressSpace and IOPCEventAreaBrowser) and information model (for example, IOPCItemProperties). In DAIS each type of data has its own object for browsing; these objects are called “home” objects. The OPC browse interface methods are hence divided among the different home objects.

2.1.1.1 Interface, method, and parameter naming

Many OPC interface, method, and parameter names have been kept but recasted according to the CORBA style guide. But many OPC names have been replaced by new names to more clearly indicate the meaning. This is particularly the case for the browse and alarms & events APIs.

2.1.1.2 Error and status codes

In OPC it is common to return arrays of HRESULTs corresponding to arrays of data. In the case where the data did not contain any errors an array with “no error occurred” codes still is returned. In DAIS such error codes will be returned as a sequence of error structs identifying the erroneous data and an error code. In case of no errors the sequence of error structs is empty.

2.1.1.3 Identifiers, handles, and blobs

OPC data accesses use both server and client side handles created based on identification texts for nodes and items. To make the translation from identification texts to handles fast and avoid repeated translation in OPC an intermediate server side identifier called the blob exists. In DAIS the server side handles and blobs are replaced by identifiers based on ResourceIDs (both for nodes and items). For a description of ResourceIDs refer to Section 2.2, “Data Access Facility (DAF),” on page 2-4. The OPC client side handles are still kept in DAIS.

2.1.1.4 Callbacks

COM/OPC use the standard interfaces IConnectionPoint to set up callbacks. In DAIS callbacks are set up directly between server and client by updating an attribute holding the callback object.

2.1.1.5 Enumerators

The COM style enumerators in OPC are replaced by CORBA style iterators in DAIS.
2.1.1.6 Parameters and structs

In OPC multiple parallel vectors often pass data. In DAIS a single vector holding structs, thus reducing the number of parameters, replaces the parallel vectors.

2.1.1.7 Method return data

In OPC all methods return the HRESULT error value. In DAIS HRESULTs are replaced by exceptions and out parameters are returned as method results instead. Only in a few cases are there additional output parameters in DAIS.

2.1.1.8 Items, structuring, and naming

In OPC data is generally organized in hierarchical structures. In data access the leafs are called leaf nodes and the branches branch nodes or items. A leaf node or item is the same as an instance of property at an object. A branch node corresponds to an:

- Object having properties.
- Arbitrary organization of other branch nodes.
- Object having both properties and other branch nodes as children.

In DAIS data access the branch nodes are called nodes and the leaf nodes are called items. In DAIS alarms & events, the branch nodes are called areas and the leaf nodes sources.

In OPC branch and leaf nodes have a name unique among the nodes or items that are children of the same node. A second name is formed by combining these names from each branch node in the path from the node or leaf to the root. Both names are called ItemIDs in OPC and the name created by following the path to the root is sometimes called a fully qualified ItemID.

In DAIS the name unique among the child’s to the same node is called label and the name including the labels from all nodes in the path to the root is called pathname. Both DAIS branch nodes and leaf nodes have both a label and a pathname.

A label is used in the same way as a name but the word label is preferred before the word name as it denotes something that is atomic; that is, it cannot be further divided.

2.1.1.9 Server side cursors

OPC provides a server side cursor from where browsing is made and the OPC interface provides methods to move the cursor.

DAIS does not provide server side cursors and requires the client to keep track of browse positions themselves. The reason for removing server side cursors is it makes clean up after crashed clients easier and simplifies the server design.
2.1.10 Properties and types

OPC does not support types meaning it is not possible in OPC to get information about the type of a node. OPC however supports properties, which means it is possible to browse the existing properties and what properties a particular node has. DAIS has extended this to also include types; that is, it is possible to browse existing types, the properties each type has and each node has a type.

OPC defines the following property sets:

• set 1 of OPC specific properties
• set 2 of OPC recommended properties
• set 3 of vendor specific properties

In DAIS, OPC set 1 properties cannot be browsed through the property-browsing interface (Property). Instead access of set 1 properties is direct in the interfaces as parameters or struct members as is the case for OPC. This means that DAIS exposes only OPC set 2 and 3 properties through the property browse interface.

2.2 Data Access Facility (DAF)

The UMS DAF and DAIS are server APIs for access of object data rather than the data objects themselves.

2.2.1 Resources and Properties

The DAF describes a generic interface for navigating and reading data from complex data structures including relations between objects. Both DAF and DAIS support navigation in a space of hierarchically structured objects (an object is called a node in DAIS and a resource in DAF). Both support identification of objects and properties at an object. In DAIS an instance of property at an object is called an item and an ItemID (item identifier) identifies an item. An ItemID consists of a ResourceID (resource identifier) for the node and a PropertyID (property identifier) for the property. The DAIS API uses ItemIDs to access data while the DAF uses ResourceIDs and PropertyIDs separated.

A system may implement both a DAF server and a DAIS server. In such a system it shall be assumed that the same object will have the same ResourceID seen through either API. This means it shall be possible to navigate to an object, retrieve its ResourceID through one of the APIs, and use that ResourceID with the other API. In the same way PropertyIDs are the same.

Textual identification of resources and properties in the DAF is by URIs (Uniform Resource Identifier). ResourceIDs and PropertyIDs have their own URIs. URIs can be translated into corresponding ResourceIDs and PropertyIDs and vice versa. A property name is a URI where the container part is a unique schema identifier and the fragment part is the property name.
For example:

http://www.epri.com/schema/CIM-07f.xml#Measurement.positiveFlowIn

DAIS does not support URIs directly but has a textual representation for nodes, items, types, and properties. This textual representation may correspond to the fragment part of a URI.

### 2.2.2 Information model/schema

In the case where a system supports both the DAF and the DAIS interfaces it is expected that the same object will be identified by the same ResourceID through both interfaces and the same property by the same PropertyID. Sameness of objects and properties; however, depends on the information model exposed through the two interfaces and if the information models are different, then a mapping is required.

In the case where a system supports both the DAF and the DAIS interfaces it is expected that the same objects will be identified by the same ResourceID and URI fragment through both interfaces and the same property by the same PropertyID and URI fragment.

The exact mapping has to be described for each DAIS item – DAF property pair according to the underlying information models. Section 4.1.5, “Utility SCADA/EMS Measurement Model,” on page 4-5 elaborates on the information model and a detailed mapping between IEC 61970-30x and the DAIS can be found in Section 4.1.5, “Utility SCADA/EMS Measurement Model,” on page 4-5.

The foregoing concepts are fundamental enough that they should find equivalents in any data repository. Some, perhaps approximate, equivalents are given in Table 2-1 as a guide.

**Table 2-1** Mappings between modeling languages

<table>
<thead>
<tr>
<th>RDF</th>
<th>DAF</th>
<th>Relational Model</th>
<th>UML</th>
<th>DAIS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Resource</td>
<td>Resource, ResourceID</td>
<td>Tuple (i.e., row)</td>
<td>Object</td>
<td>Node, ResourceID</td>
</tr>
<tr>
<td>Property</td>
<td>Property, PropertyID</td>
<td>Attribute (i.e., column) or foreign key</td>
<td>Attribute or association</td>
<td>Property, PropertyID</td>
</tr>
<tr>
<td>Class</td>
<td>Class, ClassID</td>
<td>Relation (i.e., table)</td>
<td>Class</td>
<td>Type, TypeID</td>
</tr>
<tr>
<td>Resource Description</td>
<td>Resource Description</td>
<td>Tuple value</td>
<td>-</td>
<td>Sequence &lt;ItemState&gt;</td>
</tr>
<tr>
<td>URI</td>
<td>URI, ResourceID</td>
<td>Key value</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Value</td>
<td>SimpleValue</td>
<td>Field value</td>
<td>-</td>
<td>SimpleValue</td>
</tr>
<tr>
<td>-</td>
<td>ResourceID and PropertyID pair</td>
<td>-</td>
<td>-</td>
<td>Item, ItemID</td>
</tr>
</tbody>
</table>
2.2.3 Data Types

DAIS uses the DAF SimpleValue type for data transferred over the API instead of the OPC type VARIANT. The data types used in OPC are Microsoft COM types and as DAIS is a CORBA API a mapping of the data types is needed. The translation of OPC/COM data types to DAF/CORBA data types is listed below.

Table 2-2  Mappings between OPC and DAF data types

<table>
<thead>
<tr>
<th>OPC/COM basic types</th>
<th>DAF Simple Value types</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>ResourceID (RESOURCE_TYPE)</td>
</tr>
<tr>
<td>-</td>
<td>URI (URI_TYPE)</td>
</tr>
<tr>
<td>LPWSTR (VT_BSTR)</td>
<td>string (STRING_TYPE)</td>
</tr>
<tr>
<td>BOOL (VT_BOOL)</td>
<td>boolean (BOOLEAN_TYPE)</td>
</tr>
<tr>
<td>LONG (VT_R4)</td>
<td>long (INT_TYPE)</td>
</tr>
<tr>
<td>DWORD (VT_I4)</td>
<td>unsigned long (UNSIGNED_TYPE)</td>
</tr>
<tr>
<td>double (VT_R8)</td>
<td>double (DOUBLE_TYPE)</td>
</tr>
<tr>
<td>-</td>
<td>Complex (COMPLEX_TYPE)</td>
</tr>
<tr>
<td>FILETIME (VT_Date)</td>
<td>DateTime (DATE_TIME_TYPE)</td>
</tr>
<tr>
<td>-</td>
<td>ULongLong (ULONG_LONG_TYPE)</td>
</tr>
<tr>
<td>WORD (VT_I2)</td>
<td>unsigned long (UNSIGNED_TYPE)</td>
</tr>
<tr>
<td>FLOAT (VT_R4)</td>
<td>double (DOUBLE_TYPE)</td>
</tr>
<tr>
<td>BYTE (VT_UI1)</td>
<td>-</td>
</tr>
<tr>
<td>HRESULT (VT_ERROR)</td>
<td>-</td>
</tr>
<tr>
<td>VARIANT (VARTYPE)</td>
<td>SimpleValue (SimpleValueType)</td>
</tr>
</tbody>
</table>

2.3 COM and CORBA IDL Differences

Interfaces defined in COM and CORBA IDL differ in a number of ways. Important differences concerning interface definitions are

- object referencing,
- interface management,
- error management, and
- IDL.

A detailed description of the mapping can be found in the CORBA 2.3 specification, chapters 17, 18, and 19.

2.3.1 Object Referencing

In CORBA each object is uniquely referenced in one step. In COM obtaining an object usually is a two step process. In the first step a stateless server object is obtained. In the second step the server object is loaded with state data. In CORBA an object is a unique individual that contains state from the very beginning.
2.3.2 Interface Management

A CORBA object has a single interface. This interface can be built from several other interfaces through inheritance. The resulting interface might have many methods and hence become big. A COM object usually has multiple interfaces and supports the client to detect and navigate between these interfaces at run time. As a CORBA interface is defined by inheritance it has to be fully defined at compile time. As COM allows run time detection of interfaces (the IUnknown::QueryInterface() method) a full match between interfaces implemented by a server and interfaces known to a client is not required.

Mapping OPC interfaces to DAIS interfaces can be done in two ways:

1. Inherit a number of OPC interfaces into one CORBA object. This may result in name clashes that require renaming of methods.
2. Instantiate an OPC interface as an own CORBA object referenced by a container object.

Both techniques are used in the DAIS specification.

2.3.3 Error Management

CORBA provides exceptions for error handling and COM does not. COM provides error status through a return parameter called HRESULT. The caller has to explicitly test the HRESULT to decide if the operation was successful. HRESULT return parameters are replaced by CORBA exceptions.

2.3.4 IDL

The COM and CORBA IDL have several syntactical differences and use different style guides.

2.4 IEC 1346-1, Structuring and Naming

DAIS (as well as OPC) structure nodes and items hierarchically. Nodes and items have a label uniquely identifying child’s located at the same node. The labels from the nodes in the path from a node or item to the root form a pathname. The pathname uniquely identifies a node or item in the tree.

The same principle for naming objects is described in the IEC 1346-1 standard. IEC 1346-1 call labels for single level designation and the pathname for multi level designation and supports multiple hierarchical structures. The DAIS supports multiple structures by allowing multiple views where each view exposes one structure.

2.5 IEC 61970 EMS API

The IEC 61970-30x [8] draft standard (also named CIM) describes a specific organization of power system objects in a hierarchical structure. DAIS is transparent to the structure and hence supports the IEC 61970 structure.
DAIS is also transparent to what attributes are defined as long as they can be reached through the hierarchy. DAIS however defines a set of attributes that a DAIS server is expected to support. Some of these attributes can also be found in IEC 61970-30x. A detailed attribute mapping is provided later in this specification and is further elaborated in Section 4.1.5, “Utility SCADA/EMS Measurement Model,” on page 4-5.

### 2.6 XPath

The W3C XPath [13] standard describes how navigation in a hierarchical structure (i.e., an XML document) is made using an expression.

In DAIS, navigation is made using the various browse APIs. However in one place there is a need to describe navigation paths as expressions and that is for the condition logic in Alarms and Events.
DAISServer

Contents

This chapter contains the following sections.

<table>
<thead>
<tr>
<th>Section Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Common Declarations”</td>
<td>3-1</td>
</tr>
<tr>
<td>“Server”</td>
<td>3-30</td>
</tr>
</tbody>
</table>

This section describes the DAISServer and IDL common to the server, data access, and alarms & events.

3.1 Common Declarations

3.1.1 Character Encoding

To support universal character encodings the UTF-8 Unicode standard [12] shall be used for characters and strings.

3.1.2 Common IDL Overview

The DAIS relies on the Data Access Facility (DAF) for basic data type declarations and some DAF declarations (for example, ResourceID and SimpleValue) are included because of this. DAIS common declarations (for example, DAIS::Quality) are made in DAISCommon. Interfaces for nodes, types, properties, and sessions are also common between data access and alarms & events. The Interface Definition Language (IDL) files and dependencies listed in Figure 3-1 are defined.
3.1.3 DAFIdentifiers IDL

Refer to the Data Access Facility specification [2].

3.1.4 DAFDescriptions IDL

Refer to the Data Access Facility specification [2].

3.1.5 DAISCommon IDL

```cpp
// File: DAISCommon.idl
#ifndef _DAIS_COMMON_IDL
#define _DAIS_COMMON_IDL
#pragma prefix "omg.org"
#include <DAFDescriptions.idl>
module DAIS
{
    typedef unsigned short SupportedFunctions;
    const SupportedFunctions DAIS_DATA_ACCESS =0x0001;
    const SupportedFunctions DAIS_ALARMS_AND_EVENTS =0x0002;

typedef DAFDescriptions::ResourceId ResourceId;
typedef DAFDescriptions::SimpleValueType SimpleValueType;
typedef DAFDescriptions::SimpleValue SimpleValue;
typedef DAFDescriptions::DateTime DateTime;
typedef DAFDescriptions::PropertyId PropertyId;
typedef DAFDescriptions::PropertyValueSequence PropertyValueSequence;
```
// sequences of resource ids
typedef sequence<ResourceID> ResourceIDs;
typedef sequence<PropertyID> PropertyIDs;

typedef sequence<ItemID> ItemIDs;

union ServerItemIdentification switch( ServerItemIdentificationType )
{
    case ITEM_ID: ItemID item;
    case PATH_NAME: string pathname;
};
typedef sequence<ServerItemIdentification> ServerItemIdentifications;

typedef unsigned short Error;
struct ItemError
{
    Error err;
    ClientItemHandle client_handle;
    ServerItemHandle server_handle;
    string pathname;
    string reason;
};
typedef sequence<ItemError> ItemErrors;

// error codes
const Error ERROR_DAISOK = 0;
const Error ERROR_BAD_RIGHTS = 1;
const Error ERROR_UNKNOWN_ITEMID = 2;
const Error ERROR_CLAMPED = 3;
const Error ERROR_OUT_OF_RANGE = 4;
const Error ERROR_UNKNOWN_PATHNAME = 5;
const Error ERROR_BAD_TYPE = 6;
const Error ERROR_UNKNOWN_ACCESS_PATH = 7;
const Error ERROR_INTERNAL_SERVER = 8;
const Error ERROR_INVALID_DAIS_HANDLE = 9;

enum AccessRights
{
    READABLE,
WRITEABLE, READ_AND_WRITEABLE

typedef unsigned long OPCQuality;
typedef unsigned long UserQuality;

struct Quality {
    OPCQuality opc_quality;
    UserQuality user_quality;
};

// Masks for extracting quality subfields
// (note 'status' mask also includes 'Quality' bits)

const OPCQuality OPC_QUALITY_MASK  = 0x000000C0;
const OPCQuality OPC_STATUS_MASK  = 0x000000FC;
const OPCQuality OPC_LIMIT_MASK  = 0x00000003;

// Values for QUALITY_MASK bit field
const OPCQuality OPC_QUALITY_BAD  = 0x00000000;
const OPCQuality OPC_QUALITY_UNCERTAIN  = 0x00000040;
const OPCQuality OPC_QUALITY_GOOD  = 0x000000C0;

// STATUS_MASK Values for Quality = BAD
const OPCQuality OPC_QUALITY_CONFIG_ERROR = 0x00000004;
const OPCQuality OPC_QUALITY_NOT_CONNECTED = 0x00000008;
const OPCQuality OPC_QUALITY_DEVICE_FAILURE = 0x0000000C;
const OPCQuality OPC_QUALITY_SENSOR_FAILURE = 0x00000010;
const OPCQuality OPC_QUALITY_LAST_KNOWN = 0x00000014;
const OPCQuality OPC_QUALITY_COMM_FAILURE = 0x00000018;
const OPCQuality OPC_QUALITY_OUT_OF_SERVICE = 0x0000001C;

// STATUS_MASK Values for Quality = UNCERTAIN
const OPCQuality OPC_QUALITY_LAST_USABLE = 0x00000044;
const OPCQuality OPC_QUALITY_SENSOR_CAL = 0x00000050;
const OPCQuality OPC_QUALITY_EGU_EXCEEDED = 0x00000054;
const OPCQuality OPC_QUALITY_SUB_NORMAL = 0x00000058;
const OPCQuality DAIS_QUALITY_OCILLATORY = 0x0000005C;

// STATUS_MASK Values for Quality = GOOD
const OPCQuality OPC_QUALITY_LOCAL_OVERRIDE = 0xD8;

// Values for Limit Bitfield
const OPCQuality OPC_LIMIT_OK = 0x00000000;
const OPCQuality OPC_LIMIT_LOW = 0x00000001;
const OPCQuality OPC_LIMIT_HIGH = 0x00000002;
const OPCQuality OPC_LIMIT_CONST = 0x00000003;
//DAIS Quality extension masks
const OPCQuality EXQ_SOURCE_MASK = 0x00000700;
const OPCQuality EXQ_TEST_MASK = 0x00000800;
const OPCQuality EXQ_OPERATOR_BLOCKED_MASK = 0x00001000;
const OPCQuality EXQ_TIMESTAMP_ACCURACY_MASK = 0x00006000;

//DAIS Quality source extension
const OPCQuality EXQ_SOURCE_NONE = 0x00000000;
const OPCQuality EXQ_SOURCE_PROCESS = 0x00000100;
const OPCQuality EXQ_SOURCE_PRIMARY_SUBSTITUTED = 0x00000200;
const OPCQuality EXQ_SOURCE_INHERITED_SUBSTITUTED = 0x00000300;
const OPCQuality EXQ_SOURCE_CORRECTED = 0x00000400;
const OPCQuality EXQ_SOURCE_DEFAULTED = 0x00000500;

//DAIS Time stamp accuracy
const OPCQuality EXQ_TS_ACC_10_MSEC = 0x00000000;
const OPCQuality EXQ_TS_ACC_100_MSEC = 0x00002000;
const OPCQuality EXQ_TS_ACC_SECOND = 0x00004000;
const OPCQuality EXQ_TS_ACC_BAD_TIME = 0x00006000;
}
#endif // _DAIS_COMMON_IDL

**SupportedFunctions**

This constant tells what functions the DAIS server (i.e., Data access, Alarms & Events, or both) supports. In case the server is extended with other functions they are included as well (e.g., Historical Data functionality (HDAIS)).

**DAF declarations**

These declarations (for example, ResourceID) import DAF declarations to DAIS.

**ItemID**

A pair of a ResourceID and a PropertyID. It uniquely identifies an item; that is, a property at a node.

<table>
<thead>
<tr>
<th>Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>resource</td>
<td>The ResourceID.</td>
</tr>
<tr>
<td>property</td>
<td>The PropertyID.</td>
</tr>
</tbody>
</table>

**ClientItemHandle**

A client created numeric handle used by the client to efficiently identify data coming from the server in callbacks.

**ServerItemHandle**

A server created numeric handle used by the server to efficiently identify items in client calls.
ServerItemIdentification

A union that holds either an ItemID or a pathname. Either can be used for identification of an item.

Error

Numeric error codes that are returned by ItemError.

<table>
<thead>
<tr>
<th>EnumValue</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ERROR_DAISOK</td>
<td>Used to indicate an error free result.</td>
</tr>
<tr>
<td>ERROR_BAD_RIGHTS</td>
<td>The Items AccessRights do not allow the operation.</td>
</tr>
<tr>
<td>ERROR_UNKNOWN_ITEMID</td>
<td>The resource or property in the ItemID is unknown.</td>
</tr>
<tr>
<td>ERROR_CLAMPED</td>
<td>A value passed to WRITE was accepted but the output was clamped.</td>
</tr>
<tr>
<td>ERROR_OUT_OF_RANGE</td>
<td>The value was out of range.</td>
</tr>
<tr>
<td>ERROR_UNKNOWN_PATHNAME</td>
<td>The pathname was not recognised.</td>
</tr>
<tr>
<td>ERROR_BAD_TYPE</td>
<td>The server cannot convert the data between the specified format/ requested data type and the canonical data type.</td>
</tr>
<tr>
<td>ERROR_UNKNOWN_ACCESS_PATH</td>
<td>The item's access path is unknown.</td>
</tr>
<tr>
<td>ERROR_INTERNAL_SERVER</td>
<td>An error has appeared in the server due to some server internal problem.</td>
</tr>
<tr>
<td>ERROR_INVALID_DAIS_HANDLE</td>
<td>A client or server handle was invalid.</td>
</tr>
</tbody>
</table>

ItemError

A struct for reporting of item related errors.

<table>
<thead>
<tr>
<th>Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>err</td>
<td>An error code as described below.</td>
</tr>
<tr>
<td>client_handle</td>
<td>The client side handle identifying the item.</td>
</tr>
<tr>
<td>server_handle</td>
<td>The server side handle identifying the item.</td>
</tr>
<tr>
<td>pathname</td>
<td>The pathname for display or presentation purposes.</td>
</tr>
<tr>
<td>reason</td>
<td>An additional text explaining the error.</td>
</tr>
</tbody>
</table>
AccessRights

Numeric access rights supported per item.

<table>
<thead>
<tr>
<th>EnumValue</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>READABLE</td>
<td>Read only data.</td>
</tr>
<tr>
<td>WRITEABLE</td>
<td>Write only data.</td>
</tr>
<tr>
<td>READ_AND_WRITABLE</td>
<td>Both read and write data.</td>
</tr>
</tbody>
</table>

OPCQuality

A flag word giving the OPC quality. Each flag has a specific meaning as described below. Four groups of flags exist:

1. Main quality telling if a value is good, bad, or suspected.
2. Detailed quality.
3. Limits telling if the value is stuck.
4. Historical data access flags. Those flags are described in the HDAIS specification.

Bit masks are defined to extract these flags.

Quality, status, and limit bit masks

<table>
<thead>
<tr>
<th>Mask</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OPC_QUALITY_MASK</td>
<td>Bit mask for main quality.</td>
</tr>
<tr>
<td>OPC_STATUS_MASK</td>
<td>Bit mask for detailed quality.</td>
</tr>
<tr>
<td>OPC_LIMIT_MASK</td>
<td>Bit mask for the limits.</td>
</tr>
</tbody>
</table>

Main quality enumeration numbers

<table>
<thead>
<tr>
<th>Enum</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OPC_QUALITY_BAD</td>
<td>The number for bad quality.</td>
</tr>
<tr>
<td>OPC_QUALITY_UNCERTAIN</td>
<td>The number for uncertain quality.</td>
</tr>
<tr>
<td>OPC_QUALITY_GOOD</td>
<td>The number for good quality.</td>
</tr>
</tbody>
</table>

After application of the \texttt{OPC	extunderscore QUALITY	extunderscore MASK} the quality shall be compared directly to the enumeration numbers to decide the quality.
### Detailed quality flags for bad quality

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OPC_QUALITY_CONFIG_ERROR</td>
<td>There is a server configuration error concerning this value.</td>
</tr>
<tr>
<td>OPC_QUALITY_NOT_CONNECTED</td>
<td>The source of the value is not connected.</td>
</tr>
<tr>
<td>OPC_QUALITY_DEVICE_FAILURE</td>
<td>A device failure has been detected.</td>
</tr>
<tr>
<td>OPC_QUALITY_SENSOR_FAILURE</td>
<td>A sensor failure has been detected.</td>
</tr>
<tr>
<td>OPC_QUALITY_LAST_KNOWN</td>
<td>The updating has stopped but there is an old value available.</td>
</tr>
<tr>
<td>OPC_QUALITY_COMM_FAILURE</td>
<td>Communication has failed and no value available.</td>
</tr>
<tr>
<td>OPC_QUALITY_OUT_OF_SERVICE</td>
<td>The updating of the value is manually blocked for update (the item is not active).</td>
</tr>
</tbody>
</table>

### Detailed quality flags for uncertain quality

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OPC_QUALITY_LAST_USABLE</td>
<td>The value is old. The time stamp gives the age.</td>
</tr>
<tr>
<td>OPC_QUALITY_EGU_EXCEEDED</td>
<td>The value is beyond the capability of representation (for example, counter overflow).</td>
</tr>
<tr>
<td>OPC_QUALITY_SENSOR_CAL</td>
<td>The sensor calibration is bad.</td>
</tr>
<tr>
<td>OPC_QUALITY_SUB_NORMAL</td>
<td>Value is derived from multiple sources where the majority has less than required good quality.</td>
</tr>
<tr>
<td>DAIS_QUALITY_OCILLATORY</td>
<td>If a binary value changes cyclically with a frequency higher than a specific threshold, it is oscillating. This quality compliant with IEC 61850-7-3.</td>
</tr>
</tbody>
</table>

### Definition of limit flags

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OPC_LIMIT_OK</td>
<td>The value is not limited; that is, it moves freely up or down.</td>
</tr>
<tr>
<td>OPC_LIMIT_LOW</td>
<td>The value is stuck at a low limit.</td>
</tr>
</tbody>
</table>
DAIS Quality extension masks

The part of the flag word giving the DAIS extended quality. Each flag has a specific meaning as described below. These quality definitions are based on the revised IEC 61850-7-3 definitions of quality. The following masks are defined.

<table>
<thead>
<tr>
<th>Mask</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXQ_SOURCE_MASK</td>
<td>Bit mask for the source.</td>
</tr>
<tr>
<td>EXQ_TEST_MASK</td>
<td>Bit mask for the test status. The test status indicates that the value is generated by a test and shall not be regarded as an operational value.</td>
</tr>
<tr>
<td>EXQ_OPERATOR_BLOCKED_MASK</td>
<td>Bit mask for the operator blocked status. The status indicates that the value has been blocked for update and is old. The OPC_QUALITY_LAST_USABLE quality shall be set as well.</td>
</tr>
<tr>
<td>EXQ_TIMESTAMP_ACCURACY_MASK</td>
<td>Bit mask for the time stamp accuracy.</td>
</tr>
</tbody>
</table>

Flags defining source

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXQ_SOURCE_NONE</td>
<td>There is no source for this data item. The code is used for spare items not yet allocated.</td>
</tr>
<tr>
<td>EXQ_SOURCE_PROCESS</td>
<td>The source for this value is the process.</td>
</tr>
<tr>
<td>EXQ_SOURCE_PRIMARY_SUBSTITUTED</td>
<td>The value is manually substituted.</td>
</tr>
<tr>
<td>EXQ_SOURCE_INHERITED_SUBSTITUTED</td>
<td>A substituted value has been copied or used as input to some calculation. The result value is then marked with \EXQ_SOURCE_INHERITED_SUBSTITUTED.</td>
</tr>
<tr>
<td>EXQ_SOURCE_CORRECTED</td>
<td>An alternate and more accurate value has been calculated by some application (e.g., a State Estimator). If this value has been used to correct the original value, it shall be indicated \EXQ_SOURCE_CORRECTED.</td>
</tr>
<tr>
<td>EXQ_REMOTE_DEFAULTED</td>
<td>The value is initialized by a default value.</td>
</tr>
</tbody>
</table>
Flags defining time stamp quality

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXQ_LOCAL_NONEEXQ_TS_ACC_10_MSEC</td>
<td>The value is not updated locally (i.e., it has a remote source specification). The flags (=0) indicate that the accuracy is 10 milliseconds or better (IEC61850-7-2 performance class T0).</td>
</tr>
<tr>
<td>EXQ_LOCAL_SUBSTITUTEDEXQ_TS_ACC_100_MSEC</td>
<td>The value is locally substituted (manually). The flags (=1) indicate that the accuracy is 100 milliseconds or better.</td>
</tr>
<tr>
<td>EXQ_LOCAL_SE_REPLACEDEXQ_TS_ACC_SECOND</td>
<td>The value is locally substituted by State Estimator. The flags (=2) indicate that the accuracy is in the range of seconds or better.</td>
</tr>
<tr>
<td>EXQ_TS_ACC_BAD_TIME</td>
<td>The flags (=3) indicate that the time stamp is bad.</td>
</tr>
</tbody>
</table>

Quality

The DAIS quality consists of **OPCQuality** and **ExtendedQuality**.

<table>
<thead>
<tr>
<th>Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc_quality</td>
<td>The quality as specified by OPC including extensions from DAIS.</td>
</tr>
<tr>
<td>user_quality</td>
<td>A user specific quality.</td>
</tr>
</tbody>
</table>

### 3.1.6 Iterator Methods IDL

Methods that return information about more than one resource may return an iterator. The resource description iterator allows a client to access a large query result sequentially, several resources at a time. This is necessary where the ORB limits message sizes. It also enables implementations to overlap the client and server processing of query results, if necessary.

The client and the data provider should cooperate to manage the lifetime of the iterator and the resources it consumes. The `destroy()` and `next_n()` methods allow the client and data provider respectively to indicate that the iterator may be destroyed.

In addition, the data provider may autonomously destroy the iterator at any time (for resource management or other reasons). If a client detects that an iterator has been destroyed, it will not interpret this condition in itself as either an indication that the end of the iteration has been reached, or as a permanent failure of the data provider.
next_n()

This operation returns possibly 0 and at most n resource descriptions in the form of a resource description sequence. In all cases the state of the iteration is indicated by the Boolean return value.

- True means that there may be more resource descriptions beyond those returned so far.
- False means all the resource descriptions have now been returned. No further calls are expected for this iterator and the data provider may destroy the iterator at any time after the call returns.

reset()

This operation resets the iterator to the first element.

close()

This operation returns a copy of the iterator.

destroy()

This operation is used to terminate iteration before all the resource descriptions have been returned. After destroy() is invoked no further calls are expected for this iterator. The data provider may destroy the iterator at any time after the call returns.

3.1.7 DAISNode IDL

3.1.7.1 DAIS::Node overview

A node may represent a real world object such as a location or a piece of equipment. A node may also represent a schema item such as a type or property. Nodes correspond to “branches” in the IOPCBrowseServerAddressSpace or IOPCEventAreaBrowser interfaces. Nodes correspond to Resources in the RDF model and the DAF interface. Each node has a universal identity given by its ResourceID. The ResourceID of a node is the same in all views provided by a DAIS server. DAIS servers may be coordinated with DAF servers so that a node has the same ResourceID as the corresponding resource. Each node has zero or more child(s). A child may be another node or any other type of object (for example, data access items or alarms & event sources). Each node has a type. The type determines what other types of child objects a node has.

Nodes are arranged in a strict hierarchy for naming purposes. A DAIS server may provide more than one such hierarchy, each is called a view. (The view is selected when the session is initiated.) Within a view, each node, except for the root node, has a single parent node, a label that is unique among all nodes with the same parent, and a unique pathname. A node’s pathname is a string that contains its label and the pathname of its parent. The pathname must be a valid URI, but apart from that the syntax of pathnames is implementation dependent.
The Node IDL defines a main interface `DAIS::Node::IHome` for browsing among the hierarchically structured nodes. Nodes are described by `DAIS::Node::Description` struct.

```
//File: DAISNode.idl
#ifndef _DAIS_NODE_IDL
#define _DAIS_NODE_IDL
#pragma prefix "omg.org"
#include <DAISCommon.idl>

module DAIS {
    module Node {
        struct Description {
            ResourceID id;
            ResourceID parent;
            string label;
            string descrip;
            ResourceID type;
            boolean is_leaf;
        };
        typedef sequence< Description > Descriptions;

        interface Iterator {
            boolean next_n ( ...
```

Figure 3-2  DAIS node IDL in UML

3.1.7.2 IDL
in unsigned long n,
out Descriptions nodes
);
void reset();
Iterator clone();
void destroy();
};

interface IHome
{
exception UnknownResourceID {string reason;};
exception InvalidFilter {string reason;};
exception UnknownTypeID {string reason;};

Description find(
    in ResourceID node
) raises (UnknownResourceID);

Descriptions find_each(
    in ResourceIDs nodes
) raises (UnknownResourceID);

Iterator find_by_parent(
    in ResourceID node,
    in string filter_criteria
) raises (UnknownResourceID, InvalidFilter);

Iterator find_by_type(
    in ResourceID node,
    in string filter_criteria,
    in ResourceIDs type_filter
) raises (UnknownResourceID, InvalidFilter, UnknownTypeID);

Strings get_pathnames(
    in ResourceIDs nodes
);

ResourceIDs get_ids(
    in Strings pathnames
);

};};}
#endif // _DAIS_NODE_IDL

Description

A struct describing a node.
Iterator

Refer to Section 3.1.6, “Iterator Methods IDL,” on page 3-10. This interface corresponds to the OPC interface EnumString with the difference that the Iterator returns the Description struct instead of a string.

IHome

An interface used for browsing nodes. The interface corresponds to the IOPCBrowseServerAddressSpace with the BrowseFilterType set to OPC_BRANCH or the IOPCEventAreaBrowser. A major difference to OPC is that the server does not provide a cursor for clients. Instead clients have to provide the browse position in each call.

UnknownResourceID

An exception telling that the ResourceID is unknown. For methods taking a sequence of resource ids the first found unknown id is reported. The likely reason behind this exception is some misunderstanding between the server and client code due to a programming error.

InvalidFilter

An exception telling the filter_criteria string is not correct. The likely reason behind this exception is an erroneously entered string.

UnknownTypeID

An exception telling one or more TypeIDs does not exist.

find()

For a given node, return information about that node.

<table>
<thead>
<tr>
<th>Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>The identification of this node.</td>
</tr>
<tr>
<td>parent</td>
<td>The identification of the parent node.</td>
</tr>
<tr>
<td>label</td>
<td>The label (single level designation) of the node.</td>
</tr>
<tr>
<td>descrip</td>
<td>A descriptive text for the node.</td>
</tr>
<tr>
<td>type</td>
<td>A reference to the type of the node.</td>
</tr>
<tr>
<td>is_leaf</td>
<td>Indicate if the node is a leaf (i.e., it has no child nodes).</td>
</tr>
</tbody>
</table>
find_each()

For a sequence of nodes, return information about each node.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>node</td>
<td>A node identification.</td>
</tr>
<tr>
<td>return</td>
<td>The node description.</td>
</tr>
</tbody>
</table>

find_by_parent()

For a given node, return all child nodes at that node. Hence to reach leaf nodes using this method repeated calls must be made for each level. This corresponds to the OPC method `BrowseOPCItemIDs` with the parameter `dwBrowseFilterType` set to `OPC_BRANCH`.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>node</td>
<td>The parent node identification.</td>
</tr>
<tr>
<td>filter_criteria</td>
<td>A server specific filter string. This is entirely free format and may be entered by the user via a text field. An empty string indicates no filtering. The filter selects nodes with a pathname matching the filter criteria. For a description of the filter refer to Section 3.1.11, “Filter Definitions,” on page 3-25.</td>
</tr>
<tr>
<td>return</td>
<td>An iterator holding the child node descriptions.</td>
</tr>
</tbody>
</table>

find_by_type()

For a sub-tree given by the node parameter, return all child nodes of the specified type. This will return all leaf nodes under the given sub-tree root node. There is no corresponding operation in OPC. Refer to Section 4.2.4.1, “DAIS::Item Overview,” on page 4-15 for a description of how item browsing is mapped to OPC.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>node</td>
<td>The identification of the node defining the sub-tree.</td>
</tr>
</tbody>
</table>
get_pathnames()

Translate a sequence of node identifications to the corresponding sequence of pathnames. If a node fails to translate to a pathname (due to an unknown node identification), the corresponding pathname is an empty string.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nodes</td>
<td>The sequence of nodes.</td>
</tr>
<tr>
<td>return</td>
<td>The corresponding sequence of pathnames.</td>
</tr>
</tbody>
</table>

get_ids()

Translate a sequence of pathnames to the corresponding sequence of node identifications. If a pathname fails to translate to a node identification (due to an unrecognized pathname), the corresponding node identification is NULL.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pathnames</td>
<td>The sequence of pathnames.</td>
</tr>
<tr>
<td>return</td>
<td>The corresponding sequence of node identifications.</td>
</tr>
</tbody>
</table>

3.1.8 DAISType IDL

3.1.8.1 DAIS::Type Overview

A type represents a set of related properties and associations. Each node has a type and all the properties represented by that type apply to the node. Each type is identified by a ResourceID and has a label and description. A type may be obtained for any node using the node’s TypeID. Related types may be grouped into a schema. A ResourceID identifies each schema. All the types in a schema may be obtained given the schema ResourceID. A schema and its type may be represented as nodes in one or more of views provided by a DAIS server. When a schema is represented as a node, the node’s ResourceID and the schema ResourceID are identical. Similarly, when a type is
represented as a node, the node’s ResourceID and the type’s ResourceID are identical. The type’s parent is always the node representing its schema, the type’s label is identical to the node label and the type’s description is identical to the node description.

The type IDL defines a main interface DAIS::Type::IHome for browsing supported types.

```
<<Interface>>>
DAIS::Type::IHome

find() 1 0..n
find_by_schema() 1

<<Interface>>>
DAIS::Type::Iterator

max_left() 0..n
next_n() 0..n
destroy() 0..n
```

```
//File DAISType.idl
#ifndef _DAIS_TYPE_IDL
#define _DAIS_TYPE_IDL
#pragma prefix "omg.org"
#include "DAISCommon.idl"

module DAIS {
    module Type {

        struct Description {
            ResourceID id;
            ResourceID schema;
            string label;
            string descrip;
            ResourceIDs aggregated_types;
        };
        typedef sequence<Description> Descriptions;

        interface Iterator {
            boolean next_n (in unsigned long n, out Descriptions types);
        }
    }
}
```

Figure 3-3 DAIS type IDL in UML

3.1.8.2 IDL
void reset();
Iterator clone();
void destroy();

};

interface IHome
{
   exception UnknownResourceID {string reason;};

   Description find ( 
      in  ResourceID type
   ) raises (UnknownResourceID);

   Iterator find_by_schema ( 
      in  ResourceID schema
   ) raises (UnknownResourceID);
};};
#endif // _DAIS_TYPE_IDL

Description
A struct describing a type.

<table>
<thead>
<tr>
<th>Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>The identification of this type.</td>
</tr>
<tr>
<td>schema</td>
<td>The identification of the schema where the type is defined.</td>
</tr>
<tr>
<td>label</td>
<td>The label of the type.</td>
</tr>
<tr>
<td>descrip</td>
<td>A description of the type.</td>
</tr>
<tr>
<td>aggregated_types</td>
<td>A sequence of type identifications that a node of this type may contain. This information is intended as a guide when the type filter is specified for the find_by_type() methods.</td>
</tr>
</tbody>
</table>

IHome
An object used to browse the types. There is no corresponding interface in OPC.

UnknownResourceID
An exception telling that the ResourceID is unknown. The likely reason behind this exception is some misunderstanding between the server and client code due to a programming error.
find()

For a given type, return information about that type.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>A type identification.</td>
</tr>
<tr>
<td>return</td>
<td>The type description.</td>
</tr>
</tbody>
</table>

find_by_schema()

For a given schema, find all types defined by that schema.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>schema</td>
<td>The identification of the schema.</td>
</tr>
<tr>
<td>return</td>
<td>A sequence of type descriptions.</td>
</tr>
</tbody>
</table>

3.1.9 DAISProperty IDL

3.1.9.1 DAIS::Property Overview

A property represents a characteristic of a node that can be described with a value. A given property may apply to many nodes, for each such node there will be an item corresponding to the property. (See Section 4.2.4, “DAISItem IDL,” on page 4-15).

A DAIS property corresponds to a property in RDF and the DAF. A DAIS property corresponds to the concept of a property in OPC accessed with the IOPCItemProperties::QueryAvailableProperties() method. However, the six core OPC properties (timestamp, quality, value) do not correspond to properties in DAIS. They are given special treatment in OPC (they are not the same as other OPC properties). See Section 4.2.4, “DAISItem IDL,” on page 4-15 for interfaces to handle these.

Each property is identified by a ResourceID and has a label, description, and canonical data type. The canonical data type is a member of the SimpleType enumeration and indicates the preferred CORBA atomic data type for values of this property. Every item of a given property has an identical canonical data type. A property may be obtained for any item via the property member of the ItemID. All properties that apply to a given node may be obtained, given the node’s ResourceID. All properties that apply to the nodes of a given type may be obtained, given the TypeID. A property may be represented as a node in one or more of views provided by a DAIS server. In this case the node’s ResourceID and the property’s ResourceID are identical. The property’s parent is always the node representing the type to which it belongs. The property’s label is identical to the node label and the property’s description is identical to the node description.
The property IDL defines a main interface DAIS::Property::IHome for browsing properties. The information model describing how the properties are related to nodes and items is found in Section 4.1.1, “Nodes, Items, Types, and Properties,” on page 4-2.

Figure 3-4 DAIS property IDL in UML

3.1.9.2 IDL

```
//File: DAISProperty.idl
#ifndef _DAIS_PROPERTY_IDL
#define _DAIS_PROPERTY_IDL
#pragma prefix "omg.org"
#include <DAISCommon.idl>

module DAIS {
module Property {

struct Description {
    PropertyID id;
    string label;
    string descrip;
    SimpleValueType canonical_data_type;
};
typedef sequence<Description> Descriptions;

interface IHome {
    exception UnknownResourceID {string reason;};

    Description find (
        in PropertyID property
    ) raises (UnknownResourceID);

    Descriptions find_each (
        in PropertyIDs properties
    ) raises (UnknownResourceID);

    Descriptions find_by_node (
        in ResourceID node
    ) raises (UnknownResourceID);
}
```

Descriptions find_by_type (in ResourceID type) raises (UnknownResourceID);

Description
Describe a property.

<table>
<thead>
<tr>
<th>Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>The identification of this property.</td>
</tr>
<tr>
<td>label</td>
<td>The label (single level designation) of the property.</td>
</tr>
<tr>
<td>descrip</td>
<td>A description of the property.</td>
</tr>
<tr>
<td>canonical_data_type</td>
<td>The data type used for the property in the server.</td>
</tr>
</tbody>
</table>

Home
An object used for browsing properties defined for a type or existing at a node. It corresponds to the OPC interface IOPCItemProperties.

UnknownResourceID
An exception telling that the ResourceID is unknown. The likely reason behind this exception is some misunderstanding between the server and client code due to a programming error.

find()
For a given property, return information about that property.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>property</td>
<td>A property identification.</td>
</tr>
<tr>
<td>return</td>
<td>The property description.</td>
</tr>
</tbody>
</table>

find_each()
For a given property, return information about that property.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>properties</td>
<td>A sequence of property identifications.</td>
</tr>
<tr>
<td>return</td>
<td>The sequence of property descriptions.</td>
</tr>
</tbody>
</table>
**find_by_node ()**

For a node, return information about each property describing its items. This method corresponds to `IOPCItemProperties::QueryAvailableProperties()`.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>node</td>
<td>A node identification.</td>
</tr>
<tr>
<td>return</td>
<td>A sequence of property descriptions.</td>
</tr>
</tbody>
</table>

**find_by_type ()**

For a given type, return all property descriptions.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>A type identification.</td>
</tr>
<tr>
<td>return</td>
<td>A sequence of property descriptions.</td>
</tr>
</tbody>
</table>

### 3.1.10 DAISSession IDL

#### 3.1.10.1 DAIS::Session Overview

The session is an interface inherited by data access and alarms & events sessions. A session represents a single conversation with the DAIS service. A session has a connection to a shut down callback used by a server to shut down clients in the case of an ordered shut down of the server. If the session object is destroyed or a failure is detected by the server when invoking an operation on any callback objects, then the session is terminated.

A session object corresponds to the OPC server object; hence each OPC server object will be represented by a session object in DAIS.

![Figure 3-5 DAIS session IDL in UML](image-url)
The `status()` method corresponds to the read only `SessionStatus` attribute in the IDL. The `callback()` get and set methods correspond to the `ShutDownCallback` attribute.

### 3.1.10.2 IDL

```idl
//File DAISSession.idl
#ifndef _DAIS_SESSION_IDL
#define _DAIS_SESSION_IDL
#pragma prefix "omg.org"
#include <DAISCommon.idl>

module DAIS {

    struct SessionStatus {
        string name;
        DateTime start_time;
        DateTime current_time;
        DateTime last_update_time;
        unsigned long group_count;
        long band_width;
    }

    interface ShutdownCallback {
        void shutdown_notify (in string reason);
    }

    interface Session {
        readonly attribute SessionStatus status;
        attribute ShutdownCallback callback;
        void destroy();
    }
}
#endif // _DAIS_SESSION_IDL
```

**SessionStatus**

A struct holding session status.
ShutdownCallback

An object implemented by clients and used by the server to indicate that it will shutdown soon. No further calls should be made and no further data callbacks should be expected.

**shutdown_notify()**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>reason</td>
<td>An explanation of why the server is shutting down.</td>
</tr>
</tbody>
</table>

Session

An interface representing a single conversation with the DAIS service. The interface is inherited into interfaces representing sessions supporting specific services, (for example, data access or alarms & events).

**status**

A read only attribute holding the `SessionStatus`.

**callback**

An attribute holding a reference to a `ShutdownCallback` object. A client that wants to receive shut down callbacks from a server shall update the attribute with a reference to a `ShutdownCallback` object.
3.1.11 Filter Definitions

Some methods have a filter string (for example, the browse methods `find_by_type()` and `find_by_parent()` in DAISNode and DAISItem). The filter string shall contain a pattern that is used to match a text string (e.g., the pathname). The text strings matching the pattern are passing the filter.

The strings that may appear in the filter string pattern are listed below.

<table>
<thead>
<tr>
<th>Characters in pattern</th>
<th>Matches in string</th>
</tr>
</thead>
<tbody>
<tr>
<td>?</td>
<td>Any single character.</td>
</tr>
<tr>
<td>*</td>
<td>Zero or more characters.</td>
</tr>
<tr>
<td>#</td>
<td>Any single digit (0-9).</td>
</tr>
<tr>
<td>substring</td>
<td>The specified substring</td>
</tr>
<tr>
<td>[charlist]</td>
<td>Any single character in charlist.</td>
</tr>
<tr>
<td>![charlist]</td>
<td>Any single character not in charlist.</td>
</tr>
</tbody>
</table>

Some examples of patterns and strings are given below.

<table>
<thead>
<tr>
<th>Pattern</th>
<th>String</th>
<th>String pass filter</th>
</tr>
</thead>
<tbody>
<tr>
<td>a*a</td>
<td>aBBBa</td>
<td>Yes</td>
</tr>
<tr>
<td>[A-Z]</td>
<td>F</td>
<td>Yes</td>
</tr>
<tr>
<td>![A-Z]</td>
<td>F</td>
<td>No</td>
</tr>
<tr>
<td>a#a</td>
<td>a2a</td>
<td>Yes</td>
</tr>
<tr>
<td>a[L-P]#![c-e]</td>
<td>aM5b</td>
<td>Yes</td>
</tr>
<tr>
<td>B?T*</td>
<td>BAT123khg</td>
<td>Yes</td>
</tr>
<tr>
<td>B?T*</td>
<td>CAT123khg</td>
<td>No</td>
</tr>
</tbody>
</table>

3.1.12 Logical Expressions and Navigation

This section defines a little language that is used to describe logical expressions and navigation in structures. A logical expression evaluates to true or false and can be used in filters for collection of data or definition of states, etc. The condition logic described in Section 5.2.7, “DAISConditionSpace IDL,” on page 5-32 is used to define states.
In DAIS the data model seen through the API is a hierarchical structure of typed nodes. A typed node may contain other typed nodes and/or property values (called items). This data model is described in more detail in Section 4.1, “Information Model,” on page 4-1. In addition to this hierarchy DAIS supports data models where a property value may refer to one or more other nodes. Hence navigation via such references is needed as well. An information model that includes both hierarchy and references is the IEC 61970 briefly described in Section 4.1.5, “Utility SCADA/EMS Measurement Model,” on page 4-5.

To be able to navigate to nodes in the structure the little language shall include support for describing paths. XML Path Language (XPath) [13] is a language that supports navigation in structures and this specification uses a subset of XPath as the logical expression language. XPath requires the exposed data model to be hierarchical (even though a model is exposed as hierarchical it may internally be organized in other ways).

A logical expression in XPath is called “PredicateExpr” (statement 9 in the XPath specification [13]). A PredicateExpr evaluates to true or false. A logical expression in DAIS is called DAIS_Expression. The rules for a DAIS_expression are:

- DAIS_Expression ::= PredicateExpr

The following restrictions of the functionality described in the XPath specification are defined for DAIS_Expression:

- The union operator "|" is not supported, refer to statement 18 in [13].
- The arithmetic operators "+", ".", "div", "+" and "mod" are not supported, refer to statements 25, 26 and 27 in [13].
- unabbreviated location paths is not supported.
- abbreviated location paths is supported.
- the following axes are supported; child, parent, descendant, attribute, and self.
- the following XPath functions are supported; id(), position(), not(), true(), false().

The mapping between DAIS and XML data models is listed below.

<table>
<thead>
<tr>
<th>DAIS entity</th>
<th>XML entity</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node</td>
<td>A DAIS Node element that may be contained by another DAIS node element.</td>
<td>The DAIS Node element name is the DAIS::Type name (DAIS::Type::Description.label) referenced by DAIS::Node::Description.type. If the DAIS Node element has a parent element, the parent is given by Node::Description.parent.</td>
</tr>
<tr>
<td>Node::Description.id</td>
<td>An id attribute of type ID at a DAIS Node element.</td>
<td>Accessed by the id() function</td>
</tr>
<tr>
<td>Node::Description.label</td>
<td>A child element of a DAIS Node element.</td>
<td>The element name is “label.” The value is Node::Description.label.</td>
</tr>
</tbody>
</table>
Table 3-1  Mapping between DAIS and XML data models

<table>
<thead>
<tr>
<th>DAIS entity</th>
<th>XML entity</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node::Description. descrip</td>
<td>A child element of a DAIS Node element.</td>
<td>The element name is “description.” The value is Node::Description.descrip.</td>
</tr>
<tr>
<td>Item</td>
<td>A DAIS Item element that is child of a DAIS Node element.</td>
<td>The element name is the DAIS::Property name (DAIS::Property::Description.label) concatenated with the DAIS Node element name separated by a “.”. The DAIS::Property is given by Item::Description.id.property. The DAIS Node parent element is given by Item::Description.id.resource.</td>
</tr>
<tr>
<td>Item::Description.value</td>
<td>The value of the DAIS Item element.</td>
<td></td>
</tr>
</tbody>
</table>

XML data example

```xml
<MeasurementModel>
  <Measurement id="_100">
    <label>M100</label>
    <description/>
    <Measurement.LimitSets>_200</Measurement.LimitSets>
    <MeasurementValue id="_500">
      <label>TM100</label>
      <description/>
      <MeasurementValue.value>30</MeasurementValue.value>
    </MeasurementValue>
  </Measurement>
  <LimitSet id="_200">
    <label>summer</label>
    <description/>
    <Limit id="_300">
```
3.1.13 Authorization

Through DAIS an operator gets access to data in a control system. The operator can:

- Read data (Data Access) and alarms and events (Alarms & Events).
- Write data (Data Access)
- Acknowledge alarms (Alarms & Events).

Many control systems implement an authorization scheme where it checks if an operator is allowed to read, write, or acknowledge.

A DAIS server exposes many data objects. If authorization is supported, checks must be made by the server. The server must then know who the operator is. An interface that can be used by the server to get this information is described in the Security Service Specification [14]. The Security Service Specification includes a rich interface supporting extensive security comprising:

- Identification and authentication
- Authorization and access control
- Security auditing
• Security of communication
• Non-repudiation
• Administration

The smallest need for a DAIS server is to be able to identify an operator (a principal in Security language) so that access control can be made from within the DAIS server. Authentication is assumed taken care of at the operating system login. The control of access to a DAIS server itself within the scope of a secure system is not necessarily a requirement. The other functions (auditing, secure communication, non-repudiation, and administration) supported by the Security Service are not necessarily required for a DAIS server.

3.1.14 Requirement Levels

For a DAIS server the following requirement levels for support by the system are defined:

0 - No authorization and no requirements.

1 - Identification of principal requiring SecurityLevel1 interfaces, refer to [14] appendix A.

2 - Security levels as described in [14] appendix C, mainly Security Functionality Level 1.
3.2 Server

3.2.1 DAISServer IDL Overview

The **DAISServer** IDL describes the DAIS server interface and depends on the types of session interfaces (for example, data access or alarms & events) it implements. A server not implementing a session type (data access or alarms & events) shall still be capable of reporting the type as not implemented. The IDL files and dependencies in Figure 3-6 are defined below.

![Diagram of dependencies between server IDL files](image)

Figure 3-6 Dependencies between server IDL files

3.2.2 DAIS Server IDL

3.2.2.1 DAIS::Server objects Overview

The fundamental DAIS service from which session objects may be obtained. **DAIS::Server** would normally be implemented as a persistent object accessed via the naming service or the trader service. From the **DAIS::Server** object, the session objects for data access or alarms & events are created. A client may create as many session objects as wanted.

A session can be created for a view. A view corresponds to a specific hierarchical organization of objects (also called nodes in this specification). The same object may appear in multiple views and hence in different hierarchical structures. An example is a breaker appearing in a functional structure (having the function of breaking current) and in a location structure (the place where it is located), refer also to [10]. Another example is the same object appearing in different areas of authority.
A DAIS server supporting data access may have a number of persistent public groups. Public groups can be used and managed through the group interface, refer to Section 4.2.7, “DAISGroup IDL,” on page 4-40.

Services that are not implemented will raise the CORBA standard exception NO_IMPLEMENT. Not implemented exceptions can be expected for a data access session, an alarms & event session, and inspection.

![DAIS server IDL in UML](image)

**Figure 3-7** DAIS server IDL in UML

### 3.2.2.2 IDL

```cpp
//File: DAISServer.idl
#ifndef _DAIS_SERVER_IDL
#define _DAIS_SERVER_IDL
#pragma prefix "omg.org"
#include <DAISDASession.idl>
#include <DAISAESession.idl>

module DAIS {

  enum ServerState {
    SERVER_STATE_RUNNING,
    SERVER_STATE_FAILED,
    SERVER_STATE_NOCONFIG,
    SERVER_STATE_SUSPENDED,
    SERVER_STATE_TEST
  };
```
struct ServerStatus {
  DateTime start_time;
  DateTime current_time;
  ServerState server_state;
  unsigned long session_count;
  unsigned long major_version;
  unsigned long minor_version;
  unsigned long build_number;
  string vendor_info;
};

interface Inspection {};

interface Server {
  exception DuplicateName {string reason;};
  exception InvalidView {string reason;};

  readonly attribute ServerStatus status;
  readonly attribute SupportedFunctions supported_functions;

  DataAccess::Session create_data_access_session(
    in string session_name
  ) raises (DuplicateName);

  DataAccess::Session create_data_access_session_for_view(
    in string session_name,
    in string view_name
  ) raises (DuplicateName, InvalidView);

  AlarmsAndEvents::Session create_alarms_and_events_session(
    in string session_name
  ) raises (DuplicateName);

  AlarmsAndEvents::Session create_alarms_and_events_session_for_view(
    in string session_name,
    in string view_name
  ) raises (DuplicateName, InvalidView);

  Strings find_views();

  Inspection inspect();
};

#endif // _DAIS_SERVER_IDL
An enumeration of the states a Server may have.

<table>
<thead>
<tr>
<th>EnumValue</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SERVER_STATE_RUNNING</td>
<td>The server is running normally. This is the usual state for a server.</td>
</tr>
<tr>
<td>SERVER_STATE_FAILED</td>
<td>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.</td>
</tr>
<tr>
<td>SERVER_STATE_NOCONFIG</td>
<td>The server is running but has no configuration information loaded and thus cannot function normally. This state implies that the server needs configuration information in order to function. Servers that do not require configuration information should not return this state.</td>
</tr>
<tr>
<td>SERVER_STATE_SUSPENDED</td>
<td>The server has been temporarily suspended via some vendor specific method and is not getting or sending data. Note that Quality will be returned as OPC_QUALITY_OUT_OF_SERVICE.</td>
</tr>
<tr>
<td>SERVER_STATE_TEST</td>
<td>The server is in Test Mode. The outputs are disconnected from the real hardware but the server will otherwise behave normally. Inputs may be real or may be simulated depending on the vendor implementation. Quality will generally be returned normally.</td>
</tr>
</tbody>
</table>

**ServerStatus**

<table>
<thead>
<tr>
<th>Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>start_time</td>
<td>Time the 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.</td>
</tr>
<tr>
<td>current_time</td>
<td>The current time as known by the server.</td>
</tr>
<tr>
<td>server_state</td>
<td>The current status of the server. Refer to ServerState enumeration.</td>
</tr>
<tr>
<td>session_count</td>
<td>The total number of sessions created by clients for this server.</td>
</tr>
<tr>
<td>major_version</td>
<td>The major version of the server software.</td>
</tr>
<tr>
<td>minor_version</td>
<td>The minor version of the server software.</td>
</tr>
<tr>
<td>---------------</td>
<td>------------------------------------------</td>
</tr>
<tr>
<td>build_number</td>
<td>The ‘build number’ of the server software.</td>
</tr>
<tr>
<td>vendor_info</td>
<td>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.</td>
</tr>
</tbody>
</table>

**supported_functions**

A constant that tells what functions are implemented by the DAIS server (e.g., Data Access, Alarms & Events or both.)

**Inspection**

An optional interface intended to be specialized into a vendor specific inspection object. The inspection object is intended to expose server internal details for debugging and inspection purposes.

**Server**

An object implementing the DAIS server. A DAIS server might provide more than one view on nodes. A view is a specific hierarchical organization of nodes and nodes may appear in more than one hierarchical structure (for example, a functional structure or a location structure as defined by IEC 1346-1), refer also to Section 2.4, “IEC 1346-1, Structuring and Naming,” on page 2-7.

**DuplicateName**

An exception raised when an object is created and the name already exists. No object is created if the exception is raised. Is used for session and group manager objects.

**InvalidView**

An exception raised when an invalid view is specified. An invalid view is if the view name does not exist or a view not intended for the type of session is used (for example, a view for data access is used for alarms & events).

**status**

An attribute holding the ServerStatus.

**create_data_access_session()**

A method for creation of a data access session object. The default view will be used.
create_data_access_session_for_view()

A method for creation of a data access session object using a view of nodes.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>The name of the session. If an empty name is supplied, the server will create a name for the session. If a duplicate name is supplied, no session is generated.</td>
</tr>
<tr>
<td>return</td>
<td>A reference to the created <code>DAIS::DataAccess::Session</code> object.</td>
</tr>
</tbody>
</table>

create_alarms_and_events_session()

A method for creation of an alarms & events session object. The default view will be used.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>The name of the session. If an empty name is supplied, the server will create a name for the session. If a duplicate name is supplied, no session is generated.</td>
</tr>
<tr>
<td>view_name</td>
<td>The name of the view to open. If no name is supplied, the default view will be used.</td>
</tr>
<tr>
<td>return</td>
<td>A reference to the created <code>DAIS::DataAccess::Session</code> object.</td>
</tr>
</tbody>
</table>

create_alarms_and_events_session_for_view()

A method for creation of an alarms & events session object using a view of areas. This allows a server to support different area structures for different purposes (for example, operational responsibility, workorder management).
find_views()

A method to get the names for the server supported views.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>The name of the session. If an empty name is supplied, the server will create a name for the session. If a duplicate name is supplied, no session is generated.</td>
</tr>
<tr>
<td>view_name</td>
<td>The name of the view to open.</td>
</tr>
<tr>
<td>return</td>
<td>A reference to the created <code>DAIS::DataAccess::Session</code> object.</td>
</tr>
</tbody>
</table>

inspect()

A method for creation of an inspection object.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>return</td>
<td>The inspection object.</td>
</tr>
</tbody>
</table>
Session Management

Figure 3-8  Session management interaction
Contents

This chapter contains the following sections.

<table>
<thead>
<tr>
<th>Section Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Information Model”</td>
<td>4-1</td>
</tr>
<tr>
<td>“API”</td>
<td>4-8</td>
</tr>
</tbody>
</table>

The data access interface provides a client with a way to read, write, or subscribe for data (items) held by the server. It has a discovery interface where a client can browse and select available data. Selected data is used to compose groups and groups are used in read, write, or subscribe operations. To get subscribed data, a client has to connect a callback object to the server so that the server can notify the clients with changed data.

4.1 Information Model

Equipment in any industrial process is usually modeled as entities or classes in computerized systems. The class name usually reflects the type of equipment. Objects from such classes are called Real World Objects (RWOs) in this document. RWO classes have properties describing equipment characteristics (for example, size, length, geometries, material, nameplate data) as well as application specific data (for example, impedance parameters, state estimated values). Depending on the purpose of a system and the managed (industrial) process the RWOs are of different types and contain different properties. Some RWOs are common between systems and independent of industrial process like process variables; that is, measurement (state) and control variables. Other RWOs like transmission lines and breakers are specific to power transmission.
RWO will typically have properties depending on the RWO type (for example, measurement, transformer) and what application or usage the RWO is involved in. Properties are defined per type of RWO.

4.1.1 Nodes, Items, Types, and Properties

In DAIS RWOs are represented by Nodes. Property instances at a node are represented by Items. An item may represent a measurement value, a control output value, or any parameter (for example, a limit value, a unit, a name, a description). A Type and properties belonging to a type represent an RWO type by DAISProperties. Nodes are hierarchically structured and the leaves are Items. This is shown in Figure 4-1.

![DAIS data access server information model](image)

Figure 4-1  DAIS data access server information model

The Component class inherited by the Node and Item classes models the hierarchical structure. The Node may contain any number of Components as described by the Contains role having the cardinality many (role is a UML concept, refer to [9] for an explanation). A Component is a member of one Node as described by the MemberOf role having the cardinality of 1. A Component can be both a Node and an Item through the inheritance, which means a Node can contain other Nodes or Items. An Item cannot contain any Components as it only inherits the MemberOf role and not the Contains role.
4.1.2 Naming

Node and item names follow OPC [4] and IEC 1346-1 [10]. Each Node has a label unique among other Nodes having the same parent; that is, are MemberOf the same Node. An Item does not have an own label but uses the label from the Property. Each Item in a Node is associated with different DAISProperties so that the label is unique among other items at the same node. The labels in the path from an item or node to the root form a pathname. Labels and pathnames are explained in Figure 4-2.

Figure 4-2  Labels and pathnames

A delimiter, the label delimiter, might separate the labels in a pathname. Assuming the label delimiter is a “.” An example of an item pathname for a measurement value in Figure 4-2 is

Cobden.G1.P.Value

where the labels are:
• station; Cobden
• generator; G1
• active power measurement; P
• the actual measurement value property; Value

Exactly how the pathname is composed from the labels is server specific and outside the scope of this specification.

4.1.3 Item Values

Items are associated with values. Typically an item value provided by a DAIS server is read from a device and transferred to one or more clients. In a distributed control system involving remote devices (as indicated in Figure 4-3) communication failures might make item values not available. To cope with communication failures item values are associated with a quality. The quality indicates the reliability of the item value. Devices usually scan item values at a certain rate and item values will be transferred to the DAIS server at this rate or some other. In the server, item values will appear as time stamped and quality coded samples. A server that keeps item values in a local cache is expected to always hold the latest sample. Other item related informations are access-rights and scan-rate. This information is shown in Figure 4-4. The cache_value is the latest sample received from a device, the cache_value_last_updated the time when the cache_value was last updated or validated, and cache_value_quality when the value was last updated or validated.

For items, a DAIS server exposes the following information to clients:
• the value and its data type,
• the quality of the value,
• the time stamp for the value,
• the fastest scan rate with which the value can be expected to be updated, and
• the access rights.

To make access of item values efficient and avoid reading the values from devices each time a client requests item values, a DAIS server is expected to have a local cache. The mechanism for keeping the cache up to date is server specific but a client shall expect the following from the server:
• Values delivered from the cache reflect the latest value considering update rate and update dead bands. Based on the agreed update rate (between a client and a server) a client can expect that the server will validate the values with devices with the agreed update rate.
• The dead band is expected to be checked at each update or validation. Values that don’t transgress the deadband will not be reported.
• Time stamps delivered from the cache shall reflect when the values were updated or validated with the devices according to the agreed update rate. The time stamp gives the time for the latest successful update or validation.
• The quality shall reflect how successful the server has been in keeping the values updated or validated.

### 4.1.4 OPC Recommended Properties

DAIS is flexible in terms of what types and properties may exist in a server.

In OPC there is a recommendation of what properties are expected to be supported by a data access server. These recommended properties are shown in Table 4-1. The column names correspond to the attributes in the Property class seen in Figure 4-4.

<table>
<thead>
<tr>
<th>label</th>
<th>id</th>
<th>canonical_data_type</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>engineeringUnit</td>
<td>100</td>
<td>STRING_TYPE</td>
<td>Engineering unit.</td>
</tr>
<tr>
<td>description</td>
<td>101</td>
<td>STRING_TYPE</td>
<td>Description.</td>
</tr>
<tr>
<td>maxValue</td>
<td>102</td>
<td>DOUBLE_TYPE</td>
<td>Maximum value.</td>
</tr>
<tr>
<td>minValue</td>
<td>103</td>
<td>DOUBLE_TYPE</td>
<td>Minimum value.</td>
</tr>
<tr>
<td>sensorMaximum</td>
<td>104</td>
<td>DOUBLE_TYPE</td>
<td>Maximum value from an analog input sensor.</td>
</tr>
<tr>
<td>sensorMinimum</td>
<td>105</td>
<td>DOUBLE_TYPE</td>
<td>Minimum value from an analog input sensor.</td>
</tr>
<tr>
<td>closedLabel</td>
<td>106</td>
<td>STRING_TYPE</td>
<td>Text for the closed state for a discrete status input.</td>
</tr>
<tr>
<td>openLabel</td>
<td>107</td>
<td>STRING_TYPE</td>
<td>Text for the open state for a discrete status input.</td>
</tr>
<tr>
<td>itemTimeZone</td>
<td>108</td>
<td>UNSIGNED_TYPE</td>
<td>Reserved by OPC.</td>
</tr>
<tr>
<td>109-4999</td>
<td></td>
<td></td>
<td>Reserved by OPC.</td>
</tr>
</tbody>
</table>

For a system having data described by a particular schema (for example, the CIM for power systems) the implementation of that schema as seen through DAIS has to be decided. Either a mapping to the recommended OPC properties can be made or the schema can be exposed as is through DAIS. This is however outside the scope of this specification.

### 4.1.5 Utility SCADA/EMS Measurement Model

The classes and properties defined in the IEC 61970-30x (the CIM) can be mapped onto the OPC recommended properties. As the CIM is highly structured, this will create a complicated mapping. An alternative is to expose CIM as is through the DAIS. This means that the CIM classes and their properties will appear the same as seen through the
DAIS as through the DAF interfaces. As the DAIS requires a hierarchical structuring of nodes, the CIM equipment hierarchy is selected as the hierarchy exposed through the DAIS.

Besides the equipment hierarchy the CIM also has a number of other associations. Such associations may be made visible through the DAIS interface as properties holding ResourceIDs.

The CIM has several associations between RWOs and ways to structure RWOs. The requirement from DAIS on the equipment hierarchy is that it is strictly hierarchical, else DAIS is transparent to any information model.

For RWOs in the equipment hierarchy its properties simply appear as items at the nodes representing the RWOs. The DAIS supports navigating across associations as ResourceIDs can be conveyed by SimpleValues.

Utility SCADA/EMS systems have a number of different applications calculating alternate measurement values. In the IEC61970 draft standard this is modeled by Measurements containing one or more MeasurementValues, as shown in Figure 4-3.

![Figure 4-3 Utility SCADA Measurement Modeling](image)

The Measurement and the MeasurementValue appear as Nodes and their properties as Items. In the example from Section 4.1.2, “Naming,” on page 4-3, the Measurement again is represented by the pathname:

Cobden.G1.P

Moving down to MeasurementValue will add its label to the pathname so that it may look like:

Cobden.G1.P.Telem
If the MeasurementValue property for the value has the label Value, finally the pathname for the item will be:

Cobden.16kV.G1.P.Telem.Value

In a system also having a State Estimated MeasurementValue, the pathname for state estimated value might be:

Cobden.16kV.G1.P.SE.Value

Finally if the state estimated value is to replace the telemetered, this can be implemented picking the “best” MeasurementValue from Measurement itself shortcutting the MeasurementValue resulting in the path:

Cobden.G1.P.Value

The CIM classes that shall be navigable and whose data is exposed through DAIS are listed in Table 4-2.

Table 4-2  CIM Classes

<table>
<thead>
<tr>
<th>CIM class</th>
<th>Properties and references</th>
</tr>
</thead>
<tbody>
<tr>
<td>Measurement</td>
<td>All</td>
</tr>
<tr>
<td>MeasurementValue</td>
<td>All. MeasurementValues are expected to appear in the hierarchical structure so that all MeasurementValue instances are visible in the browser.</td>
</tr>
<tr>
<td>MeasurementUnit</td>
<td>All</td>
</tr>
<tr>
<td>LimitSet</td>
<td>All. The Measurement to LimitSet reference has a cardinality 1..*. It is expected that one LimitSet is the current used and exposed through DAIS.</td>
</tr>
<tr>
<td>Limit</td>
<td>As the LimitSet to Limit reference has the cardinality 1..* the mapping through DAIS is expected to be a number of properties where each property corresponds to one limit value.</td>
</tr>
<tr>
<td>ValueAliasSet</td>
<td>As the ValueAliasSet to ValueToAlias reference has cardinality 1..* the mapping through DAIS is expected to be a number of properties where each property corresponds to one translation from numeric to symbolic value.</td>
</tr>
<tr>
<td>Classes in the hierarchical structure above Measurement</td>
<td>An implementation is free to expose any hierarchical structure of classes above the Measurement. Multiple and different structures are allowed using different views (for views refer to Section 3.2.2, “DAIS Server IDL,” on page 3-30). An implementation is also free to expose any properties from these classes.</td>
</tr>
</tbody>
</table>

The table is based on the CIM version given by [11].
4.2 API

4.2.1 Data Access IDL Overview

The IDL is divided into files. Each file is modeled as a package in UML. A file that depends on declarations made in another file needs to include it. Figure 4-4 shows how the IDL files depend on each other.

4.2.2 DAISDASession IDL

4.2.2.1 DAIS::DataAccess::Session objects overview

The DAIS::DataAccess::Session object implements the data access service on a per client basis. A data access session object has a number of services provided by one singleton home object each. Each home object provides methods for manipulation of the data of the specific type they provide. Rather than exposing data as objects with interfaces it is exposed as structs or sequences of structs. The reason is that a large number of data items will become a performance bottleneck if instantiated as objects over an interface. The
DAIS::Group::IHome object is the only object to expose its data as objects; that is, the DAIS::Group::Manager. Each DAIS::Group::Manager is expected to connect to one callback object implemented by the client.

Each client may instantiate one or more Sessions. The Session objects have one Type and Property Home object each. The client shall expect that all Type and Property Home objects expose the same types and properties.

The session object corresponds to an OPCServer object.

Figure 4-5  DAIS data access session IDL in UML
4.2.2.2 IDL

//File: DAISDASession.idl
#ifndef _DAIS_DA_SESSION_IDL
#define _DAIS_DA_SESSION_IDL
#pragma prefix "omg.org"

// Common Information
#include <DAISNode.idl>
#include <DAISProperty.idl>
#include <DAISSession.idl>

// Data Access interface
#include <DAISType.idl>
#include <DAISItem.idl>
#include <DAISDANode.idl>
#include <DAISGroup.idl>
#include <DAISDASimpleIO.idl>

module DAIS {
    module DataAccess {

        interface Session : DAIS::Session {
            readonly attribute Group::IHome group_home;
            readonly attribute SimpleIO::IHome simple_io_home;
            readonly attribute Node::IHome node_home;
            readonly attribute Item::IHome item_home;
            readonly attribute Type::IHome type_home;
            readonly attribute Property::IHome property_home;
        }
    }
}
#endif // _DAIS_DA_SESSION_IDL

Session

Session is an object implementing the data access functions. It inherits common functionality as shut down callbacks and session status from DAIS::Session.

group_home

A read only attribute holding a reference to a singleton Group::IHome object.
node_home

A read only attribute holding a reference to a singleton Node::IHome object.

item_home

A read only attribute holding a reference to a singleton Item::IHome object.

type_home

A read only attribute holding a reference to a singleton Type::IHome object.

property_home

A read only attribute holding a reference to a singleton Property::IHome object.

4.2.3 DAISDANode IDL

4.2.3.1 DAIS::DataAccess::Node overview

The DAIS::DataAccess::Node inherits most of the functionality from DAIS::Node. The only difference is that it supports to get the tree root node.

Figure 4-6 DAIS data access node IDL in UML
4.2.3.2 IDL

//File: DAISDANode.idl
#ifndef _DAIS_DANODE_IDL
#define _DAIS_DANODE_IDL
#pragma prefix "omg.org"
#include <DAISNode.idl>

module DAIS {
    module DataAccess {
        module Node {

            interface IHome : DAIS::Node::IHome {

                ResourceID get_root();
            }
        }
    }
#define _DAIS_DANODE_IDL
#endif // _DAIS_DANODE_IDL

IHome

An object used for browsing nodes. Most functionality is inherited from the DAIS::Node::IHome interface.

get_root()

Get the root node of the whole tree of nodes.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>return</td>
<td>The root node identification.</td>
</tr>
</tbody>
</table>
Hierarchical browsing

Browsing by type

Figure 4-7  Hierarchical browsing interaction

Figure 4-8  Browsing by type interaction
4.2.4 DAISItem IDL

4.2.4.1 DAIS::Item Overview

An item is a property of a node. While a node generally represents a real-world object, an item represents some characteristic of that object (for example, measurement value, control variable, or parameter related to the measurement/control process). The concept of an item in DAIS corresponds to the item in OPC. More precisely, it corresponds to the “leafs” in the OPC IOPCBrowseServerAddressSpace interface. In the Resource Description Framework (RDF) data model, the item corresponds to a combination of a subject and a predicate; that is, a resource and a property.

A node may have many items, each representing a different characteristic of the same real world object. An item might represent a measured variable, a calculated variable, a control variable, or a configuration parameter. An item will typically have many values where each value corresponds to a time stamped sample (a single item value corresponds to a single statement in the RDF model). Each sample will also have its own quality. An item value is then qualified by a time stamp and a quality.

An item value and its qualifications are represented by six fixed attributes:

- **value** - the value.
- **time_stamp** - the time when the value was last updated.
- **quality** - the quality of the value.
- **canonical_data_type** - the data type of the value. This actually belongs to the property (see Property) but it is mirrored at the item.
- **access_rights** - tells if the value is read only, write only, or both read/write. The access right is common for all item values and belongs to the item.
- **scan_rate** - the fastest rate with which the value can be expected to be updated. The scan rate is common for all item values and belongs to the item.

For an information model describing this refer to Section 4.1.1, “Nodes, Items, Types, and Properties,” on page 4-2 and Section 4.1.3, “Item Values,” on page 4-4.

Each item has a universal identity given by its ItemID. The ItemID is made up of the ResourceID of a node and the PropertyID of a property. The ItemID of an item is the same in all views provided by a DAIS server. Clients may construct ItemIDs given the identities of nodes and properties.

DAIS servers may be coordinated with DAF servers so that valid ItemIDs can be constructed from DAF ResourceIDs.

Within each view provided by the server, an item has a label that is unique among all items belonging to the same node. Within each view, an item has a unique pathname. The pathname is a string that contains the item’s label and the pathname of its node.

The pathname must be a valid URI, but apart from that the syntax of pathnames is implementation dependent.
Note – In OPC the pathname is the primary way to identify items and is called the ItemID. In DAIS the ResourceID, PropertyID pair is the primary identification and so it is called the ItemID. This is a potential point of confusion. The two ItemIDs play approximately the same role in OPC and DAIS respectively, but they are not the same type.

The item interfaces permit the stock of items to be browsed. Once an item is located, the group interfaces are used to deliver its values, selected by source, at successive times. In addition, the item interfaces provide the most current value for the default source. In either case the item value and qualifications are represented by the six fundamental attributes listed above.

The Item IDL defines a main interface Home for browsing among hierarchically structured items (leaf nodes). The information model describing the hierarchical structure is found in Section 4.1.1, “Nodes, Items, Types, and Properties,” on page 4-2.

**Figure 4-9**  DAIS data access item IDL in UML

### 4.2.4.2 IDL

```idl
//File: DAISItem.idl
#ifndef _DAIS_ITEM_IDL
#define _DAIS_ITEM_IDL
#pragma prefix "omg.org"
#include <DAISCommon.idl>
```

```idl
DAIS::DataAccess::Item::Iterator
max_left() next_n() reset() clone() destroy()
```

```idl
DAIS::DataAccess::Item::Home
find() find_each() find_by_parent() find_by_type() get_pathnames() get_ids() get_access_paths()
```

```idl
NodeItemComponent (from DAISNode)
pathname : string
```

```idl
DAIS::DataAccess::Item::Home
cache_value : SimpleValue
cache_value_last_updated : DateTime
cache_value_quality : DAIS::Quality
scan_rate : unsigned long
access_rights : DAIS::AccessRights
```
module DAIS {
    module DataAccess {
        module Item {

            struct Description {
                ItemID id;
                string label;
                SimpleValue value; // includes the canonical_data_type
                Quality dais_quality;
                DateTime time_stamp;
                AccessRights access_rights;
                unsigned long scan_rate;
            }

            typedef sequence< Description >Descriptions;

            interface Iterator {
                boolean next_n (
                    in unsigned long n,
                    out Descriptions items);
                void reset();
                Iterator clone();
                void destroy();
            }

            interface IHome {
                exception UnknownResourceID {string reason;};
                exception UnknownItemID {string reason;};
                exception InvalidFilter {string reason;};
                exception InvalidValueType {string reason;};
                exception UnknownTypeID {string reason;};
                exception InvalidAccessRight {string reason;};

                Description find (
                    in ItemID item
                ) raises (UnknownItemID);

                Descriptions find_each(  
                    in ItemIDs items
                ) raises (UnknownItemID);

                Iterator find_by_parent (  
                    in ResourceID node,
                    in string filter_criteria,
                    in SimpleValueType data_type_filter,
                    in AccessRights access_rights_filter
                ) raises (UnknownResourceID,
            }
InvalidFilter,
InvalidValueType,
InvalidAccessRight);

Iterator find_by_type (  
in ResourceID node,
in ResourceIDs type_filter,
in string filter_criteria,
in SimpleValueType data_type_filter,
in AccessRights access_rights_filter  
) raises (UnknownResourceID,
InvalidFilter,
InvalidValueType,
UnknownTypeID,
InvalidAccessRight);

Strings get_pathnames (  
in ItemIDs items
);

ItemIDs get_ids (  
in Strings pathnames
);

Strings get_access_paths (  
in ItemID item
) raises (UnknownItemID);

Description

A struct describing an item.

<table>
<thead>
<tr>
<th>Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>The identification of this item.</td>
</tr>
<tr>
<td>label</td>
<td>The label (single level designation) of the item.</td>
</tr>
<tr>
<td>value</td>
<td>The current value sample for the item. The SimpleValue also contains the data type.</td>
</tr>
<tr>
<td>dais_quality</td>
<td>The current quality of the value.</td>
</tr>
<tr>
<td>time_stamp</td>
<td>The time stamp for the value sample.</td>
</tr>
<tr>
<td>access_rights</td>
<td>States if the value is read, write, or both read and write.</td>
</tr>
<tr>
<td>scan_rate</td>
<td>States the highest update rate that can be expected.</td>
</tr>
</tbody>
</table>
Iterator

Refer to Section 4.1.5, “Utility SCADA/EMS Measurement Model,” on page 4-5. This interface corresponds to the OPC interface EnumString with the difference that the Iterator return the Description struct instead of a single string.

IHome

An object used for browsing items and corresponds to the IOPCBrowseServerAddressSpace.

UnknownResourceID

An exception telling that the ResourceID is unknown. The likely reason behind this exception is some misunderstanding between the server and client code due to a programming error.

UnknownItemID

An exception stating that the resource or property in the ItemID is unknown. For methods taking a sequence of item ids the first found unknown id is reported. The likely reason behind this exception is some misunderstanding between the server and client code due to a programming error.

InvalidFilter

An exception stating the filter_criteria string is not correct. The likely reason behind this exception is an erroneously entered string.

InvalidValueType

An exception stating that the SimpleValueType does not exist.

UnknownTypeID

An exception stating one or more TypeIDs does not exist.

InvalidAccessRight

An exception stating that the access rights do not exist.

**find()**

For a given item browse position, return information about that item.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>item</td>
<td>An item identification.</td>
</tr>
<tr>
<td>return</td>
<td>The item description.</td>
</tr>
</tbody>
</table>
**find_each()**

For a sequence of items, return information about each item.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>items</td>
<td>A sequence of item identifications.</td>
</tr>
<tr>
<td>return</td>
<td>An iterator holding the item descriptions.</td>
</tr>
</tbody>
</table>

**find_by_parent()**

For a given node identification, return child items to that node. Hence to reach all items using this method repeated calls must be made for each node level. This corresponds to the OPC method BrowseOPCItemIDs with the parameter dwBrowseFilterType set to OPC_LEAF.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>node</td>
<td>The parent node identification.</td>
</tr>
<tr>
<td>filter_criteria</td>
<td>A server specific filter string. This is entirely free format and may be entered by the user via a text field. An empty string indicates no filtering. The filter selects only items with pathnames matching the filter criteria. For a description of the filter refer to Section 3.1.11, “Filter Definitions,” on page 3-25.</td>
</tr>
<tr>
<td>data_type_filter</td>
<td>Select items having the specified canonical data type.</td>
</tr>
<tr>
<td>access_rights_filter</td>
<td>Select items having the specified access rights.</td>
</tr>
<tr>
<td>return</td>
<td>An iterator holding item descriptions for items \</td>
</tr>
<tr>
<td></td>
<td>• that are child to the parent node.</td>
</tr>
<tr>
<td></td>
<td>• matching the filter_criteria, data_type_filter, and access_rights_filter.</td>
</tr>
</tbody>
</table>

**find_by_type()**

For a sub-tree given by the node parameter, return all child items matching the filter criteria. This will return all items under the given sub-tree root node. This will make the items in the sub-tree to appear flattened out. This corresponds to the OPC method BrowseOPCItemIDs with the parameter dwBrowseFilterType set to OPC_FLAT.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>node</td>
<td>The identification for the node defining the sub-tree.</td>
</tr>
<tr>
<td>type_filter</td>
<td>Select nodes in the sub-tree having a type matching any of the types listed in the type_filter.</td>
</tr>
</tbody>
</table>
get_pathnames()

Translate a sequence of item identifications to the corresponding sequence of pathnames. If an item fails to translate to a pathname (due to an unknown identification), the corresponding pathname is an empty string.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>items</td>
<td>The sequence of items.</td>
</tr>
<tr>
<td>return</td>
<td>The corresponding sequence of pathnames.</td>
</tr>
</tbody>
</table>

get_ids()

Translate a sequence of pathnames to the corresponding sequence of item identifications. If a pathname fails to translate to an item identification (due to an unrecognized pathname) the corresponding item identification is NULL.

get_access_paths()

Get the possible communication paths how data can be retrieved for the node. An access path is expected to be human readable so that a human can pick one and feed it back to the server as the preferred path (via other interfaces).

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>item</td>
<td>An item identification.</td>
</tr>
<tr>
<td>return</td>
<td>A sequence of possible access paths for the item.</td>
</tr>
</tbody>
</table>
Browsing Items

Figure 4-10  Browsing items interaction
Navigate across associations

![Diagram showing data flow between client and server]

Figure 4-11  Navigating across associations interaction

4.2.5 DAIS::DAIO IDL

4.2.5.1 DAIS::DataAccess::IO Overview

These are definitions for transmitting item values to clients. Interfaces are defined for server side read and write operations and client side callback operations. Clients shall implement the Callback object for the server to use at transfer of data. A client may have any number of callback objects. The client shall connect each callback object to a server object implementing ConnectionPoint.

The IO interfaces support three different ways to read data and two different ways to write data.

**Read data**
- Synchronous read where the data is received at return from the sync_read() method.
- Asynchronous read where the data is returned at the Callback object.
- Subscription where data is sent spontaneously by the server at the callback object.

**Write data**
- Synchronous write returning to the client once all the written data has reached the devices.
• Asynchronous write returning when the data is received by the DAIS server. A callback on Callback is sent by the server once the written data has reached the devices.

Each item value is transmitted in a struct with a timestamp and quality indication. A sequence of this struct is either sent via the callback object or directly in read or write calls.

In OPC the IO interface corresponds to the OPC interfaces IOPCSyncIO, IOPCAsyncIO, and IOPCDataCallback.

Figure 4-12 DAIS data access IO IDL in UML

The DAISConnectionPoint callback() methods correspond to a get or set method for the callback attribute.

4.2.5.2 IDL

//File: DAISDAIO.idl
#ifndef _DAIS_DAIO_IDL
#define _DAIS_DAIO_IDL
#pragma prefix "omg.org"
#include <DAISCommon.idl>
module DAIS {
module DataAccess {
module IO {

DAIS::DataAccess::IO::SyncIO
<<Interface>>

sync_read()
sync_write()

DAIS::DataAccess::IO::AsyncIO
<<Interface>>

async_read()
async_write()
refresh()
cancel()
set_enable()
get_enable()

DAIS::DataAccess::IO::Item State
<<struct>>

client_handle : ClientItemHandle
time_stamp : DateTime
daish_quality : DAIS::Quality
value : SimpleValue

DAIS::DataAccess::IO::ConnectionPoint
<<Interface>>
callback() : DAIS::DataAccess::IO::Callback
callback(callback : DAIS::DataAccess::IO::Callback)

callback(callback : DAIS::DataAccess::IO::Callback)

DAIS::DataAccess::ItemState
<<Interface>>

0..* 1

Client
(from DAIS Server)

Figure 4-12 DAIS data access IO IDL in UML

The DAISConnectionPoint callback() methods correspond to a get or set method for the callback attribute.

4.2.5.2 IDL

module IO {

module DAIS {
enum DataSource {
    DS_CACHE,
    DS_DEVICE
};

struct ItemState {
    SimpleValue value;
    DateTime time_stamp;
    Quality dais_quality;
    ClientItemHandle client_handle;
};
typedef sequence<ItemState> ItemStates;

struct ItemUpdate {
    ServerItemHandle server_handle;
    SimpleValue value;
};
typedef sequence<ItemUpdate> ItemUpdates;

interface SyncIO {
    ItemStates sync_read (
        in DataSource data_source,
        in ServerItemHandles server_handles,
        out ItemErrors errors
    );

    ItemErrors sync_write (in ItemUpdates updates);
};
typedef unsigned long CancelID;

interface AsyncIO {
    exception NotConnected(string reason);
    exception InvalidCancelID(string reason);
    exception NotActive(string reason);

    CancelID async_read (in ServerItemHandles server_handles,
        in DataSource data_source,
        in unsigned long transaction_id
    ) raises (NotConnected, NotActive);

    CancelID async_write (in ItemUpdates updates,
CancellID refresh (in DataSource data_source, ) raises (NotConnected,NotActive);

void cancel (in CancellID cancel_id in unsigned long transaction_id ) raises (InvalidCancellID);

attribute boolean enabled;
};

interface Callback {

    void on_data_change (in unsigned long transaction_id, in boolean all_quality_good, in ItemStates states);

    void on_read_complete (in unsigned long transaction_id, in boolean all_quality_good, in ItemStates states, in ItemErrors errors);

    void on_write_complete (in unsigned long transaction_id, in ItemErrors errors);

    void on_cancel_complete (in unsigned long transaction_id);
};

interface ConnectionPoint {
    attribute Callback cllbck;
};

#endif // _DAIS_DAIO_IDL
**DataSource**

<table>
<thead>
<tr>
<th>Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DS_CACHE</td>
<td>Data cached in the server is requested.</td>
</tr>
<tr>
<td>DS_DEVICE</td>
<td>Data from the device is requested. This will force a read from the device or RTU. A read from device will be made regardless of the group or item active status and no group NotActive exception will be forced.</td>
</tr>
</tbody>
</table>

**ItemState**

The struct is the major carrier of data conveyed over the interface. It is the “message” holding the payload.

<table>
<thead>
<tr>
<th>Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>The value itself.</td>
</tr>
<tr>
<td>time_stamp</td>
<td>The time stamp when the value was last updated.</td>
</tr>
<tr>
<td>quality</td>
<td>The quality for the value.</td>
</tr>
<tr>
<td>client_handle</td>
<td>A client side handle enabling the client to make a quick look up of the item in its internal data structures.</td>
</tr>
</tbody>
</table>

**ItemUpdate**

The struct carry an update for an item.

<table>
<thead>
<tr>
<th>Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>server_handle</td>
<td>A server side handle enabling the server to make a quick look up of the item in its internal data structures.</td>
</tr>
<tr>
<td>value</td>
<td>The value that shall be used in the update.</td>
</tr>
</tbody>
</table>

**SyncIO**

An interface for the synchronous operations.

**sync_read()**

Synchronous read of items. Inactive items will be reported with OPCQuality set to OPC_QUALITY_OUT_OF_SERVICE.
**sync_write()**

Synchronous write of item values to devices (not the internal server cache). The active state of the group or the items is ignored.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>data_source</td>
<td>The source from where to read the data.</td>
</tr>
<tr>
<td>server_handles</td>
<td>A sequence specifying the whole or a subset of the server side handles as defined via the DAIS::GroupEntry::Manager interface.</td>
</tr>
<tr>
<td>errors</td>
<td>A sequence reporting items for which the read failed. An empty sequence indicates all read operations succeeded. Reported errors are:</td>
</tr>
<tr>
<td></td>
<td>• ERROR_BAD_RIGHTS (item is write only)</td>
</tr>
<tr>
<td></td>
<td>• INVALID_DAIS_HANDLE</td>
</tr>
<tr>
<td>return</td>
<td>A sequence of ItemStates for the items.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>updates</td>
<td>A sequence of ItemUpdates specifying all or a subset of the items defined for a GroupEntry::Manager. The ItemUpdate::value member is used to update the items in devices.</td>
</tr>
<tr>
<td>return</td>
<td>A sequence reporting items for which the write failed. An empty sequence indicates write operations for all items succeeded. Reported errors are:</td>
</tr>
<tr>
<td></td>
<td>• ERROR_BAD_RIGHTS (item is read only)</td>
</tr>
<tr>
<td></td>
<td>• ERROR_INVALID_DAIS_HANDLE</td>
</tr>
<tr>
<td></td>
<td>• ERROR_CLAMPED</td>
</tr>
<tr>
<td></td>
<td>• ERROR_OUT_OF_RANGE</td>
</tr>
<tr>
<td></td>
<td>• ERROR_BAD_TYPE</td>
</tr>
</tbody>
</table>

**AsyncIO**

An interface for asynchronous read or write operations.

**NotConnected**

An exception telling that there is no callback object connected by the client.

**InvalidCancelID**

An exception telling that the supplied cancel id number is not recognized.
NotActive

An exception telling that the group or all items in the group is inactive. Only issued when read from cache.

async_read()

Asynchronous read of items from devices. OPC may report read errors both at return from async_read() and at on_read_complete(). DAIS will report all errors at on_read_complete().

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>server_handles</td>
<td>A sequence specifying the whole or a subset of the server side handles, as defined via the DAIS::GroupEntry::Manager interface.</td>
</tr>
<tr>
<td>data_source</td>
<td>The source from where to read the data. When reading from cache, inactive items will be reported with OPCQuality set to OPC_QUALITY_OUT_OF_SERVICE.</td>
</tr>
<tr>
<td>transaction_id</td>
<td>A transaction number unique for the client. The number is returned in the corresponding on_read_complete call.</td>
</tr>
<tr>
<td>return</td>
<td>A cancellation number unique for the client. The number is used by a client to cancel an ongoing asynchronous read operation.</td>
</tr>
</tbody>
</table>

async_write()

Asynchronous write of item values to devices (not the internal server cache). OPC may report write errors both at return from async_write() and at on_write_complete(). DAIS will report all errors at on_write_complete().

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>updates</td>
<td>A sequence of ItemUpdates specifying all or a subset of the items defined for a DAIS::GroupEntry::Manager. The ItemUpdate::value member is used to update the items in devices.</td>
</tr>
<tr>
<td>transaction_id</td>
<td>A transaction number unique for the client. The number is returned in the corresponding on_read_complete call.</td>
</tr>
<tr>
<td>return</td>
<td>A cancellation number unique for the client. The number is used by a client to cancel an ongoing asynchronous write operation.</td>
</tr>
</tbody>
</table>
refresh()

Initiate a complete asynchronous read transfer for all item entries defined via the DAIS::DataAccess::GroupEntry::Manager interface. Inactive items will be reported with OPCQuality set to OPC_QUALITY_OUT_OF_SERVICE.

The cyclic on_data_change reporting continues unaffected by a refresh call. However, items still unchanged after a refresh will not be reported in a succeeding on_data_change call.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>data_source</td>
<td>The source from where to read the data.</td>
</tr>
<tr>
<td>transaction_id</td>
<td>A transaction number unique for the client. The number is returned in the corresponding on_data_change call.</td>
</tr>
<tr>
<td>return</td>
<td>A cancellation number unique for the client. The number is used by a client to cancel an ongoing asynchronous refresh operation.</td>
</tr>
</tbody>
</table>

cancel()

Cancel on ongoing refresh, async read, or async write operations. The server is expected to acknowledge a successfully initiated cancel operation with an on_cancel_complete() callback.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cancel_id</td>
<td>The server generated cancellation number for the operation to cancel.</td>
</tr>
</tbody>
</table>

enable

An attribute used to enable or disable the spontaneous on_data_change() callbacks. The enable state does not affect on data change response to refresh calls. When a group is created it is enabled by default.

Callback

An interface implemented by the client and used by the server to send data to the client.

on_data_change()

The method is called by the server when spontaneous changes occur or when the client has requested an explicit refresh. Only active items are reported in spontaneous calls.
on_read_complete()

The method is used by the server to report data in response to an asynchronous read.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>transaction_id</td>
<td>If the call is in response to a refresh, the transaction number for that refresh call. If the call is autonomous due to one or more spontaneous changes, the number is zero.</td>
</tr>
<tr>
<td>all_quality_good</td>
<td>All item quality values are good.</td>
</tr>
<tr>
<td>item_states</td>
<td>A sequence of requested or spontaneously changed ItemStates.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>transaction_id</td>
<td>The transaction number for the corresponding read.</td>
</tr>
<tr>
<td>all_quality_good</td>
<td>All item quality values are good. This requires that no errors are reported in the error parameter below.</td>
</tr>
<tr>
<td>item_states</td>
<td>A sequence of ItemStates matching the read operation.</td>
</tr>
<tr>
<td>errors</td>
<td>A sequence reporting items for which the read failed. An empty sequence indicates all read operations initially succeeded. Reported errors are:</td>
</tr>
<tr>
<td></td>
<td>• ERROR_BAD_RIGHTS (item is write only)</td>
</tr>
<tr>
<td></td>
<td>• ERROR_INVALID_DAIS_HANDLE</td>
</tr>
</tbody>
</table>

on_write_complete()

The method is used to report the success of an asynchronous write operation.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>transaction_id</td>
<td>The transaction number for the corresponding write.</td>
</tr>
<tr>
<td>errors</td>
<td>A sequence reporting items for which the write failed. An empty sequence indicates all write operations initially succeeded. Reported errors are:</td>
</tr>
<tr>
<td></td>
<td>• ERROR_BAD_RIGHTS (item is read only)</td>
</tr>
<tr>
<td></td>
<td>• ERROR_INVALID_DAIS_HANDLE</td>
</tr>
<tr>
<td></td>
<td>• ERROR_CLAMPED</td>
</tr>
<tr>
<td></td>
<td>• ERROR_OUT_OF_RANGE</td>
</tr>
<tr>
<td></td>
<td>• ERROR_BAD_TYPE</td>
</tr>
</tbody>
</table>
on_cancel_complete()

The method is used to acknowledge the completion of a successfully initiated cancel call.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>transaction_id</td>
<td>The transaction number for the corresponding cancel.</td>
</tr>
</tbody>
</table>

ConnectionPoint

An interface used by the client to connect or disconnect a client callback object at the server.

callback

An attribute referencing the callback object.

In an implementation one get and one set method will implement the callback attribute. Due to limitation in the UML tool used to draw the diagrams, the attribute is represented by the two methods connect and disconnect.

4.2.6 DAISGroupEntry IDL

4.2.6.1 DAIS::DataAccess::GroupEntry Overview

A group has a collection of group entries. Each group entry associates the group with an item. An ItemID identifies a group entry within its group. The pathname of an item may be used as an alternative to the ItemID when the group entry is created (see Section 4.2.4, “DAISItem IDL,” on page 4-14).

In OPC the GroupEntry interface corresponds to the interface IOPCItemMgt.
4.2.6.2 IDL

//File: DAISGroupEntry.idl
ifndef DAIS_GROUP_ENTRY_IDL
#define DAIS_GROUP_ENTRY_IDL
#pragma prefix "omg.org"
#include <DAISCommon.idl>

module DAIS {
  module DataAccess {
    module GroupEntry {

      struct Description {
        ServerItemIdentification server_item_id;
        string access_path;
        ClientItemHandle client_handle;
        SimpleValueType requested_data_type;
        boolean active;
      };
      typedef sequence<Description> Descriptions;

      struct DetailedDescription {
        ItemID item;
        string pathname;
        string access_path;
        ServerItemHandle server_handle;
      };
    }
  }
}

Figure 4-13  DAIS data access group entry IDL in UML
typedef sequence<DetailedDescription> DetailedDescriptions;

struct Result {
  ServerItemHandle server_handle;
  ClientItemHandle client_handle;
  AccessRights access_rights;
  SimpleValueType canonical_data_type;
};
typedef sequence<Result> Results;

struct HandleAssociation {
  ServerItemHandle server_handle;
  ClientItemHandle client_handle;
};
typedef sequence<HandleAssociation> HandleAssociations;

struct DataTypeDescription {
  ServerItemHandle server_handle;
  SimpleValueType requested_data_type;
};
typedef sequence<DataTypeDescription> DataTypeDescriptions;

interface Iterator {
  boolean next_n (in unsigned long n, out DetailedDescriptions entries);
  void reset();
  Iterator clone();
  void destroy();
};

interface Manager {
  Results create_entries (in Descriptions entries, out ItemErrors errors);
  Results validate_entries (}
in Descriptions entries, 
out ItemErrors errors 
); 

ItemErrors remove_entries ( 
in ServerItemHandles server_handles 
); 

ItemErrors set_active_state ( 
in ServerItemHandles server_handles 
); 

ItemErrors set_inactive_state ( 
in ServerItemHandles server_handles 
); 

ItemErrors set_client_handles ( 
in HandleAssociations handleAssociations 
); 

ItemErrors set_data_types ( 
in DataTypeDescriptions descriptions 
); 

Iterator create_group_entry_iterator (); 
);));}); 
#endif // _DAIS_GROUP_ENTRY_IDL 

Description

The struct describes a group entry for an item. The client to configure new entries in a group uses it. It directly corresponds to the OPCItemDef struct.

<table>
<thead>
<tr>
<th>Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>server_item_id</td>
<td>The identification of the item.</td>
</tr>
<tr>
<td>access_path</td>
<td>The access path used by the server to connect to the device and sensor. An empty string as input tells the server to select the access path.</td>
</tr>
<tr>
<td>client_handle</td>
<td>The client provided handle to the item.</td>
</tr>
<tr>
<td>requested_data_type</td>
<td>The data type requested by the client for the value.</td>
</tr>
<tr>
<td>active</td>
<td>Tells if the item is active and data from devices is updated in the cache.</td>
</tr>
</tbody>
</table>
**DetailedDescription**

The struct is used to deliver group entry information to the client. In OPC this is made with the struct OPCITEMATTRIBUTES. Description is used for both these OPC structs.

<table>
<thead>
<tr>
<th>Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>item</td>
<td>The item identification by an ItemID.</td>
</tr>
<tr>
<td>pathname</td>
<td>A string concatenating the labels for all nodes in the path from the item up to the root.</td>
</tr>
<tr>
<td>access_path</td>
<td>The access path used by the server to connect to the device and sensor. An empty string as input tells the server to select the access path.</td>
</tr>
<tr>
<td>server_handle</td>
<td>Server provided handle for the item.</td>
</tr>
<tr>
<td>client_handle</td>
<td>Client provided handle for the item.</td>
</tr>
<tr>
<td>requested_data_type</td>
<td>The data type requested by the client for the value.</td>
</tr>
<tr>
<td>canonical_data_type</td>
<td>The data type the server uses internally for the value.</td>
</tr>
<tr>
<td>access_rights</td>
<td>The access rights (read, write, and read-write)</td>
</tr>
<tr>
<td>active</td>
<td>Tells if the item is active and data from devices is updated in the cache.</td>
</tr>
</tbody>
</table>

**Result**

The struct is used to transfer the result from a create_entries() or validate_entries() call back to a client. In OPC the corresponding struct is OPCITEMRESULT.

<table>
<thead>
<tr>
<th>Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>server_handle</td>
<td>The handle given to the client for the item.</td>
</tr>
<tr>
<td>client_handle</td>
<td>The supplied client handle for the item.</td>
</tr>
<tr>
<td>access_rights</td>
<td>The access rights (read, write, and read-write)</td>
</tr>
<tr>
<td>canonical_data_type</td>
<td>The data type the server uses internally for the value.</td>
</tr>
</tbody>
</table>

**HandleAssociation**

The struct is used to change the association between server handles and client handles in the set_client_handle() method.

<table>
<thead>
<tr>
<th>Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>server_handle</td>
<td>The handle given to the client for the item.</td>
</tr>
<tr>
<td>client_handle</td>
<td>The new client handle wanted by the client.</td>
</tr>
</tbody>
</table>
**DataTypeDescription**

The struct is used to change the data type of the value delivered in the ItemState struct.

<table>
<thead>
<tr>
<th>Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>server_handle</td>
<td>The handle given to the client for the item.</td>
</tr>
<tr>
<td>requested_data_type</td>
<td>The new data type wanted by the client.</td>
</tr>
</tbody>
</table>

**Iterator**

Refer to Section 3.1.6, “Iterator Methods IDL,” on page 3-10. The interface directly corresponds to the EnumOPCItemAttributes interface.

**Manager**

An interface for creation and browsing of group entries. The interface directly corresponds to the IOPCItemMgt interface.

**create_entries()**

Adds one or more entries to a group.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>entries</td>
<td>Group entry descriptions for entries to be created.</td>
</tr>
<tr>
<td>errors</td>
<td>A sequence of structs reporting the items that were not entered due to an error. Reported errors are:</td>
</tr>
<tr>
<td></td>
<td>• ERROR_UNKNOWN_ITEMID</td>
</tr>
<tr>
<td></td>
<td>• ERROR_UNKNOWN_PATHNAME</td>
</tr>
<tr>
<td></td>
<td>• ERROR_BAD_TYPE</td>
</tr>
<tr>
<td></td>
<td>• ERROR_UNKNOWN_ACCESS_PATH</td>
</tr>
<tr>
<td>return</td>
<td>A sequence of result descriptions for the entries that were entered in the group.</td>
</tr>
</tbody>
</table>

**validate_entries()**

Is used to determine if an item is valid (could it be added without error). Also returns information about the item such as canonical datatype. Does not affect the group in any way.
### remove_entries()

Used to remove entries from a group.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>entries</td>
<td>Group entry descriptions for entries to be validated.</td>
</tr>
<tr>
<td>errors</td>
<td>A sequence of structs reporting the items that could not be validated due to an error. Reported errors are:</td>
</tr>
<tr>
<td></td>
<td>• ERROR_UNKNOWN_ITEMID</td>
</tr>
<tr>
<td></td>
<td>• ERROR_UNKNOWN_PATHNAME</td>
</tr>
<tr>
<td></td>
<td>• ERROR_BAD_TYPE</td>
</tr>
<tr>
<td></td>
<td>• ERROR_UNKNOWN_ACCESS_PATH</td>
</tr>
<tr>
<td>return</td>
<td>A sequence of result descriptions for the entries that were validated.</td>
</tr>
</tbody>
</table>

### set_active_state()

Used to activate individual items in a group. Activate state means that the server acquires data from devices. Inactive state means that the server does not acquire any data. The group enable state control if acquired data shall be sent further to subscribers via on_data_change.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>server_handles</td>
<td>Server handles for items to activate.</td>
</tr>
<tr>
<td>return</td>
<td>A sequence of structs reporting the items that were not recognized or could not be activated due to an error.</td>
</tr>
<tr>
<td></td>
<td>Reported error(s):</td>
</tr>
<tr>
<td></td>
<td>• ERROR_UNKNOWN_ITEMID</td>
</tr>
</tbody>
</table>
set_inactive_state()

Used to deactivate individual items in a group.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>server_handles</td>
<td>Server handles for items to deactivate.</td>
</tr>
<tr>
<td>return</td>
<td>A sequence of structs reporting the items that were not recognized or could not be deactivated due to an error. Reported error(s):</td>
</tr>
<tr>
<td></td>
<td>• ERROR_UNKNOWN_ITEMID</td>
</tr>
</tbody>
</table>

set_client_handles()

Used to change the client handles for items.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle_associations</td>
<td>Change descriptions for the items where to change the client handles.</td>
</tr>
<tr>
<td>return</td>
<td>A sequence of structs reporting the items that were not recognized or could not be updated due to an error. Reported error(s):</td>
</tr>
<tr>
<td></td>
<td>• ERROR_UNKNOWN_ITEMID</td>
</tr>
</tbody>
</table>

set_data_types()

Used to change the requested data types for items.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>descriptions</td>
<td>Change descriptions for the items where to change the data types.</td>
</tr>
<tr>
<td>return</td>
<td>A sequence of structs reporting the items that were not recognized or could not be updated due to an error. Reported errors are:</td>
</tr>
<tr>
<td></td>
<td>• ERROR_UNKNOWN_ITEMID</td>
</tr>
<tr>
<td></td>
<td>• ERROR_BAD_TYPE</td>
</tr>
</tbody>
</table>

create_group_entry_iterator()

Used to create a group entry iterator. Used by clients to inspect existing group entries.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>return</td>
<td>The GroupEntry Iterator.</td>
</tr>
</tbody>
</table>
4.2.7 DAISGroup IDL

4.2.7.1 DAIS::DataAccess::Group Overview

A group is a collection of items and a connection to one or more consumers of item values. Clients create groups and their lifetime is bounded by the session to which they belong. (See Section 4.2.2, “DAISDASession IDL,” on page 4-8).

The purpose of a group is to convey selected item values to a client. A callback object may be connected to a group to receive item value information (see Section 4.2.5, “DAISDAIO IDL,” on page 4-22). Items may be added and removed from a group as group entries (see Section 4.2.6, “DAISGroupEntry IDL,” on page 4-31). A group has an update rate that determines how frequently updated values for its entries are notified to its connected callback objects. A group also has other states that control its notification behavior.

A group may also be initialized with a predefined set of entries. A set of entries is called a public group and is identified by a ResourceID. A client can create or remove public groups. A server may represent a public group as a node such that the ResourceID of the public group and the node are identical. This would allow clients to locate public groups by name.

The DAIS::DataAccess::Group::Manager object implements interfaces from the DAISDAIO and DAISGroupEntry IDLs. This is specified by inheritance of interfaces as seen in Figure 4-14. A DAIS::DataAccess::Group::Manager has a state given by the DAIS::DataAccess::Group::State struct and a DAIS::DataAccess::Group::Manager object is created from the DAIS::DataAccess::Group::IHome object.

In OPC the Group interface corresponds to the interfaces IOPCGroupStateMgt and IOPCPublicGroupStateMgt.
4.2.7.2  IDL

//File: DAISGroup.idl
#ifndef _DAIS_GROUP_IDL
#define _DAIS_GROUP_IDL
#pragma prefix "omg.org"

DAIS::DataAccess::IO::SyncIO
(from DAIS::DataAccess::IO)
<<Interface>>
sync_read()
sync_write()

DAIS::DataAccess::IO::AsyncIO
(from DAIS::DataAccess::IO)
<<Interface>>
async_read()
async_write()
refresh()
cancel()
set_enable()
get_enable()

DAIS::DataAccess::Group::State
update_rate : unsigned long
active : boolean
time_bias : long
percent_dead_band : double
locale_id : unsigned long
name : string

DAIS::DataAccess::Group::Iterator
(from DAIS::DataAccess::Group)
<<Interface>>

DAIS::DataAccess::Group::Manager
(from DAIS::DataAccess::Group)
<<Interface>>
create_group()
#include <DAISCommon.idl>
#include <DAISGroupEntry.idl>
#include <DAISDAIO.idl>

module DAIS {
    module DataAccess {
        module Group {

            exception DuplicateName {string reason;};

            struct State {
                string name;
                unsigned long update_rate;
                boolean active;
                long time_bias;
                double percent_deadband;
                unsigned long locale_id;
            };

            struct PublicGroupDescription {
                ResourceID id;
                State group_state;
            };

            typedef sequence<PublicGroupDescription> PublicGroupDescriptions;

            interface Manager : GroupEntry::Manager, IO::AsyncIO, IO::SyncIO, IO::ConnectionPoint {
                State get_state ();

                unsigned long set_state (in State group_state) raises (DuplicateName);

                Manager clone (in string name) raises (DuplicateName);

                PublicGroupDescription clone_group_to_public (in string name) raises (DuplicateName);

                void destroy ();
            };
        }
    }
}

DAIS, v1.1: API
interface IHome
{
    exception UnknownResourceID {string reason;};

    PublicGroupDescriptions find_public_groups();

    PublicGroupDescription find (
        in ResourceID public_group
    ) raises (UnknownResourceID);

    Manager create_group ( 
        in State group_state,
        out unsigned long revised_update_rate
    ) raises (DuplicateName);

    Manager clone_group_from_public ( 
        in ResourceID public_group,
        in string name
    ) raises (DuplicateName, UnknownResourceID);

    void remove_public_group ( 
        in ResourceID public_group
    ) raises (UnknownResourceID);
};};};
#endif // _DAIS_GROUP_IDL

DuplicateName

An exception raised when an object is created and the name already exists. No object is created if the exception is raised. Used for session and group manager objects.

State

The struct contains information about the group state.

<table>
<thead>
<tr>
<th>Members</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Within the session and public groups, unique name of the group.</td>
</tr>
<tr>
<td>update_rate</td>
<td>Update rate for the group in milliseconds. When used as input it specifies</td>
</tr>
<tr>
<td></td>
<td>the fastest rate at which data changes may be sent to on_data_change() for</td>
</tr>
<tr>
<td></td>
<td>items in this group. This also indicates the desired accuracy of cached</td>
</tr>
<tr>
<td></td>
<td>data. This is intended only to control the behavior at the interface. How</td>
</tr>
<tr>
<td></td>
<td>the server deals with the update rate and how often it actually polls the</td>
</tr>
<tr>
<td></td>
<td>hardware internally is an implementation detail. Passing 0 indicates the</td>
</tr>
<tr>
<td></td>
<td>server should use the fastest practical rate.</td>
</tr>
</tbody>
</table>
Manager

An object used to manage a group. It has a set of methods related to the group itself. It also inherits methods from interfaces for group entry management and data transfer. For group entry management refer to Section 4.2.6, “DAISGroupEntry IDL,” on page 4-31 and for data transfer refer to Section 4.2.4, “DAISItem IDL,” on page 4-14. The DAIS::DataAccess::Group::Manager interface corresponds to the IOPCGroupStateMgt interface.

get_state()

The method gets the group status.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>return</td>
<td>State</td>
</tr>
</tbody>
</table>

set_state()

The method sets the group status.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>group_state</td>
<td>The State with the updates. All members will be updated. If the name already</td>
</tr>
<tr>
<td></td>
<td>exists, a DuplicateName exception is raised and no update is made.</td>
</tr>
<tr>
<td>return</td>
<td>The closest update rate the server is able to provide for the group.</td>
</tr>
</tbody>
</table>
clone()
Create a copy of a group.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>The name to be given for the new group. If the name already exists, a DuplicateName exception is raised and no clone is created.</td>
</tr>
<tr>
<td>return</td>
<td>A description of the public group.</td>
</tr>
</tbody>
</table>

clone_group_to_public()
Create a public copy of a group including all items and the group state.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>The name to be given for the new group. If the name already exists, a DuplicateName exception is raised and no public group is created.</td>
</tr>
<tr>
<td>return</td>
<td>A description of the public group.</td>
</tr>
</tbody>
</table>

destroy()
Delete the group.

PublicGroupDescription
A struct describing public groups.

<table>
<thead>
<tr>
<th>Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>A ResourceID identifying the public group.</td>
</tr>
<tr>
<td>group_state</td>
<td>The group state struct including the group name.</td>
</tr>
</tbody>
</table>

IHome
The factory object for groups. The corresponding OPC interface is IOPCServer.

UnknownResourceID
An exception telling that the ResourceID is unknown. For methods taking a sequence of resource ids the first found unknown id is reported. The likely reason behind this exception is some misunderstanding between the server and client code due to a programming error.
find_public_groups()

Find all public groups defined in the server.

<table>
<thead>
<tr>
<th>Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>return</td>
<td>A sequence of public group descriptions.</td>
</tr>
</tbody>
</table>

find()

For a given public group, return information about that group.

<table>
<thead>
<tr>
<th>Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>public_group</td>
<td>A ResourceID identifying the public group.</td>
</tr>
<tr>
<td>return</td>
<td>A public group description.</td>
</tr>
</tbody>
</table>

create_group()

Create a new initially empty group.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>group_state</td>
<td>The State to be set for the new group.</td>
</tr>
<tr>
<td>revised_update_rate</td>
<td>The closest update rate the server is able to provide for the group.</td>
</tr>
<tr>
<td>return</td>
<td>The new group.</td>
</tr>
</tbody>
</table>

clone_group_from_public()

Create a copy from a public group having an existing set of entries and state.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>public_group</td>
<td>The identification of the public group.</td>
</tr>
<tr>
<td>name</td>
<td>The name of the new group.</td>
</tr>
<tr>
<td>return</td>
<td>The new group.</td>
</tr>
</tbody>
</table>

remove_public_group()

Remove a public group.
### Group management

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>public_group</td>
<td>The identification of the public group.</td>
</tr>
<tr>
<td>return</td>
<td>None.</td>
</tr>
</tbody>
</table>

---

*Figure 4-15  Group management interaction*
Activate subscription

: Client  : DAIS::DataAccess::Group::Manager  : DAIS::DataAccess::IO::Callback

callback(DAIS::Callback)

refresh()

on_data_change()

Forced by refresh

Spontaneous

on_data_change()

Spontaneous

Figure 4-16  Active subscription interaction
Activate a subscription silently

Figure 4-17  Activate a subscription silently interaction
4.2.8 DAISDASimpleIO IDL

4.2.8.1 DAIS::DataAccess::SimpleIO Overview

The purpose of the interface is to provide a simple read and write of data. The functionality is the same as the synchronous read and write described in Section 4.2.5, “DAISDAIO IDL,” on page 4-22 but without having to create a group.

4.2.8.2 IDL

```c++
//File: DAISDASimpleIO.idl
#ifndef _DAIS_DA_SIMPLE_IO_IDL
#define _DAIS_DA_SIMPLE_IO_IDL
#include <DAISCommon.idl>

module DAIS {
  module DataAccess {
    module SimpleIO {
```

---

**Figure 4-18**  Cancellation interaction
enum DataSource {
    DS_CACHE,
    DS_DEVICE
};

struct ItemError
{
    Error err;
    ServerItemIdentification id;
    string reason;
};
typedef sequence<ItemError> ItemErrors;

struct ItemState
{
    SimpleValue value;
    DateTime time_stamp;
    Quality dais_quality;
    ServerItemIdentification id;
};
typedef sequence<ItemState> ItemStates;

struct ItemUpdate
{
    ServerItemIdentification id;
    SimpleValue value;
};
typedef sequence<ItemUpdate> ItemUpdates;

interface IHome
{
    ItemStates read (in DataSource data_source,
                     in ServerItemIdentifications ids,
                     out ItemErrors errors);

    ItemErrors write_with_qt (in ItemStates updates);

    ItemErrors write (in ItemUpdates updates);
};}}}}};
#endif // _DAIS_DA_SIMPLE_IO_IDL

Data Acquisition from Industrial Systems, v1.1  June 2005
**DataSource**

Find all public groups defined in the server.

<table>
<thead>
<tr>
<th>Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DS_CACHE</td>
<td>Data cached in the server is requested.</td>
</tr>
<tr>
<td>DS_DEVICE</td>
<td>Data from the device is requested. This will force a read from the device or RTU. A read from device will be made regardless of the group or item active status and no group NotActive exception will be forced.</td>
</tr>
</tbody>
</table>

**ItemError**

A struct for reporting of item related errors. The struct is different from the DAIS::ItemError because no handles are used to identify the item in SimpleIO.

<table>
<thead>
<tr>
<th>Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>err</td>
<td>An error code as described for DAIS::ItemError.</td>
</tr>
<tr>
<td>id</td>
<td>The identification of the item.</td>
</tr>
<tr>
<td>reason</td>
<td>An additional text explaining the error.</td>
</tr>
</tbody>
</table>

**ItemState**

The struct is the carrier of data conveyed over the interface. It is the “message” holding the payload. This is basically the same struct as DataAccess::IO::ItemState with the difference that instead of a handle for identification the full server identification is used.

<table>
<thead>
<tr>
<th>Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>The value itself.</td>
</tr>
<tr>
<td>time_stamp</td>
<td>The time stamp when the value was last updated.</td>
</tr>
<tr>
<td>quality</td>
<td>The quality for the value.</td>
</tr>
<tr>
<td>id</td>
<td>The identification of the value in the server.</td>
</tr>
</tbody>
</table>

**ItemUpdate**

The struct carries an update for an item.

<table>
<thead>
<tr>
<th>Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>The identification of the value in the server.</td>
</tr>
<tr>
<td>value</td>
<td>The value that shall be used in the update.</td>
</tr>
</tbody>
</table>
IHome

The interface for simple IO operations.

read

Synchronous read of items. Inactive items will be reported with OPCQuality set to OPC_QUALITY_OUT_OF_SERVICE.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>data_source</td>
<td>The source from where to read the data, see DataSource above.</td>
</tr>
<tr>
<td>ids</td>
<td>A sequence specifying the server identifications to read.</td>
</tr>
</tbody>
</table>
| errors      | A sequence reporting items for which the read failed. An empty sequence indicates all read operations succeeded. Reported errors are:  
  • ERROR_BAD_RIGHTS (item is write only)  
  • ERROR_UNKNOWN_PATHNAME  
  • ERROR_UNKNOWN_ITEMID |
| return      | A sequence of ItemStates for the items.                                      |

write_with_qt

Synchronous write of item values including quality and time stamp. The active state of the group or the items is ignored.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>updates</td>
<td>A sequence of ItemStates specifying the updates including time stamp and quality.</td>
</tr>
</tbody>
</table>
| return      | A sequence reporting items for which the write failed. An empty sequence indicates write operations for all items succeeded. Reported errors are:  
  • ERROR_BAD_RIGHTS (item is read only)  
  • ERROR_UNKNOWN_PATHNAME  
  • ERROR_UNKNOWN_ITEMID  
  • ERROR_CLAMPED  
  • ERROR_OUT_OF_RANGE  
  • ERROR_BAD_TYPE |
write

Synchronous write of item values only. The active state of the group or the items is ignored.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>updates</td>
<td>A sequence of ItemUpdates specifying the update values.</td>
</tr>
<tr>
<td>return</td>
<td>A sequence reporting items for which the write failed. An empty sequence indicates write operations for all items succeeded. Reported errors are:</td>
</tr>
<tr>
<td></td>
<td>• ERROR_BAD_RIGHTS (item is read only)</td>
</tr>
<tr>
<td></td>
<td>• ERROR_UNKNOWN_PATHNAME</td>
</tr>
<tr>
<td></td>
<td>• ERROR_UNKNOWN_ITEMID</td>
</tr>
<tr>
<td></td>
<td>• ERROR_CLAMPED</td>
</tr>
<tr>
<td></td>
<td>• ERROR_OUT_OF_RANGE</td>
</tr>
<tr>
<td></td>
<td>• ERROR_BAD_TYPE.</td>
</tr>
</tbody>
</table>
The alarms & events interface provides a client with a way to subscribe for alarms and events generated within RTUs, devices, or any software. The server supports various filter functions so that the client can compose a filter matching its current interest in alarms and events. Once the client sets up a filter specification it has to supply the server a callback object used by the server to notify the client with generated alarms and events.

Events just reports state changes while alarms have an associated alarm state and fault state. Alarms are generated with the intent to gain attention to a condition that needs operator intervention. Hence alarms and events are recorded and presented to operators. Alarm presentation usually involves acoustic annunciation and some highlighting (e.g., red color and/or blink, etc.). An operator shall acknowledge the alarm state and the fault state disappear when the fault causing the alarm disappears.

Equipment and functions that may generate alarms and events are:

- Process instrumentation making sensor data and actuation capabilities available.
- Remote terminal units (RTUs) or substation control systems, reading sensor data, and controlling actuators.
- Process communication units connecting to RTUs or substation control systems.
- SCADA subsystem making processed sensor data and control capabilities available to operators, applications, or other systems.
• Energy Management System (EMS) - subsystem using the SCADA subsystem for extended processing and control.

An example flow of alarms, events, and acknowledgments are shown in Figure 5-1.

![Figure 5-1](alarming.png)

**Figure 5-1**  Alarms, events and acknowledgment flows in a SCADA/EMS

### 5.1 Information Model

Alarms and events are generated by a source represented by Source. A source might be a single measurement, a collection of measurements, or some other object. A collection of measurements may represent a complex real world object like a generator or some control function for the generator. A source has a name property. The source is also associated with a type that describes any additional properties. If a DAIS server implements both data access and alarms & events, a client shall expect there is a mapping between sources and nodes. A client shall however not make assumptions on how the mapping is made and use the method translate_to_item_ids() to get the mapping from the server.

Sources are organized in areas represented by Area. An area typically represents area of responsibility concerning supervision and operation of sources. Areas may however also be used for other groupings of sources. Areas can be hierarchically structured to allow creation of hierarchically organized responsibilities. Multiple views of areas are supported.

Alarms and events are categorized. Category represents the categories. Three main categories are defined and must be implemented by a server:

- simple - describing events that do not have an explicitly modeled condition space.
• tracking - describing events generated due to an operator action.
• condition - describing events generated based on an explicitly modeled condition space.

The categories simple and tracking are used for events, and the category condition is used for alarms.

For each main category it is possible to define a number of sub-categories. A server is free to implement any sub-categories. Each type of source is expected to be associated with at least one main category and one or more sub-categories for each main category. The main categories are not expected to have an association with a source type. Sub-categories however are all expected to have an associated source type.

A Condition category is associated with one or more condition spaces. ConditionSpace represents a condition space. Depending on how limits are applied to the properties defined by the type associated with a condition category it is possible to create a space consisting of different discrete conditions. Condition describes each discrete condition. A condition space will have a number of conditions defined for it. A Transition describes each possible transition between a pair of conditions. The alarms & event session does not however provide any methods for direct access of transitions. When a transition is traversed a condition event is generated.

A source may be associated with one or more condition spaces. Each such association has status information describing the current condition. The association and its data are called SourceCondition. The source condition is identified by its associated source and condition space.

A source is of a specific type (e.g., measurement, breaker, generator, tank, etc.) and the type has one or more properties in the same way as a node (refer to Section 4.1.1, “Nodes, Items, Types, and Properties,” on page 4-2). A category may have any number of properties associated with it. An alarm or event may contain values for the properties associated with the corresponding category. This is used to convey additional server specific information with alarms and events. Specification of what properties are associated with categories is outside the scope of this specification.

The described classes are shown in Figure 5-2. The attributes are described later with the description of the interfaces.
Figure 5-2 DAIS Alarms and Events Information Model
The objects in DAIS alarms and events has the following mapping to OPC alarms and events objects.

<table>
<thead>
<tr>
<th>DAIS A&amp;E object</th>
<th>OPC A&amp;E object</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source</td>
<td>Source</td>
</tr>
<tr>
<td>Area</td>
<td>Area</td>
</tr>
<tr>
<td>Property</td>
<td>EventAttribute</td>
</tr>
<tr>
<td>Category</td>
<td>EventCategory</td>
</tr>
<tr>
<td>The categories; simple, tracking and condition</td>
<td>EventType</td>
</tr>
<tr>
<td>ConditionSpace</td>
<td>Condition</td>
</tr>
<tr>
<td>Condition</td>
<td>SubCondition</td>
</tr>
<tr>
<td>SourceCondition</td>
<td>SourceCondition</td>
</tr>
</tbody>
</table>

### 5.1.1 OPC Recommended Properties

In OPC there is a recommendation of what properties are expected to be supported by an alarm & event server. The recommended properties are shown in Table 5-1. The column names correspond to the attributes in the Property class.

**Table 5-1**  Recommended Properties

<table>
<thead>
<tr>
<th>label</th>
<th>id</th>
<th>canonical_data_type</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Condition Status</td>
<td>300</td>
<td>STRING_TYPE</td>
<td>The current alarm or condition status associated with the Item (for example, “NORMAL,” “ACTIVE,” “HI ALARM”).</td>
</tr>
<tr>
<td>Alarm Quick Help</td>
<td>301</td>
<td>STRING_TYPE</td>
<td>A short text string providing a brief set of instructions for the operator to follow when this alarm occurs.</td>
</tr>
<tr>
<td>Alarm Area List</td>
<td>302</td>
<td>STRING_TYPE sequence</td>
<td>An array of strings indicating the plant or alarm areas that include this ItemID.</td>
</tr>
<tr>
<td>Primary Alarm Area</td>
<td>303</td>
<td>STRING_TYPE</td>
<td>A string indicating the primary plant or alarm area including this ItemID.</td>
</tr>
<tr>
<td>Condition Logic</td>
<td>304</td>
<td>STRING_TYPE</td>
<td>An arbitrary string describing the test being performed (for example, “High Limit Exceeded” or “TAG.PV &gt;= TAG.HILIM”). Refer to Section 5.2.7.3, “Condition Logic,” on page 5-38.</td>
</tr>
</tbody>
</table>
5

5.2 API

5.2.1 Alarms & Events IDL Overview

The dependencies among the different IDL files are shown in Figure 5-3.

<table>
<thead>
<tr>
<th>Limit</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Limit Exceeded</td>
<td>STRING</td>
<td>For multistate alarms, the condition exceeded (for example, HIHI, HI, LO, LOLO).</td>
</tr>
<tr>
<td>Deadband</td>
<td>DOUBLE</td>
<td>Deadband</td>
</tr>
<tr>
<td>HiHi Limit</td>
<td>DOUBLE</td>
<td>HiHi Limit</td>
</tr>
<tr>
<td>Hi Limit</td>
<td>DOUBLE</td>
<td>Hi Limit</td>
</tr>
<tr>
<td>Lo Limit</td>
<td>DOUBLE</td>
<td>Lo Limit</td>
</tr>
<tr>
<td>LoLo Limit</td>
<td>DOUBLE</td>
<td>LoLo Limit</td>
</tr>
<tr>
<td>Rate of Change Limit</td>
<td>DOUBLE</td>
<td>Rate of Change Limit</td>
</tr>
<tr>
<td>Deviation Limit</td>
<td>DOUBLE</td>
<td>Deviation Limit</td>
</tr>
<tr>
<td></td>
<td>312-4999</td>
<td>Reserved by OPC</td>
</tr>
</tbody>
</table>
5.2.2 Alarms and Events Common IDL Definitions

5.2.2.1 IDL

```c
// File: DAISAECCommon.idl
#ifndef _DAIS_AECOMMON_IDL
#define _DAIS_AECOMMON_IDL
#pragma prefix "omg.org"
#include <DAISCommon.idl>

module DAIS {
    module AlarmsAndEvents {
        typedef ResourceID EventID;
        struct ResourceError {
            Error err;
            ResourceID id;
        }
    }
}
```

Figure 5-3 Dependencies between alarms and events IDL files
string reason;
);
typedef sequence<ResourceError> ResourceErrors;

// error codes
const Error RES_ERROR_DAISOK = 0;
const Error RES_ERROR_UNKNOWNRESOURCE = 1;

typedef unsigned long SourceConditionOpState;
const SourceConditionOpState CONDITION_ENABLED = 0x0001;
const SourceConditionOpState CONDITION_ACTIVE = 0x0002;
const SourceConditionOpState CONDITION_ACKED = 0x0004;

typedef unsigned short EventFormat;
const EventFormat OPC_SIMPLE_EVENT = 0x0001;
const EventFormat OPC_TRACKING_EVENT = 0x0002;
const EventFormat OPC_CONDITION_EVENT = 0x0004;
const EventFormat OPC_ALL_EVENTS = 0x0007;
#endif // _DAIS_AECOMMON_IDL

EventID

A ResourceID uniquely identifying an event notification.

ResourceError

A struct for reporting of resource related errors.

<table>
<thead>
<tr>
<th>Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>err</td>
<td>An error code as described below.</td>
</tr>
<tr>
<td>id</td>
<td>The identification of the resource.</td>
</tr>
<tr>
<td>reason</td>
<td>An additional text explaining the error.</td>
</tr>
</tbody>
</table>

ResourceErrors

ResourceErrors is a sequence containing errors. If no errors are present, no entry shall be included in the sequence rather than including a large number of no errors. An empty sequence means no errors.
Error

The error codes for ResourceError.

<table>
<thead>
<tr>
<th>Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RES_ERROR_DAISOK</td>
<td>No error.</td>
</tr>
<tr>
<td>RES_ERROR_UNKNOWN_RESOURCE</td>
<td>The resource was not found.</td>
</tr>
</tbody>
</table>

SourceConditionOpState

Flag word holding for the operational state of a SourceCondition. The definitions of the state variable in the flag word are listed below.

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONDITION_ENABLED</td>
<td>The Condition is enabled and supervision is active.</td>
</tr>
<tr>
<td>CONDITION_ACTIVE</td>
<td>The Condition is active; that is, the supervision has determined that a fault has activated the condition.</td>
</tr>
<tr>
<td>CONDITION_ACKED</td>
<td>The Condition alarm has been acknowledged.</td>
</tr>
</tbody>
</table>

The combinations of the state variables result in eight states. The valid SourceCondition operational states are listed below.

<table>
<thead>
<tr>
<th>State</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disabled</td>
<td>Not supervised by server.</td>
</tr>
<tr>
<td>Enabled, Inactive, Acedd</td>
<td>Supervised by server, no fault detected and all alarms are acknowledged.</td>
</tr>
<tr>
<td>Enabled, Inactive, Unacked</td>
<td>Supervised by server, no fault detected and unacknowledged alarms exist.</td>
</tr>
<tr>
<td>Enabled, Active, Unacked</td>
<td>Supervised by server, a fault is detected and unacknowledged alarms exist.</td>
</tr>
<tr>
<td>Enabled, Active, Acedd</td>
<td>Supervised by server, a fault is persistent and all alarms are acknowledged.</td>
</tr>
</tbody>
</table>

When enabled the state {Enabled, Inactive, Acked} is entered and from there the supervision will generate the appropriate state depending on the result of the supervision. Each state change results in sending an alarm and event notification. All notifications contain the state.
EventFormat

This constant tells the format of an event.

<table>
<thead>
<tr>
<th>Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OPC_SIMPLE_EVENT</td>
<td>The event is a simple event.</td>
</tr>
<tr>
<td>OPC_TRACKING_EVENT</td>
<td>The event is a tracking event.</td>
</tr>
<tr>
<td>OPC_CONDITION_EVENT</td>
<td>The event is a condition event.</td>
</tr>
<tr>
<td>OPC_ALL_EVENTS</td>
<td>This constant value is used to ask for all event formats in a subscription set up.</td>
</tr>
</tbody>
</table>

5.2.3 DAISAE SESion IDL

5.2.3.1 DAIS::AlarmsAndEvents::Session Objects Overview

The DAIS::AlarmsAndEvents::Session object implements the alarms & events service on a per client basis. An alarm & event session object has a number of services provided by one singleton home object each. Each home object provides methods for manipulation of the data of the specific type they provide.

The session object corresponds to an OPCEventServer object.
Figure 5-4  DAIS alarms and events session IDL in UML

5.2.3.2  IDL

//File: DAISAESession.idl
#ifndef _DAIS_AESERVER_IDL
#define _DAIS_AESERVER_IDL
#define _DAIS_AESERVER_IDL
#pragma prefix "omg.org"
// Common Information
#include <DAISType.idl>
#include <DAISProperty.idl>
#include <DAISSession.idl>

// Events and Alarms
#include <DAISAESubscription.idl>
#include <DAISAEArea.idl>
#include <DAISAESource.idl>
#include <DAISAEConditionSpace.idl>
#include <DAISAESourceCondition.idl>
#include <DAISAECategory.idl>
#include <DAISAEIO.idl>

module DAIS {  
module AlarmsAndEvents {  

interface Session : DAIS::Session {  
  readonly attribute Subscription::IHome subscription_home;  
  readonly attribute Area::IHome area_home;  
  readonly attribute Source::IHome source_home;  
  readonly attribute ConditionSpace::IHome condition_space_home;  
  readonly attribute SourceCondition::IHome source_condition_home;  
  readonly attribute Category::IHome category_home;  
  readonly attribute Type::IHome type_home;  
  readonly attribute Property::IHome property_home;  
  };};
#endif // _DAIS_AESESSION_IDL;

Session

Session is an object implementing the alarms & events functions. It inherits common functionality as shut down callbacks and session status from DAIS::AlarmsAndEvents::Session.

subscription_home

A read only attribute holding a reference to a singleton Subscription::IHome object.
area_home
A read only attribute holding a reference to a singleton Area::IHome object.

source_home
A read only attribute holding a reference to a singleton Source::IHome object.

condition_space_home
A read only attribute holding a reference to a singleton ConditionSpace::IHome object.

source_condition_home
A read only attribute holding a reference to a singleton SourceCondition::IHome object.

category_home
A read only attribute holding a reference to a singleton Category::IHome object.

type_home
A read only attribute holding a reference to a singleton Type::IHome object.

property_home
A read only attribute holding a reference to a singleton Property::IHome object.

5.2.4 DAISAESESubscription IDL

5.2.4.1 DAIS::AlarmsAndEvents::Subscription Overview

A DAIS::AlarmsAndEvents::Subscription::Manager is an object holding a filter specification set up by a client. The filter is used to specify what notifications shall be sent to the client. A server can support various filter functions and a client can ask the DAIS::AlarmsAndEvents::Subscription::IHome object what filter functions are supported. The subscription home is also used to create any number of subscription manager objects. Each subscription manager shall be associated with a client implemented callback object so that the server can send alarm and event notifications to the client.

A server may optionally support an event history. The event history shall record all alarms and events appearing in a server. The method async_read_history() gives clients access to the event history. The size of the event history is server specific and outside the control of a client.

In OPC the Subscription interface corresponds to the interface IOPCEventSubscription.
5.2.4.2 IDL

//File: DAISAESubscription.idl
#ifndef _DAIS_AESUBSCRIPTION_IDL
#define _DAIS_AESUBSCRIPTION_IDL
#define _DAIS_AESUBSCRIPTION_IDL
#pragma prefix "omg.org"
#include <DAISAEIO.idl>

module DAIS {

DAIS::AlarmsAndEvents::Subscription::SubscriptionSpec

reason_filter : ReasonFilter
low_severity : unsigned long
high_severity : unsigned long
source_filter : SourceFilter
type_filter : TypeIDs

DAIS::AlarmsAndEvents::Subscription::Subscription::State
active : boolean
buffer_time : unsigned long
max_size : unsigned long
keep_alive_time : unsigned long

DAIS::AlarmsAndEvents::Subscription::Subscription::Manager
callback() : DAIS::AlarmsAndEvents::IO::Callback
callback(in callback : DAIS::AlarmsAndEvents::IO::Callback)
set_filter()
get_filter()
select_returned_properties()
get_returned_properties()
refresh()
synch_read_history()
Cancel()
get_state()
set_state()
clone()
destroy()

DAIS::AlarmsAndEvents::Subscription::Subscription::IHome
query_available_filters()
create_subscription()

DAIS::AlarmsAndEvents::Subscription::IO::Callback
0..1
1
0..1
1
0..n
1
0..n
1
1
1
1
1
1
1
1
1
1
1

Figure 5-5  DAIS alarms and events subscription IDL in UML
module AlarmsAndEvents {
    module Subscription {

        struct State {
            boolean active;
            unsigned long buffer_time;
            unsigned long max_size;
            unsigned long keep_alive_time;
        };

        struct OPCSourceFilter {
            ServerItemIdentifications areas;
            ServerItemIdentifications sources;
        };

        typedef short SourceFilterType;
        const SourceFilterType OPC_SOURCE_FILTER_TYPE = 1;
        const SourceFilterType XPATH_SOURCE_FILTER_TYPE = 2;

        union SourceFilter switch(SourceFilterType) {
            case OPC_SOURCE_FILTER_TYPE : OPCSourceFilter opc_source_filters;
            case XPATH_SOURCE_FILTER_TYPE : string xpath_source_filter;
        };

        typedef short CategoryFilterType;
        const CategoryFilterType OPC_REASON_FILTER_TYPE = 1;
        const CategoryFilterType XPATH_REASON_FILTER_TYPE = 2;

        union CategoryFilter switch(ReasonFilterType) {
            case OPC_CATEGORY_FILTER_TYPE : ServerItemIdentifications opc_category_filters;
            case XPATH_CATEGORY_FILTER_TYPE : string xpath_category_filter;
        };

        struct FilterSpec {
            EventFormat event_format;
            CategoryFilter category_filter;
            unsigned long low_severity;
            unsigned long high_severity;
            SourceFilter source_filter;
            ResourceIDs type_filter;
        };

        struct PropSelection {
            ResourceID category;
            PropertyIDs properties;
        };

        typedef sequence<PropSelection> PropSelections;
enum ReadDirection {
    READ_FORWARDS,
    READ_BAKWARDS
};

typedef unsigned long CancelID;

interface Manager
{
    exception BusyDueToRefresh{string reason;};
    exception HistoryNotImplemented{string reason;};

    attribute IO::Callback callback;

    void set_filter (in FilterSpec filter_spec)
        raises (BusyDueToRefresh);

    FilterSpec get_filter ();

    void select_returned_properties (in PropSelections prop_selections);

    PropSelections get_returned_properties (in ResourceIDs categories);

    CancelID refresh () raises (BusyDueToRefresh);

    CancelID async_read_history (in DateTime start_time,
        in unsigned long number_of_events,
        in ReadDirection direction,
        in unsigned long transaction_id)
        raises (HistoryNotImplemented);

    void cancel (in CancelID cancel_id);

    void refresh () raises (BusyDueToRefresh);

    void refresh_with_history (in DateTime start_time,
        in DateTime end_time)
        raises (HistoryNotImplemented);

    void cancel_refresh ();

    State get_state ();
State set_state (in State subscription_spec

Manager clone ();

void destroy ();
);

typedef unsigned long Filter;
const Filter FILTER_BY_EVENT_FORMAT = 0x0001;
const Filter FILTER_BY_CATEGORY = 0x0002;
const Filter FILTER_BY_SEVERITY = 0x0004;
const Filter FILTER_BY_AREA = 0x0008;
const Filter FILTER_BY_SOURCE = 0x0010;
const Filter FILTER_WITH_XPATH = 0x0020;
const Filter FILTER_BY_SOURCE_TYPE = 0x0040;

interface IHome
{
    Filter query_available_filters ();

    Manager create_subscription (in State subscription_spec,
                                    out State revised_subscr_spec);
};};
#endif // _DAIS_AESUBSCRIPTION_IDL

State

A struct describing the state for the subscription.

<table>
<thead>
<tr>
<th>Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>active</td>
<td>Indicates if the subscription is active; that is, is sending data on the call back interface. Events that are appearing in the server will be lost for clients connected to inactive subscriptions.</td>
</tr>
<tr>
<td>buffer_time</td>
<td>The requested buffer time in milliseconds. This is a minimum time - do not send event notifications any faster that this UNLESS max_size is greater than 0, in which case the server will send an event notification sooner to obey the max_size value. A value of 0 for buffer_time means that the server should send event notifications as soon as it gets them. This parameter along with the max_size parameter is 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 revised_buffer_time.</td>
</tr>
</tbody>
</table>
The struct specifies source filtering by enumerating the areas and sources the way OPC does it.

### OPCSourceFilter

The union specifies OPCSourceFiltering or XPath filtering. A prerequisite for XPath filtering is that the area and source hierarchy has an XML mapping. The same mapping is used as described in Section 3.1.12, “Logical Expressions and Navigation,” on page 3-25 with the additional comments:

- an Area corresponds to a Node (Area inherits Node).
- a Source corresponds to a Node (Source inherits Node).

<table>
<thead>
<tr>
<th>Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>areas</td>
<td>A sequence of ResourceIDs for the areas that shall be notified. All sources below an area node will be notified.</td>
</tr>
<tr>
<td>sources</td>
<td>A sequence of ResourceIDs for sources that shall be notified. An empty sequence means all sources shall be notified.</td>
</tr>
</tbody>
</table>

### SourceFilter

The union specifies OPCSourceFiltering or XPath filtering. A prerequisite for XPath filtering is that the area and source hierarchy has an XML mapping. The same mapping is used as described in Section 3.1.12, “Logical Expressions and Navigation,” on page 3-25 with the additional comments:

<table>
<thead>
<tr>
<th>Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc_source_filters</td>
<td>An OPCSourceFilter</td>
</tr>
<tr>
<td>xpath_source_filter</td>
<td>An XPath filter.</td>
</tr>
</tbody>
</table>
CategoryFilter

The union specifies OPC style filtering or XPath filtering of categories. A prerequisite for XPath filtering is that the resource hierarchy has an XML mapping. The same mapping is used as described in Section 3.1.12, “Logical Expressions and Navigation,” on page 3-25 with the additional comments:

- a Resource corresponds to a Node (Resource inherits Node).

<table>
<thead>
<tr>
<th>Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc_category_filters</td>
<td>An OPCSourceFilter</td>
</tr>
<tr>
<td>xpath_reason_filter</td>
<td>An XPath filter</td>
</tr>
</tbody>
</table>

FilterSpec

A struct holding the specification for how the server shall filter notifications sent to the client. The struct is used to specify the filtering a client wants a server to do for it. All specified filter criteria shall be fulfilled for a notification to be sent.

<table>
<thead>
<tr>
<th>Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>event_format</td>
<td>An EventFormat bit mask specifying the event formats to be returned.</td>
</tr>
<tr>
<td>reasons</td>
<td>A sequence of CategoryFilter for the categories for which the client wants to get notifications. Observe that specifying a main category (for example, Simple, Tracking, Condition) will result in notifications for all sub-categories specified for that main category.</td>
</tr>
<tr>
<td>low_severity</td>
<td>The client wants all events with greater or equal severity.</td>
</tr>
<tr>
<td>high_severity</td>
<td>The client wants all events with less or equal severity.</td>
</tr>
<tr>
<td>sources</td>
<td>A SourceFilter for the sources that shall be notified.</td>
</tr>
<tr>
<td>types</td>
<td>A sequence of TypeIDs for the types that shall be notified.</td>
</tr>
</tbody>
</table>

PropSelection

A struct identifying what properties shall be included in an event notification for a specific reason.

<table>
<thead>
<tr>
<th>Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>reason</td>
<td>The reason for which the property selection is made.</td>
</tr>
<tr>
<td>properties</td>
<td>A sequence of PropertyIDs for the selected properties.</td>
</tr>
</tbody>
</table>
**ReadDirection**

An enumeration that tells if a read of history shall be forward or backwards in time.

<table>
<thead>
<tr>
<th>Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ_FORWARDS</td>
<td>Read events after the given start time.</td>
</tr>
<tr>
<td>READ_BAKWARDS</td>
<td>Read events before the given start time.</td>
</tr>
</tbody>
</table>

**Manager**

An object managing subscription and filtering of event notifications per client.

**BusyDueToRefresh**

An exception stating that a refresh is already going on.

**HistoryNotImplemented**

An exception stating that the server does not record the event history.

**callback**

An attribute holding a reference to the callback object. The client is required to update the attribute with a by the client implemented callback object to get event notifications from the server. The attribute corresponds to the IConnectionPoint interface in OPC.

**set_filter()**

The method updates the filter specification. A try to change the filter spec during an ongoing refresh will give an exception. The method corresponds to IOPCEventSubscriptionMgt::SetFilter().

<table>
<thead>
<tr>
<th>Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>filer_spec</td>
<td>The filter specification.</td>
</tr>
<tr>
<td>return</td>
<td>none.</td>
</tr>
</tbody>
</table>

**get_filter()**

The method gets the filter specification. The method corresponds to IOPCEventSubscriptionMgt::GetFilter().

<table>
<thead>
<tr>
<th>Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>return</td>
<td>The filter specification.</td>
</tr>
</tbody>
</table>
**select_returned_properties()**

The method sets what properties shall be included in a notification specified per reason. The method corresponds to IOPCEventSubscriptionMgt::SelectReturnedAttributes().

<table>
<thead>
<tr>
<th>Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>prop_selections</td>
<td>A sequence specifying what properties to include per reason.</td>
</tr>
<tr>
<td>return</td>
<td>none.</td>
</tr>
</tbody>
</table>

**get_returned_properties()**

The method gets what properties currently are included in a notification specified per category. The method corresponds to IOPCEventSubscriptionMgt::GetReturnedAttributes().

<table>
<thead>
<tr>
<th>Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>categories</td>
<td>The categories for that shall be used to fetch the properties.</td>
</tr>
<tr>
<td>return</td>
<td>A sequence specifying what properties currently are included per reason.</td>
</tr>
</tbody>
</table>

**refresh()**

The method forces notifications for all currently active source conditions. The notifications will not include the history but only the current source condition operational state. The method corresponds to IOPCEventSubscriptionMgt::Refresh().

<table>
<thead>
<tr>
<th>Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>return</td>
<td>A server generated handle that the client can use to cancel the operation.</td>
</tr>
</tbody>
</table>

**async_read_history()**

The method read historic events recorded by the server. If the server does not support recording of alarms and events an exception is raised. No corresponding method exists in OPC.

<table>
<thead>
<tr>
<th>Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>start_time</td>
<td>The time where to start the read. An unspecified time (=0) will make the read start from the oldest recorded event.</td>
</tr>
<tr>
<td>number_of_events</td>
<td>The max number of events to deliver as a response to the read operation.</td>
</tr>
</tbody>
</table>
cancel()

The method cancels an ongoing refresh operation or async_read_history() operations. It corresponds to IOPCEventSubscriptionMgt::CancelRefresh().

<table>
<thead>
<tr>
<th>Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cancel_id</td>
<td>A server generated handle that is given back to the server when a started operation shall be cancelled.</td>
</tr>
<tr>
<td>return</td>
<td>none.</td>
</tr>
</tbody>
</table>

get_state()

The method gets the current subscription state and corresponds to IOPCEventSubscriptionMgt::getState().

<table>
<thead>
<tr>
<th>Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>return</td>
<td>The current subscription state.</td>
</tr>
</tbody>
</table>

set_state()

The method sets the current subscription state and corresponds to IOPCEventSubscriptionMgt::setState(). The actual parameters set are returned and an exception is raised if any input parameters are not accepted and hence changed.

<table>
<thead>
<tr>
<th>Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>subscription_spec</td>
<td>New state wanted for the subscription.</td>
</tr>
<tr>
<td>return</td>
<td>The accepted subscription state.</td>
</tr>
</tbody>
</table>

clone()

The method creates a copy of the subscription.
destroy()

The method destroys the subscription object.

Filter

A flag word holding flags specifying the filters that are implemented by the server.

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FILTER_BY_MAIN_CATEGORY</td>
<td>Filtering by main category is supported.</td>
</tr>
<tr>
<td>FILTER_BY_CATEGORY</td>
<td>Filtering by sub-category is supported.</td>
</tr>
<tr>
<td>FILTER_BY_SEVERITY</td>
<td>Filtering by severity is supported.</td>
</tr>
<tr>
<td>FILTER_BY_AREA</td>
<td>Filtering by area is supported.</td>
</tr>
<tr>
<td>FILTER_BY_SOURCE</td>
<td>Filtering by source is supported.</td>
</tr>
<tr>
<td>FILTER_WITH_XPATH</td>
<td>XPath as filter description is supported.</td>
</tr>
<tr>
<td>FILTER_BY_SOURCE_TYPE</td>
<td>Filtering by source type is supported.</td>
</tr>
</tbody>
</table>

IHome

A factory object for creation of subscription management objects.

query_available_filters()

Gets by the server implemented filter functions. The method corresponds to IOPCEventServer::QueryAvailableFilters().

<table>
<thead>
<tr>
<th>Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>return</td>
<td>Filter specification flagword.</td>
</tr>
</tbody>
</table>

create_subscription()

Creates a subscription management object. An invalid state will give an exception.

<table>
<thead>
<tr>
<th>Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>subscription_spec</td>
<td>State wanted for the subscription.</td>
</tr>
</tbody>
</table>
revise_subscr_spec | State accepted by the server.
---|---
return | The new subscription manager.

**Set up Subscription**

```
query_available_filters( )
Check what filters are supported by the server
create_subscription( )
callback(DAIS::Callback)
Set the callback for server to use
selectReturnedProperties( )
Do some browsing to prepare for selecting properties
selectReturnedProperties( )
Set what property values to be included in event notifications
setFilter( )
Do some browsing to prepare for setting up the filter. Browse for reasons, areas and sources
setFilter( )
Set up filter to get only wanted event notifications
```
5.2.5 DAISAE Area IDL

5.2.5.1 DAIS::AlarmsAndEvents::Area Overview

An area is a specialization of a node and is a collection of other areas or sources. Areas are intended for arbitrary hierarchical structuring of sources for various usages (for example, areas for authority or responsibility).

DAIS::AlarmsAndEvents::Area::IHome is used for browsing the area hierarchy. The find methods in the interface correspond to the IO PCEventAreaBrowser with the filter type parameter set to OPC_AREA. The interface also implements the following OPC methods:

- IOPCEventServer::EnableConditionByArea()
- IOPCEventServer::DisableConditionByArea().
5.2.5.2 IDL

// File: DAISAEArea.idl
#ifndef _DAIS_AEAREA_IDL
#define _DAIS_AEAREA_IDL
#pragma prefix "omg.org"
#include <DAISNode.idl>
#include <DAISAECommon.idl>

module DAIS { 
    module AlarmsAndEvents { 
        module Area { 

            interface IHome : Node::IHome 
            { 
                ResourceID get_root();
                ResourceErrors enable_condition ( 
                    in ResourceIDs areas 
                );
                ResourceErrors disable_condition ( 
            } 

            AreaSourceComponent 
            { 
                id : ResourceID 
                name : string 
                description : string 
            } 

            Area 
            { 
                +parent 
            } 

            Source 
            (from DAISAECommon) 

        } 
    } 
} 

Figure 5-8  DAIS alarms and events area IDL in UML
in ResourceIDs areas;
);
};
};
};
#ifndef _DAIS_AEAREA_IDL

IHome

An object for browsing areas.

get_root()

A method to get the root node for the area tree.

<table>
<thead>
<tr>
<th>Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>return</td>
<td>The root area node.</td>
</tr>
</tbody>
</table>

enable_condition()

A method for enabling the sources contained by the specified areas. The corresponding OPC method is IOPCEventServer::EnableConditionByArea().

<table>
<thead>
<tr>
<th>Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>areas</td>
<td>A sequence of area identifications.</td>
</tr>
<tr>
<td>return</td>
<td>The resource identifications that failed.</td>
</tr>
</tbody>
</table>

disable_condition()

A method for disabling the sources contained by the specified areas. The corresponding OPC method is IOPCEventServer::DisableConditionByArea().

<table>
<thead>
<tr>
<th>Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>areas</td>
<td>A sequence of area identifications.</td>
</tr>
<tr>
<td>return</td>
<td>The resource identifications that failed.</td>
</tr>
</tbody>
</table>
5.2.6 DAISAESource IDL

5.2.6.1 DAIS::AlarmsAndEvents::Source Overview

A source is a specialization of a node and is contained by areas. A source represents an object that generates alarms and events. A source may have source conditions. The find methods in the interface correspond to the IOPCEventAreaBrowser with the filter type parameter set to OPC_SOURCE. The interface also implements the following OPC methods:

- IOPCEventServer::TranslateToItemIDs
- IOPCEventServer::EnableConditionBySource()
- IOPCEventServer::DisableConditionBySource()
5.2.6.2 IDL

```idl
//File: DAISAESource.idl
#ifndef __DAIS_AESOURCE_IDL
#define __DAIS_AESOURCE_IDL
#pragma prefix "omg.org"
#include <DAISNode.idl>
#include <DAISAECommon.idl>
module DAIS {
    module AlarmsAndEvents {
        module Source {
            DAIS::Node::IHome
            (from DAISNode)
            find()
            find_each()
            find_by_parent()
            find_by_type()
            get_pathnames()
            get_ids()

            DAIS::AlarmsAndEvents::Source::IHome
            translate_to_item_ids()
            enable_conditions()
            disable_conditions()

            SourceCondition
            (from DAISAESourceCondition)
            condition_op_state : DAISConditionOpState
            quality : DAIS::Quality
            last_acknowledge : DateTime
            condition_last_active : DateTime
            condition_space_last_active : DateTime
            condition_space_last_inactive : DateTime
            acknowledger_name : string
            comment : string

            AreaSourceComponent
            (from DAISAEArea)
            id : ResourceID
            name : string
            description : string

            Area
            (from DAISAEArea)
            +parent

            Source
            0..n

            AreaSourceComponent
            0..n

            AreaSourceComponent
            0..n

            AreaSourceComponent
            0..n

            AreaSourceComponent
            0..n

            AreaSourceComponent
            0..n

            AreaSourceComponent
            0..n

            AreaSourceComponent
            0..n

            AreaSourceComponent
            0..n

            AreaSourceComponent
            0..n

            AreaSourceComponent
            0..n

            AreaSourceComponent
            0..n

            AreaSourceComponent
            0..n

            AreaSourceComponent
            0..n

            AreaSourceComponent
            0..n

            AreaSourceComponent
            0..n

            AreaSourceComponent
            0..n

            AreaSourceComponent
            0..n

            AreaSourceComponent
            0..n

            AreaSourceComponent
            0..n

            AreaSourceComponent
            0..n

            AreaSourceComponent
            0..n

            AreaSourceComponent
            0..n

            AreaSourceComponent
            0..n

            AreaSourceComponent
            0..n

            AreaSourceComponent
            0..n

            AreaSourceComponent
            0..n

            AreaSourceComponent
            0..n

            AreaSourceComponent

```
interface IHome : Node::IHome
{
    exception PropertyDidNotTranslate(string reason);;

    ItemIDs translate_to_item_ids (
        in  ResourceID source,
        in  ResourceID category,
        in  PropertyIDs properties
    ) raises (PropertyDidNotTranslate);

    ResourceErrors enable_conditions (in ResourceIDs sources);

    ResourceErrors disable_conditions (in ResourceIDs sources);
};};};};
#endif // __DAIS_AESOURCE_IDL

IHome

An object for browsing sources at areas.

PropertyDidNotTranslate

An exception telling that one or more properties did not translate to ItemIDs.

translate_to_item_ids()

A method for translation of information about a source to ItemIDs for use with the data access interface. If one or more properties did not translate to ItemIDs, an exception is raised. The corresponding OPC method is IOPCEventServer::TranslateToItemIDs().

<table>
<thead>
<tr>
<th>Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>source</td>
<td>The identification of the source.</td>
</tr>
<tr>
<td>category</td>
<td>The identification of the category.</td>
</tr>
<tr>
<td>properties</td>
<td>A sequence of properties for which ItemIDs are wanted.</td>
</tr>
<tr>
<td>return</td>
<td>A sequence of ItemIDs. Properties that did not translate to ItemIDs are returned as empty ItemIDs.</td>
</tr>
</tbody>
</table>
**enable_conditions()**

A method for enabling the specified sources. The corresponding OPC method is `IOPCEventServer::EnableConditionBySource()`.

<table>
<thead>
<tr>
<th>Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sources</td>
<td>A sequence of area identifications.</td>
</tr>
<tr>
<td>return</td>
<td>The resource identifications that failed.</td>
</tr>
</tbody>
</table>

**disable_conditions()**

A method for disabling specified sources. The corresponding OPC method is `IOPCEventServer::DisableConditionBySource()`.

<table>
<thead>
<tr>
<th>Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sources</td>
<td>A sequence of area identifications.</td>
</tr>
<tr>
<td>return</td>
<td>The resource identifications that failed.</td>
</tr>
</tbody>
</table>

**Browse sources**

*Figure 5-11* Browse sources interaction
5.2.7 DAISConditionSpace IDL

5.2.7.1 DAIS::AlarmsAndEvents::ConditionSpace Overview

A condition space describes a set of conditions; that is, it is a space of conditions. Each condition space is associated with a particular sub-reason of the main reason ConditionReason. Each condition has logic describing when the condition is active. The logic is described in a little language having the following constructs:

- arithmetic operators
- logic operators
- references to properties

The referred properties must be included in the set of properties defined by the associated reason. The little language grammar is server specific and is outside the scope of this specification. Transitions describe what transitions between conditions are allowed. The interface does not support exploration of the transitions.

In OPC the ConditionSpace interface is implemented by the methods:

- IOPCEventServer::QueryConditionNames()
- IOPCEventServer::QuerySubConditionNames()
Figure 5-12  DAIS alarms and events condition space IDL in UML

5.2.7.2  IDL

```
//File: DAISAEConditionSpace.idl
#ifndef _DAIS_AECONDITION_SPACE_IDL
#define _DAIS_AECONDITION_SPACE_IDL
#pragma prefix "omg.org"
#include <DAISAECommon.idl>

module DAIS {
  module AlarmsAndEvents {
    module ConditionSpace {
```
struct ConditionDescription {
    unsigned long id_number;
    string name;
    string condition_logic;
    unsigned long severity;
    string descrip;
};
typedef sequence<ConditionDescription> ConditionDescriptions;

struct Description {
    ResourceID id;
    string name;
    string descrip;
    ConditionDescriptions conditions;
};
typedef sequence<Description> Descriptions;

interface IHome {
    exception UnknownResourceID {string reason;};
    Description find (in ResourceID condition_space) raises (UnknownResourceID);
    Descriptions find_each (in ResourceIDs condition_spaces) raises (UnknownResourceID);
    Descriptions find_by_category (in ResourceID category) raises (UnknownResourceID);
    Descriptions find_by_source (in ResourceID source) raises (UnknownResourceID);
    Strings get_names (in ResourceIDs condition_spaces);
    ResourceIDs get_ids (in Strings names);
};
#endif // _DAIS_AECONDITION_SPACE_IDL
**ConditionDescription**

A struct describing a condition.

<table>
<thead>
<tr>
<th>Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id_number</td>
<td>A numeric identification unique within the condition space.</td>
</tr>
<tr>
<td>name</td>
<td>The name of the condition.</td>
</tr>
<tr>
<td>condition_logic</td>
<td>The logic telling when the condition is active. Refer to Section 5.2.7.3, “Condition Logic,” on page 5-38 for a description.</td>
</tr>
<tr>
<td>severity</td>
<td>Severity is a number between 1 and 1000 having the following meaning:</td>
</tr>
<tr>
<td></td>
<td>• Low severity 1-200</td>
</tr>
<tr>
<td></td>
<td>• Medium low severity 201-400</td>
</tr>
<tr>
<td></td>
<td>• Medium severity 401-600</td>
</tr>
<tr>
<td></td>
<td>• Medium high severity 601-800</td>
</tr>
<tr>
<td></td>
<td>• High severity 801-1000</td>
</tr>
<tr>
<td>description</td>
<td>A text that to be included in event notifications when the condition is active.</td>
</tr>
</tbody>
</table>

**Description**

A struct describing the condition space.

<table>
<thead>
<tr>
<th>Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>A ResourceID identifying the condition space.</td>
</tr>
<tr>
<td>name</td>
<td>The name of the condition space.</td>
</tr>
<tr>
<td>description</td>
<td>A description of the condition space.</td>
</tr>
<tr>
<td>conditions</td>
<td>A sequence of the conditions creates the condition space. In OPC the conditions are called sub-conditions and are retrieved by the method IOPCEventServer::QuerySubConditionNames().</td>
</tr>
</tbody>
</table>

**IHome**

An object for browsing the condition spaces defined by a server.
find()

A method for getting the description of a condition space.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>condition_space</td>
<td>A ResourceID identifying a condition space.</td>
</tr>
<tr>
<td>return</td>
<td>The condition space description.</td>
</tr>
</tbody>
</table>

find_each()

A method for getting the descriptions for a number of condition spaces.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>condition_spaces</td>
<td>A sequence of ResourceID identifying condition spaces.</td>
</tr>
<tr>
<td>return</td>
<td>A sequence of condition space descriptions.</td>
</tr>
</tbody>
</table>

find_by_category()

A method for finding all condition spaces defined for a category. The corresponding OPC method is IOPCEventServer::QueryConditionNames().

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>category</td>
<td>A ResourceID identifying the category for which to get the condition spaces.</td>
</tr>
<tr>
<td>return</td>
<td>A sequence of condition space descriptions.</td>
</tr>
</tbody>
</table>

find_by_source()

A method for finding all condition spaces defined for a source. The corresponding OPC method is IOPCEventServer::QuerySourceConditions().

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>source</td>
<td>A ResourceID identifying the source for which to get the condition spaces.</td>
</tr>
<tr>
<td>return</td>
<td>A sequence of condition space descriptions.</td>
</tr>
</tbody>
</table>

get_names()

A method translating a number of condition space identifications into name strings.
**get_ids**

A method translating a number of condition space names into ResourceIDs.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>condition_spaces</td>
<td>A sequence of ResourceID identifying condition spaces.</td>
</tr>
<tr>
<td>return</td>
<td>A sequence of condition space names. Non-translated identifications are returned as empty strings.</td>
</tr>
</tbody>
</table>

**Browse condition space by source**

![Browse condition space by source interaction](image1)

*Figure 5-13  Browse condition space by source interaction*

**Browse condition space by category**

![Browse condition space by category interaction](image2)

*Figure 5-14  Browse condition space by category interaction*
5.2.7.3 Condition Logic

The reduced XPath language as defined in Section 3.1.12, “Logical Expressions and Navigation,” on page 3-25 is used to describe the condition_logic. This allows description of a path through a data structure picking up property values to be used in a logical expression.

A common case is property values at objects associated with a source, hence the starting point for navigation is the source. An example from the CIM model where the Measurement is the source can be found in Figure 4-3 on page 4-6.

Examples how to use DAIS_Expressions starting from a source are shown below:

Measurement.value < 500 assuming that the tested value is an element at the source.

MeasurementValue[id(MeasurementValue.MeasurementValueSource)/label='telemetry']/
MeasurementValue.value < id(Measurement.LimitSets)[LimitSet.label = 'summer']/Limit[label = 'highalarm']/Limit.value

Based on the model in Figure 4-3 on page 4-6, the XML example in Section 3.1.12, “Logical Expressions and Navigation,” on page 3-25 and the source as a Measurement.

5.2.8 DAISAECondition Source IDL

5.2.8.1 SourceCondition Overview

A source condition associates a source with a condition space. A source condition holds the current information specific to a source using a particular condition space for the supervision. A source condition has no own identification and is identified by its associated source and condition space. Information about a source condition is accessed through its home object. In OPC a source condition is sometimes called an instance of a condition.

In OPC the SourceCondition interface is implemented by the methods:

• IOPCEventServer::QuerySourceConditions()
• IOPCEventServer::AckCondition().
Figure 5-15  DAIS alarms and events source condition IDL in UML

5.2.8.2 IDL

```
//File: DAISAESourceCondition.idl
#ifndef _DAIS_AESOURCE_CONDITION_IDL
#define _DAIS_AESOURCE_CONDITION_IDL
#pragma prefix "omg.org"
#include <DAISAECommon.idl>

module DAIS {
module AlarmsAndEvents {
module SourceCondition {

struct Id {
    ResourceID source;
    ResourceID condition_space;
};
typedef sequence<Id>Ids;

struct Description {

```
Id source_condition;
SourceConditionOpState source_condition_op_state;
unsigned long active_condition;
string ac_logic;
unsigned long ac_severity;
string ac_description;
Quality dais_quality;
DateTime last_acknowledge;
DateTime condition_last_active;
DateTime condition_space_last_active;
DateTime condition_space_last_inactive;
string acknowledger_name;
string comment;
PropertyValues property_values;
};

typedef sequence< Description > Descriptions;

struct AcknowledgeSpec {
    Id source_condition;
    DateTime active_time;
    EventID cookie;
};

typedef sequence<AcknowledgeSpec>AcknowledgeSpecs;

interface Iterator {
    boolean next_n (in unsigned long n, out Descriptions c_descriptions);
    void reset();
    Iterator clone();
    void destroy();
};

interface IHome {
    exception UnknownId {string reason;};
    exception UnknownPropertyID {string reason;};

    Description find (in Id source_condition,
                      in PropertyIDs properties)
                      raises (UnknownId, UnknownPropertyID);

    Iterator find_each( in Ids source_conditions,
                        in PropertyIDs properties)
                        raises (UnknownId, UnknownPropertyID);

    Descriptions ack_condition (};
### Id

A struct that identifies a source condition.

<table>
<thead>
<tr>
<th>Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>source</td>
<td>The ResourceID identifying the associated source.</td>
</tr>
<tr>
<td>condition_space</td>
<td>The ResourceID identifying the associated condition space.</td>
</tr>
</tbody>
</table>

### Description

A struct describing the source condition.

<table>
<thead>
<tr>
<th>Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>The Id identifying the source condition.</td>
</tr>
<tr>
<td>source_condition_op_state</td>
<td>The DAISSourceConditionOpState as described in Section 5.2.2, “Alarms and Events Common IDL Definitions,” on page 5-7.</td>
</tr>
<tr>
<td>active_condition</td>
<td>The identification number of the currently active condition.</td>
</tr>
<tr>
<td>ac_logic</td>
<td>The condition logic from the active condition.</td>
</tr>
<tr>
<td>ac_severity</td>
<td>The severity from the active condition.</td>
</tr>
<tr>
<td>ac_description</td>
<td>The description from the active condition.</td>
</tr>
<tr>
<td>dais_quality</td>
<td>The quality is evaluated from the qualities from the properties used to evaluate the condition logic.</td>
</tr>
<tr>
<td>last_acknowledge</td>
<td>The last time the condition was acknowledged.</td>
</tr>
<tr>
<td>condition_last_active</td>
<td>Time for the latest condition transition.</td>
</tr>
<tr>
<td>Member</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------</td>
<td>------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>condition_space_last_active</td>
<td>The last time when the condition space became active. After this time more condition transitions may occur. The condition_last_active will then be later than condition_space_last_active.</td>
</tr>
<tr>
<td>condition_space_last_inactive</td>
<td>The last time when the condition space became inactive; that is, no conditions are active.</td>
</tr>
<tr>
<td>acknowledger_name</td>
<td>The name of the client making an acknowledge.</td>
</tr>
<tr>
<td>comment</td>
<td>A comment passed by the client making an acknowledge.</td>
</tr>
<tr>
<td>property_values</td>
<td>A sequence of property values as selected by the Manager::select_returned_properties call.</td>
</tr>
</tbody>
</table>

**AcknowledgeSpec**

A struct specifying an alarm to acknowledge.

**Member**

<table>
<thead>
<tr>
<th>Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>source_condition</td>
<td>The Id identifying the source condition for which to acknowledge an alarm.</td>
</tr>
<tr>
<td>active_time</td>
<td>The time when the alarm activated.</td>
</tr>
<tr>
<td>cookie</td>
<td>A identification of the alarm.</td>
</tr>
</tbody>
</table>

**Iterator**

A standard iterator. Refer to Section 3.1.6, “Iterator Methods IDL,” on page 3-10.

**IHome**

An object for browsing and accessing source conditions.

**UnknownDAISSourceConditionID**

An exception telling that the source condition identification was not recognized.

**UnknownPropertyID**

An exception telling that a property identification was not recognized.
find()

A method for getting the description of a source condition. The corresponding OPC method is IOPCEventServer::GetConditionState().

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>source_condition</td>
<td>A ResourceID identifying a source condition.</td>
</tr>
<tr>
<td>return</td>
<td>The source condition description.</td>
</tr>
</tbody>
</table>

find_each()

A method for getting the descriptions for a number of source conditions.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>condition_spaces</td>
<td>A sequence identifying source conditions.</td>
</tr>
<tr>
<td>return</td>
<td>A sequence of source condition descriptions.</td>
</tr>
</tbody>
</table>

ack_condition()

A method to acknowledge a number of source condition alarms. The corresponding OPC method is IOPCEventServer::AckCondition().

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>acknowledged_name</td>
<td>The name of the client making the acknowledge.</td>
</tr>
<tr>
<td>comment</td>
<td>A comment to be added to source condition and the event.</td>
</tr>
<tr>
<td>ack_spec</td>
<td>A sequence specifying the alarms to acknowledge.</td>
</tr>
<tr>
<td>return</td>
<td>A sequence containing descriptions for the acknowledged source conditions.</td>
</tr>
</tbody>
</table>

Inspect a specific source condition

Figure 5-16  Inspect a specific source condition interaction
5.2.9 DAISCategory IDL

5.2.9.1 DAIS::AlarmsAndEvents::Category Overview

These are definitions for manipulating categories. A category describes the content of an event. Categories are organized in a two level hierarchy where the first level has the three mandatory main categories; simple, condition, and tracking.

Simple category does not have a condition space (refer to ConditionSpace). Generation of simple category alarms or events is coded into some software function and cannot be configured. Typical simple categories are program errors, hardware device failures, etc.

Sources of different types (e.g., breaker position, breaker current, generator active power, etc.) may have different ways an alarm or event is generated (e.g., breaker trip, breaker over current, generator active power generation overload, etc.). Condition categories are used to describe source type specific alarm and event contents.

The alarm and event generation for a condition category is configured using condition spaces and several condition spaces can be created for the same condition category. This corresponds to variations in the way the alarms and events are generated using different condition logics.

Alarms or events due to operator actions or control functions (intended alarms or events rather than spontaneous) use tracking categories.

A category has a set of properties associated with it. Some or all of the properties may be included in event notifications. The properties come from the source type and may be used by the logic generating an event notification.

The get_main_category() method is used to obtain the three main categories. The find_by_parent() is used to get the child categories. Categories at this second level correspond to the OPC EventCategories. The labels for the three main reasons are “DAIS_CONDITION_CATEGORY,” “DAIS_TRACKING_CATEGORY,” and “DAIS_SIMPLE_CATEGORY.”
In OPC the Category interface is implemented by the methods IOPCEventServer::QueryEventCategories() and IOPCEventServer::QueryEventAttributes().

5.2.9.2 IDL

```
//File: DAISAECategory.idl
ifndef _DAIS_AECATEGORY_IDL
#define _DAIS_AECATEGORY_IDL
#pragma prefix "omg.org"
#include <DAISNode.idl>

module DAIS {
  module AlarmsAndEvents {
    module Category{

      interface IHome : Node::IHome {
        Node::Descriptions get_main_categories();
        get_event_properties();
      }
    }
  }
}
```
IHome

IHome is an object for browsing categories. Most of the browsing functionality is inherited from DAIS::Node.

get_main_categories()

Get the three main categories; simple, condition, and tracking. The three main categories appear as three different roots.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>return</td>
<td>The descriptions for the three main categories. The labels for them are:</td>
</tr>
<tr>
<td></td>
<td>- &quot;DAIS_CONDITION_CATEGORY&quot;</td>
</tr>
<tr>
<td></td>
<td>- &quot;DAIS_TRACKING_CATEGORY&quot;</td>
</tr>
<tr>
<td></td>
<td>- &quot;DAIS_SIMPLE_CATEGORY&quot;</td>
</tr>
</tbody>
</table>

get_event_properties()

Get all properties that are used in the supervision of the source type for a specific category. The corresponding OPC method is IOPCEventServer::QueryEventAttributes().

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>category</td>
<td>The identification of the categories to get the properties for.</td>
</tr>
<tr>
<td>return</td>
<td>A sequence of property identifications.</td>
</tr>
</tbody>
</table>
Browse categories

: Client

get_main_reasons()

Get the three main reasons

find_by_parent()

Get the sub reasons for one main reason. For each sub reason explore
1) the condition spaces that are defined, refer to ConditionSpace
2) the properties that are used in supervision of sources

Continue and explore all main reasons

Figure 5-19  Browse category interaction
5.2.10 DAISAEIO IDL

5.2.10.1 DAIS::AlarmsAndEvents::IO Overview

The IO interface for alarms & events only supports callbacks and the client shall implement callback.

The event notifications sent over the callback are of three different formats corresponding to the three main categories (simple event, condition event, and tracking event). All three event formats have the common part Event. The tracking event has additional information on the operator and the condition event has additional information on the source condition causing the notification.

In OPC the IO interface is implemented by the interface IOPCEventSink
5.2.10.2 IDL

```c++
//File: DAISAEIO.idl
#define _DAIS_AEIO_IDL
#define DAIS_AEIO_IDL
#pragma prefix "omg.org"
#include <DAISAECommon.idl>

module DAIS {
    module AlarmsAndEvents {
        module IO {

            typedef unsigned long ChangeSpec;
            const ChangeSpec CHANGE_ACTIVE_STATE = 0x0001;

            class Event {
                Source source : ResourceID
                source_pathname : string
                time_stamp : DateTime
                message : string
                event_format : EventFormat
                category : ResourceID
                category_name : string
                severity : unsigned long
                property_values : PropertyValues
            }

            class SimpleEvent {
                SimpleCategory (from DAISAECategory)
            }

            class ConditionEvent {
                ConditionCategory (from DAISAECategory)
                condition_space_name : string
                condition_space : ResourceID
                condition_name : string
                condition_number : unsigned long
                ack_required : boolean
                active_time : DateTime
                event_id : EventID
                change_specification : ChangeSpec
                source_condition_op_state : SourceConditionOpState
                dais_quality : Quality
            }

            class TrackingEvent {
                TrackingCategory (from DAISAECategory)
                actor_name : string
            }

            <<Interface>>
            DAIS::AlarmsAndEvents::IO::Callback
            on_event()
            on_read_complete()
        }
    }
}
```
const ChangeSpec CHANGE_ACK_STATE = 0x0002;
const ChangeSpec CHANGE_ENABLE_STATE = 0x0004;
const ChangeSpec CHANGE_QUALITY = 0x0008;
const ChangeSpec CHANGE_SEVERITY = 0x0010;
const ChangeSpec CHANGE_CONDITION = 0x0020;
const ChangeSpec CHANGE_MESSAGE = 0x0040;
const ChangeSpec CHANGE_ATTRIBUTE = 0x0080;

struct SimpleEvent {
    ResourceID source;
    string source_pathname;
    DateTime time_stamp;
    string message;
    EventFormat event_format;
    ResourceID category;
    string category_name;
    unsigned long severity;
    PropertyValues property_values;
};

struct TrackingEvent {
    string actor_name;
};

struct ConditionEvent {
    string condition_space_name;
    ResourceID condition_space;
    string condition_name;
    unsigned long condition_number;
    boolean ack_required;
    DateTime active_time;
    EventID event_id;
    ChangeSpec change_specification;
    SourceConditionOpState source_condition_op_state;
    Quality dais_quality;
};

struct Event {
    SimpleEvent simple_event;
    TrackingEvent tracking_event;
    ConditionEvent condition_event;
};

typedef sequence<Event> Events;

interface Callback {
    void on_event (
        in boolean refresh,
        in boolean last_refresh,
        in Events the_events
    );
}
void on_read_complete (  
    in Events the_events,  
    in unsigned long transaction_id  
); 
}; 
}; 
}; 
}; 
#endif // _DAIS_AEIO_IDL

ChangeSpec

A flag word having a number of flags telling what change caused the event notification.

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHANGE_ACTIVE_STATE</td>
<td>Alarm active has changed.</td>
</tr>
<tr>
<td>CHANGE_ACK_STATE</td>
<td>Alarm acknowledge has changed.</td>
</tr>
<tr>
<td>CHANGE_ENABLE_STATE</td>
<td>Enable has changed.</td>
</tr>
<tr>
<td>CHANGE_QUALITY</td>
<td>The quality has changed.</td>
</tr>
<tr>
<td>CHANGE_SEVERITY</td>
<td>The severity has changed.</td>
</tr>
<tr>
<td>CHANGE_CONDITION</td>
<td>A condition has become active/inactive.</td>
</tr>
<tr>
<td>CHANGE_MESSAGE</td>
<td>The message has been updated.</td>
</tr>
<tr>
<td>CHANGE_ATTRIBUTE</td>
<td>An attribute value has changed.</td>
</tr>
</tbody>
</table>

SimpleEvent

A struct holding the simple event data.

<table>
<thead>
<tr>
<th>Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>source</td>
<td>The identification of the source for which the event notification was created.</td>
</tr>
<tr>
<td>source_pathname</td>
<td>The full pathname of the source.</td>
</tr>
<tr>
<td>time_stamp</td>
<td>Time of the event occurrence - for conditions, time that the condition transitioned into the new state or condition. For example, if the event notification is for acknowledgment of a condition, this would be the time that the condition became acknowledged.</td>
</tr>
<tr>
<td>message</td>
<td>Event notification message describing the event.</td>
</tr>
<tr>
<td>event_format</td>
<td>The identification for one of the three main formats.</td>
</tr>
<tr>
<td>Member</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>category</td>
<td>The identification of the sub-category why the event notification was sent.</td>
</tr>
<tr>
<td>category_name</td>
<td>The label for the sub-category.</td>
</tr>
<tr>
<td>severity</td>
<td>The severity for the event, a number between 0 and 1000.</td>
</tr>
<tr>
<td>property_values</td>
<td>A sequence of property values as selected by the select_returned_properties() method.</td>
</tr>
</tbody>
</table>

**TrackingEvent**

A struct holding tracking event data.

<table>
<thead>
<tr>
<th>Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>actor_name</td>
<td>The name of the actor or operator causing the event notification.</td>
</tr>
</tbody>
</table>

**ConditionEvent**

A struct holding the condition event data.

<table>
<thead>
<tr>
<th>Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>condition_space_name</td>
<td>The name of the condition space that caused the event notification.</td>
</tr>
<tr>
<td>condition_space</td>
<td>Identification of the condition space.</td>
</tr>
<tr>
<td>condition_name</td>
<td>The name of the condition that caused the event notification.</td>
</tr>
<tr>
<td>condition_number</td>
<td>Identification of the condition.</td>
</tr>
<tr>
<td>ack_required</td>
<td>An indication that an alarm is generated and that an acknowledgment is required.</td>
</tr>
<tr>
<td>active_time</td>
<td>The time when the condition became active.</td>
</tr>
<tr>
<td>event_id</td>
<td>An identification of the event notification.</td>
</tr>
<tr>
<td>change_specification</td>
<td>Indicates to the client which properties of the condition have changed.</td>
</tr>
<tr>
<td>source_condition_op_state</td>
<td>The new state for the source condition.</td>
</tr>
<tr>
<td>quality</td>
<td>The quality.</td>
</tr>
</tbody>
</table>
Event

A struct composed of the three above event structs. The simple event struct always contains valid information and which of the tracking or condition event structs are valid is decided from the main_reason simple event member.

<table>
<thead>
<tr>
<th>Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>simple_event</td>
<td>The simple event.</td>
</tr>
<tr>
<td>tracking_event</td>
<td>The tracking event.</td>
</tr>
<tr>
<td>condition_event</td>
<td>The condition event.</td>
</tr>
</tbody>
</table>

Callback

The callback object implemented by the client and used by the server to send event notifications.

on_event()

The callback method.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>refresh</td>
<td>Indicates if this callback is due to a refresh.</td>
</tr>
<tr>
<td>last_refresh</td>
<td>Indicates if this callback is the last in a sequence initiated by a refresh.</td>
</tr>
<tr>
<td>the_events</td>
<td>A sequence of event notifications.</td>
</tr>
<tr>
<td>return</td>
<td>None.</td>
</tr>
</tbody>
</table>

on_read_complete()

A callback method for the async_read_history() method. After an async_read_history() call from a client the server will deliver the historical events using this method. There is no corresponding OPC method.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>the_events</td>
<td>A sequence of historical events.</td>
</tr>
<tr>
<td>transaction_id</td>
<td>The transaction number matching the client call number.</td>
</tr>
</tbody>
</table>
Event notification callbacks

The on_event method is called by the server at:
1) expiration of the subscription buffer_time timeout
2) subscription event buffer full

For alarms the client may want to call SourceCondition::ack_alarm() to acknowledge. All information needed for doing this is found in the event notifications.

Refresh callbacks

On_event is called as response to a refresh. All refresh calls are made with the refresh parameter set and the last call also has the last_refresh parameter set.
References

A.1 List of References

1. OMG DAIS RFP dtc/99-01-02
2. OMG Utility Management Systems Data Access Facility (DAF) formal/01-06-01
4. OPC Data access version 2.03; www.opcfoundation.org.
5. OPC Alarm and events 1.02; www.opcfoundation.org.
7. Guidelines for Control Center APIs; EPRI TR-106324
8. Energy management system APIs; IEC standard 61970 and 61970-30x.
10. Structuring principles and reference designations; IEC standard 1346-1
11. CIM UML model from the Rose file cimu09a.mdl.
The complete IDL can be found in the zip archive mfg/2001-01-04. The following URL should be used to access this file:

http://www.omg.org/cgi-bin/doc?mfg/01-01-04
The complete UML model can be found in the Rose file mfg/2001-01-05. The following URL should be used to access this file:

http://www.omg.org/cgi-bin/doc?mfg/01-01-05

Note – This file is produced in Rose 2000 and is backward compatible at least to Rose 98. It is also possible to open the file with Microsoft Visual Studio 6 Enterprise, Visual Modeler.
Glossary

List of Definitions

API - Application Program Interface.

CIM - Common Information Model. The CIM is a power system information model that was developed in an EPRI sponsored project. The EPRI report was published in 1996.

DAF - The Utility Management System Data Access Facility.

DAF Client - A program or software entity that uses the DAF interfaces to obtain information. Abbreviated to client in most of this specification.

Data Provider - An implementation of the DAF. That is, a program or software entity that supplies information via the DAF interfaces. Also referred to as a DAF server or just a server.

DMS - A Distribution Management System. This is a UMS for operating an electric power sub-transmission and distribution system.

EMS - An Energy Management System. This is a UMS for operating an electric power main transmission and/or production system.

EPRI - Electric Power Research Institute. A power industry body that is engaged in an effort to define APIs and data models for EMS systems and applications.

EPRI CIM - The EPRI Common Information Model. A data model defined in UML that can be used to describe power systems and related concepts.

HMI - Human Machine Interface (also GUI - Graphical User Interface).

OPC - OLE for Process Control.

PLC - Programmed Logic Controller, a device that controls an item or items of equipment. A PLC may transmit data it gathers to a UMS and receive control commands from the UMS. In this case it fills a role similar to an RTU.
Power System - The integrated facilities and resources that produce, transmit and/or distribute electric energy.

RDF - Resource Description Framework. A model of data that has been defined by a W3C recommendation and is used in conjunction with XML notation.

RTU - Remote Terminal Unit, a device located at a (usually) remote site that connects equipment with a central UMS. An RTU gathers data from equipment, and transmits that data back to the UMS. It also receives commands from the UMS and controls the equipment.

SCADA - Supervisory Control and Data Acquisition, a system that gives operators oversight and control of geographically dispersed facilities.

UML - Unified Modeling Language. The OMG standard modeling language, which has been used to define the EPRI Common Information Model.

UMS - Utility Management System, a control system that incorporates simulation and analysis applications used by a water, gas or electric power utility for operations or operational decision support.

WQEMS - A Water Quality and Energy Management System. This is a UMS for operating water supply and/or waste water systems.

XML - Extensible Markup Language. A generic syntax defined by a W3C recommendation that can be used to represent UMS data and schema, among other things.
Index

A
Alarms & Events 1-7, 5-6
API 4-8
Authorization 3-28

C
Callbacks 2-2
Character encoding 3-1
CIM classes 4-7
COM and CORBA IDL Differences 2-6
Common declarations 3-1
Common IDL Overview 3-1
Common Information Model (CIM) 1-3
Concurrency Control 1-5
Conformance 1-7
CORBA
  contributors vii
  documentation set vi

D
DAIS Server IDL 3-30
DAISAEArea 5-23
DAISAEIO 5-44
DAISAESEdition 5-8
DAISAESEsession 5-28
DAISAESEsourceCondition 5-34
DAISAESEsubscription 5-11
DAISCommon IDL 3-2
DAISConditionSpace 5-28
DAISDAO 4-23
DAISDASession IDL 4-9
DAISDASimpleIO 4-49
DAISGroup 4-40
DAISGroupEntry 4-32
DAISItem 4-14
DAISNode IDL 3-12
DAISProperty IDL 3-19
DAISReason 5-40
DAISSession IDL 3-22
DAISType IDL 3-16
Data Access 1-4, 1-7
Data Access Facility (DAF) 2-4
Data Semantics 1-5
Data Types 2-6

E
Enumerators 2-2
Error and status codes 2-2
Error management 2-7

F
Filter definitions 3-25

H
Hierarchical structure diagram 1-2
High Performance Implementations 1-6

I
Identifiers, handles, and blobs 2-2
IEC 1346-1, Structuring and Naming 2-7
IEC 61970 structure 2-8
Information Model 4-1
Information model/schema 2-5
Interface management 2-7
Item Values 4-4
Items, structuring, and naming 2-3

L
Logical Expressions and Navigation 3-25

M
Method return data 2-3

N
Naming 4-3

O
Object Management Group v
  address of vii
Object referencing 2-7
OLE for Process Control (OPC) 2-1
OPC 1-3
OPC names 2-2
OPC Recommended Properties 4-5, 5-5

P
Parameters and structs 2-3
Properties and types 2-4

R
Real World Objects (RWOs) 4-1
Remote terminal units (RTUs) 1-1
Requirement levels 3-29

S
SCADA system 1-1
Security Service A-1, B-1, C-1, 1
Server 1-7, 3-29
Server side cursor 2-3

T
Typographical conventions vii

U
UNIX based systems 1-3
Utility SCADA/EMS Measurement Model 4-6

W
Windows NT based systems 1-3

X
XPath 2-8
The document history for this specification is as follows:

- dtc/04-10-02 - RTF report
- dtc/04-10-05 - convenience document

This document supersedes formal/02-11-07 (v1.0).