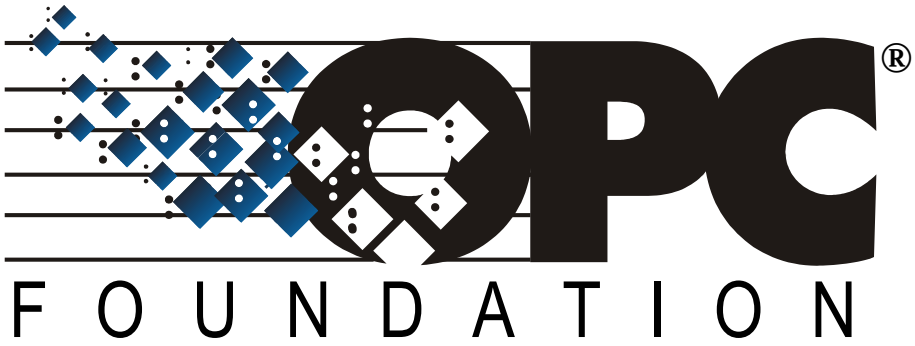

OPC Unified Architecture

Specification

Part 11: Historical Access

Release Candidate

Version 1.01

December 1, 2009

Send comments to:
UAcomments@opcfoundation.org

Specification Type	Industry Standard Specification		
Title:	OPC Unified Architecture Historical Access	Date:	December 1, 2009
Version:	Release Candidate 1.01	Software Source:	MS-Word OPC UA Part 11 - Historical Access RC 1.01.06.doc
Author:	OPC Foundation	Status:	Release Candidate

CONTENTS

FIGURES

TABLES

OPC FOUNDATION

UNIFIED ARCHITECTURE –

FOREWORD

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

AGREEMENT OF USE

with these materials. Products developed using this specification may claim compliance or conformance with this specification if and only if the software satisfactorily meets the certification requirements set by the OPC Foundation. Products that do not meet these requirements may claim only that the product was based on this specification and must not claim compliance or conformance with this specification.

TRADEMARKS

Most computer and software brand names have trademarks or registered trademarks. The individual trademarks have not been listed here.

GENERAL PROVISIONS

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

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

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

ISSUE REPORTING

The OPC Foundation strives to maintain the highest quality standards for its published specifications, hence they undergo constant review and refinement. Readers are encouraged to report any issues and view any existing errata here:
http://www.opcfoundation.org/errata

1 Scope

This specification is part of the overall OPC Unified Architecture specification series and defines the information model associated with Historical Access (HA). It particularly includes additional and complementary descriptions of the *NodeClass*es and *Attributes* needed for Historical Access, additional standard *Properties*, and other information and behaviour.

The complete *AddressSpace* model including all *NodeClass*es and *Attributes* is specified in [UA Part 3]. The predefined *Information Model* is defined in [UA Part 5]. The *Services* to detect and access historical data and events, and description of the *ExtensibleParameter* types are specified in [UA Part 4].

This specification includes functionality to compute and return *Aggregates* like minimum, maximum, average etc. The *Information Model* and the concrete working of *Aggregates* are defined in [UA Part 13]~~[UA Part 13]~~.

2 Reference documents

[UA Part 1] OPC UA Specification: Part 1 – Concepts, Version 1.0 or later
http://www.opcfoundation.org/UA/Part1/

[UA Part 3] OPC UA Specification: Part 3 – Address Space Model, Version 1.0 or later
http://www.opcfoundation.org/UA/Part3/

[UA Part 4] OPC UA Specification: Part 4 – Services, Version 1.0 or later
http://www.opcfoundation.org/UA/Part4/

[UA Part 5] OPC UA Specification: Part 5 – Information Model, Version 1.0 or later
http://www.opcfoundation.org/UA/Part5/

[UA Part 7] OPC UA Specification: Part 7 – Profiles, Version 1.0 or later
http://www.opcfoundation.org/UA/Part7/

[UA Part 8] OPC UA Specification: Part 8 – Data Access, Version 1.0 or later
http://www.opcfoundation.org/UA/Part8/

[UA Part 9] OPC UA Specification: Part 9 – Alarm & Conditions, Version 1.0 or later
http://www.opcfoundation.org/UA/Part9/

[UA Part 13] OPC UA Specification: Part 13 – Aggregates, Version 1.0 or later
http://www.opcfoundation.org/UA/Part13/

3 Terms, definitions, and abbreviations

3.1 OPC UA Part 1 terms

The following terms defined in [UA Part 1] apply.

1) AddressSpace
2) Alarm
3) Attribute
4) Client
5) Event

6) EventNotifier
7) Information Model
8) Message
9) Node
10) NodeClass
11) Notification
12) Object
13) ObjectType
14) Profile
15) Reference
16) ReferenceType
17) Server
18) Service
19) Session
20) Subscription
21) Variable
22) View

3.2 OPC UA Part 3 terms

The following terms defined in [UA Part 3] apply.

1) DataType
2) DataVariable
3) EventType
4) Property
5) SourceNode

3.3 OPC UA Part 13 terms

The following terms defined in [UA Part 13]~~[UA Part 13]~~ apply.

1) Aggregate
2) Interval
3) Interpolated
4) SlopedInterpolation
5) SteppedInterpolation
6) Bounding Values

3.4 OPC UA Historical Access terms

3.4.1 Annotation

An *Annotation* is metadata that is associated with an item at a given instance in time. There does not have to be a value stored at that time.

3.4.2 BoundingValues

BoundingValues are the values that are associated with the starting and ending time of an *Interval* specified when reading from the historian. *BoundingValues* may be required by

Clients to determine the starting and ending values when requesting raw data over a time range. If a raw data value exists at the start or end point, it is considered the bounding value even though it is part of the data request. If no raw data value exists at the start or end point, then the server will determine the boundary value, which may require data from a data point outside of the requested range. See Clause 4.4 for details on using *BoundingValues*.

3.4.3 HistoricalNode

A *HistoricalNode* is a term used in this document to represent any *Object*, *Variable, Property* or *View* in the *AddressSpace* for which a *Client* may read and/or update historical data or *Events*. The terms "*HistoricalNode's* history" or "history of a *HistoricalNode*" will refer to the time series data or *Events* stored for this *HistoricalNode* where *HistoricalNode* is an *Object*, *Variable*, *Property* or *View*. The term *HistoricalNode* refers to both *HistoricalDataNodes* and *HistoricalEventNodes*, and is used when referencing aspects of the specification that apply to accessing historical data and *Events*.

3.4.4 HistoricalDataNode

A *HistoricalDataNode* represents any *Variable* or *Property* in the *AddressSpace* for which a *Client* may read and/or update historical data. The terms "*HistoricalDataNode's* history" or "history of a *HistoricalDataNode*" will refer to the time series data stored for this *HistoricalNode* where *HistoricalNode* is an *Object*, *Variable*, or *Property*. Some examples of such data are:

- device data (like temperature sensors)
- calculated data
- status information (open/closed, moving)
- dynamically changing system data (like stock quotes)
- diagnostic data

The term *HistoricalDataNodes* is used when referencing aspects of the specification that apply to accessing historical data only.

3.4.5 HistoricalEventNode

A *HistoricalEventNode* represents any *Object* or *View* in the *AddressSpace* for which a *Client* may read and/or update historical *Events*. The terms "*HistoricalEventNode's* history" or "history of a *HistoricalEventNode*" will refer to the time series *Events* stored in some historical system. Some examples of such data are:

- *Notifications*
- system *Alarms*
- operator action events
- system triggers (such as new orders to be processed)

The term *HistoricalEventNode* is used when referencing aspects of the specification that apply to accessing historical *Events* only.

3.4.6 Modified values

A modified value is a *HistoricalDataNode's* value that has been changed (or manually inserted or deleted) after it was stored in the historian. For some servers, a lab data entry value is not a modified value, but if a user corrects a lab value, the original value would be considered a modified value, and would be returned during a request for modified values. Also manually inserting a value that was missed by a standard collection system may be considered a modified value. Unless specified otherwise, all historical *Services* operate on the current, or most recent, value for the specified *HistoricalDataNode* at the specified timestamp. Requests for modified values are used to access values that have been superseded, deleted or inserted. It is up to a system to determine what is considered a modified value. Whenever a

server has modified data available for an entry in the historical collection it shall set the *ExtraData* bit in the *StatusCode.*

3.4.7 Raw data

Raw data is data that is stored within the historian for a *HistoricalDataNode* . The data may be all data collected for the *DataValue* or it may be some subset of the data depending on the historian and the storage rules invoked when the item's values were saved.

3.4.8 StartTime / EndTime

The *StartTime* and *EndTime* specify the bounds of a history request and define the time domain of the request. For all requests, a value falling at the end time of the time domain is not included in the domain, so that requests made for successive, contiguous time domains will include every value in the historical collection exactly once.

3.4.9 TimeDomain

The interval of time covered by a particular request, or by a particular response. In general, if the start time is earlier than or the same as the end time, the time domain is considered to begin at the start time and end just before the end time; if the end time is earlier than the start time, the time domain still begins at the start time and ends just before the end time, with time "running backward" for the particular request and response. In both cases, any value which falls exactly at the end time of the *TimeDomain* is not included in the *TimeDomain*. See the examples in section 4.4. *BoundingValues* effect the time domain as described in section 4.4.

All timestamps which can legally be represented in an *UtcTime DataType* are valid timestamps, and the server may not return an invalid argument result code due to the timestamp being outside of the range for which the server has data. See [UA Part 3] for a description of the range and granularity of this *DataType*. Servers are expected to handle out-of-bounds timestamps gracefully, and return the proper *StatusCodes* to the *Client.*

3.4.10 Structured History Data

Structured History Data is structured data stored in a history collection where one or more parameters of the structure are used to uniquely identify the data within the data collection. Most historical data applications assume only one current value per timestamp. Therefore the timestamp of the data is considered the unique identifier for that value. Some data or meta data such as Annotations may permit multiple values to exist at a single timestamp. In such cases the Server would use one or more parameters of the *Structured History Data* entry to uniquely identifiy each within the history collection. *Annotations* are examples of Structured History Data.

3.5 Abbreviations and symbols

DA	Data Access
HDA	Historical Data Access
UA	Unified Architecture

4 Concepts

4.1 General

The OPC UA Historical Access specification defines of the handling of historical time series data and historical *Event* data in the OPC Unified Architecture. Included is the specification of the representation of historical data and *Events* in the OPC UA *AddressSpace.*

4.2 Data Architecture

An OPC UA Server supporting Historical Access provides one or more OPC UA *Client* with transparent access to different historical data and/or historical *Event* sources (e.g. process historians, event historians etc.).

The historical data or *Events* may be located in a proprietary data collection, database or a short term buffer within memory. An OPC UA Server supporting Historical Access may or may not provide historical data and *Events* for some or all available *Variables*, *Objects, Properties* or *Views* within the server *AddressSpace*. As with the other *Information Models*, the *AddressSpace* of an OPC UA Server supporting Historical Access is accessed via the *View* or *Query Service* sets.

An OPC UA Server supporting Historical Access provides a way to access or communicate to a set of historical data and/or historical *Event* sources. The types of sources available are a function of the server implementation.

Figure 1~~Figure 1~~ illustrates how the *AddressSpace* of a UA server might consist of a broad range of different historical data and/or historical *Event* sources.

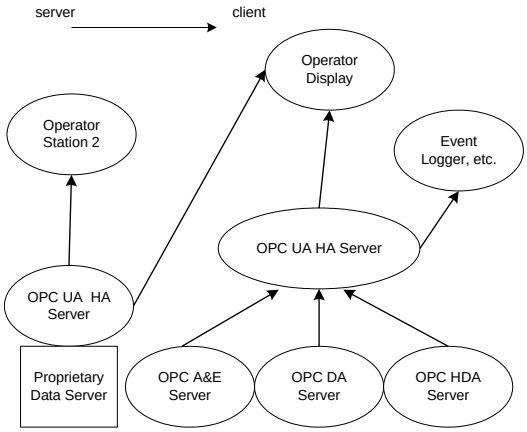

Figure 1 - Possible OPC UA Server supporting Historical Access

The server may be implemented as a standalone OPC UA Server that collects data from another OPC UA Server, a legacy OPC HDA Server, a legacy OPC DA Server, a legacy OPC A&E Server or another data source. The *Client* that references the OPC UA Server supporting Historical Access for historical data may be simple trending packages that just desire values over a given time frame or they may be complex reports that require data in multiple formats.

4.3 Timestamps

The nature of OPC UA Historical Access requires that a single timestamp reference be used to relate the multiple data points, and *Client* may request which timestamp will be used as the reference. See [UA Part 4] for details on the *TimestampsToReturn* enumeration. An OPC UA Server supporting Historical Access will treat the various timestamp settings as described below. A HistoryRead with invalid settings will be rejected with Bad_TimestampsToReturnInvalid see [UA Part 4]

For *HistoricalDataNodes,* the *SourceTimestamp* is used to determine which historical data values are returned.

SOURCE_0 Return the *SourceTimestamp*.

SERVER_1 Return the *ServerTimestamp*.

BOTH_2 Return both the *SourceTimestamp* and *ServerTimestamp*.

NEITHER_3 This is not a valid setting for any HistoryRead accessing *HistoricalDataNodes.*

Any reference to Timestamps in this context through out this specification will represent either *ServerTimestamp* or *SourceTimestamp* as dictated by the type requested in the HistoryRead *Service*. Some servers may not support historizing both *SourceTimestamp* and *ServerTimestamp*, but it is expect that all servers will support historizing *SourceTimestamp* (see [UA Part 7] for details on Server *Profiles*).

For *HistoricalEventNodes* this parameter does not apply. This parameter is ignored since the entries returned are dictated by the *Event* Filter. See section **Error! Reference source not found.**~~5.3.2.1~~ for details.

4.4 Bounding values and time domain

When accessing *HistoricalDataNodes via* the *HistoryRead Service*, requests can set a flag, returnBounds, indicating that a *BoundingValues* are requested. For a complete description of the *Extensible Parameter HistoryReadDetails* that include StartTime, EndTime and NumValuesPerNode, see Section 6.4. The concept of bounding values and how they affect the time domain that is requested as part of the *HistoryRead* request is further explained in this section. This section also provides examples of *TimeDomains* to further illustrate the expected behaviour.

When making a request for historical data using the *HistoryRead Service*, the required parameters include at least 2 of these three parameters: startTime, endTime and numValuesPerNode. What is returned when bounding values are requested varies according to which of these parameters are provided. For a historian that has values stored at 5:00, 5:02, 5:03, 5:05 and 5:06, the data returned when using the Read *Raw* functionality is given by Table 1~~Table 1~~. In the table, FIRST stands for a tuple with a value of Null, a timestamp of the specified StartTime, and a *StatusCode* of *Bad_BoundNotFound*. LAST stands for a tuple with a value of Null, a timestamp of the specified EndTime, and a *StatusCode* of *Bad_BoundNotFound*In some cases attempting to locate bounds, particularly FIRST or LAST points may be resource intensive for servers. Therefore how far back or forward to look in history for bounding values is server dependent, and the server search limits may be reached before a bounding value can be found. There are also cases, such as reading *Annotations* or *Attribute* data where Bounding values may not be appropriate. For such use cases it is permissible for the Server to return a *StatusCode* of *Bad_BoundNotSupported.*

Table 1 – Bounding Value Examples

Start Time	End Time	numValues PerNode	Bounds	Data Returned
5:00	5:05	0	Yes	5:00, 5:02, 5:03, 5:05
5:00	5:05	0	No	5:00, 5:02, 5:03
5:01	5:04	0	Yes	5:00, 5:02, 5:03, 5:05
5:01	5:04	0	No	5:02, 5:03
5:05	5:00	0	Yes	5:05, 5:03, 5:02, 5:00
5:05	5:00	0	No	5:05, 5:03, 5:02
5:04	5:01	0	Yes	5:05, 5:03, 5:02, 5:00
5:04	5:01	0	No	5:03, 5:02
4:59	5:05	0	Yes	FIRST, 5:00, 5:02, 5:03, 5:05
4:59	5:05	0	No	5:00, 5:02, 5:03
5:01	5:07	0	Yes	5:00, 5:02, 5:03, 5:05, 5:06, LAST
5:01	5:07	0	No	5:02, 5:03, 5:05, 5:06
5:00	5:05	3	Yes	5:00, 5:02, 5:03
5:00	5:05	3	No	5:00, 5:02, 5:03
5:01	5:04	3	Yes	5:00, 5:02, 5:03
5:01	5:04	3	No	5:02, 5:03
5:05	5:00	3	Yes	5:05, 5:03, 5:02
5:05	5:00	3	No	5:05, 5:03, 5:02
5:04	5:01	3	Yes	5:05, 5:03, 5:02
5:04	5:01	3	No	5:03, 5:02
4:59	5:05	3	Yes	FIRST, 5:00, 5:02
4:59	5:05	3	No	5:00, 5:02, 5:03
5:01	5:07	3	Yes	5:00, 5:02, 5:03
5:01	5:07	3	No	5:02, 5:03, 5:05
5:00	UNSPECIFIED	3	Yes	5:00, 5:02, 5:03
5:00	UNSPECIFIED	3	No	5:00, 5:02, 5:03
5:00	UNSPECIFIED	6	Yes	5:00, 5:02, 5:03, 5:05, 5:06, LAST*
5:00	UNSPECIFIED	6	No	5:00, 5:02, 5:03, 5:05, 5:06
UNSPECIFIED	5:06	3	Yes	5:06,5:05,5:03
UNSPECIFIED	5:06	3	No	5:06,5:05,5:03
UNSPECIFIED	5:06	6	Yes	5:06,5:05,5:03,5:02,5:00,FIRST**
UNSPECIFIED	5:06	6	No	5:06, 5:05, 5:03, 5:02, 5:00
4:48	4:48	0	Yes	FIRST,5:00
4:48	4:48	0	No	NODATA
4:48	4:48	1	Yes	FIRST
4:48	4:48	1	No	NODATA
4:48	4:48	2	Yes	FIRST,5:00
5:00	5:00	0	Yes	5:00,5:02
5:00	5:00	0	No	5:00
5:00	5:00	1	Yes	5:00
5:00	5:00	1	No	5:00
5:01	5:01	0	Yes	5:00, 5:02
5:01	5:01	0	No	NODATA
5:01	5:01	1	Yes	5:00
5:01	5:01	1	No	NODATA

*The timestamp of LAST cannot be the specified End Time because there is no specified End Time. In this situation the timestamp for LAST will be equal to previous timestamp returned plus one second.

** The timestamp of FIRST cannot be the specified End Time because there is no specified Start Time. In this situation the timestamp for FIRST will be equal to previous timestamp returned minus one second.

4.5 Changes in AddressSpace over time

Client use the browse *Services* of the *View Service Set* to navigate through the *AddressSpace* to discover the *HistoricalNodes* and their characteristics. These *Services* provide the most current information about the *AddressSpace*. It is possible and probable that the *AddressSpace* of a *Server* will change over time (i.e. *TypeDefinitions* may change; *NodeIds* may be modified, added or deleted).

Server developers and administrators need to be aware that modifying the *AddressSpace* may impact a *Client's* ability to access historical information. If the history for a *HistoricalNode* is still required, but the *HistoricalNode* is no longer historized, the *Object* should be maintained in the AddressSpace, with the appropriate *AccessLevel Attribute* and Historizing *Attribute* settings (see [UA Part 3] for details on access levels).

5 Historical Information Model

5.1 HistoricalNodes

5.1.1 General

The Historical Access model defines additional *Properties* that are applicable for both *HistoricalDataNodes* and *HistoricalEventNodes.*

5.1.2 Annotations Property

The *DataVariable* or *Object* that has annotation data will add the *Annotations Property*.

Table 2 – Annotations Property

Name	Use	Data Type	Description
Standard Properties			
Annotations	O	Annotation	The *Annotations Property* is used to indicate that annotation data exists for the history collection exposed by a *HistoricalDataNode*. Annotation *DataType* is defined in clause 5.5.1

Since it is not allowed for *Properties* to have *Properties*, the *Annotation property* is only available for *DataVariables* or *Objects*.

Not every *HistoricalDataNode* in the *AddressSpace* might contain annotation data. The *Annotations Property* indicates whether or not a *HistoricalDataNode* supports *Annotations*. *Annotation* data is accessed using the standard *HistoryRead* functions. *Annotations* are modified, inserted or deleted using the standard *HistoryUpdate* functions.

As with all *HistoricalNodes*, modifications, deletions or addition of *Annotations* will raise the appropriate Historical Audit Event with the corresponding *NodeId*.

5.2 HistoricalDataNodes

5.2.1 General

The Historical Data model defines additional *ObjectTypes and Objects*. These descriptions also include required use cases for *HistoricalDataNodes.*

5.2.2 HistoricalDataConfigurationType

The Historical Access Data model extends the standard type model by defining the *HistoricalDataConfigurationType.* This *Object* defines the general characteristics of a *Node*

that defines the historical configuration of any *HistoricalDataNode* that is defined to contain history. It is formally defined in Table 3~~Table 3~~

All *Instances* of the *HistoricalDataConfigurationType* use the standard *BrowseName* as defined in Table 6~~Table 6~~

Table 3 – HistoricalDataConfigurationType Definition

Attribute	Value				
BrowseName	HistoricalDataConfigurationType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
HasComponent	Object	AggregateConfiguration	--	AggregateConfigurationType	Mandatory
HasProperty	Variable	Stepped	Boolean	PropertyType	Mandatory
HasProperty	Variable	Definition	String	PropertyType	Optional
HasProperty	Variable	MaxTimeInterval	Duration	PropertyType	Optional
HasProperty	Variable	MinTimeInterval	Duration	PropertyType	Optional
HasProperty	Variable	ExceptionDeviation	Double	PropertyType	Optional
HasProperty	Variable	ExceptionDeviationFormat	Enum	PropertyType	Optional

AggregateConfiguration Object represents the browse entry point for information on how the *Server* treats *Aggregate* specific functionality such as handling *Uncertain data*. This *Object* is required to be present even if it contains no *Aggregate* configuration *Objects*. *Aggregates* are defined in [UA Part 13]~~[UA Part 13]~~.

The *Stepped Variable* specifies whether the historical data was collected in such a manner that it should be displayed as SlopedInterpolation (sloped line between points) or as *SteppedInterpolation* (vertically-connected horizontal lines between points) when raw data is examined. This *Property* also effects how some *Aggregates* are calculated. A value of True indicates stepped interpolation mode. A value of False indicates *SlopedInterpolated* mode. The default value is False.

The *Definition Variable* is a vendor-specific, human readable string that specifies how the value of this *HistoricalDataNode* is calculated. Definition is non-localized and will often contain an equation that can be parsed by certain *Client*.

Example: *Definition* ::= “(TempA – 25) + TempB”

The *MaxTimeInterval Variable* specifies the maximum interval between data points in the history repository regardless of their value change (see [UA Part 3] for definition of *Duration*).

The *MinTimeInterval Variable* specifies the minimum interval between data points in the history repository regardless of their value change (see [UA Part 3] for definition of *Duration*).

The *ExceptionDeviation Variable* specifies the minimum amount that the data for the *HistoricalDataNode* must change in order for the change to be reported to the history database.

The *ExceptionDeviationFormat Variable* specifies how the ExceptionDeviation is determined. Its values are defined in Table 4~~Table 4~~.

Table 4 – ExceptionDeviationFormat Values

Value	Description
ABSOLUTE_VALUE_0	ExceptionDeviation is an absolute Value.
PERCENT_OF_VALUE_1	ExceptionDeviation is a percent of Value.
PERCENT_OF_RANGE_2	ExceptionDeviation is a percent of InstrumentRange (See [UA Part 8])
PERCENT_OF_EU_RANGE_3	ExceptionDeviation is a percent of EURange (See [UA Part 8])
UNKNOWN_4	ExceptionDeviation type is Unknown or not specified.

5.2.3 HasHistoricalConfiguration ReferenceType

This *ReferenceType* is a concrete *ReferenceType* that can be used directly. It is a subtype of the *Aggregates ReferenceType* and will be used to refer from a Historical *Node* to one or more *HistoricalAccessConfigurationType Objects*.

The semantic indicates that the target *Node* is "used" by the source *Node* of the *Reference*. Figure 2~~Figure 2~~ informally describes the location of this *ReferenceType* in the OPC UA hierarchy. Its representation in the *AddressSpace* is specified in Table 5~~Table 5~~.

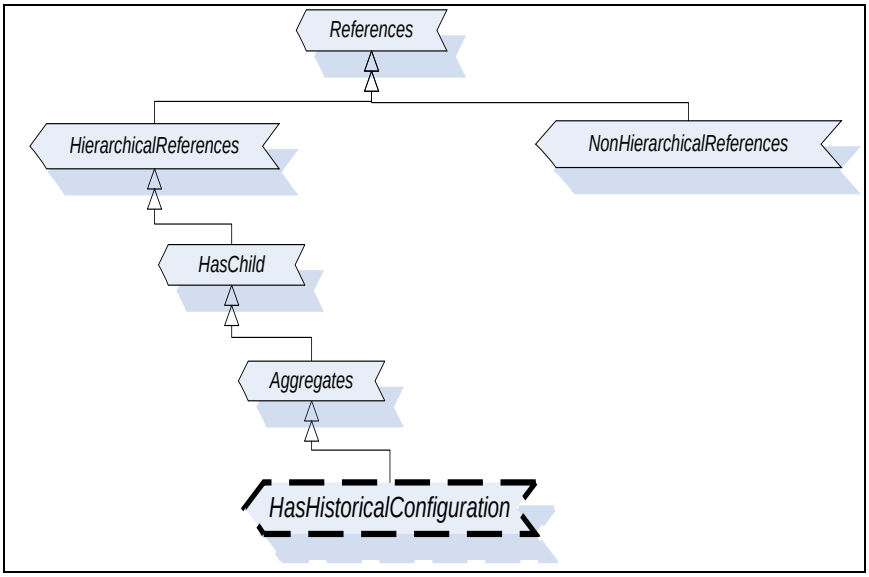

Figure 2 – ReferenceType Hierarchy

Table 5 – HasHistoricalConfiguration ReferenceType

Attributes	**Value**		
BrowseName	HasHistoricalConfiguration		
InverseName	HistoricalConfigurationOf		
Symmetric	False		
IsAbstract	False		
References	**NodeClass**	**BrowseName**	**Comment**
Subtype of Aggregates *ReferenceType* defined in [UA Part 5].			

5.2.4 Historical Data Configuration Object

This *Object* is used as the browse entry point for information about *HistoricalDataNode* configuration. The content of this *Object* is already defined by its type definition in Table 3~~Table 3~~. It is formally defined in Table 6~~Table 6~~. If a *HistoricalDataNode* has configuration defined then one instance shall have a *BrowseName* of 'HA Configuration'. Additional configurations may be defined with different *BrowseNames*. All Historical Configuration *Objects* must be *Referenced* using the *HasHistoricalConfiguration ReferenceType*. It is also highly recommended that display names be chosen that more clearly describe the historical configuration e.g. "1 Second Collection", "Long Term Configuration" etc.

Table 6 – Historical Access Configuration Definition

Attribute	**Value**				
BrowseName	HA Configuration				
References	**Node Class**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
HasTypeDefinition	Object Type	HistoricalDataConfigurationType	Defined in Table 3~~Table 3~~		

5.2.5 HistoricalDataNodes Address Space Model

HistoricalDataNodes are always part of other *Nodes* in the AddressSpace. They are never defined by themselves. A simple example of a container for *HistoricalDataNodes* would be a "Folder Object".

Figure 3~~Figure 3~~ illustrates the basic *AddressSpace* model of a *DataVariable* that includes History.

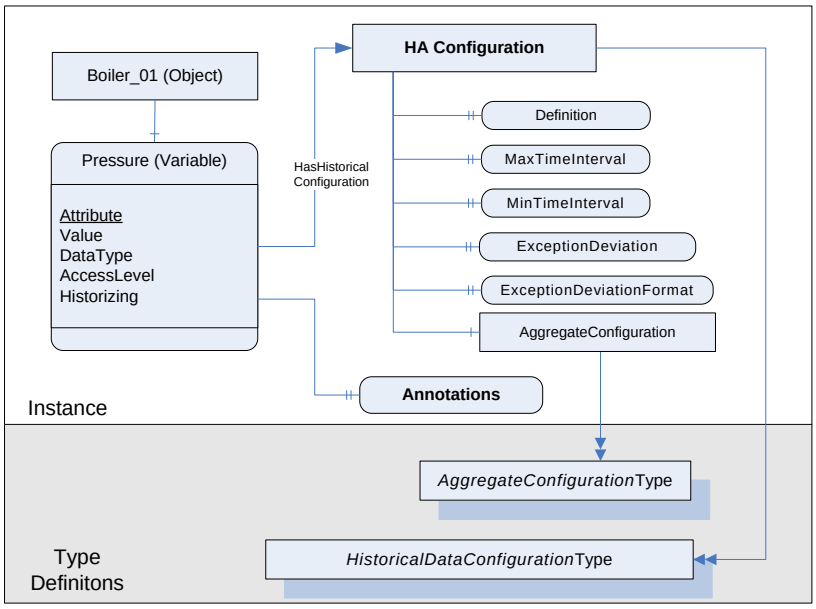

Figure 3 – Historical *Variable* with Historical Data Configuration and Annotations

Each *HistoricalDataNode* with history shall have the *Historizing Attribute* (see [UA Part 3]*)* defined and may *Reference* a *HistoricalAccessConfiguration Object*. In the case where the

HistoricalDataNode is itself a *Property* then the *HistoricalDataNode* inherits the values from the Parent of the *Property*.

Not every *Variable* in the *AddressSpace* might contain history data. To see if history data is available, a *Client* will look for the `HistoryRead/Write` states in the *AccessLevel Attribute* (see [UA Part 3] for details on use of this *Attribute).*

Figure 3~~Figure 3~~ only shows a subset of *Attributes* and *Properties*. Other *Attributes* that are defined for *Variables* in [UA Part 3], may also be available.

5.2.6 Attributes

This section lists the *Attributes* of *Variables* that have particular importance for historical data. They are specified in detail in [UA Part 3].

- AccessLevel
- Historizing

5.3 HistoricalEventNodes

5.3.1 General

The Historical *Event* model defines additional *Properties*. These descriptions also include required use cases for *HistoricalEventNodes.*

Historical access of *Events* uses an *EventFilter*. It is important to understand the differences between applying an *EventFilter* to current *Event Notifications*, and historical *Event* retrieval.

In real time monitoring *Events* are received via *Notifications* when subscribing to an *EventNotifier*. The *EventFilter* provides for the filtering and content selection of *Event Subscriptions*. If an *Event Notification* conforms to the filter defined by the *where* parameter of the *EventFilter*, then the *Notification* is sent to the *Client*.

In historical *Event* retrieval the *EventFilter* represents the filtering and content selection used to describe what parameters of *Events* are available in history. These may or may not include all the parameters of the real-time *Event*, i.e. not all fields available when the *Event* was generated may have been stored in history.

The *HistoricalEventFilter* may change over time so a *Client* may specify any field for any *EventType* in the *EventFilter*. If a field is not stored in the historical collection then the field is set to NULL when it is referenced in the *selectClause* or the *whereClause*.

5.3.2 HistoricalEventFilter Property

A *HistoricalEventNode* that has *Event* history available will provide the *Property*. This *Property* is formally defined in Table 7~~Table 7~~

Table 7 – Historical Events Properties

Name	Use	Data Type	Description
Standard Properties			
HistoricalEventFilter	M	EventFilter	A filter used by the Server to determine which HistoricalEventNode fields are available in history. It may also include a where clause that indicates the types of *Events* or restrictions on the *Events* that are available via the *HistoricalEventNode*

5.3.3 HistoricalEventNodes Address Space model

HistoricalEventNodes are *Objects* or *Views* in the *AddressSpace* that expose historical *Events.* These *Nodes* are identified via the *EventNotifier Attribute,* and provide some historical subset of the *Events* generated by the server.

Each *HistoricalEventNode* is represented by an *Object* or *View* with a specific set of *Attributes*. The *HistoricalEventFilter Property* specifies the fields available in the history.

Not every *Object* or *View* in the *AddressSpace* may be a *HistoricalEventNode*. To qualify as *HistoricalEventNodes*, a *Node* has to contain historical *Events*. To see if historical *Events* are available, a *Client* will look for the `HistoryRead/Write` states in the *EventNotifier Attribute*. See [UA Part 3] for details on use of this *Attribute.*

Figure 4~~Figure 4~~ illustrates the basic *AddressSpace* model of an *Event* that includes History.

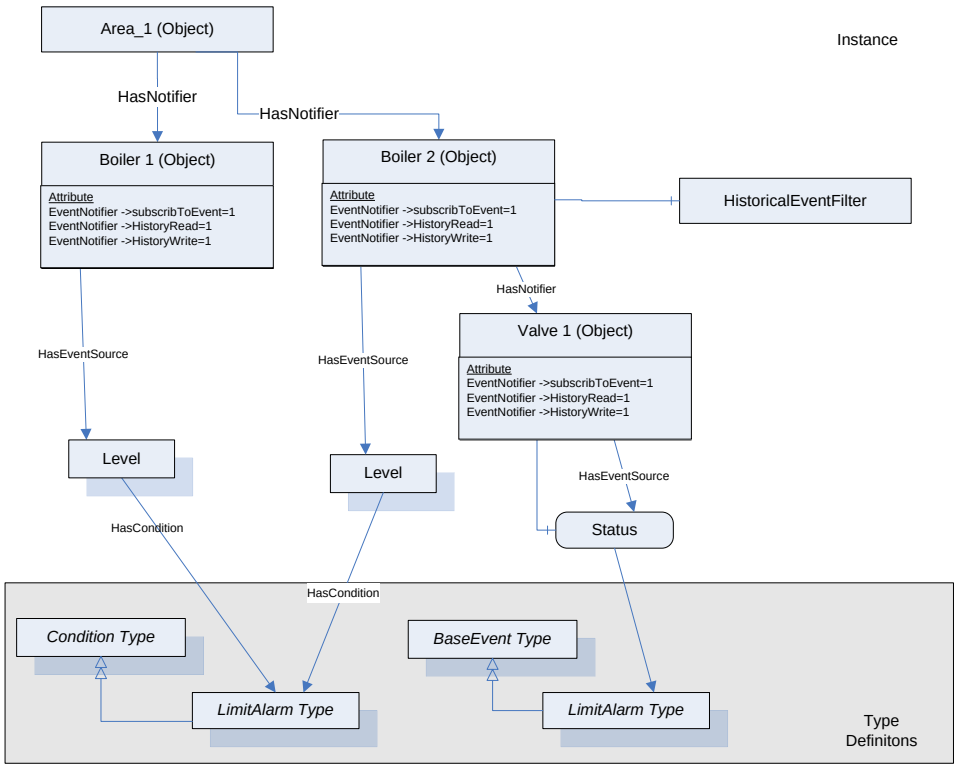

Figure 4 – Representation of an *Event* with History in the AddressSpace

5.3.4 HistoricalEventNodes Attributes

This section lists the *Attributes* of *Objects* or *Views* that have particular importance for historical *Events*. They are specified in detail in [UA Part 3]. The following *Attributes* are particularly important for *HistoricalEventNodes*.

- *EventNotifier*

The *EventNotifier Attribute* is used to indicate if the *Node* can be used to read and/or update historical *Events*.

5.4 Exposing Supported Functions and Capabilities

5.4.1 General

OPC UA servers can support several different functionalities and capabilities. The following standard *Objects* are used to expose these capabilities in a common fashion, and there are several standard defined concepts that can be extended by vendors. The *Objects* are outlined Figure 5~~Figure 5~~.

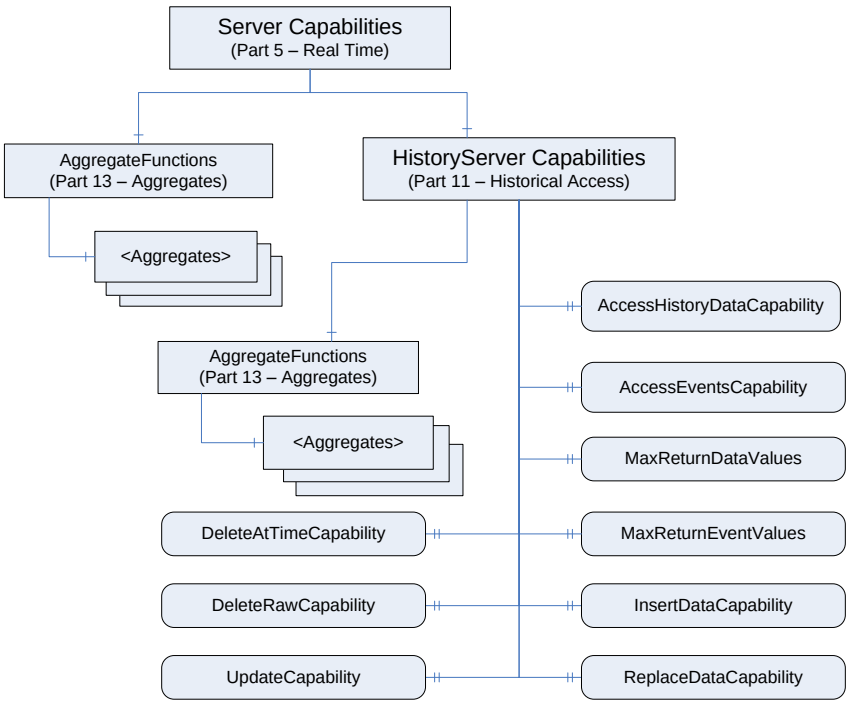

Figure 5 – Server and HistoryServer Capabilities

5.4.2 HistoryServerCapabilitiesType

The *ServerCapabilitiesType Objects* for any OPC UA Server supporting Historical Access must contain a *Reference* to a *HistoryServerCapabilitiesType Object*.

The content of this *BaseObjectType* is already defined by its type definition in [UA Part 5]. The *Object* extensions are formally defined in Table 8~~Table 8~~.

These properties are intended to inform *Client* of the general expected capabilities of the server. They do not guarantee that all capabilities will be available for all nodes. For example not all nodes will support *Events*, or in the case of an aggregating server where underlying servers may not support Insert or a particular *Aggregate*. In such cases the HistoryServerCapabilities *Property* would indicate the capability is supported, and the server would return appropriate *StatusCodes* for situations where the capability does not apply.

Table 8 – HistoryServerCapabilitiesType Definition

Attribute	Value				
BrowseName	HistoryServerCapabilitiesType				
IsAbstract	False				
References	**NodeClass**	**Browse Name**	**Data Type**	**Type Definition**	**Instantiation Rule**
HasProperty	Variable	AccessHistoryDataCapability	Boolean	PropertyType	Mandatory
HasProperty	Variable	AccessHistoryEventsCapability	Boolean	PropertyType	Mandatory
HasProperty	Variable	MaxReturnDataValues	UInt32	PropertyType	Mandatory
HasProperty	Variable	MaxReturnEventValues	UInt32	PropertyType	Mandatory
HasProperty	Variable	InsertDataCapability	Boolean	PropertyType	Mandatory
HasProperty	Variable	ReplaceDataCapability	Boolean	PropertyType	Mandatory
HasProperty	Variable	UpdateDataCapability	Boolean	PropertyType	Mandatory
HasProperty	Variable	DeleteRawCapability	Boolean	PropertyType	Mandatory
HasProperty	Variable	DeleteAtTimeCapability	Boolean	PropertyType	Mandatory
HasProperty	Variable	InsertEventCapability	Boolean	PropertyType	Mandatory
HasProperty	Variable	ReplaceEventCapability	Boolean	PropertyType	Mandatory
HasProperty	Variable	UpdateEventCapability	Boolean	PropertyType	Mandatory
HasProperty	Variable	DeleteEventCapability	Boolean	PropertyType	Mandatory
HasProperty	Variable	InsertAnnotationsCapability	Boolean	PropertyType	Mandatory
HasComponent	Object	AggregateFunctions	--	AggregateFunctions Type	Mandatory

All UA servers that support Historical access must include the HistoryServerCapabilities as part of its ServerCapabilities.

The *AccessHistoryDataCapability Variable* defines if the server supports access to historical data values. A value of True indicates the server supports access to history for *HistoricalDataNodes*, a value of False indicates the server does not support access to history for *HistoricalDataNodes*. The default value is False. At least one of *AccessHistoryDataCapability* or *AccessHistoryEventsCapability* must have a value of True for the server to be a valid OPC UA Server supporting Historical Access.

The *AccessHistoryEventCapability Variable* defines if the server supports access to historical *Events*. A value of True indicates the server supports access to history of events, a value of False indicates the server does not support access to history of *Events*. The default value is False. At least one of *AccessHistoryDataCapability* or *AccessHistoryEventsCapability* must have a value of True for the server to be a valid OPC UA Server supporting Historical Access.

The *MaxReturnDataValues Variable* defines maximum number of values that can be returned by the server for each *HistoricalDataNode* accessed during a request. A value of 0 indicates that the server forces no limit on the number of values it can return. It is valid for a server to limit the number of returned values and return a continuation point even if *MaxReturnValues* = 0. For example, it is possible that although the server does not impose any restrictions, the underlying system may impose a limit that the server is not aware of. The default value is 0.

Similarily, the MaxReturnEventValues specifies the maximum number of *Events* that a server can return for a HistoricalEventNode.

The *InsertDataCapability Variable* indicates support for the Insert capability. A value of True indicates the server supports the capability to insert new data values in history, but not overwrite existing values. The default value is False.

The *ReplaceDataCapability Variable* indicates support for the Replace capability. A value of True indicates the server supports the capability to replace existing data values in history, but will not insert new values. The default value is False.

The *UpdateDataCapability Variable* indicates support for the Update capability. A value of True indicates the server supports the capability to insert new data values into history if none exists, and replace values that currently exist. The default value is False.

The *DeleteRawCapability Variable* indicates support for the delete raw values capability. A value of True indicates the server supports the capability to delete raw data values in history. The default value is False.

The *DeleteAtTimeCapability Variable* indicates support for the delete at time capability. A value of True indicates the server supports the capability to delete a data value at a specified time. The default value is False.

The *InsertEventCapability Variable* indicates support for the Insert capability. A value of True indicates the server supports the capability to insert new *Events* in history. An insert is not a replace. The default value is False.

The *ReplaceEventCapability Variable* indicates support for the Replace capability. A value of True indicates the server supports the capability to replace existing *Events* in history. A replace is not an insert. The default value is False.

The *UpdateEventCapability Variable* indicates support for the Update capability. A value of True indicates the server supports the capability to insert new *Events* into history if none exists, and replace values that currently exist. The default value is False.

The *DeleteEventCapability Variable* indicates support for the deletion of events capability. A value of True indicates the server supports the capability to delete *Events* in history. The default value is False.

The *InsertAnnotationCapability Variable* indicates support for *Annotations*. A value of True indicates the server supports the capability to insert Annotations. Some *Servers* that support Inserting of *Annotations* will also support editing and deleting of *Annotations*. The default value is False.

AggregateFunctions is an entry point to browse to all *Aggregate* capabilities supported by the server for Historical Access. All *HistoryAggregates* supported by the *Server* should be able to be browsed starting from this *Object*. *Aggregates* are defined in [UA Part 13]~~[UA Part 13]~~. If the *Server* does not support *Aggregates* the *Folder* is left empty.

5.5 History DataType definitions

5.5.1 Annotation DataType

This *DataType* describes *Annotation* information for the history data items. Its elements are defined in Table 9~~Table 9~~.

Table 9 – Annotation Structure

Name	Type	Description
Annotation	Structure	
message	String	*Annotation* message or text
username	String	The user that added the *Annotation*, as supplied by underlying system.
annotationTime	UtcTime	The time the *Annotation* was added. This will probably be different than the *SourceTimestamp*

5.6 Historical Audit Events

5.6.1 General

AuditEvents are generated as a result of an action taken on the server by a *Client* of the server. For example, in response to a *Client* issuing a write to a *Variable*, the server would generate an *AuditEvent* describing the *Variable* as the source and the user and *Client Session* as the initiators of the *Event*. Not all servers support auditing, but if a server supports auditing then it must support audit events as described in this section. *Profiles* can be used to determine if a server supports auditing see [UA Part 13]~~[UA Part 7]~~Servers must generate

Events of the *AuditHistoryUpdateEventType* or a sub-type of this type for all invocations of the *HistoryUpdate Service* on any *HistoricalNode*. See [UA Part 3] and [UA Part 5] for details on the *AuditHistoryUpdateEventType* model. In the case where the *HistoryUpdate Service* is invoked to insert Historical *Events*, the *AuditHistoryEventUpdateEventType* Event must include the EventId of the inserted *Event* and a description that indicates that the *Event* was inserted. In the case where the HistoryUpdate *Service* is invoked to delete records, the *AuditHistoryDeleteEventType* or one of its sub-types must be generated. See Section 6.7 for details on updating historical data or *Events*.

In particular using the Delete raw or Delete modified functionality must generate an AuditHistoryRawModifyDeleteEventType *Event* or a sub-type of it. Using the Delete at time functionality must generate an AuditHistoryAtTimeDeleteEventType *Event* or a sub-type of it. Using the Delete *Event* functionality must generate an AuditHistoryEventDeleteEventType *Event* or a sub-type of it. All other updates must follow the guidelines provided in the *AuditHistoryUpdateEventType* Model.

5.6.2 AuditHistoryEventUpdateEventType

This is a subtype of AuditHistoryUpdateEventType and is used for categorization of History *Event* update related *Events*. This type follows all behaviour of its parent type. Its representation in the *AddressSpace* is formally defined in Table 10~~Table 11~~

Table 10~~11~~ – AuditHistoryEventUpdateEventType Definition

Attribute	**Value**				
BrowseName	AuditHistoryEventUpdateEventType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Inherit the *Properties* of the *AuditHistoryUpdateEventType* defined in [UA Part 3], i.e. it has *HasProperty References* to the same *Nodes*.					
HasProperty	Variable	UpdatedNode	NodeId	PropertyType	Mandatory
HasProperty	Variable	PerformInsertReplace	Enumeration	PropertyType	Mandatory
HasProperty	Variable	Filter	EventFilter	PropertyType	Mandatory
HasProperty	Variable	NewValues	HistoryEventNotification[]	PropertyType	Mandatory
HasProperty	Variable	OldValues	HistoryEventNotification[]	PropertyType	Mandatory

This *EventType* inherits all *Properties* of the *AuditHistoryUpdateEventType*. Their semantic is defined in [UA Part 5].

The *UpdateNode* identifies the *Attribute* that was written on the *SourceNode*.

The *PerformInsertReplace* enumeration reflect the parameter on the *Service* call

The *Filter* reflects the *Event* filter passed on the call to select the *Events* that are to be updated.

The *NewValues* identifies the value that was written to the *Event*.

The *OldValues* identifies the value that the *Event* contained before the write. It is acceptable for a server that does not have this information to report a null value. And in the case of an insert it is expected to be a null value

Both the *NewValue* and the *OldValue* will contain an *Event* with the appropriate fields, each with appropriately encoded values.

5.6.3 AuditHistoryValueUpdateEventType

This is a subtype of AuditHistoryUpdateEventType and is used for categorization of history value update related *Events*. This type follows all behaviour of its parent type. Its representation in the *AddressSpace* is formally defined in Table 11~~Table 12~~.

Table 11~~12~~ – AuditHistoryValueUpdateEventType Definition

Attribute	Value				
BrowseName	AuditHistoryValueUpdateEventType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Inherit the *Properties* of the *AuditHistoryUpdateEventType* defined in [UA Part 3], i.e. it has *HasProperty References* to the same *Nodes*.					
HasProperty	Variable	UpdatedNode	NodeId	PropertyType	Mandatory
HasProperty	Variable	PerformInsertReplace	Enumeration	PropertyType	Mandatory
HasProperty	Variable	NewValues	DataValue[]	PropertyType	Mandatory
HasProperty	Variable	OldValues	DataValue[]	PropertyType	Mandatory

This *EventType* inherits all *Properties* of the *AuditHistoryUpdateEventType*. Their semantic is defined in [UA Part 5].

The *UpdatedNode* identifies the *Attribute* that was written on the *SourceNode*.

The *PerformInsertReplace* enumeration reflect the parameter on the *Service* call

The *NewValue* identifies the value that was written to the *Event*.

The *OldValue* identifies the value that the *Event* contained before the write. It is acceptable for a server that does not have this information to report a null value. And in the case of an insert it is expected to be a null value

Both the *NewValue* and the *OldValue* will contain a value in the *DataType* and encoding used for writing the value.

5.6.4 AuditHistoryDeleteEventType

This is a subtype of AuditHistoryUpdateEventType and is used for categorization of history delete related *Events*. This type follows all behaviour of its parent type. Its representation in the *AddressSpace* is formally defined in Table 12~~Table 13~~.

Table 12~~13~~ – AuditHistoryDeleteEventType Definition

Attribute	Value				
BrowseName	AuditHistoryDeleteEventType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Inherit the *Properties* of the *AuditHistoryUpdateEventType* defined [UA Part 3], i.e. it has *HasProperty References* to the same *Nodes*.					
HasProperty	Variable	UpdatedNode	NodeId	PropertyType	Mandatory
HasSubtype	ObjectType	AuditHistoryRawModifyDeleteEventType			
HasSubtype	ObjectType	AuditHistoryAtTimeDeleteEventType			
HasSubtype	ObjectType	AuditHistoryEventDeleteEventType			

This *EventType* inherits all *Properties* of the *AuditUpdateEventType*. Their semantic is defined in [UA Part 5].

The *NodeId* identifies the *NodeId* that was used for the delete operation.

5.6.5 AuditHistoryRawModifyDeleteEventType

This is a subtype of AuditHistoryDeleteEventType and is used for categorization of history delete related *Events*. This type follows all behaviour of its parent type. Its representation in the *AddressSpace* is formally defined in Table 13~~Table 14~~.

Table 13~~14~~ – AuditHistoryRawModifyDeleteEventType Definition

Attribute	Value				
BrowseName	AuditHistoryRawModifyDeleteEventType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Inherit the *Properties* of the *AuditHistoryDeleteEventType* defined in Clause Table 12~~Table 13~~, i.e. it has *HasProperty References* to the same *Nodes*.					
HasProperty	Variable	IsDeleteModified	Boolean	PropertyType	Mandatory
HasProperty	Variable	StartTime	UtcTime	PropertyType	Mandatory
HasProperty	Variable	EndTime	UtcTime	PropertyType	Mandatory
HasProperty	Variable	OldValues	DataValue[]	PropertyType	Mandatory

This *EventType* inherits all *Properties* of the *AuditHistoryDeleteEventType*. Their semantic is defined in Clause 5.6.4.

The *isDeleteModified* reflect the isDeleteModified parameter of the call

The *StartTime* reflect the starting time parameter of the call.

The *EndTime* reflect the ending time parameter of the call.

The *OldValues* identifies the value that history contained before the delete. A server should report all deleted values. It is acceptable for a server that does not have this information to report a null value. The *OldValues* will contain a value in the *DataType* and encoding used for writing the value.

5.6.6 AuditHistoryAtTimeDeleteEventType

This is a subtype of AuditHistoryDeleteEventType and is used for categorization of history delete related *Events*. This type follows all behaviour of its parent type. Its representation in the *AddressSpace* is formally defined in Table 14~~Table 15~~.

Table 14~~15~~ – AuditHistoryAtTimeDeleteEventType Definition

Attribute	Value				
BrowseName	AuditHistoryAtTimeDeleteEventType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Inherit the *Properties* of the *AuditHistoryDeleteEventType* defined in Clause Table 12~~Table 13~~, i.e. it has *HasProperty References* to the same *Nodes*.					
HasProperty	Variable	ReqTimes	UtcTime[]	PropertyType	Mandatory
HasProperty	Variable	OldValues	DataValues[]	PropertyType	Mandatory

This *EventType* inherits all *Properties* of the *AuditHistoryDeleteEventType*. Their semantic is defined in Clause 5.6.7.

The *ReqTimes* reflect the request time parameter of the call.

The *OldValues* identifies the value that history contained before the delete. A server should report all deleted values. It is acceptable for a server that does not have this information to report a null value. The *OldValues* will contain a value in the *DataType* and encoding used for writing the value.

5.6.7 AuditHistoryEventDeleteEventType

This is a subtype of AuditHistoryDeleteEventType and is used for categorization of history delete related *Events*. This type follows all behaviour of its parent type. Its representation in the *AddressSpace* is formally defined in Table 15~~Table 16~~.

Table 15~~16~~ – AuditHistoryEventDeleteEventType Definition

Attribute	Value				
BrowseName	AuditHistoryEventDeleteEventType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Inherit the *Properties* of the *AuditHistoryDeleteEventType* defined in Clause Table 12~~Table 13~~, i.e. it has *HasProperty References* to the same *Nodes*.					
HasProperty	Variable	EventIds	ByteString[]	PropertyType	Mandatory
HasProperty	Variable	OldValues	HistoryEventNotification[]	PropertyType	Mandatory

This *EventType* inherits all *Properties* of the *AuditHistoryDeleteEventType*. Their semantic is defined in Clause 5.6.4

The *EventIds* reflect the EventIds parameter of the call

The *OldValues* identifies the value that history contained before the delete. A server should report all deleted values. It is acceptable for a server that does not have this information to report a null value. The *OldValues* will contain will contain an *Event* with the appropriate fields, each with appropriately encoded values.

6 Historical Access specific usage of Services

6.1 General

[UA Part 4] specifies all *Services* needed for OPC UA Historical Access. In particular:

- The *Browse Service Set* or *Query Service Set* to detect *HistoricalNodes* and their configuration.
- The *HistoryRead* and *HistoryUpdate Services* of the *Attribute Service Set* to read and update history of *HistoricalNodes*.

6.2 Historical Nodes StatusCodes

6.2.1 Overview

This section defines additional codes and rules that apply to the *StatusCode* when used for *HistoricalNodes*.

The general structure of the *StatusCode* is specified in [UA Part 4]. It includes a set of common operational result codes which also apply to historical data and/or *Events*.

6.2.2 Operation level result codes

In OPC UA Historical Access the *StatusCode* is used to indicate the conditions under which a *Value* or *Event* was stored, and thereby can be used as an indicator its usability. Due to the nature of historical data and/or *Events*, additional information beyond the basic quality and call result code needs to be conveyed to the *Client*. For example, whether the value is actually stored in the data repository, was the result *Interpolated*, were all data inputs to a calculation of good quality, etc.

In the following, Table 16~~Table 17~~ contains codes with *Bad* severity indicating a failure; Table 17~~Table 18~~ contains *Good* (success) codes.

It is Important to note, that these are the codes that are specific for OPC UA Historical Access and supplement the codes that apply to all types of data and are therefore defined in [UA Part 4] , [UA Part 8] and [UA Part 13].

Table 1617 – Bad operation level result codes

Comment [PEH1]: These code all deal wi data type history information – I think there ar some event related code that should be listed make it easier for developers

Symbolic Id	Description
Bad_NoData	No data exists for the requested time range or *Event* filter
Bad_BoundNotFound	No data found to provide upper or lower bound value.
Bad_BoundNotSupported	Bounding values are not applicable or the server has reached its search limit and will not return a bound.
Bad_DataLost	Data is missing due to collection started / stopped / lost.
Bad_DataUnavailable	Expected data is unavailable for the requested time range due to an un-mounted volume, an off-line historical collection, or similar reason for temporary unavailability.
Bad_EntryExists	The data or *Event* was not successfully inserted because a matching entry exists.
Bad_NoEntryExists	The data or *Event* was not successfully updated because no matching entry exists.
Bad_TimestampNotSupported	The *Client* requested history using a timestamp format the server does not support (i.e. requested *ServerTimestamp* when server only supports *SourceTimestamp*)
Bad_ArgumentInvalid	One or more arguments are invalid or missing.
Bad_AggregateListMismatch	The list of aggregates does not have the same length as the list of operations.
Bad_AggregateConfigurationRejected	The server does not support the specified *AggregateConfiguration* for the *Node*.
Bad_AggregateNotSupported	The specified aggregate is not valid for the specified node.

Table 1718 – Good operation level result codes

Symbolic Id	Description
Good_NoData	No data exists for the requested time range or *Event* filter.
Good_EntryInserted	The data or *Event* was successfully inserted into the historical database
Good_EntryReplaced	The data or *Event* field was successfully replaced in the historical database
Good_DataIgnored	The Event field was ignored and was not inserted into the historical database.

It may be noted that there are both Good and Bad Status codes that deal with cases of no or missing data. In general Good_NoData is used for cases where no data was found when performing a simple 'Read' request. Bad_NoData is used in cases where some action is requested on an interval, and no data could be found. The distinction exists since if users are attempting an action on a given interval, they would expect data to exist, or would at least wished to be notified that the requested action could not be performed.

Good_NoData is returned for cases such as…

- ReadEvents where startTime=endTime
- ReadEvent data is requested and does not exist
- ReadRaw where data is requested and does not exist

Bad_NoData is returned for cases such as…

- ReadEvent data is requested and underlying historian does not support requested field.
- ReadProcessed where data is requested and does not exist
- Any Delete requests where data does not exist.

The above use cases are illustrative examples. Detailed explanations on when each status code is returned are found in sections 6.4 and 6.7

6.2.3 Semantics changed

The *StatusCode* in addition contains an informational bit called *Semantics Changed*. (See [UA Part 4])

UA Servers that implement OPC UA Historical Access should not set this Bit; rather propagate the *StatusCode* which has been stored in the data repository. *Client* should be aware that the returned data values may have this bit set.

6.3 Continuation Points

The *continuationPoint* parameter in the *HistoryRead Service* is used to mark a point from which to continue the read if not all values could be returned in one response. The value is opaque for the *Client* and is only used to maintain the state information for the *Server* to continue from. For *HistoricalDataNode* requests, a *Server* may use the timestamp of the last returned data item if the timestamp is unique. This can reduce the need in the *Server* to store state information for the continuation point.

The *Client* specifies the maximum number of results per operation in the request *Message*. A *Server* shall not return more than this number of results but it may return fewer results. The *Server* allocates a *ContinuationPoint* if there are more results to return. The *Server* shall always return at least one result if it returns a *ContinuationPoint*. The *Server* may return fewer results due to buffer issues or other internal constraints. It may also be required to return a *continuationPoint* due to *HistoryRead* parameter constraints. For additional discussions regarding *ContinuationPoints* and *HistoryRead* please see the individual extensible *historyReadDetails* parameter sections.

If the *Client* specifies a *ContinuationPoint*, then the *HistoryReadDetails* parameter is ignored, because it does not make sense to request different HistoryReadDetails parameters when continuing from a previous call. It is permissible to change the *dataEncoding* parameter with each request.

If the *Client* specifies a *ContinuationPoint* that does not correspond with last returned *ContinuationPoint* from the *Server*, then the *Server* shall return a *Bad_ContinuationPointInvalid* error.

If the *releaseContinuationPoints* parameter is set in the request the *Server* shall not return any data and shall release all *ContinuationPoints* passed in the request. If the *ContinuationPoint* for an operation is missing or invalid the *StatusCode* for the operation shall be *Bad_ContinuationPointInvalid.*

6.4 HistoryReadDetails parameters

6.4.1 Overview

The *HistoryRead Service* defined in [UA Part 4] can perform several different functions. The *historyReadDetails* parameter is an *Extensible Parameter* that specifies which function to perform and the details that are specific to that function. See [UA Part 4] for the definition of *Extensible Parameter*. Table 18~~Table 19~~ lists the symbolic names of the valid Extensible Parameter structures. Some structures will perform different functions based on the setting of its associated parameters. For simplicity a functionality of each structure is listed. For example text such as ‘using the Read modified functionality’ refers to the function the *HistoryRead Service* performs using the *Extensible Parameter* structure ReadRawModifiedDetails with the isReadModified Boolean parameter set to TRUE.

Table 18~~19~~ – HistoryReadDetails parameterTypeIds

Symbolic Name	Functionality	Description

ReadEventDetails	Read event	This structure selects a set of *Events* from the history database by specifying a filter and a time domain for one or more *Objects* or *Views*. See Clause 6.4.2.1. When this parameter is specified the *Server* returns a *HistoryEvent* structure for each operation (See Clause 6.5.4).
ReadRawModifiedDetails	Read raw	This structure selects a set of values from the history database by specifying a time domain for one or more *Variables*. See Clause 6.4.3.1 When this parameter is specified the *Server* returns a *HistoryData* structure for each operation (See Clause 6.5.2)
ReadRawModifiedDetails	Read modified	This parameter selects a set of modified values from the history database by specifying a time domain for one or more *Variables*. See Clause 6.4.3.1. When this parameter is specified the *Server* returns a *HistoryModifiedData* structure for each operation (See Clause 6.5.3).
ReadProcessedDetails	Read processed	This structure selects a set of *Aggregate* values from the history database by specifying a time domain for one or more *Variables*. See Clause 6.4.4.1. When this parameter is specified the *Server* returns a *HistoryData* structure for operation (See Clause 6.5.2)
ReadAtTimeDetails	Read at time	This structure selects a set of raw or interpolated values from the history database by specifying a series of timestamps for one or more *Variables*. See Clause 6.4.5.1. When this parameter is specified the *Server* returns a *HistoryData* structure for each operation (See Clause 6.5.2)

6.4.2 ReadEventDetails structure

6.4.2.1 ReadEventDetails structure details

Table 19~~Table 20~~ defines the *ReadEventDetails* structure. This parameter is only valid for *Objects* that have the *EventNotifier Attribute* set to TRUE (See [UA Part 3]). Two of the three parameters, numValuesPerNode, startTime, and endTime must be specified.

Table 19~~20~~ – ReadEventDetails

Name	Type	Description
ReadEventDetails	Structure	Specifies the details used to perform an *Event* history read.
numValuesPerNode	Counter	The maximum number of values returned for any *Node* over the time range. If only one time is specified, the time range must extend to return this number of values. The default value of 0 indicates that there is no maximum.
startTime	UtcTime	Beginning of period to read. The default value of *DateTime.MinValue* indicates that the startTime is Unspecified.
endTime	UtcTime	End of period to read. The default value of *DateTime.MinValue* indicates that the endTime is Unspecified.
Filter	EventFilter	A filter used by the *Server* to determine which HistoricalEventNode should be included. This parameter must be specified and at least one *EventField* is required. The *EventFilter* parameter type is an extensible parameter type. It is defined and used in the same manner as defined for monitored data items which are specified [UA Part 4]. This filter also specifies the *EventFields* that are to be returned as part of the request.

6.4.2.2 Read Event functionality

ReadEventDetails structure is used to read the *Events* from the history database for the specified time domain for one or more HistoricalEventNodes. The *Events* are filtered based on the filter structure provided. This filter includes the *EventFields* that are to be returned. For a complete description of filter refer to [UA Part 4].

The time domain of the request is defined by startTime, endTime, and numValuesPerNode; at least two of these must be specified. If endTime is less than startTime, or endTime and numValuesPerNode alone are specified, the data will be returned in reverse order, with later data coming first, as if time were flowing backward. If all three are specified, the call shall return up to numValuesPerNode results going from startTime to endTime, in either ascending or descending order depending on the relative values of startTime and endTime. If numValuesPerNode is 0, then all the values in the range are returned. The default value is used to indicate when startTime, endTime or numValuesPerNode is not specified.

It is specifically allowed for the startTime and the endTime to be identical. This allows the *Client* to request the *Event* at a single instance in time. When the startTime and endTime are

identical, time is presumed to be flowing forward. If no data exists at the time specified then the server must return the *Good_NoData StatusCode*.

If a startTime, endTime and numValuesPerNode are all provided, than if more than numValuesPerNode *Events* exist within that time range for a given node, only numValuesPerNode *Events* per Node are returned along with a *continueationPoint.* When a *continuationPoint* is returned, *Client* wanting the next numValuesPerNode values should call HistoryRead again with the *continuationPoint*.

For an interval in which no data exists, the corresponding *StatusCode* shall be *Good_NoData.*

The filter parameter is used to determine which historical *Events* and their corresponding fields are returned. It is possible that the fields of an *EventType* are available for real time updating, but not available from the historian. In this case a *StatusCode* value will be returned for any *Event* field that cannot be returned. The value of the *StatusCode* must be *Bad_NoData*.

If the requested timestamp format is not supported for a *Node*, the operation shall return the *Bad_TimestampNotSupported StatusCode*. When reading *Events* this only applies to *Event* fields that are of type *DataValue*.

6.4.3 ReadRawModifiedDetails structure

6.4.3.1 ReadRawModifiedDetails structure details

Table 20~~Table 21~~ defines the *ReadRawDetails* structure. Two of the three parameters, numValuesPerNode, startTime, and endTime must be specified.

Table 20~~21~~ – ReadRawModifiedDetails

Name	Type	Description
ReadRawModifiedDetails	Structure	Specifies the details used to perform a “raw” or “modified” history read.
isReadModified	Boolean	TRUE for Read Modified functionality, FALSE for Read Raw functionality. Default value is FALSE.
startTime	UtcTime	Beginning of period to read. Set to default value of *DateTime.MinValue* if no specific start time is specified.
endTime	UtcTime	End of period to read. Set to default value of *DateTime.MinValue* if no specific end time is specified.
numValuesPerNode	Counter	The maximum number of values returned for any *Node* over the time range. If only one time is specified, the time range must extend to return this number of values. The default value 0 indicates that there is no maximum.
returnBounds	Boolean	A Boolean parameter with the following values : TRUE bounding values should be returned FALSE all other cases.

6.4.3.2 Read raw functionality

When this structure is used for reading *Raw* Values (isReadModified is set to FALSE); it reads the values, qualities, and timestamps from the history database for the specified time domain for one or more *HistoricalDataNodes*. This parameter is intended for use by *Client* wanting the actual data saved within the historian. The actual data may be compressed or may be all data collected for the item depending on the historian and the storage rules invoked when the item values were saved. When returnBounds is TRUE, the bounding values for the time domain are returned. The optional bounding values are provided to allow *Client* to interpolate values for the start and end times when trending the actual data on a display.

The time domain of the request is defined by startTime, endTime, and numValuesPerNode; at least two of these must be specified. If endTime is less than startTime, or endTime and numValuesPerNode alone are specified, the data will be returned in reverse order, with later data coming first, as if time were flowing backward. If all three are specified, the call shall return up to numValuesPerNode results going from startTime to endTime, in either ascending or descending order depending on the relative values of startTime and endTime. If numValuesPerNode is 0, then all the values in the range are returned. A default value of *DateTime.MinValue* [UA Part 6] is used to indicate when startTime or endTime is not specified.

Comment [MAR2]: I recently became aw of a significant efficiency issue related to this functionality, especially reading raw data. It m be too late in the process to change this, but wanted to discuss it and see what everyone thinks.

There is only one time range specified for all the nodes that are passed to the service. In interactive clients that are reading raw data, l a trend client, this can cause inefficiencies and/or great complexity in the client. The problem is that the raw data changes at different rates. When the trend chart scrolls, t client needs to fill in what data is missing. If th client has some data cached from previous calls, the data it needs will be different for eac tag. As it is now, the client has to look at all o the tags that need data. Try to figure out whic ones need the same time range so that it can match those up in one read raw call and then make seperate calls for all tags that do not match up thus creating more round trips to th server. As I said this is complex and inefficien

I propose that we create an inline array of structures that contain the startTime, endTim and returnBounds (and possibly numValuesPerNode if we think that makes sense) where there is one for each NodeId ju as there are for aggregates in the read processed function.

If we do not want to change the ReadRawModifiedDetails sturcture this late, perhaps we could create a ReadRawModifiedDetails2 structure. This wo have the added benefit of letting the client us whichever type of call is most convient.

EJM: Will be added as a Mantis issue for 2.0

It is specifically allowed for the startTime and the endTime to be identical. This allows the *Client* to request just one value. When the startTime and endTime are identical, time is presumed to be flowing forward. It is specifically not allowed for the server to return a *Bad_ArgumentInvalid StatusCode* if the requested time domain is outside of the server's range. Such a case shall be treated as an interval in which no data exists.

If a startTime, endTime and numValuesPerNode are all provided, than if more than numValuesPerNode values exist within that time range for a given *Node*, only numValuesPerNode values per *Node* are returned along with a *continueationPoint*. When *a continuationPoint* is returned, *Client* wanting the next numValuesPerNode values should call ReadRaw again with the *continuationPoint* set.

If bounding values are requested and a non-zero numValuesPerNode was specified, any bounding values returned are included in the numValuesPerNode count. If numValuesPerNode is 1, then only the start bound is returned (the End bound if reverse order is needed). If numValuesPerNode is 2, the start bound and the first data point is returned (the End bound if reverse order is needed).

When bounding values are requested and no bounding value is found, the corresponding *StatusCode* entry will be set to *Bad_BoundNotFound*, a timestamp equal to the start or end time, as appropriate, and a value of Null. How far back or forward to look in history for bounding values is server dependent.

For an interval in which no data exists, if bounding values are not requested, the corresponding *StatusCode* must be *Good_NoData*. If bounding values are requested and one or both exist, the result code returned is Success and the bounding value(s) are returned.

For cases where there are multiple values for a given timestamp, all but the most recent are considered to be Modified values and the server must return the most recent value. If the server returns a value which hides other values at a timestamp then it must set the *ExtraData* bit in the *StatusCode* associated with that value. If the Server contains additional information regarding a value then the ExtraData bit shall also be set. This it indicates that ModifiedValues are available for retrieval, see 6.4.3.3

If the requested timestamp format is not supported for a *Node*, the operation shall return the *Bad_TimestampNotSupported StatusCode*.

6.4.3.3 Read modified functionality

When this structure is used for reading *Modified* Values (isReadModified is set to TRUE); it reads the values, *StatusCodes*, timestamps, modification type, theuser identifier, and the timestamp of the modification from the history database for the specified time domain for one or more *HistoricalDataNodes*. If there are multiple replaced values the server must return all of them. The updateType specifies what value is returned in the modification record. If the updateType is INSERT the value is the new value that was inserted. If the updateType is anything else the value is the old value that was changed.

The purpose of this function is to read values from history that have been *Modified.* The returnBounds parameter must be set to FALSE for this case otherwise the server returns a *Bad_ArgumentInvalid StatusCode*.

The domain of the request is defined by startTime, endTime, and numValuesPerNode; at least two of these must be specified. If endTime is less than startTime, or endTime and numValuesPerNode alone are specified, the data shall be returned in reverse order, with later data coming first. If all three are specified, the call shall return up to numValuesPerNode results going from *StartTime* to *EndTime*, in either ascending or descending order depending on the relative values of *StartTime* and *EndTime*. If more than *numValuesPerNode* values exist within that time range for a given *Node*, only *numValuesPerNode* values per *Node* are returned along with a *continuationPoint*. When *a continuationPoint* is returned, *Client* wanting the next *numValuesPerNode* values should call ReadRaw again with the *continuationPoint* set. If numValuesPerNode is 0, then all the values in the range are returned. If the Server cannot return all modified values for a given timestamp in a single response it shall return modified values with the same timestamp in subsequent calls.

If a value has been modified multiple times, all values for the time are returned. This means that a timestamp can appear in the array more than once. The order of the returned values

with the same timestamp should be from most recent to oldest modification timestamp, if startTime is less than or equal to endTime. If endTime is less than startTime, the order of the returned values will be from oldest modificfication timestamp to most recent. It is server dependent whether multiple modifications are kept or only the most recent.

A server does not have to create a modification record for data when it is first added to the historical collection. If it does then it shall set the ExtraData bit and the *Client* can read the modification record using a ReadModified call. . If the data is subsequently modified the server shall create a second modification record which is returned along with the original modification record whenever a *Client* uses the ReadModified call if the Server supports multiple modification records per timestamp.

If the requested timestamp format is not supported for a *Node*, the operation shall return the *Bad_TimestampNotSupported StatusCode*.

6.4.4 ReadProcessedDetails structure

6.4.4.1 ReadProcessedDetails structure details

Table 21~~Table 22~~ defines the structure of the ReadProcessedDetails structure.

Table 21~~22~~ – ReadProcessedDetails

Name	Type	Description
ReadProcessedDetails	Structure	Specifies the details used to perform a “processed” history read
startTime	UtcTime	Beginning of period to read.
endTime	UtcTime	End of period to read.
resampleInterval	Duration	*Interval* between returned *Aggregate* values. The value 0 indicates that there is no *Interval* defined.
aggregateType[]	NodeId	The *NodeId* of the HistoryAggregate object that indicates the list of *Aggregates* to be used when retrieving processed history. See [UA Part 13]~~[UA Part 13]~~ for details.
aggregateConfiguration	Aggregate Configuration	*Aggregate* configuration structure
useSeverCapabilitiesDefaults	Boolean	As described in [UA Part 4].
TreatUncertainAsBad	Boolean	As described in [[UA Part 13]~~[UA Part 13]~~
PercentDataBad	UInt8	As described in [UA Part 13]~~[UA Part 13]~~
PercentDataGood	UInt8	As described in [[UA Part 13]~~[UA Part 13]~~
SteppedSlopedExtrapolation	Boolean	As described in [UA Part 13]~~[UA Part 13]~~

See [UA Part 13]~~[UA Part 13]~~ for details on possible *NodeId* values for the *HistoryAggregateType* parameter.

6.4.4.2 Read processed functionality

This structure is used to compute *Aggregate* values, qualities, and timestamps from data in the history database for the specified time domain for one or more *HistoricalDataNodes*. The time domain is divided into *Intervals* of duration resampleInterval. The specified *Aggregate*Type is calculated for each *Interval*beginning with startTime by using the data within the next resampleInterval.

For example, this function can provide hourly statistics such as Maximum, Minimum and Average for each item during the specified time domain when resampleInterval is 1 hour.

The domain of the request is defined by startTime, endTime, and resampleInterval. All three must be specified. If endTime is less than startTime, the data shall be returned in reverse order, with later data coming first. If startTime and endTime are the same, the server shall return *Bad_ArgumentInvalid*, as there is no meaningful way to interpret such a case.

The aggregateType[] parameter allows *Client* to request multiple *Aggregate* calculations per requested NodeId. If multiple *Aggregates* are requested, then a corresponding number of entries are required in the NodesToRead array.

For example to request Min *Aggregate* for *NodeId* FIC101, FIC102 and both Min and Max *Aggregates* for *NodeId* FIC103 would require *NodeId* FIC103 to appear twice in the NodesToRead array request parameter.

aggregateType[]	NodesToRead[]
Min	FIC101
Min	FIC102
Min	FIC103
Max	FIC103

If the array of *Aggregates* does not match the array of NodesToRead, the Server shall return a *StatusCode* of *Bad_AggregateListMismatch.*

The aggregateConfiguration parameter allows *Client* to override the *Aggregate* configuration settings supplied by the AggregateConfiguration *Object* on a per call basis. See [UA Part 13]~~[UA Part 13]~~ for more information on *Aggregate* configurations. If the Server does not support the ability to override the *Aggregate* configuration settings it shall return a *StatusCode* of *Bad_ AggregateConfigurationRejected.*If the *Aggregate* is not valid for the node then the *StatusCode* shall be *Bad_AggregateNotSupported.*

The values used in computing the *Aggregate* for each *Interval* shall include any value that falls exactly on the timestamp beginning the *Interval*, but shall not include any value that falls directly on the timestamp ending the *Interval*. Thus, each value shall be included only once in the calculation. If the time domain is in reverse order, we consider the later timestamp to be the one beginning the subinterval, and the earlier timestamp to be the one ending it. Note that this means that simply swapping the start and end times will not result in getting the same values back in reverse order, as the *Interval*being requested in the two cases are not the same.

Refer to [UA Part 13]~~[UA Part 13]~~ for handling of *Aggregate* specific cases.

6.4.5 ReadAtTimeDetails structure

6.4.5.1 ReadAtTimeDetails structure details

Table 22~~Table 23~~ defines the ReadAtTimeDetails structure.

Table 22~~23~~ – ReadAtTimeDetails

Name	Type	Description
ReadAtTimeDetails	Structure	Specifies the details used to perform an “at time” history read
reqTimes []	UtcTime	The entries define the specific timestamps for which values are to be read.

6.4.5.2 Read at time functionality

The ReadAtTimeDetails structure reads the values and qualities from the history database for the specified timestamps for one or more *HistoricalDataNodes*. This function is intended to provide values to correlate with other values with a known timestamp. For example, a *Client* may need to read the values of sensors when lab samples were collected.

The order of the values and qualities returned shall match the order of the time stamps supplied in the request.

When no value exists for a specified timestamp, a value shall be *Interpolated* from the surrounding values to represent the value at the specified timestamp. The interpolation will follow the same rules as the standard *Interpolated Aggregate* as outlined in [UA Part 13]~~[UA Part 13]~~

If a value is found for the specified timestamp, the server will set the *StatusCode InfoBits* to be *Raw*. If the value is *Interpolated* from the surrounding values, the server will set the *StatusCode InfoBits* to be *Interpolated.*

If the requested timestamp format is not supported for a *Node*, the operation shall return the *Bad_TimestampNotSupported StatusCode*.

6.5 HistoryData parameters returned

6.5.1 Overview

The *HistoryRead Service* returns different types of data depending on whether the request asked for the value *Attribute* of a *Node* or the history *Events* of a node. The historyData is an *Extensible Parameter* whose structure depends on the functions to perform for the *historyReadDetails* parameter. See [UA Part 4] for details on *Extensible Parameters*.

6.5.2 HistoryData type

Table 23~~Table 24~~ defines the structure of the *HistoryData* used for the data to return in a *HistoryRead*.

Table 23~~24~~ – HistoryData Details

Name	Type	Description
dataValues[]	DataValue	An array of values of history data for the node. The size of the array depends on the requested data parameters.

6.5.3 HistoryModifiedData type

Table 23~~Table 24~~ defines the structure of the *HistoryModifiedData* used for the data to return in a *HistoryRead* when IsReadModified = True.

Table 24~~25~~ – HistoryModifiedData Details

Name	Type	Description
dataValues[]	DataValue	An array of values of history data for the node. The size of the array depends on the requested data parameters.
modificationInfos[]	ModificationInfo	
Username	String	The name of the user that made the modification. Support for this field is optional. A NULL shall be returned if it is not defined.
modificationTime	UtcTime	The time the modification was made. Support for this field is optional. A NULL shall be returned if it is not defined.
updateType	HistoryUpdateMode	The modification type for the item.

6.5.4 HistoryEvent type

Table 25~~Table 26~~ defines the HistoryEvent parameter used for Historical *Event* reads.

The HistoryEvent defines a table structure that is used to return *Event* fields to a *Historical Read*. The structure is in the form of a table consisting of one or more *Events*, each containing an array of one or more fields. The selection and order of the fields returned for each *Event* is identical to the selected parameter of the *EventFilter*.

Table 25~~26~~ – HistoryEvent Details

Name	Type	Description
Events []	HistoryEventFieldList	The list of *Events* being delivered
eventFields []	BaseDataType	List of selected *Event* fields. This will be a one to one match with the fields selected in the *EventFilter*.

6.6 HistoryUpdateMode Enumeration

Table 25~~Table 26~~ defines the HistoryUpdate enumeration.

Table 2627 – HistoryUpdateMode Enumeration

Name	Description
INSERT_1	Data was inserted
REPLACE_2	Data was replaced
UPDATE_3	Data was inserted or replaced
DELETE_4	Data was deleted.

6.7 HistoryUpdateDetails parameter

6.7.1 Overview

The *HistoryUpdate Service* defined in [UA Part 4] can perform several different functions. The *historyUpdateDetails* parameter is an *Extensible Parameter* that specifies which function to perform and the details that are specific to that function. See [UA Part 4] for the definition of *Extensible Parameter*. Table 27~~Table 28~~ lists the symbolic names of the valid *Extensible Parameter* structures. Some structures will perform different functions based on the setting of its associated parameters. For simplicity a functionality of each structure is listed. For example text such as ‘using the Replace data functionality’ refers to the function the *HistoryUpdate Service* performs using the *Extensible Parameter* structure *UpdateDataDetails* with the performInsertReplace enumeration parameter set to REPLACE_2

Table 2728 – HistoryUpdateDetails parameter TypeIds

Symbolic Name	Functionality	Description
UpdateDataDetails	Insert data	This function inserts new values into the history database at the specified timestamps for one or more HistoricalDataNodes. The *Variable*'s value is represented by a composite value defined by the *DataValue* data type.
UpdateDataDetails	Replace data	This function replaces existing values into the history database at the specified timestamps for one or more HistoricalDataNodes. . The *Variable*'s value is represented by a composite value defined by the *DataValue* data type.
UpdateDataDetails	Update data	This function inserts or replaces values into the history database at the specified timestamps for one or more HistoricalDataNodes. . The *Variable*'s value is represented by a composite value defined by the *DataValue* data type.
UpdateStructureDataDetails	Insert data	This function inserts new *Structure Data* or *Annotations* into the history database at the specified timestamps for one or more HistoricalDataNodes. The *Variable*'s value is represented by a composite value defined by the *DataValue* data type.
UpdateStructureDataDetails	Replace data	This function replaces existing *Structure Data* or *Annotations* into the history database at the specified timestamps for one or more HistoricalDataNodes. The *Variable*'s value is represented by a composite value defined by the *DataValue* data type.
UpdateStructureDataDetails	Update data	This function inserts or replaces *Structure Data* or *Annotations* into the history database at the specified timestamps for one or more HistoricalDataNodes. The *Variable*'s value is represented by a composite value defined by the *DataValue* data type.
UpdateStructureDataDetails	Remove data	This function removes *Structure Data* or *Annotations* from the history database at the specified timestamps for one or more HistoricalDataNodes. The *Variable*'s value is represented by a composite value defined by the *DataValue* data type.
UpdateEventDetails	Insert events	This function inserts new *Events* into the history database for one or more HistoricalEventNodes.
UpdateEventDetails	Replace events	This function replaces values of fields in existing *Events* into the history database for one or more HistoricalEventNodes.
UpdateEventDetails	Update events	This function inserts new events or replaces existing *Events* in the history database for one or more HistoricalEventNodes.
DeleteRawModifiedDetails	Delete raw	This function deletes all values from the history database for the specified time domain for one or more HistoricalDataNodes.
DeleteRawModifiedDetails	Delete modified	Some historians may store multiple values at the same Timestamp. This function will delete specified values and qualities for the specified timestamp for one or more HistoricalDataNodes.
DeleteAtTimeDetails	Delete at time	This function deletes all values in the history database for the specified timestamps for one or more HistoricalDataNodes.
DeleteEventDetails	Delete event	This function deletes *Events* from the history database for the specified filter for one or more HistoricalEventNodes.

The HistoryUpdate *Service* is used to update or delete both *DataValues* and *Events*. For simplicity the term "entry" will be used to mean either *DataValue* or *Event* depending on the context in which it is used. Auditing requirements for History *Services* is described in [UA Part 4]. This description assumes the user issuing the request and the server that is processing the request, support the capability to update entries. See [UA Part 3] for a description of *Attributes* that expose the support of Historical Updates.

6.7.2 UpdateDataDetails structure

6.7.2.1 UpdateDataDetails structure details

Table 28~~Table 29~~ defines the UpdateDataDetails structure.

Table 28~~29~~ – UpdateDataDetails

Name	Type	Description
UpdateDataDetails	Structure	The details for insert, replace, and insert/replace history updates.
nodeId	NodeId	Node id of the *Variable* to be updated.
performInsertReplace	PerformUpdate Enumeration	Value determines which action of insert, replace, or update is performed.
updateValue[]	DataValue	New values to be inserted or to replace.

Value	Description
INSERT_1	See Clause 6.7.2.2
REPLACE_2	See Clause 6.7.2.3
UPDATE_3	See Clause 6.7.2.4

6.7.2.2 Insert data functionality

Setting performInsertReplace = INSERT_1 inserts entries into the history database at the specified timestamps for one or more *HistoricalDataNodes*. If an entry exists at the specified timestamp, the new entry shall not be inserted; instead the *StatusCode* shall indicate *Bad_EntryExists*.

This function is intended to insert new entries at the specified timestamps; e.g., the insertion of lab data to reflect the time of data collection.

6.7.2.3 Replace data functionality

Setting performInsertReplace = REPLACE_2 replaces entries in the history database at the specified timestamps for one or more *HistoricalDataNodes*. If no entry exists at the specified timestamp, the new entry shall not be inserted; otherwise the *StatusCode* shall indicate *Bad_NoEntryExists*.

This function is intended to replace existing entries at the specified timestamp; e.g., correct lab data that was improperly processed, but inserted into the history database.

6.7.2.4 Update data functionality

Setting performInsertReplace = UPDATE_3 inserts or replaces entries in the history database for the specified timestamps for one or more *HistoricalDataNodes*. If the item has an entry at the specified timestamp, the new entry will replace the old one. If there is no entry at that timestamp, the function will insert the new data.

This function is intended to unconditionally insert/replace values and qualities; e.g., correction of values for bad sensors.

Good as a *StatusCode* for an individual entry is allowed when the server is unable to say whether there was already a value at that timestamp. If the server can determine whether the new entry replaces an entry that was already there, it should use *Good_EntryInserted* or *Good_EntryReplaced* to return that information.

6.7.3 UpdateStructureDataDetails structure

6.7.3.1 UpdateStructureDataDetails structure details

Table 28~~Table 29~~ defines the UpdateStructureDataDetails structure.

Table 29 – UpdateStructureDataDetails

Name	Type	Description
UpdateStructureDataDetails	Structure	The details for data history updates.
nodeId	NodeId	Node id of the *Variable* to be updated.
performInsertReplace	PerformUpdate Enumeration	Value determines which action of insert, replace, or update is performed.
updateValue[]	DataValue	New values to be inserted, replaced or removed.

Value	Description
INSERT_1	See Clause 6.7.3.3
REPLACE_2	See Clause 6.7.3.4
UPDATE_3	See Clause 6.7.3.5
REMOVE_4	See Clause 6.7.3.6

6.7.3.2 Specfied Uniqueness of Structured History Data

S*tructured History Data* provides metadata describing an entry in the history database. The server shall define what uniqueness means for each Structured History Data structure type. For example, a server may only allow one Annotation per timestamp which means the timestamp is the unique key for the structure. Another server may allow for multiple Annotations to exist per user, so a combination username, Timestamp and message may be used as the unique key for the structure. In the following sections the terms '*Structured Histroy Data* exists' and 'at the specificed parameters' means a matching entry has been found at the specified timestamp using the *Server's* criteria for uniqueness.

In the case where the Client wishes to Replace a parameter that is part of the uniqueness criteria, the resulting StatusCode would be *Bad_NoEntryExists*. They will have to Remove the existing structure and Insert the new structure.

6.7.3.3 Insert functionality

Setting performInsertReplace = INSERT_1 inserts *Structured History Data* such as *Annotations,* into the history database at the specified parameters for one or more *Properties* of *HistoricalDataNodes*

If a *Structured History Data* entry already exists at the specified parameters the *StatusCode* shall indicate *Bad_EntryExists*.

6.7.3.4 Replace functionality

Setting performInsertReplace = REPLACE_2 replaces *Structured History Data* such as *Annotations* in the history database at the specified parameters for one or more *Properties* of *HistoricalDataNodes.*

If a *Structured History Data* entry does not already exist at the specified parameters, the *StatusCode* shall indicate *Bad_NoEntryExists*.

6.7.3.5 Update functionality

Setting performInsertReplace = UPDATE_3 inserts or replaces *Structure Data* such as *Annotations* in the history database at the specified parameters for one or more *Properties* of *HistoricalDataNodes.*

If a *Structure History Data* entry already exists at the specified paramters it is deleted and the value provided by the *Client* is inserted. If no existing entry exists the new entry is inserted.

If an existing entry was replaced successfully the *StatusCode* shall be *Good_EntryReplaced.* If a new entry was created the *StatusCode* shall be *Good_EntryInserted.* If the server cannot determine whether it replaced or inserted an entry it the *StatusCode* shall be *Good.*

6.7.3.6 Remove functionality

Setting performInsertReplace = REMOVE_4 removes *Structure Data* such as *Annotations* from the history database at the specified parameters for one or more *Properties* of *HistoricalDataNodes.*

If a *Structure History Data* entry exists at the specified parameters it is deleted. If *Structured History Data* does not already exist at the specified parameters, the *StatusCode* shall indicate *Bad_NoEntryExists*.

6.7.4 UpdateEventDetails structure

6.7.4.1 UpdateEventDetails structure detail

Table 30~~Table 30~~ defines the UpdateEventDetails structure.

Table 30 – UpdateEventDetails

Name	Type	Description
UpdateEventDetails	Structure	The details for insert, replace, and insert/replace history *Event* updates.
nodeId	NodeId	Node id of the *Node* to be updated.
performInsertReplace	PerformUpdate Enumeration	Value determines which action of insert, replace, or update is performed. **Value** / **Description** INSERT_1 – Perform Insert *Event* (See Clause 6.7.4.2~~6.7.3.2~~) REPLACE_2 – Perform Replace *Event* (See Clause 6.7.4.3~~6.7.3.3~~) UPDATE_3 – Perform Update *Event* (See Clause 6.7.4.4~~6.7.3.4~~)
filter	EventFilter	If the history of *Notification* conforms to the *EventFilter*, the history of the *Notification* is updated.
eventData[]	HistoricalEventFieldList[]	*Events Notification* data to be inserted or updated.

6.7.4.2 Insert event functionality

This function is intended to insert new entries; e.g., backfilling of historical *Events*.

Setting performInsertReplace = INSERT_1 inserts entries into the *Event* history database for one or more *HistoricalEventNodes*. The *whereClause* parameter of the *EventFilter* shall be empty. The *SelectClause* shall specify the *EventType* and the Time. The *selectClause* should specify the *SourceNode* and the *SourceName*. If the historian does not support archiving the specified *EventType* the *StatusCode* shall indicate *Bad_TypeDefinitionInvalid*. If the *SourceNode* is not a valid source for *Events* the *StatusCode* shall indicate *Bad_SourceNodeIdInvalid.* If the *Time* does not fall within range that can be stored the *StatusCode* shall indicate *Bad_OutOfRange.*If the *selectClause* does not include fields which are mandatory for the *EventType* the *StatusCode* shall indicate *Bad_ArgumentsMissing.* If the *selectClause* specifies fields which are not valid for the *EventType* or cannot be saved by the historian the *StatusCode* shall indicate *Good_DataIgnored* and the *OperationResults* array shall specify *Bad_NotSupported* for each ignored field.

The *EventId* is a server generated opaque value and a *Client* cannot assume it knows how to create value *EventIds*. If a *Client* does specify the *EventId* in the *selectClause* and it matches an exitsting *Event* the *StatusCode* shall indicate *Bad_EntryExists.* A *Client* must use a *HistoryRead* to discover any automatically generated *EventIds*.

If any errors occur while processing individual fields the *StatusCode* shall indicate *Bad_ArgumentInvalid* and the *OperationResults* array shall specify the exact error for each invalid field. The *IndexRange* parameter of the *SimpleAttributeOperand* is not valid for insert operations and the *OperationResults* shall specify Bad_*IndexRangeInvalid* if one is specified.

If no errors occur the *StatusCode* shall indicate *Good* and the *OperationResults* array shall be empty. If errors occur *OperationResults* array will have one element for each field specified in the *selectClause*.

A *Client* may instruct the Server to choose a suitable default value for a field by specifying a value of null. If the server is not able to select a suitable default the corresponding entry in the OperationResults array shall be *Bad_InvalidArgument*.

6.7.4.3 Replace event functionality

This function is intended to replace fields in existing *Event* entries; e.g., correct *Event* data that contained incorrect data due to a bad sensor.

Setting performInsertReplace = REPLACE_2 replaces entries in the *Event* history database for the specified filter for one or more *HistoricalEventNodes*. The *whereClause* parameter of the *EventFilter* shall specify the *EventId Property*. If no entry exists matching the specified filter, no updates will be performed, instead the *StatusCode* shall indicate *Bad_NoEntryExists*.

If the *selectClause* specifies fields which are not valid for the *EventType* or cannot be saved by the historian the *StatusCode* shall indicate *Good_DataIgnored* and the *OperationResults* array shall specify *Bad_NotSupported* for each ignored field.

If a field is valid for the *EventType* but cannot be changed the *StatusCode* shall indicate *Good_DataIgnored* and the corresponding entry in the *OperationResults* array shall be *Bad_NotWriteable*.

If fatal errors occur while processing individual fields the *StatusCode* shall indicate *Bad_ArgumentInvalid* and the *OperationResults* array shall specify the exact error.

If no errors occur the *StatusCode* shall indicate *Good* and the *OperationResults* array shall be empty. If errors occur *OperationResults* array will have one element for each field specified in the *selectClause*.

If a *Client* specifies a value of null for any field the *Server* shall set the value of the field to null. If a null value is not valid for the corresponding entry in the *OperationResults* array shall be *Bad_InvalidArgument*.

6.7.4.4 Update event functionality

This function is intended to unconditionally insert/replace *Events*; e.g., synchronizing a backup *Event* database.

Setting performInsertReplace = UPDATE_3 inserts or replaces entries in the *Event* history database for the specified filter for one or more *HistoricalEventNodes*.

The server will, based on its own criteria, attempt to determine if the *Event* already exists, if it does the existing *Event* will be deleted and the new *Event* will be inserted (retaining the *EventId*). If the event does not exist then a new *Event* will be inserted, including the generation of a new *EventId*.

All of the restrictions, behavoirs errors specified for the Insert functionality also apply to this function.

If an existing entry was replaced successfully the *StatusCode* shall be *Good_EntryReplaced.* If a new entry was created the *StatusCode* shall be *Good_EntryInserted.* If the server cannot determine whether it replaced or inserted an entry it the *StatusCode* shall be *Good.*

6.7.5 DeleteRawModifiedDetails structure

6.7.5.1 DeleteRawModifiedDetails structure detail

Table 31~~Table 31~~ defines the DeleteRawModifiedDetails structure.

Table 31 – DeleteRawModifiedDetails

Name	Type	Description
DeleteRawModifiedDetails	Structure	The details for delete raw and delete modified history updates.
nodeId	NodeId	Node id of the *Variable* for which history values are to be deleted.
isDeleteModified	Boolean	TRUE for MODIFIED, FALSE for RAW. Default value is FALSE.
startTime	UtcTime	beginning of period to be deleted
endTime	UtcTime	end of period to be deleted

These functions are intended to be used to delete data that has been accidentally entered into the history database; e.g., deletion of data from a source with incorrect timestamps. Both startTime and endTime must be defined. The startTime must be less than the endTime, and values up to but not including the endTime are deleted, It is permissible for startTime = endTime in which case the value at the startTime is deleted.

6.7.5.2 Delete raw functionality

Setting isDeleteModified = FALSE deletes all *Raw* entries from the history database for the specified time domain for one or more *HistoricalDataNodes*.

If no data is found in the time range for a particular HistoricalDataNode, the *StatusCode* for that item is *Bad_NoData.*

6.7.5.3 Delete modified functionality

Setting isDeleteModified = TRUE deletes all *Modified* entries from the history database for the specified time domain for one or more *HistoricalDataNodes*.

If no data is found in the time range for a particular *HistoricalDataNode*, the *StatusCode* for that item is *Bad_NoData.*

6.7.6 DeleteAtTimeDetails structure

6.7.6.1 DeleteAtTimeDetails structure detail

Table 32~~Table 32~~ defines the structure of the DeleteAtTimeDetails structure.

Table 32 – DeleteAtTimeDetails

Name	Type	Description
DeleteAtTimeDetails	Structure	The details for delete raw history updates
nodeId	NodeId	Node id of the *Variable* for which history values are to be deleted.
reqTimes []	UtcTime	The entries define the specific timestamps for which values are to be deleted.

6.7.6.2 Delete at time functionality

The DeleteAtTime structure deletes all entries in the history database for the specified timestamps for one or more *HistoricalDataNodes*.

This parameter is intended to be used to delete specific data from the history database; e.g., lab data that is incorrect and cannot be correctly reproduced.

6.7.7 DeleteEventDetails structure

6.7.7.1 DeleteEventDetails structure detail

Table 32~~Table 32~~ defines the structure of the DeleteEventDetails structure.

Table 33 – DeleteEventDetails

Name	Type	Description
DeleteEventDetails	Structure	The details for delete raw and delete modified history updates.
nodeId	NodeId	Node id of the *Variable* for which history values are to be deleted.
eventId[]	ByteString	An array of *EventIds* to identify which *Events* are to be deleted.

6.7.7.2 Delete event functionality

The DeleteEventDetails structure deletes all *Event* entries from the history database matching the *EventId* for one or more *HistoricalEventNodes*.

If no *Events* are found that match the specified filter for a *HistoricalEventNode*, the *StatusCode* for that *Node* is *Bad_NoData.*

Annex A Client Conventions

A.1 How clients may request timestamps

The OPC HDA COM based specifications allowed *Client* to programmatically request historical time periods as absolute time (Jan 01, 2006 12:15:45) or a string representation of relative time (NOW -5M). The OPC UA specification does not allow for using a string representation to pass date/time information using the standard *Services*.

OPC UA *Client* applications that wish to visually represent date/time in a relative string format must convert this string format to UTC DateTime values before sending requests to the UA server. It is recommended that all OPC UA *Client* use the syntax defined in this section to represent relative times in their user interfaces.

The format for the relative time is:

```
keyword+/-offset+/-offset…
```

where keyword and offset are as specified in the table below. Whitespace is ignored. The time string must begin with a keyword. Each offset must be preceded by a signed integer that specifies the number and direction of the offset. If the integer preceding the offset is unsigned, the value of the preceding sign is assumed (beginning default sign is positive). The keyword refers to the beginning of the specified time period. DAY means the timestamp at the beginning of the current day (00:00 hours, midnight). MONTH means the timestamp at the beginning of the current month, etc.

For example, “DAY -1D+7H30M” could represent the start time for data requested for a daily report beginning at 7:30 in the morning of the previous day (DAY = the first timestamp for today, -1D would make it the first timestamp for yesterday, +7H would take it to 7 a.m. yesterday, +30M would make it 7:30 a.m. yesterday (the + on the last term is carried over from the last term).

Similarly, “MONTH-1D+5H” would be 5 a.m. on the last day of the previous month, “NOW-1H15M” would be an hour and fifteen minutes ago, and “YEAR+3MO” would be the first timestamp of April 1 this year.

Resolving relative timestamps is based upon what Microsoft has done with Excel, thus for various questionable time strings, we have these results:

10-Jan-2001 + 1 MO = 10-Feb-2001

29-Jan-1999 + 1 MO = 28-Feb-1999

31-Mar-2002 + 2 MO = 30-May-2002

29-Feb-2000 + 1 Y = 28-Feb-2001

In handling a gap in the calendar (due to different numbers of days in the month, or in the year), when one is adding or subtracting months or years:

Month: if the answer falls in the gap, it is backed up to the same time of day on the last day of the month.

Year: if the answer falls in the gap (February 29), it is backed up to the same time of day on February 28.

Note that the above does not hold for cases of adding or subtracting weeks or days, but only for adding or subtracting months or years, which may have different numbers of days in them.

Note that all keywords and offsets are specified in uppercase.

Table 34 –Time Keyword Definitions

Keyword	Description
NOW	The current UTC time as calculated on the server.
SECOND	The start of the current second.
MINUTE	The start of the current minute.
HOUR	The start of the current hour.
DAY	The start of the current day.
WEEK	The start of the current week.
MONTH	The start of the current month.
YEAR	The start of the current year.

Table 35 –Time Offset Definitions

Offset	Description
S	Offset from time in seconds.
M	Offset from time in minutes.
H	Offset from time in hours.
D	Offset from time in days.
W	Offset from time in weeks.
MO	Offset from time in months.
Y	Offset from time in years.

A.2 Determining the First Historical Data Point

In some cases Servers are required to return the first available data point for a historical node, this section recommends the way that *Client* should request this information so that Servers can optimize this call if desired. Although there are multiple calls that would return the first data value, the recommended practice will be to use the following ReadRawModifiedDetails parameters:

returnBounds=false

numValuesPerNode=1

startTime=DateTime.MinValue+0x1000

endTime=DateTime.MaxValue-0x1000