ADDENDA

BSR/ASHRAE Addendum am to
ANSI/ASHRAE Standard 135-2012

Data Communication Protocol for Building Automation and Control Networks

Approved by ASHRAE on April 29, 2016, and by the American National Standards Institute on April 29, 2016.

This addendum was approved by a Standing Standard Project Committee (SSPC) for which the Standards Committee has established a documented program for regular publication of addenda or revisions, including procedures for timely, documented, consensus action on requests for change to any part of the standard. The change submittal form, instructions, and deadlines may be obtained in electronic form from the ASHRAE website (www.ashrae.org) or in paper form from the Senior Manager of Standards.

The latest edition of an ASHRAE Standard may be purchased on the ASHRAE website (www.ashrae.org) or from ASHRAE Customer Service, 1791 Tullie Circle, NE, Atlanta, GA 30329-2305. E-mail: orders@ashrae.org. Fax: 678-539-2129. Telephone: 404-636-8400 (worldwide), or toll free 1-800-527-4723 (for orders in US and Canada). For reprint permission, go to www.ashrae.org/permissions.

 ISSN 1041-2336

Approved American National Standard
ANSI
www.ansi.org

[This foreword and the "rationales" on the following pages are not part of this standard. They are merely informative and do not contain requirements necessary for conformance to the standard.]

FOREWORD

The purpose of this addendum is to present changes to ANSI/ASHRAE Standard 135-2012 and Addenda. These modifications are the result of change proposals made pursuant to the ASHRAE continuous maintenance procedures and of deliberations within Standing Standard Project Committee 135. The changes are summarized below.

In the following document, language to be added to existing clauses is indicated through the use of *italics*, while deletions are indicated by ~~strikethrough~~. Where entirely new subclauses are added, plain type is used throughout. All other material in this addendum is provided for context only.

[This Table of Contents is provided only for convenience in this addendum. It will not be part of the standard]

Table of Contents

135-2012*am*-1. Extend BACnet/WS with RESTful Services for Complex Data Types and Subscriptions.

Rationale

The existing ANNEX N, BACnet Web Services, was designed for simple data exchange and simple history retrieval for simple clients. More sophisticated clients and use cases, such as those needed by the Smart Grid, require the addition of more capable services.

Solution:

Add new services to provide structured data exchange, full history retrieval, and subscriptions.

Requirements:

These additions to the BACnet/WS standard are designed to meet the following requirements:

0) Provide state of the art security and authorization mechanisms (TLS and OAuth 2.0) with flexible PKI certificate management and provide a capability for fine-grained and extensible authorization schemes.

1) Allow the exchange of structured data: The existing services only returned plain text values for primitive, and array of primitive, datatypes. These new services allow the exchange of structured data using XML or JSON. Structured data is exchanged using the XML syntax defined in ANNEX Q and the JSON syntax of Addendum Z. The structured data is retrieved and updated using the HTTP GET and PUT methods.

2) Allow the creation and deletion of server data: The existing services only allow the reading or writing of existing data items on the server. These new services provide a means of creating or deleting data on the server, wherever the server allows such operations. The data is created and deleted using the HTTP POST and DELETE methods.

3) Allow retrieval of non-periodic trend histories: The existing services only allow the retrieval of periodic samples of primitive data in plain text, where the timestamps of the data are implied by its periodicity. The new services return data in XML allowing not only the inclusion of explicit timestamp and sequence information, but also error information, log buffer activity information, and even has the possibility of returning non-primitive data structures.

4) Allow current plain text services in REST: The exchange of simple text data, as provided by the existing SOAP messages, has a strong use case and is retained in the new REST interface. All the primitive values and metadata, like "writable" etc., may be retrieved and set in plain text.

5) Allow items to have globally unique identifiers: Some data items can have a permanent, globally unique identifier. This identifier is expressed in the 'id' metadata. The identifier is permanent and not based on location, so for data that can move, the 'id' will move with it.

6) Allow the creation of subscriptions: These new services allow the creation of subscriptions for change-of-value, trend reporting, and event reporting.

7) Allow the definitions of active callbacks: For clients that can accept active callbacks, a mechanism is defined to allow them to specify how the server should send them new records for change of value, trend, or event notification.

8) Creation of subscriptions by a third party or setup tool: Subscriptions can be created on the server "on behalf of" the actual recipient by a third device or setup tool.

9) Movement of trend history from source to archives: Through the use of subscriptions and active callbacks, trended data can be automatically moved from the source history list to an archive server. The

archive server can record the original source id and source location in link metadata which can be queried by a client when trying to find the archived records on the archive server.

10) Allow exchange of XML data defined by others (e.g., Smart Grid): The protocol supports this "foreign XML" (XML defined by others) as well as our "native XML" (CSML). The difference is that our servers can "drill down" to access subsets of CSML data but must treat foreign XML as a whole. Nonetheless, it can be retrieved, updated, created, and deleted.

11) Allow exchange of arbitrary media or bulk data: The protocol format also supports arbitrary "media" entries. These can be used to retrieve, update, create, and delete any kind of media type, documents, pictures, etc.

12) Allow client to specify/filter the contents of data: To prevent transmission of redundant or unwanted data, the client can specify what kinds of data it wants returned. This is accomplished by various filters that select the degree of metadata included in the transfer as well as filtering selected entries from large collections.

13) Allow the client to specify the range of data: For handling large sets of data, or to work within limited client or server capabilities, the client can read only a restricted range of the available data. This is accomplished by time ranges and/or index and count restrictions.

14) Allow for chaining the retrieval of large data sets: If the client specifies more records than can be returned by the server, or by the client's count limitation, the server indicates the availability of more data with a 'next' link. The client can continue to follow this 'next' link until all available data is returned.

15) Allow a resource to mirror data for another resource: In the case where an aggregating server has collected data from other servers, either to provide a common place for convenience, or to provide archiving services for trends or alarms, resources that hold data that originated in another location need to have a standard way to point to their source. This is done with "sourceId" and "via" metadata. The "via" metadata contains the dereferenceable URL of the source of the data and the "sourceId" contains the nondereferenceable "id" of the data.

16) Allow searching for data: Metadata can be present on all addressable data. To make searching a reasonable thing to implement for servers, by convention, most interesting metadata, like tags and links, are placed at the "object" granularity. The server then provides an "objects" lists which can be filtered to find particular objects based on their metadata or object property value. The client specifies the values it is interested in with the 'filter' query parameter and the server populates the results list with the objects matching the query. To further control the size of the response, the client can select only the properties it is interested in.

17) Allow discovery of data by semantic meaning: The use of "tags" allows a flexible and extensible means to tag objects and other data with meaning and usage information that can be queried with the 'filter' query parameter. Tags can be formally defined and published by their defining organization. This allows them to have machine readable localizable text for human presentation.

18) Allow access to multiple resources at a time, for efficiency or atomicity reasons. The existing SOAP services allow getting or setting multiple paths at one time, which is a very un-RESTful activity, but nonetheless desirable. These new services provide a way to access multiple items as RESTfully as possible by creating the concept of a "multi" resource, which is a single addressable resource whose contents are a collection of other resources.

19) In addition to accessing multiple resources at a time, subscribing to multiple items is also desirable and should be done in a simple and consistent manner. Thus, the "multi" resource described above can be subscribed to, just as any other single resource.

20) Add "data revisioning" to CSML. Many kinds of data definitions, for example those defined in Clause 21, change over time. New members, usually optional, are added to data constructs, new values are defined for enumerations and bit strings, and new properties are added to objects. Occasionally, some of these items are even modified or removed. CSML as currently defined in Annex Q does not allow this and requires that new data definitions be created whenever "structural" changes happen. In practice, creating new data definition names for these changes becomes untenable. Recognizing the dynamic nature of data structures, therefore, CSML needs the ability to indicate revisions without creating new, seemingly unrelated, data types. Clients need the ability to work with simple extensible data types for the most common use cases, but also to be able to know about the full details of the revision history for more advanced use cases.

[Add new **ANNEX W** p. 1026]

ANNEX W - BACnet/WS RESTful WEB SERVICES INTERFACE (NORMATIVE)

(This annex is part of this standard and is required for its use.)

This annex defines a secure service interface for integrating facility data from disparate data sources for a variety of applications. The data model and services are abstract and protocol independent, and can therefore be used to model and access data from any source, whether the server owns the data locally or is acting as a gateway to other standard or proprietary protocols.

Clients can determine the version of the standard that a server device implements by querying the specific numerical values defined in Clause W.5.1.

The services defined in this standard use the HTTP REST (REpresentational State Transfer) model. These services exchange resource data using a format appropriate to the resource. Control systems data is represented by CSML, defined in Annex Q and Annex Z. The services can also host data from other XML dialects as well as arbitrary media types. While this protocol is based on HTTP and conforms to basic HTTP rules, not all optional features of HTTP are expected to be implemented by either client or server, and only the subset required to implement the features and functions specifically called for in this standard (e.g. the specific use of the 'Location' header in W.28) are required. Therefore clients and servers are not required to handle advanced HTTP features like redirection, cache-control, transfer-encoding, retry-after, etc.

Strong security is provided by TLS standard RFC 2246 as amended and updated, and a restricted subset of the OAuth 2.0 standards RFC 6749 and RFC 6750 as amended and updated, which are incorporated here by reference.

Note: the examples of HTTP exchanges in the following clauses are written for human understanding purposes. In most cases, they are not the literal character-for-character exchanges. In all of these examples, the "HTTP/1.1" is left off of requests and responses, most HTTP headers are not shown, and all URLs are left unencoded. This concession for readability is not to be taken as a requirement of this specification. All operations and encodings shall conform to relevant standards.

W.1 Data Model

The data model for the data exchanged by these services is defined by Annex Y. The arrangement of data into hierarchies and the naming of the data is generally a local matter. However, this standard defines a number of standardized data items with standardized locations that allow clients to obtain common information for interoperability. These standardized data items are described more fully in Clause W.5.

W.2 Paths

The services defined in this annex use URIs to access data and associated metadata. The path segments of the URI are made up of the name of the data items. Metadata items are separated from regular data items by a dollar sign "$" character as a path segment, e.g., /somedata/$maximum. Data names shall not begin with whitespace or the "$" character, and names beginning with a dot "." are reserved for use by this standard. For a complete list of restrictions on names, see Clause Y.4.1. For data item names that contain characters that are not representable in a URI, the rules of RFC 3987 for conversion of IRI to URI shall be applied to the names for representation as URI path components.

Examples:

/.info/version
/east_campus/building_41/zone_5B/occ_cool_setpoint
/east_campus/building_41/zone_5B/occ_cool_setpoint/$maximum

To provide a balance between flexibility and uncomplicated interoperability, this standard defines several fixed paths that clients can assume. However, to provide some deployment flexibility, the root of these paths has an optional prefix that is discoverable in a well-known manner, as defined by RFC 5785. The server shall provide a resource at the absolute path /.well-known/ashrae. This resource shall be accessible by an HTTP GET using either "http" or "https" on their default ports, with any client certificate accepted for "https". There shall be no authorization required to read this resource. The media type of this resource is "text/plain". The response body shall be a collection of links, each on a separate line in the format of an HTTP Link Header Field, as defined by section 5 of RFC 5988. One or more of these links shall have a relation type rel="http://bacnet.org/csml/rel#server-root". Links with other relation types in this resource are not restricted nor specified by this standard. If a single host supports multiple BACnet/WS server devices, then a link for each BACnet/WS server device shall be present in the /.well-known/ashrae resource. Multiple BACnet/WS server devices can share a common TCP port for "http", but cannot share a common port for "https". See Example W.41.1.

W.3 Security

The security requirements defined by this standard provide for confidentiality and authorization that allow for multiple applications or security domains to exist in the same server. All data in the server can be assigned a "security scope" that is required to access that data and the data below it in the hierarchy. Separate scope identifiers can be provided for reading and writing, allowing deployment flexibility in visibility and modification restrictions. See Clause W.3.5.

Some examples of security scopes could be:

(a) HVAC data that can be freely read in the clear without any authorization but requires TLS and a "config" scope to update, create, or delete.
(b) Physical access control data (card credentials, etc.), that cannot be read at all in the clear and that require TLS and a "555-acc-read" scope to read and a separate "555-acc-write" scope to write.
(c) A critical resource that has a specific scope like, "555-crit39245", assigned uniquely to itself.

The above identifiers are examples only.

W.3.1 Certificate Management

All secure communications requires the use of TLS. The creation of TLS certificates and the management of the certificate signing authority (or authorities) are site-specific deployment options beyond the scope of this standard. However, to ensure interoperability, both client and server implementations of this standard shall support the storage and use of certificates as defined in the following clauses.

W.3.1.1 Required Certificates

Before deployment to an active network, each server device shall be configured with one or more "authority certificates" and a unique "device certificate". The device certificate shall be issued by a CA that is verifiable with one of the authority certificates. This allows peer-to-peer mutual authentication so that a server device can verify that a client certificate presented to it was issued from an approved CA.

W.3.1.2 Signing CA

The choice of CA to generate the device certificates shall be dictated by site policy. If a site-local CA is used, it is the responsibility of that CA to properly safeguard the private key used to issue device certificates.

W.3.1.3 Configuring Certificates and Activating TLS

The server device requires the authority certificate(s), device certificate, and the device's private key in order to activate TLS communications. The certificates are configured by writing to"/.auth/ca-certs-pend" and "/.auth/dev-cert-pend" and the private key is written to write-only "/.auth/dev-key-pend" (the key data cannot be read back under any circumstances). Note that if the device acts as a client, then the device certificate is also used as the client certificate and shall therefore have both client and server usages enabled, as defined by RFC 5280 as amended and updated.

After these have been successfully written, the Boolean at "/.auth/tls-activate" can be written with "true" to cause the pending values to become the active set. The currently active set is reflected in the read-only "/.auth/ca-certs" and "/.auth/dev-cert". The key data is never available for reading.

When the "/.auth/tls-activate" is written "true", the server device shall validate the pending certificates and private key. If any of the pending information is empty or malformed, or if the pending dev-cert cannot be verified by any of the ca-certs, or if the pending dev-key does not match the pending dev-cert, then the server device shall return a WS_ERR_TLS_CONFIG error and not activate TLS with the pending values. Successfully writing "/.auth/tls-activate" to "true" causes the pending information to be copied into the working copies at "/.auth/ca-certs" and "/.auth/dev-cert" and all of the pending data items are set to zero length and the "/.auth/tls-activate" is set back to "false".

W.3.1.4 Factory Default Condition

From the factory, server devices shall have neither the authority certificate(s) nor device certificates configured. Since the use of TLS requires the pre-existence of certificates, it is not possible to use the normal authorization mechanism to write the certificates into a server device. While in this mode, TLS and OAuth security is not available and only clear text HTTP can be used. Therefore, it is critical that server devices in this condition are not deployed to nonsecure networks until they have been properly configured.

While in the factory defaults mode, the authority certificate(s) shall be writable in plain text, each using an HTTP POST to the List at "/.auth/ca-certs-pend", and the device certificate shall be writable in plain text with a PUT to "/.auth/dev-cert-pend". In addition to the two public certificates, the private key corresponding to the device certificate shall be writable in plain text using a PUT to "/.auth/dev-key-pend" and the activation command shall be writable in plain text using a PUT to "/.auth/tls-activate".

If not factory set and fixed, the device unique identifier shall default to all zeros and shall be writable in plain text by a PUT to /.auth/dev-uuid

After TLS has been successfully activated (see Clause W.3.1.3), the device shall enable the internal authorization server and all future writes to the "/.auth" structure shall require the use of TLS and OAuth using the "auth" scope.

W.3.1.5 Reset to Factory Defaults

Devices shall provide a suitably secure out-of-band mechanism to place a server device into "factory defaults" mode. It is recommended that this requires physical access to the server device.

Performing a "reset to factory defaults" operation shall erase all certificates, the private key, and any sensitive data that the server contains, and reset the default user name, password, client id, and client secret to their default values, and, if not factory preset, reset the device unique identifier to all zeros. It is not allowed to simply block access to existing sensitive data while in the factory defaults mode because an attacker with physical access can use this mode to insert new certificates and then use that false trust relationship to access sensitive data that was not erased.

W.3.2 OAuth

The scope of authorization is provided to the server by an OAuth 2.0 bearer token, which was obtained by the client from an OAuth 2.0 authorization server. Implementation and deployment options allow the authorization server function to be located in the same physical host as the resource server, or on a separate host. To ensure interoperability and to simplify basic installations, this standard requires that all servers implement a basic "internal authorization server", as defined in Clause W.3.3.

RFC 6750 provides a set of recommendations for safeguarding bearer tokens. These recommendations are summarized and further strengthened by this standard in the following table.

RFC recommendation	Requirements by this standard
Safeguard bearer tokens	The client shall never transmit a token in the clear, transmit a token to an unverified destination, or store a token in an unsecured manner.
Validate TLS certificate chains	The client shall validate the TLS certificate chain of the destination. Clients shall provide capability to configure a trusted certificate root for site-specific Certificate Authorities.
Always use TLS (https)	Clients shall only transmit tokens in TLS to verified destinations. Servers shall provide the ability to configure an installer-provided certificate.
Don't store bearer tokens in cookies	Clients shall never store tokens in cookies. No exceptions are allowed by this standard.
Issue short-lived bearer tokens	It is recommended that external authorization servers issue tokens with a lifetime appropriate for the security scope (i.e. high security scopes should receive shorter lifetimes). However, configuration of external authorization servers is beyond the scope of this standard. The "internal authorization server" is not required to support such configuration.
Issue scoped bearer tokens	In this context, the word "scoped" refers to audience rather than the OAuth "Scope" parameter. All access tokens shall have an "audience" parameter which identifies either a single device or a group of devices. The format of access tokens is defined by Clause W.3.7
Don't pass bearer tokens in page URLs	Bearer tokens shall only be provided in the HTTP "Authorization" message header, never as a query parameter or in the message body.
Restricting the use of the token to a specific scope is also RECOMMENDED	All access tokens shall specify at least one scope for which they apply. Because of this, all client requests for tokens shall specify a non-empty OAuth "scope" parameter. The format of access tokens is defined by Clause W.3.7
The token integrity protection MUST be sufficient to prevent the token from being modified	All access tokens shall be digitally signed to prevent tampering. The format of access tokens is defined by Clause W.3.7

W.3.3 Internal Authorization Server

To support basic security needs, and to bootstrap more advanced configurations, all server devices are required to support a basic OAuth 2.0 authorization server function that can be used to issue access tokens for the same server. This "internal" authorization server is not required to issue access tokens for other servers.

The internal server is required to support the OAuth "Resource Owner Password Credentials Grant" and the "Client Credentials Grant" types. Support for other OAuth grant types is optional, and the details of such support are a local matter.

The internal authorization server's "Authorization endpoint" is "/.auth/int/authorize", and the "Token endpoint" is "/.auth/int/token". Note that the addendum only requires the use of the "Resource Owner Password Credentials Grant" and the "Client Credentials Grant" authorization flows, both of which use the "token endpoint" directly. Therefore the definition of the "Authorization endpoint" is included in this standard only to reserve the path name for the case where a server optionally implements OAuth authorization flows that make use of it.

For the "Resource Owner Password Credentials Grant" type, the server shall support at least one configurable user name and user password pair, at "/.auth/int/user" and "/.auth/int/pass", with a storage minimum of 16 bytes and 32 bytes respectively. This pair shall authorize any scope presented by the client, including the "auth" scope. Access tokens issued for this pair shall set the user-id field to 0 and the user-role field to 1. See Clause 24.2.11 for definition of user-id and user-role. Support for additional usernames, passwords, and a corresponding limiting of scope is optional and is a local matter.

For the "Client Credentials Grant" type, the server shall support at least one configurable client id and client secret pair at "/.auth/int/id" and "/.auth/int/secret", with a storage minimum of 16 bytes and 32 bytes respectively. This pair shall authorize any scope presented by the client, except the "auth" scope. Access tokens issued for this pair shall set the user-id field to 0 and the user-role field to 0. See Clause 24.2.11 for definition of user-id and user-role. Support for additional client ids and secrets, and a corresponding limiting of scope is optional and is a local matter.

The Boolean switch at "/.auth/int/enable" shall be "true" by default (or when restored to factory default mode) and can be set to "false" to disable the internal authorization server after external authorization servers have been configured. Note that disabling the internal authorization before correctly configuring external authorizations servers shall result in a state where no further authorized actions can occur since the internal server is disabled and tokens previously issued by the internal server will be rejected. Therefore, the configuration of the external authorization servers and their operational status shall be verified by the installer before disabling the internal authorization server.

If external authorization servers are not configured, then the internal server will continue to be the only way to authenticate access to the restricted data in the server device. It is recommended, however, that the internal authorization server be disabled once external authorizations servers have been configured. Leaving the internal authorization server running after the external servers have been configured could lead to a stale backdoor password situation and the owner's site policy shall determine if this deployment is allowed.

W.3.3.1 Factory Default Condition

By default (or when reset to factory defaults), the username, password, client id, and client secret of the internal authorization server shall all be equal to a single period character (".") and all addresses for external authorization servers shall contain empty strings. This is considered a "bootstrap" security condition and is generally not suitable for deployment on a non-physically-secure network. A combination of site policy and the contents of a server will determine whether a server in this default condition is deployable to the field (e.g., possibly allowed if the server device contains only read-only public data that requires no authorization). If it is deployed in this condition, it is recommended that the internal authorization server be disabled to prevent malicious settings of bogus passwords or external authorization server information. When designing a user interface to edit these default values for field deployment, it is strongly recommended that the interface requires the user to change both the user 'password' and the client 'secret', since changing only one of these and leaving the other in the default state will leave an unintended backdoor condition.

While in the default condition, the configuration of other usernames or passwords or the writing of any external authorization server information shall be forbidden. Therefore, the only security configuration operations that are allowed in this condition are:

(a) to set the base username and password at "/.auth/int/user" and "/.auth/int/pass" to non-default values,
(b) to set the base client's id and secret at "/.auth/int/id" and "/.auth/int/secret" to non-default values,
(c) to set the server UUID at "/.auth/dev-uuid" if not factory preset and fixed, or
(d) to configure the external authorization server information and then disable the internal server by writing "false" to "/.auth/int/enable".

Note that the username, password, client id, and client secret are write-only resources and cannot be read back.

W.3.4 External Authorization Servers

For deployments where one or more centralized authorization servers provide authorization services for a number of server devices, each server device can be configured to refer to an "external authorization server" that provides authorization for that server device.

W.3.4.1 Indication of External Authorization Servers

When external authorization servers are to be used for a server device, the server device shall provide a network visible way for clients to discover the location of the authorization server(s). This is provided by the following resources:

Path	Meaning
{prefix}/.auth/ext/pri-uri	URI of the primary authorization server token endpoint
{prefix}/.auth/ext/pri-cert	Public certificate of the primary server
{prefix}/.auth/ext/pri-pubkey	Public key used to verify tokens signed by primary server
{prefix}/.auth/ext/sec-uri	URI of the secondary authorization server token endpoint
{prefix}/.auth/ext/sec-cert	Public certificate of the secondary server
{prefix}/.auth/ext/sec-pubkey	Public key used to verify tokens signed by secondary server

URIs in these data items shall be empty strings by default (or when restored to factory defaults). Writing to this data shall only be allowed with an access token using the "auth" scope.

In deployments where there is no redundant backup authorization server, it is recommended that the "secondary" URI be left blank, rather than being set equal to the primary URI, to prevent unnecessary client actions.

At a minimum, server devices shall support storage of an 80 byte URI and a 1500 byte certificate for both primary and secondary authorization servers.

W.3.4.1.1 Server Device Use of This Information

The server device uses the public key from the /.auth/ext/{'pri' or 'sec'}-pubkey to verify tokens signed by the corresponding authorization server. If the server device has no client capability itself, then it has no use for /.auth/ext/{'pri' or 'sec'}-uri and the corresponding certificate. However, if the server device does have a need to communicate with the external authorization server, it shall use the corresponding certificate to verify that connection.

W.3.4.1.2 Client Use of This Information

If the "/.auth/ext/..." information is present, a client shall use the referenced external authorization server(s) to request authorization, even if the internal authorization server is still enabled, unless specifically directed by the user to use the internal authorization sever.

To support cases where communications with the referenced authorization server is temporarily unavailable (for example during installation, maintenance, or periods of network outage), it is recommended that clients be capable of communicating with the referenced authorization server in some local manner and then retaining the access tokens for use at another location. To support this, external authorization servers are required to have user configurable lifetimes for the tokens they issue so that temporary long-lived tokens can be issued for these special cases.

It is recommended that clients consider that a compromised server device could present bogus information about external authorization servers, and that trusting the presented uri and certificate exclusively could lead to disclosure of user credentials to a bogus authorization server. Therefore, it is recommended that clients maintain their own set of acceptable external authorization server certificates, or acceptable root certificates, that have been provided to them in a secure manner.

W.3.4.2 Capabilities of External Authorization Servers

Whereas the internal authorization servers can only issue tokens for one server device, external authorization servers can issue tokens for an unlimited number of devices and for groups of devices. Clients use the "audience" parameter to request a single device or a group of devices.

W.3.4.3 Requirements of External Authorization Servers

External authorization servers communicate with BACnet/WS clients using the message flows defined by the OAuth standard. Additionally, they are required to accept the "audience" parameter as defined by Clause W.3.6.1, to produce tokens as defined by Clause W.3.7, and to have user configurable token lifetimes as required by Clause W.3.4.1.2. They are otherwise not controlled by this specification. Therefore, the configuration and operation of external authorization servers are beyond the scope of this specification, and the configuration of users, credentials, authentication mechanisms, and the assignment of user-id and user-role in the external authorization servers is a local matter.

W.3.4.4 Deployment of External Authorization Servers

An "external authorization server" is an OAuth function that is not associated with any particular BACnet/WS server device. The external authorization server function might be hosted on the same HTTP server as one or more BACnet/WS server device(s), but it is an independent function with independent URI endpoints and there is no implicit association with any BACnet/WS server device. Therefore, even if they share the same HTTP server, the BACnet/WS server device(s) are still required to point at the external authorization server with the "/.auth/ext/..." data items.

W.3.5 Scope

The naming of scope identifiers is an application-specific matter with the exception that they shall conform to the subset allowed by OAuth, and the scope "auth" is reserved for very limited uses. For interoperability, several other scope identifiers are predefined as well, and it is recommended that server devices use the predefined scope identifiers whenever possible.

The number of scope identifiers (the granularity) supported by the server device is a local matter and will generally match the capabilities of the server device and the expected kinds of data that it will contain. The assignment of scope identifiers to data is also a local matter and some server devices might allow the scope identifiers to be assigned by a client and some might be fixed. If the server device allows client configuration of the scope identifiers for a particular piece of data, then it shall make the 'authRead', 'authWrite', and 'authVisible' metadata writable for that data. Writing to this metadata shall only be allowed with an access token containing the "auth" scope.

The scope identifier string value and the meaning and intended usage for the predefined scope identifiers is defined in the following table.

Table W-1. Predefined Scopes

Scope Identifier	Meaning
view	View (view private data - public data does not need authorization)
adjust	Adjust setpoints
control	Runtime Control (write values for the purpose of controlling actions)
override	Command Override (placing objects out of service, commanding at priorities that would limit runtime control, etc.)
config	Configuration and Programming (configuration and programming actions, adding objects etc.)
bind	Configuration of external references for which the server device will use its own credentials to read or write.
install	Installation (more technical configuration actions, adding IO points, uploading different application programs)
auth	Configuration of authorization-related data.

To aid interoperability and simplify deployment, it is recommended that simple server devices that do not have a need for more granularity than provided by the above table should support the predefined scope meanings and identifiers.

W.3.5.1 Extended Scope Identifiers

While predefined scope identifiers are desirable for the consistency of the authorization server's policies and user access rules, a server device is nonetheless allowed to use scope identifiers that are as specific as needed for data being modeled. This is allowed for highly granular cases for highly sensitive data that requires unique access rules. In this case, the authorization server's policies will have to be configured to match the server device's requirements.

To prevent collisions between implementations, scope identifiers not defined in this standard shall use a prefix in one of two forms:

1) A reversed registered DNS name, followed by a period character. e.g., "com.example.", or
2) A BACnet vendor identifier in decimal, followed by a dash character. e.g., "555-"

Since extended scope identifiers are nonstandard, a server device can describe them for the benefit of clients presenting them to humans. This is done with the optional "/.auth/scope-desc" collection.

For example:

/.auth/scope-desc/com.example.user = "User Configuration Information"
/.auth/scope-desc/555-codes = "Key Codes"

It is not required that a server device provides descriptions for all of the extended scope identifiers that it supports.

W.3.5.2 Use of Scope Identifiers

Although there seems to be an implied hierarchy of the predefined scope identifiers, server devices are not required to know any such hierarchy or overlap. Server devices are normally kept simple by associating a particular piece of data with only a single scope identifier (either predefined or extended). This may be thought of as a "minimum" scope for the data, but any hierarchy or overlap in scope identifiers is a client concept.

Since an OAuth client can request any number of scope identifiers for which it is requesting authorization, the client is free to interpret the scope identifiers as a hierarchy and request multiple scope identifiers as it sees fit. However, the server device is not required to know any such relationships or ordering. The server device simply compares the list of authorized scope identifiers to the collection of scope identifiers associated with the data.

For example, a configuration tool client could request the authorization server for the scope "view adjust override configure". When the server device is presented with a token authorizing "view adjust override configure", it only needs to check that a data item's particular scope, e.g., "adjust", has all of its identifiers present in the authorized list.

This ask-for-more-than-you-need method works as well for extended scope identifiers. Since extended scope identifiers are not defined by this standard, support for any such hierarchy or overlap knowledge of extended scopes is a local matter for the client.

Note that the ability for the authorization server to return fewer scope identifiers than requested is specified in the OAuth RFC 6749 which states: "The authorization server MAY fully or partially ignore the scope requested by the client". Therefore, the client shall always check the returned scope and not assume that it is equal to the requested scope.

If the resource server uses scope identifiers other than the default set defined in W.3.5, then the Authorization Server policies will need to know about these scope identifiers in its policies. It is therefore a possible scenario that an Authorization Server's policies are configured based on knowledge about a logical hierarchy of scope identifiers in the resource server, perhaps based on documentation about the resource server, even if the resource server itself is unaware of any such hierarchy since it simply performs the comparison required by Clause W.3.5.3. For example, if the Authorization Server knows by some means that for a particular resource server, scope "A" > scope "B" > scope "C", so if the client asks for scope "A" it gets in return "A B C". This allows the Authorization Server to anticipate

the needs of the client before the client later encounters data that needs scope "B". This is not considered a "broadening of scope" (which is forbidden by OAuth) because the server has been configured to know the hierarchy and knows that this is safe.

W.3.5.3 Use of Multiple Scope Identifiers

While data will normally be associated with a single scope identifier, there is no prohibition against requiring multiple scope identifiers for a single data item, e.g., "555-foo 555-bar". To successfully access the data, a client shall provide a token with at least the required set of identifiers, e.g., "555-bar 555-foo 555-baz" will work, but "555-foo" will not.

The scope identifiers required for reading and writing are independent and can be different. Therefore a data item can require "555-foo 555-bar" to write but only "555-foo" to read.

The order of scope identifiers is not significant and server devices shall perform the comparison regardless of order.

W.3.6 Audiences

In OAuth, an "audience" is the intended recipient of an access token. As defined by this standard, that means either an individual server device or a group of devices. The audience specifier for an individual server device is the globally unique identifier at "/.auth/dev-uuid". The list of groups that a server device participates in is controlled by the list of globally unique identifiers at the "/.auth/group-uuids" array. Writing to either of these is a critical part of the security configuration of a server device and therefore requires an access token with the "auth" scope.

When a client requests an access token from an internal authorization server, the audience is implicitly the server device and does not need to be specified by the client. An internal authorization server cannot issue tokens for group audiences.

When a client requests an access token from an external authorization server, the client shall specify the identifier of either the individual server device or the group using the "audience" parameter in the OAuth request.

When the audience specifies a group, all members of that group shall share common entries for the external authorization server(s) in order to unambiguously interpret the "iss" field of an access token.

W.3.6.1 The "Audience" Parameter

RFC 6750 states the need for limiting the audience of a token:

> "To deal with token redirect, it is important for the authorization server to include the identity of the intended recipients (the audience), typically a single resource server (or a list of resource servers), in the token."

However, the mechanism for conveying this information from the client to the authorization server is not defined in the RFC. Therefore, this protocol will use the following mechanism to convey this information.

When the client interacts with the authorization server, it shall construct the access token request by adding the audience parameter using the "application/x-www-form-urlencoded" format with a character encoding of UTF-8 in the HTTP request body. The name of the parameter shall be "audience" and the value shall be a URI specifying the UUID of desired audience, as defined in Clause W.3.6. This URI shall be in RFC 4122 format, without the "urn:uuid" prefix.

W.3.7 Access Token Format

The OAuth 2.0 standard does not define the format of the bearer access tokens nor the means by which a resource server verifies the validity of an access token. There are generally two possibilities for such validation: 1) the resource server checks with the authorization server for such validation, or 2) the token is digitally signed such that the resource server can validate it on its own. The latter method is used by this standard.

An access token is a signed unencrypted JSON Web Token (JWT) defined by RFC 7519, which defines the format of a JWT and defines many required and optional items. This standard makes further restrictions and additions for the use of a JWT as an access token.

Requirements for the JWT 'header' object:

The "alg" member shall have a value of either "RS256" or "ES256". Note that the RFC requires support for "HS256" and "none" and defines support for "RS256" and "ES256" to be optional. However, this specification defines "RS256" and "ES256" to be required and does not have defined uses for "HS256" and "none".

Requirements for the JWT 'claims' object:

The "iss" member is required. It shall have the value as follows:

"0" = internal authorization server
"1" = primary external authorization server
"2" = secondary external authorization server

External authorizations servers will therefore need to be configurable to know whether they are primary or secondary and all members of a group shall have identical primary and secondary configurations so that group access tokens can be correctly matched to the issuing authorization server.

The "exp" member is required. If a token is received after the 'exp' time, it shall be rejected with WS_ERR_AUTH_EXPIRED. The lifetime of a token is determined by the policies of the issuing authorization server.

The "aud" member is required and shall be a single string, not an array, with a value in the form of a UUID expressed in RFC 4122 format without the "urn:uuid:" prefix. e.g.,"2d4d11a2-f814-46a7-890a-274a72a7309e".

The "sub" member is optional. If present, it shall consist of a space separated concatenation of the string representations of the numeric "user-id'" and "user-role" fields as defined in Clause 24.2.11. If absent, it is assumed to have the value "0 0". The use of "user-id" and "user-role" for authorization decisions in the resource server is a local matter. It is generally expected that authorization decisions have already been made by the authorization server and are represented by the "scope" member. Therefore the presence of "user-id" and "user-role" in the token is generally only for auditing purposes in the resource server. However, they may also be provided by the authorization server for use by "downstream" communications by gateways to Clause 24 secured networks and devices which may require this information.

Requirements for "private claim names", defined by Section 4.3 of the RFC, to be added to the JWT 'claims' object:

A "ver" member is required and shall have a JSON numeric value of 1.

A "scope" member is required and contains the space-separated list of scope identifiers as a single JSON string value.

W.3.7.1 Access Token Signature Keys

Access tokens are digitally signed to validate that they originated at the appropriate authorization server, both internal and external. These signatures are created with PKI methods, using either the RSA or the Elliptical Curve algorithms.

The generation and storage of the private key used by the internal authorization server is a local matter and there is no need to assign it externally since it is only used between the resource server and its own internal authorization server. Nonetheless, the value for the internal private key shall be sufficiently unique for each instance of a server to prevent an access token issued by one internal authorization server to be accepted by another server device.

The public keys for the external authorization servers shall be configured in /.auth/ext/pri-pubkey and /.auth/ext/sec-pubkey.

W.3.8 Refresh Tokens

This standard does not require the use of OAuth refresh tokens, but neither does it prohibit their use by either the internal or external authorization servers, or by the client. Clients that do not support refresh tokens shall ignore one if it is provided. Therefore, the refresh token mechanism will work when both sides support it and will be harmless when either side does not:

(a) If an authorization server includes a refresh token and the client does not support it, the client shall ignore it.
(b) If an authorization server does not include a refresh token but the client was prepared to work with it, the client shall simply get a new token when needed.

W.3.9 Revocable Access Tokens

This standard does not require the use of revocable access tokens as described in RFC 7009, but neither does it prohibit their use by either the internal or external authorization servers, or by the client. Guidance for issuing and working with revocable tokens through the use of Revocation Requests is defined by RFC 7009, and the details of their implementation in a BACnet/WS server device or client is a local matter.

W.3.10 Summary of Minimum Requirements

Since this specification necessarily enables a broad range of deployment options for security, this clause serves as a reference for creating the smallest compatible secure device. Because of this minimization goal, the optional {prefix} is assumed to be absent in the following URIs.

Table W-2. Minimum ".auth" Requirements

Path	Readability	Writability	Token Source	Audience and Scope
/.auth/int/user /.auth/int/pass /.auth/int/id /.auth/int/secret	Not readable	Required writable, but only with token	From internal authorization server only	Token shall have audience equal to /.auth/dev-uuid (even if it is all zeros) and scope including "auth".
/.auth/int/enable	Readable without authorization			
/.auth/ext/pri-uri /.auth/ext/pri-cert /.auth/ext/pri-pubkey /.auth/ext/sec-uri /.auth/ext/sec-cert /.auth/ext/sec-pubkey	Readable without authorization	Required writable, but only with token	From internal authorization server or a token verifiable with current certificate.	
/.auth/dev-uuid	Readable without authorization	Required writable if not factory set	From internal authorization server only	
/.auth/dev-cert /.auth/ca-certs	Readable without authorization	Always read-only	N/A	N/A
/.auth/dev-cert-pend /.auth/ca-certs-pend /.auth/tls-activate	Readable without authorization	Required writable without TLS and with no authorization when in "factory default" mode. Otherwise, required writable, but only with token.	N/A when in "factory default" mode; otherwise, from internal authorization server or a token verifiable with current certificate	N/A when in "factory default" mode; otherwise, token shall have audience equal to /.auth/dev-uuid and scope including "auth"
/.auth/dev-key-pend	Not readable			

W.4 Sessions

The Web services defined by this standard are stateless and establish no sessions between clients and servers. There is no requirement for any information to be retained on the server from one service invocation to the next.

For computationally expensive operations, like database search operations, the server may wish to retain some cached data to be used on subsequent queries to similar or related data. Since the services defined here are URI based, the server may adorn the URIs returned so that when they are used subsequently the server can locate a cached context from a previous invocation. However, in all cases, such adornment shall not cause that URI to become an invalid identifier for the resource in the future.

For example, if a server wishes to keep a database cursor active in anticipation that the client will request the "next" set of data, the server can adorn the 'next' link with a query parameter like ...&cursor=0Zf93sg to indicate the temporary context. However, if this URI is used at a point in the future when the context identified by the adornment is no longer valid, the adornment shall be ignored and the server shall correctly locate the specified resource, establishing a new context if desired.

W.5 Standard Data Items

While the arrangement of data into hierarchies and the naming of that data is generally a local matter, this standard also defines a number of standardized data items with standardized names and locations that allow clients to obtain information about the server.

The locations, names, types, and presence requirements of the standard data items are summarized in the following table.

The optional path prefix shown as {prefix} in the table is obtained from the data at /.well-known/ashrae. See Clause W.2.

Table W-3. Standard Data Items

Path	Base Type	Presence	Meaning of the Value
/ (when {prefix} is empty) {prefix} (when {prefix} is not empty)	Collection	Required	Merely a container for all the other data items
{prefix}/.info	Composition	Required	Fixed "boiler plate" information about the server device. See Clause W.5.1
{prefix}/.data	Composition	Required	Information about the server device's configuration and queriable lists of things of interest. See Clause W.5.2
{prefix}/.auth	Composition	Required	Contains information for secure communications and authorization. See Clause W.3.3
{prefix}/.defs	Collection	Required	A list of all the definitions that the server has been configured to contain (required, but allowed to be empty).
{prefix}/.subs	Collection	Optional	Required if server supports subscriptions. See Clause W.36
{prefix}/.trees	Collection	Optional	A home for logically modeled arrangements of data. Contains a hierarchy of data items representing a logical arrangement of data.
"{prefix}/.multi	Collection	Optional	Container for reading and writing multiple resources. See Clause W.37.
{prefix}/.{protocol-name}	varies	Optional	A home for protocol specific mappings for protocol modeling. e.g., "/.bacnet" and "/.blt"

W.5.1 The .info Data Item

The .info data item, of type Composition, contains information related to the server's identity, capabilities.

The complete list of children is defined in the following table.

Table W-4. ".info" Data Items

Path	Type	Presence	Description
{prefix}/.info/vendor-identifier	Unsigned	Required	The Vendor_Identifier number, as defined in Clause 23
{prefix}/.info/vendor-name	String	Required	The name of the vendor of this server (unrestricted contents)
{prefix}/.info/model-name	String	Required	The model name and/or number of this server (unrestricted contents)
{prefix}/.info/software-version	String	Required	The version/revision of the software running in this server (unrestricted contents)
{prefix}/.info/protocol-version	Unsigned	Required	The version of the standard that the server device is implementing. It shall be equal to the Protocol_Version defined for the Device object in Clause 12.11.12.
{prefix}/.info/protocol-revision	Unsigned	Required	The revision of the standard that the server device is implementing. It shall be equal to the Protocol_Revision defined for the Device object in Clause 12.11.13.
{prefix}/.info/default-locale	String	Optional	The locale (RFC 3066) that is the default for data read or written without an explicit 'locale' provided.

{prefix}/.info/supported-locales	StringSet	Optional	A list of the locales (RFC 3066) that are supported by the server device. An entry of "*" means the server does not restrict the allowed locale(s), however, the server is allowed to restrict the number of different locales that it will accept subject to its resource constraints.
{prefix}/.info/max-uri	Unsigned	Required	The length limit of the URI that will be accepted by the server for HTTP operations. This shall be a value greater than or equal to 255.

W.5.2 The .data Data Item

The .data data item, of type Composition, contains information related to the server device's configuration and dynamic state.

The complete list of children is defined in the following table.

Table W-5. ".data" Data Items

Path	Type	Presence	Description
{prefix}/.data/database-revision	Unsigned	Required	Indicates the version number for the database - incremented on significant change. Affected by creation or deletion of persistent data like objects. Not affected by dynamic data like subscriptions, values, etc.
{prefix}/.data/histories	List of Link	Required	The paths of all the Trend Logs in the server.
{prefix}/.data/events	List of Link	Required	The paths of all the Event Logs in the server.
{prefix}/.data/objects	List of Links	Required	The paths of all the Objects in the server.
{prefix}/.data/nodes	Collection of List of Link	Required	The collection of all 'nodeType' identifiers in use in the server.
{prefix}/.data/nodes/{node-type}	List of Link	Optional	The paths of all the data items with nodeType={node-type}.

The "histories", "events", and "nodes" constructs are provided as a convenience to the client so that commonly queriable items are gathered into one place and the client does not have to traverse multiple trees or make deep filtered queries looking for interesting things. Each of the Links found here contain an absolute path to an item on the server device. Only data modeled on the server device that is accessible with the protocol defined in this annex shall be included, i.e. no references to other servers or other protocols.

W.5.3 The .auth Data Item

The .auth data item contains information related to the server device's security. The meaning of this data is discussed in Clause W.3. All data under the /.auth path, with the exception of the "{item}-pend" items, shall be nonvolatile. All Certificates shall be X.509 certificates in binary DER format with a mediaType "application/x-x509-ca-cert" and all keys shall be in PKCS #8 binary DER format (RFC 5958) with a mediaType "application/pkcs8". The complete list of children is defined in the following table.

Table W-6. ".auth" Data Items

Path	Type	Presence	Description
{prefix}/.auth/dev-uuid	OctetString	Required	The UUID (RFC 4122) that uniquely identifies this server device.
{prefix}/.auth/ca-certs	List of OctetString	Required	The certificate that was used to sign the dev-cert, plus other certificates that are used to sign device certificates in use by peers
{prefix}/.auth/ca-certs-pend	List of OctetString	Required	The pending certificate that was used to sign the dev-cert-pend, plus other certificates that are used to sign device certificates in use by peers
{prefix}/.auth/dev-cert	OctetString	Required	The device's active unique certificate
{prefix}/.auth/dev-cert-pend	OctetString	Required	The device's pending unique certificate
{prefix}/.auth/dev-key-pend	OctetString	Required	The pending private key corresponding to the pending device certificate
{prefix}/.auth/tls-activate	Boolean	Required	The self-clearing trigger to activate the xxx-pend items.
{prefix}/.auth/int	Composition	Required	A container for things related to the internal authorization server
{prefix}/.auth/int/authorize	N/A	Optional	This is not a data item. It is the "Authorization endpoint" for OAuth 2.0 interactions with the internal authorization server.
{prefix}/.auth/int/token	N/A	Required	This is not a data item. It is the "Token endpoint" for OAuth 2.0 interactions with the internal authorization server.
{prefix}/.auth/int/enable	Boolean	Required	Enables or disables the internal authorization server function. See Clause W.3.3.
{prefix}/.auth/int/user	String	Required	The required base user name.
{prefix}/.auth/int/pass	String	Required	The required base password.
{prefix}/.auth/int/id	String	Required	The required base client id.
{prefix}/.auth/int/secret	String	Required	The required base client secret.
{prefix}/.auth/int/config	Any	Optional	The configuration data (permissions, groups, etc.), for the internal authorization server. The data format and meaning is determined by /.info/vendor-identifier.
{prefix}/.auth/ext	Composition	Required	External authorization server information
{prefix}/.auth/ext/pri-uri	String	Required	The URI end point of the primary external authorization server. See Clause W.3.4.
{prefix}/.auth/ext/pri-cert	OctetString	Required	The public certificate of the primary authorization server
{prefix}/.auth/ext/pri-pubkey	OctetString	Required	The public key used to verify tokens signed by the primary authorization server
{prefix}/.auth/ext/sec-uri	String	Required	The URI end point of the secondary (redundant backup) external authorization server.
{prefix}/.auth/ext/sec-cert	OctetString	Required	The public certificate of the secondary authorization server.

{prefix}/.auth/ext/sec-pubkey	OctetString	Required	The public key used to verify tokens signed by the secondary authorization server
{prefix}/.auth/group-uuids	List of OctetString	Optional	The UUIDs (RFC 4122) for the groups that this server device is part of.
{prefix}/.auth/scope-desc	Collection of String	Optional	Descriptions for extended scope identifiers in use in the server. See Clause W.3.5.1.

W.5.4 The .trees Data Item

The .trees data item is the home of logical arrangements of data in the server. A tree arranges data into hierarchies. An item in the hierarchy can refer to data elsewhere in the server and thus serves to arrange the data for a variety of purposes. Since a common arrangement is by geographic location, a reserved tree name of ".geo" is defined for that purpose. The naming of other trees is a local matter. Semantic tags can be used to give meaning to the other trees.

See Clause Y.2 for more information on trees.

For example, an air handler and its VAVs are modeled as peers under a building in the geographic tree, but a separate air distribution tree gathers data from other locations into an arrangement where VAV terminals are children of the air handler. In the ".geo" tree, therefore, the children of the air handler are its points, while in the "air" tree, the children of the data item referencing the air handler are data items referencing the VAVs.

```
<Collection name=".trees" nodeType="collection">
  <Collection name=".geo" nodeType="tree">
    <Collection name="east" displayName="East Campus" nodeType="area">
      <Collection name="b41" displayName="Chemistry Building" nodeType="building">
        <Composition name="ahu-2" displayName="AHU #2" tags="App-AHU-type-1">
          <Real name="sat" displayName="Supply Air Temperature" nodeType="point"/>
          … more points …
        </Composition >
        <Composition name="vav-2-1" displayName="Room 332B" tags="App-VAV-type-1">
          <Real name="damper" value="100.0" displayName="Damper Position" nodeType="point"/>
          … more points …
        </Composition >
        ... more AHUs, VAVs, and other equipment in this building ...
      </Collection >
    </Collection >
  </Collection >
  <Collection name="air" nodeType="tree">
    <Collection name="a-41-2" represents="/.trees/.geo/east/b41/ahu-2">
      <Collection name="v-41-2-1" represents="/.trees/.geo/east/b41/vav-2-1"/>
      ... more VAVs ...
    </Collection >
    ... more AHUs ...
  </Collection >
  … other trees …
</Collection >
```

W.5.5 The .defs Collection

The data item at path /.defs is a collection of all data type definitions that the server has been configured to contain. There is no requirement for a server to contain the definitions for the data types declared by the data in its database. However, it is very helpful for client interactions if that information is available in the server. This avoids the need for the client to be guided by some other means to finding the definitions. This is potentially a very large collection. However, the most common use case would be for a client to ask for a specific definition using the 'filter' query parameter. See Example W.41.18

W.5.6 The .subs Collection

The data item at path /.subs is a collection of all current subscriptions in the server. Subscriptions are generally created by a client and can be deleted either by the client when they are no longer needed or by the server when they time out. See Clause W.36

W.6 Metadata

All the metadata defined in Annex Y are available by name with the {data-path}/${metadataName} path syntax. e.g., /foo/bar/$displayName. Some metadata in the model can have their representations affected by query parameters. This is used for limiting large sets of data and may actually be required by the server for some metadata, like 'descendants'.

Metadata is also a kind of data; therefore, it can have its own metadata. e.g., /foo/bar/$displayName/$writable is a valid URI. While most metadata will naturally never have a need for metadata itself, there are two strict prohibitions: 'name' and 'value'. There is no URI syntax for accessing either of these. e.g., /foo/$value and /foo/$name are both invalid. This prohibition prevents the server from needing to process the possibly infinite construction /foo/$value/$value/$value..., and the pointless query /foo/$name.

W.7 Functions

In addition to metadata, this standard defines a mechanism for invoking computations on data using "functions". The parameters for these computations, if any, are enclosed by parentheses following the function name.

The computations of functions are affected by the parameters provided by the client and dynamic results are returned. However, as a RESTful protocol design, these computations are not intended to persistently create or delete data. Appropriate POST, PUT, or DELETE operations are used for those actions and are not allowed for functions.

The results of functions are expressed as data items, but the server is not required to further process the results as if the data items were persistent resources, i.e. query parameters like 'filter', 'select', and 'depth' are not required to operate on the results of a function.

Functions are invoked as a URI child of a data item on which they operate. The function's name is followed immediately by an open parenthesis and then by a comma separated series of optionally named parameters and a close parenthesis. Whitespace is not allowed. For example, /some/data/functionname(arg1,arg2). Parameter values containing comma, space, equals, ampersand, or parentheses characters shall have those characters percent-encoded before the entire URI is encoded.

Function parameters have both names and positions, and either or both methods can be used in an invocation, following the same rules as Python. Unnamed parameters provide values for the parameters in the order they are given, and named parameters can be specified in any order. If both are used, the named parameters shall follow all the unnamed parameters. If this rule is violated, e.g., f(a=1,75.5), a WS_ERR_PARAM_SYNTAX shall be returned.

Function parameters can be required or optional. Optional parameters have default values defined by the function. A parameter that is not provided takes on the default value. If a required parameter is missing, a WS_ERR_MISSING_PARAMETER shall be returned. Functions shall define all required parameters positionally ahead of all optional parameters.

Examples: For the function named "doit" that takes three parameters, in order: a String named "foo", an Unsigned named "bar", and a Boolean named "baz" with a default value of 'false', the following are allowed and are equivalent:

doit(hi,2,false) doit(hi,2) doit(hi,baz=false,bar=2) doit(bar=3,foo=hi)

And the following are not allowed:

doit(hi) doit(hi,bar=2,false)

Function names and parameter names shall conform to the restrictions for naming data items. Simple function names, e.g., 'contains', are reserved for definition by ASHRAE. Function names defined by organizations other than ASHRAE shall use a prefix to ensure uniqueness. This prefix shall be either:

1) A reversed registered DNS name, followed by a period character. e.g., "com.example.", or
2) A BACnet vendor identifier in decimal, followed by a dash character. e.g., "555-"

W.7.1 tagged()

The 'tagged' function returns a List of Links pointing to descendants that have the given set of tags. This function returns the same results that can be returned by filtering the 'descendants' metadata with an appropriate series of "$tags/contains({tag}) or $valueTags/exists({tag})" terms along with the 'descendants-depth' query parameter. However, by having a dedicated function, the server device can optimize the execution of this common client action.

The function takes two parameters: "tags" and "depth". The "tags" parameter is a String that is a semicolon separated collection of tags that are required to all be present on the descendant data item in order to be included in the resultant list. The "depth" parameter limits the descendant traversal depth that the function uses to search for tagged data. A "depth" of one searches only the immediate children. The "depth" parameter is optional and if not provided, the search depth is not limited.

W.7.2 historyPeriodic()

The 'history' function returns a processed selection of the trend history for the given data item in a simple plain text format. If the data item does not have an associated history, WS_ERR_NO_HISTORY shall be returned. The function takes four parameters: "start", "period", "count", and "method". These parameters and the format of the returned value are defined in Clause W.11.2

W.7.3 exists()

The 'exists' function returns a Boolean that is 'true' if a specified child data item or metadata item exists, and 'false otherwise. The function takes one parameter named "item", of type String, that is either the name of a child or the name of metadata item prefixed with a "$" character.

W.7.4 remote()

The 'remote' function applies only to Link data and returns a Boolean that is 'false' if the Link's value resolves to a location on the same server device, and 'true' otherwise. The function takes no parameters.

W.7.5 contains()

The 'contains' function applies only to String, StringSet, and BitString data and returns a Boolean that is 'true' if the data item's value contains the provided string and 'false' otherwise. The function takes one parameter named "match", of type String.

For String data, the "match" parameter is compared as a literal case-dependent substring of the data item's value and an empty parameter string will match anything. For example, given a data value of "abcde", parameter values of "ab", "bcd", and "" will all return true.

For BitString and StringSet data, the data value is a semicolon concatenation of individual components. In this case, the "match" parameter is compared to each component in its entirety, not as a substring. For example, given a data value of "abc;def", parameter values of "abc" and "def" will return true, but "ab", "c;d", and "" will return false.

W.7.6 startsWith()

The 'startsWith' function applies only to String data and returns a Boolean that is 'true' if the data item's value starts with the provided string. The function takes one parameter named "match", of type String.

The "match" parameter is compared as a literal case-dependent substring of the data item's value starting at the first character, and an empty parameter string will match anything. For example, given a data value of "abcde", parameter values of "ab", "abc", and "" will all return true.

W.7.7 endsWith()

The 'endsWith' function applies only to String data and returns a Boolean that is 'true' if the data item's value ends with the provided string. The function takes one parameter named "match", of type String.

For String data, the "match" parameter is compared as a literal case-dependent substring of the data item's value from the last character backwards, and an empty parameter string will match anything. For example, given a data value of "abcde", parameter values of "cde", "e", and "" will all return true.

W.8 Query Parameters

Query parameters shall be concatenated and encoded into the 'query' portion of the URI as defined by RFC 3986 using the formatting rules for media type "application/x-www-form-urlencoded". The set of allowed and reserved characters and the required escaping is defined by RFC 3986. If a query parameter is specified more than once, the last one is used and any previous ones are ignored.

Simple query parameters names, e.g., 'skip', are reserved for definition by ASHRAE. Query parameter names defined by organizations other than ASHRAE shall use a prefix to ensure uniqueness. This prefix shall be either:

1) A reversed registered DNS name, followed by a period character. e.g., "com.example.", or
2) A BACnet vendor identifier in decimal, followed by a dash character. e.g., "555-"

If a query parameter has no defined function for a given context, it shall be ignored. Unrecognized query parameters in the ASHRAE reserved name space shall be rejected with WS_ERR_PARAM_NOT_SUPPORTED. Unrecognized query parameters not in the ASHRAE reserved name space shall be ignored. If a bad value is provided for a supported standard parameter, then it shall be rejected with WS_ERR_PARAM_SYNTAX, WS_ERR_PARAM_VALUE_FORMAT, or WS_ERR_PARAM_OUT_OF_RANGE, as appropriate.

Table W-7. Query Parameters

Option Name	Datatype	Applies To	Default if Not Specified
"alt"	string	all	json
"skip"	integer	constructed types, value of String and OctetString	undefined
"max-results"	nonNegativeInteger	constructed types, value of String and OctetString	unlimited
"select"	string	constructed types	all children
"filter"	string	constructed types	no filtering
"depth"	nonNegativeInteger	constructed types	no limit
"descendant-depth"	nonNegativeInteger	'descendant' metadata	no limit
"metadata"	string	all	value and the minimum set of metadata only (see Clause W.15)
"published-gt", "published-ge"	dateTime	time series lists	start of all time
"published-lt", "published-le"	dateTime	time series lists	end of all time
"sequence-gt", "sequence-ge"	nonNegativeInteger	time series lists with inherent sequence numbers	start of sequence
"sequence-lt", "sequence-le"	nonNegativeInteger	time series lists with inherent sequence numbers	end of sequence
"locale"	string	String base type in plain text	system default locale
"error-prefix"	string	all	"?"
"error-string"	string	all	server defined
"reverse"	boolean	time series lists	return records in increasing time or sequence order
"priority"	nonNegativeInteger	commandable data	required when writing a Null, otherwise defaults to 16

W.8.1 alt

Specifies the representation format for the returned data. The choices are "xml", "json", "plain", or "media". Any other value shall result in a WS_ERR_PARAM_OUT_OF_RANGE.

W.8.2 filter

Sets the criteria for which items are to be included in a response. See Clause W.12.

W.8.3 select

Selects by name which children are to be included in a response. See Clause W.14

W.8.4 metadata

Selects which metadata are to be included in a response. See Clause W.15

W.8.5 skip

Controls the starting point for the child data items of a constructed datatype, the characters in a character string, or the octets in an octet string. See Clause W.16.

W.8.6 max-results

Controls the maximum count of the child data items of a constructed datatype, characters in a character string, or octets in an octet string in a response. See Clause W.16.

W.8.7 depth

Limits the depth of the returned data. Applies to the children of constructed data, starting with the first level of data as depth 1. Constructed data at the level of 'depth'+1 are returned with 'truncated' = true and no children. This does not apply to metadata and thus does not limit the depth of constructed metadata like 'choices'. The server device is allowed to truncate responses at levels less than the specified 'depth' if the server is unable to construct or return the data for some reason. The presence of 'truncated' at a level less than 'depth' alerts the client that some of the requested data is missing and that follow-up requests may be required.

W.8.8 descendant-depth

Limits the depth of the 'descendants' metadata. A 'descendant-depth' of 1 will process only the immediate children, a 'descendant-depth' of 2 will include grandchildren, etc. In addition to 'descendant-depth', the 'filter' query parameter can also be used to limit the data that are included in the response. Since the 'descendants' list without filtering could become immense, the server device is allowed to impose its own limitations on the descent. If the server cannot process all of the requested data, it shall return WS_ERR_TOO_DEEP; it shall not silently impose its own depth limiting that would be unknown to the client.

W.8.9 published-gt, published-ge, published-lt, published-le

Controls the range of entries returned in a time series list. The "-xx" postfix determines if the comparison is greater-than, greater-than-or-equal, less-than, or less-than-or-equal. See Clause W.16.4.

W.8.10 sequence-gt, sequence-ge, sequence-lt, sequence-le

Controls the range of entries returned in a sequenced list. The "-xx" postfix determines if the comparison is greater-than, greater-than-or-equal, less-than, or less-than-or-equal. See Clause W.16.5.

W.8.11 reverse

Controls the direction of the records that are returned when reading a List with a range specified by 'published-gt', 'published-ge', 'published-lt', 'published-le', 'sequence-gt', 'sequence-ge', 'sequence-lt', or 'sequence-le'. See Clause W.16.4 and W.16.5.

W.8.12 locale

Controls the locale of the value of String data when accessed in plain text (alt=plain). See Clause W.17.

W.8.13 error-prefix

Controls the format of an error response. Can be used to replace the default "?" prefix. See Clause W.40.

W.8.14 error-string

Controls the format of an error response. Can be used to replace the error text entirely. See Clause W.40.

W.8.15 priority

Controls the priority of a write to a commandable data value. See Clause W.24.

W.9 Representation of Data

The 'alt' query parameter controls the format for representing data.

For the format, alt=xml, data shall be represented as the XML element corresponding to the data item's base type, such as <Real>, <Sequence>, <Array>, etc., as defined in Annex Q. The names for <Array>, <List>, and <SequenceOf> members are required in this context. The HTTP Content-Type shall be set to "application/xml". The CMSL namespace shall be set as the default namespace on the topmost element. The XML format applies to GET, PUT, POST operations. Other methods shall generate a WS_ERR_BAD_METHOD error response.

For the format alt=json, data shall be represented as a combination of JSON values and/or objects as appropriate to represent the data and metadata requested by the client. The HTTP Content-Type shall be set to "application/json", and the topmost data item shall be the anonymous top level JSON object in the HTTP body. The alt=json format applies to GET, PUT, POST operations. Other methods shall generate a WS_ERR_BAD_METHOD error response.

For the format alt=plain, the representation of primitive value data is to return or accept the value in plain text. The representation of other base types in plain text is not defined and shall generate a WS_ERR_NOT_REPRESENTABLE error response. The HTTP Content-Type shall be set to "text/plain". The text is placed directly in the body of the HTTP request or response, and, since escaping and quoting are not necessary, they shall not be used. The textual format shall be the same as the 'value' attribute of the XML format, with the exception that for Date, Time, and DateTime base types, the textual format when 'unspecifiedValue' is true shall be "----/--/--", "--:--:--" and "----/--/--T--:--:--Z", respectively. The alt=plain format applies to GET and PUT operations only. Other methods shall generate a WS_ERR_BAD_METHOD error response.

For the format alt=media, the representation of String or OctetString data is to return or accept the value in a native media format, like "application/pdf". See Clause W.21.1. If 'mediaType' is unknown, then a GET operation cannot generate the correct HTTP Content-Type header and the server shall return a WS_ERR_NOT_REPRESENTABLE response.

For PUT and POST operations, if the provided HTTP Content-Type header is not equal to the appropriate value, a WS_ERR_UNSUPPORTED_MEDIA_TYPE shall be returned.

W.10 Representation of Metadata

If metadata items are represented as part of their associated data item, they are represented in a brief "short form" because their type is fixed and therefore assumed. The "short form" encodes only the metadata name and its value or children.

For example, when included along with it associated data item, (e.g., /path/to/example), the 'minimum' metadata is encoded as:

```
<Real value="74.0" minimum="0.0" displayName="A Real Example" />
```

Or:

```
{"$value":74.0, "$minimum":0.0, "$displayName":"A Real Example"}
```

If an individual metadata item is addressed directly by a URI, it is encoded in the "long form".

For example, when addressed directly (e.g., /path/to/example/$minimum), the 'minimum' metadata is encoded at the top level as:

```
<Real  "value="0.0" />
```

Or:

```
{"$value":0.0}
```

W.11 Representation of Logs

A logically modeled Log is an abstract concept. It can either represent the contents of a protocol-specific source, like the Log_Buffer property of a BACnet Trend Log or BACnet Event Log object, or it can represent log records from any other source. In all cases, a Log Buffer is a List of sequenced and timestamped records.

There are two kinds of Logs - Trends and Events. Each record in a Trend Log is in the form of a BACnetTrendLogRecord, even if the source of the data is not BACnet. Each record in an Event Log is in the form of a BACnetEventLogRecord, even if the source of the data is not BACnet.

Logs shall maintain a record number that monotonically increases as records are added to the log. This is independent of the timestamp and allows the timestamp to not be monotonically increasing. For example, for BACnet Log_Buffer data, this would be the record's "sequence" number. The record number shall be represented as

a decimal number as the name of each record in the List. Clients can use this to detect duplicates and to subsequently access a log buffer using the 'sequence-xx' query parameters.

Logs can be subscribed to for pushed updates. When subscribing to a Log, range parameters like 'published-gt' shall not be given. Subscribers can control how often the list of records is pushed with parameters provided when the subscription is made.

While subscribing to a Log can cause data delivery to be delayed, based on the parameters set in the subscription, polling a Log allows all the records up to the current time to be retrieved. Since Logs are List base types, servers are required to support the query parameters 'published-gt', 'published-ge', 'published-lt', 'published-le', and 'max-results'. In addition, for Log Lists, servers shall support the query parameters 'sequence-gt', 'sequence-ge', 'sequence-lt', and 'sequence-le'.

Logs can be purged by writing an empty List, if allowed by the server.

W.11.1 Trend Logs

Any primitive data can have an associated trend log history. The presence of a history is indicated by the metadata 'hasHistory' set to true, and the availability of 'history' metadata and 'historyPeriodic' function.

If a history is provided by a protocol mapped trend log, e.g., a BACnet Trend Log object, the 'history' metadata and 'historyPeriodic' function shall have a 'viaMap' metadata link so a client can associate that mapped trend log object with the history of the data item. It is possible that a single data item is configured to have multiple histories, perhaps based on different sampling frequencies, triggering, or change of value. In this case, the selection of the log, or a combination of logs, for 'history' and 'historyPeriodic', and the selection of the 'viaMap' metadata, is a local matter.

Some protocol mapped log buffers, like BACnet Trend Log Multiple, can trend multiple data values in a single record. For this reason, multiple data items can be associated with a single mapped trend log, however, each of the data items is a single independent primitive value. Correlation between the data items, if desired, shall be done by the client.

W.11.1.1 Trend Log Records

Trend Log records have all the features of a BACnet TrendLog record even if the source of the data is from another protocol or calculation. This provides a "superset" of features commonly available in control and automation protocols that can be normalized into a format that provides properly sequenced samples (possibly out of timestamp order), buffer status indicators (disabled, enabled, purged, time change, etc.), and detailed error indications.

Trend records are available with the 'history' metadata that is a List of BACnetTrendRecord constructs.

For example, the following shows a time series where a normal sensor value is sampled, then the sensor goes into fault also causing it to go into alarm, and finally in indication that an operator has overridden the value.

```
<List xmlns="http://bacnet.org/csml/1.2" >
   <Sequence name="24216" >
      <DateTime name="timestamp" value="2012-04-02T09:01:00Z" />
      <Choice name="log-datum">
         <Real name="real-value" value="75.2"/>
      </Choice>
      <BitString name="status-flags" value=""/>
   </Sequence>
   <Sequence name="24217" >
      <DateTime name="timestamp" value="2012-04-02T09:02:00Z" />
      <Choice name="log-datum">
         <Real name="real-value" value="99.9"/>
      </Choice>
      <BitString name="status-flags" value="in-alarm;fault"/>
   </Sequence>
   <Sequence name="24218" >
      <DateTime name="timestamp" value="2012-04-02T09:03:00Z" />
      <Choice name="log-datum">
         <Real name="real-value" value="74.0"/>
      </Choice>
      <BitString name="status-flags" value="overridden"/>
   </Sequence>
</List>
```

```
{
   "24216":{
      "timestamp":"2012-04-02T09:01:00Z" ,
      "log-datum":{
         "real-value":75.2
      },
      "status-flags":""
   },
   "24217":{
      "timestamp":"2012-04-02T09:02:00Z",
      "log-datum":{
         "real-value":99.9
      },
      "status-flags":"in-alarm;fault"
   },
   "24218":{
      "timestamp":"2012-04-02T09:03:00Z",
      "log-datum":{
         "real-value":74.0
      },
      "status-flags":"overridden"
   }
}
```

Log status and error records are represented in Log Buffers as separate choices of the "log-datum" Choice to distinguish them from the data values. The possible choices are:

Choice	Type	Value
"time-change"	Real	number of seconds the time has changed
"log-status"	BitString	consists of bit flags "disabled", "interrupted", and "purged"
"failure"	Sequence	the "failure" construct provides "error-class", "error-code", and "error-desc" fields. It is recommended that non-BACnet sources use appropriate "error-class" and "error-code" values when possible, and "error-desc" to provide protocol-specific information.

For example, this list shows a normal sample followed by a time change record and then an error record:

```
<List xmlns="http://bacnet.org/csml/1.2" >
   <Sequence name="24218" >
      <DateTime name="timestamp" value="2012-04-02T09:03:00Z" />
      <Choice name="log-datum">
         <Real name="real-value" value="74.0"/>
      </Choice>
   </Sequence>
   <Sequence name="24219" >
      <DateTime name="timestamp" value="2012-04-02T08:59:57Z" />
      <Choice name="log-datum">
         <Real name="time-change" value="-4.0"/>
      </Choice>
   </Sequence>
   <Sequence name="24220" >
      <DateTime name="timestamp" value="2012-04-02T09:00:25Z" />
      <Choice name="log-datum">
         <Sequence name="failure">
            <Enumerated name="error-class" value="communications" />
            <Enumerated name="error-code" value="other" />
            <String name="error-desc" value="XBus error 9999: invalid foo" />
         </Sequence>
      </Choice>
   </Sequence>
</List>
```

```
{
   "24218":{
      "timestamp":"2012-04-02T09:03:00Z",
      "log-datum":{
         "real-value":74.0
      }
   },
   "24219":{
      "timestamp":"2012-04-02T08:59:57Z",
      "log-datum":{
         "time-change":-4.0
      }
   },
   "24220":{
      "timestamp":"2012-04-02T09:00:25Z",
      "log-datum":{
         "failure":{
            "error-class":"communications",
            "error-code":"other",
            "error-desc":"XBus error 9999: invalid foo"
         }
      }
   }
}
```


W.11.2 Processed Trend Results

In addition to retrieving each trend record individually and performing processing on the client side, it may be desirable and more efficient to perform certain processing on the server side to avoid transmitting a large amount of data when only a few results or calculations are ultimately needed. This server-side processing is available through the 'historyPeriodic' function. All data that have an associated history shall also have a 'historyPeriodic' function available that can be used to perform resampling and certain simple statistical answers for the client.

The 'historyPeriodic' function, e.g., /path/to/data/historyPeriodic(2013-10-14T00:00:00,3600,24), returns a fixed number of plain text results based on the client's query parameters. Each result is either a data value or an error string indicating why the data was not available for that period.

The parameters are, in order: 'start', 'period', 'periods', and 'method'. The following provides information about types and optionality.

When invoking the 'historyPeriodic' function, the client shall provide a starting time with the 'start' parameter, formatted as an xs:dateTime, the duration of each period with the 'period' parameter (see format below), and the number of periods with the 'periods' parameter, formatted as an xs:nonNegativeInteger. With these parameters, the client is specifying the desired sampling rate for the trend series, regardless of the actual sampling rate or timestamps of the data stored in the historical records of the server. If there is a mismatch in the requested sample times and the actual sample times, the server shall process or "resample" the data to find a value for each requested period. If the data is known to the server to not be available for a particular period, the server shall construct an error string for that value. The format of an error string is a question mark character "?" followed by a space followed by an error number defined by Clause W.40 followed by a space followed by a human readable message whose content is a local matter.

The 'period' parameter is either a xs:float formatted number of seconds, or one of the literal strings "minute", "hour", "day", "month", or "year". Note that the special values "month" and "year" have no numerical equivalent.

The client can optionally provide the 'method' parameter, formatted as xs:string, which has a default value of "default", to determine how the resampling or statistical calculation is to be performed. If the 'method' is absent, it shall default to a server-determined method for finding an appropriate value to return for the period. If the 'method'

query parameter is provided, the server shall perform the calculation of the returned values as described in the following table.

Table W-8. History Processing Methods

Parameter Value	Description
"interpolation"	Each data sample returned is determined by straight line interpolation between the record before and the record after the desired sample time. If the server knows that the trend records have a fixed sampling interval and one of the records to be used in this calculation is missing, then an error shall be returned for the sample.
"average"	Each data sample returned is the average of all collected records within the interval centered on the desired sample time. If all records are missing from this interval, then an error shall be returned for the sample.
"minimum"	Each data sample returned is the minimum of all collected records within the interval centered on the desired sample time. If all records are missing from this interval, then an error shall be returned for the sample.
"maximum"	Each data sample returned is the maximum of all collected records within the interval centered on the desired sample time. If all records are missing from this interval, then an error shall be returned for the sample.
"after"	Each data sample returned is the value of the closest record at or after the desired sample time. If the server knows that the trend records have a fixed sampling interval and the appropriate record is missing or is in error, then an error shall be returned for the sample.
"before"	Each data sample returned is the value of the closest record at or before the desired sample time. If the server knows that the trend records have a fixed sampling interval and the appropriate record is missing or is in error, then an error shall be returned for the sample.
"closest"	Each data sample returned is the value of the record closest to the desired sample time. If the server knows that the trend records have a fixed sampling interval and the appropriate record is missing or is in error, then an error shall be returned for the sample.
"default"	The server shall use the most appropriate resample method. The server is not restricted to the standard resample methods and may use any proprietary method suited to the data.
"ending-average"	Each data sample returned is the average of all collected records within the interval that ends, inclusively, at the desired sample time. If all records are missing from this interval, then an error shall be returned for the sample.
"ending-minimum"	Each data sample returned is the minimum of all collected records within the interval that ends, inclusively, at the desired sample time. If all records are missing from this interval, then an error shall be returned for the sample.
"ending-maximum"	Each data sample returned is the maximum of all collected records within the interval that ends, inclusively, at the desired sample time. If all records are missing from this interval, then an error shall be returned for the sample.

Given these capabilities, the 'historyPeriodic' function can also be used to retrieve "statistics" like minimum, maximum, and average for a given period. For example, if the client only wants daily maximums for a week, it can use function parameters like "historyPeriodic(2013-10-14T00:00:00,"day",7,"ending-maximum").

W.12 Filtering Items

Filtering is very useful with large constructed datatypes (for example, the "/.data/objects" list) which might be too large to handle if not filtered. The items returned from a constructed datatype can be filtered with the query parameter 'filter'. Only those items matching the filter criteria shall be returned in the results.

When processing the 'descendants' list, only descendant items that match the 'filter' criteria shall be included in the result. Note that if a data item does not match the criteria, its descendants are nonetheless considered for inclusion in the results, since the normal 'descendants' list is a list of all the descendants individually, subject to limitation only by 'descendant-depth', and filtering is logically applied to every member in the list.

Some filter operations, such as those based dynamic data values, can take a considerable amount of time to execute fully. In order to alleviate this problem, the server is allowed to return partial results to a query and to provide a 'next' link for the client to return at a later time to get the next portion of the results. This keeps the client/server request/response mechanism reasonably responsive even for large difficult queries.

The server shall return the 'partial' metadata as "true" on the data items that have been limited by filtering.

W.12.1 Expression Syntax

The syntax for filter expressions is defined by the following grammar. Whitespace (space or tab in any combination) is not shown in the grammar definition. Whitespace is allowed between all items in the grammar and is required on either side of textual operators like "or" and "not".

```
filterQuery =       "filter=" filterExpression

filterExpression = boolExp  /
                   filterExpression "and" filterExpression  /
                   filterExpression "or" filterExpression  /
                   "(" filterExpression ")"  /
                   "not" filterExpression

boolExp =          relativePath  /
                   relativePath compOp literal  /
                   function  /
                   function  compOp literal  /
                   relativePath "/" function  /
                   relativePath "/" function compOp literal

compOp =           "eq"  /  "ne"  / "gt"  / "ge"  /  "lt" /  "le"

relativePath =     { a relative path (Clause N.2) with respect to each child }

function =         { a function name and arguments (Clause W.7) }
```

The "not" operator takes precedence over the "and" operator which takes precedence over the "or" operator. Parentheses can be used to group sub-expressions to control the application of this precedence.

Literals shall be formatted in the same textual format as they would appear in an XML value attribute.

Paths and literals containing comma, space, equals, ampersand, or parentheses characters shall have those characters percent-encoded prior to encoding the entire URI. Note that function parameters have a similar requirement. See Clause W.7. This rule means that an evil String data item named "object-type eq" would be percent-encoded before

the entire URI is encoded. Thus, if the client wanted to compare that evil data item's value to the literal " eq ", then the first round encoding would result in this:

?filter=object-type%20eq eq %20eq%20

And the final URI will then be finally encoded as:

?filter=object-type%20eq+eq+%20eq%20

This rule is the same for function arguments. This simple encoding scheme eliminates a great deal of complexity that would otherwise go into defining how to quote literals and paths that contain reserved chars.

W.12.2 Expression Evaluation

The valid 'relativePath' components for a filter expression are those strings that can be appropriately appended to the path for the children of the data on which the filter is applied. e.g., if a data item has a path of "http://theserver/thepath", then a 'relativePath' of "foo/bar/$writable" would evaluate the resource at "http://theserver/thepath/{each-child}/foo/bar/$writable". If a path refers to optional data that is absent, it shall evaluate to null in the rules below. Note that the definition of the 'target' metadata is to be an alias for the referent of a Link, or for the data item itself when applied to a non-Link base type.

The evaluation rules are defined as follows:

1. If either side of a binary operation evaluates to a null, the operation shall evaluate to null, with the exception that the "eq" operation will evaluate to 'true' if both sides are null.
2. When a boolExp is evaluating a non-boolean value, it shall evaluate to 'false' if:
   (a) the value is null,
   (b) a value is of type <Null>,
   (c) a numeric value is zero,
   (d) a character string or octet string value is of zero length,
   (e) a bit string value contains no true bits,
   (f) a string set has no members, or
   (g) a date, time, or datetime value is unspecified

W.12.3 Filter Examples

This filter specifies that only items in the "/.data/objects" list that have a property named 'group-number' with a value of 143 will be returned.

```
GET /.data/objects/?filter=group-number eq 143
```

This query returns all objects with a child named "status-flags" that has the "fault" bit set.

```
GET /.data/objects?filter=status-flags/contains(fault)
```

In this example, filtering is used to find data in an archive of data gathered from other sources by looking for references to the original source's 'id'. This query will return all items in the list with a 'sourceId' metadata containing the 'id' metadata of the source data.

```
GET /Our/Log/Archives/?filter=$sourceId eq urn:uuid:1225c695-cfb8-4ebb-aaaa-80da344efa6a
```

W.13 Limiting Number of Items

The number of items returned from a constructed datatype can be limited with the query parameters 'max-results', and 'skip'. These allow the retrieval of a selection of items from any constructed datatype. The 'skip' parameter indicates how many items to skip over, thus a 'skip' of 0 has no effect and a 'skip' of 3 would start the response at the fourth item. The 'max-results' query parameter limits the number of items returned for a response. If the response

would contain more than 'max-results' items, then only the first 'max-results' items are returned, and a 'next' metadata is included for clients to use to obtain the next set of entries.

For constructed datatypes that are defined to be unordered, clients shall be prepared for the contents to change while reading the portions of the data structures using 'skip' and 'max-results'. The strategy for handling this possibility is a local matter.

The server shall return the 'partial' metadata as "true" on the data items that have been limited by this method.

W.14 Selecting Children

The selection of children to be returned for constructed base types can be controlled with the query parameter 'select'. Selecting children that do not exist is not considered an error and shall be ignored.

The syntax for the selection expression is defined by the following grammar. Whitespace (space or tab in any combination) is not shown in this grammar definition. Whitespace is allowed between any of the items of the grammar and shall be ignored by the server device.

selectQuery = "select=" selectClause

selectClause = selectPath [";" selectClause]

selectPath = selectName / ".required" / ".optional" ["/" selectPath]

selectName = {string matching the name of the data item)

The pseudo-name ".optional" matches all the children that are modeled as optional (the 'optional' metadata is true) and ".required" matches all the children that are not modeled as optional.

The server shall return the 'partial' metadata as "true" on the data items that have been limited by this method.

W.15 Controlling Content of Data Representations

During run-time, a client typically only needs to work with the "values" of the data it is querying or updating. However, during discovery, the client needs to know much more about the data, possibly including metadata like datatypes, display names, writability, units, limits, volatility, named values, etc.

Some metadata, like 'etag', 'truncated', 'next', and 'subscription', are always present when required and are not subject to client query parameters.

W.15.1 Default Content

By default, the server shall include only the value and default set of metadata for the top level item in the response. For sub-items, the server shall include the name, value, and default set of metadata. The default set of metadata is defined by the "cat-value" category identifier defined in Clause W.15.2.

Clients need to always be able to determine the base type of all data. Therefore, in addition to this default set of value oriented metadata, the 'base' metadata shall always be present for any data that does not have a corresponding definition or which was defined as an Any base type.

W.15.2 Enhanced Content

The amount of extra metadata returned can be controlled by the query parameter, 'metadata'. If the 'metadata' query parameter is provided, then the "default content" defined by Clause W.15.1 is no longer in effect and the set of values and metadata returned is defined by the table in this clause.

The 'metadata' query parameter value is a comma separated list of one or more metadata names and/or categories of metadata. For example, setting this query parameter as metadata=units,cat-type,cat-ui shall cause the server to return the units metadata and various metadata related to datatype and user interface on all the data for which it knows this

information. See example W.41.7. Additionally, the string "defs" can be added to request that definitions be included in the response as well. See Clause W.15.5.

See Clause W.34 for the server device's requirements for metadata.

The category identifiers allowed for the 'metadata' query parameter are defined by the following table:

Table W-9. Metadata Query Options

Category	Metadata Included
"cat-all"	All the metadata the server knows for all returned data except metadata that is explicitly defined to not be included, like 'count', 'children', 'descendants', and 'history'.
"cat-types"	'base', 'type', 'extends', 'memberType', 'memberTypeDefinition', 'overlays', 'allowedTypes', 'choices', 'allowedChoices','optional', and 'absent'
"cat-tags"	'nodeType', 'nodeSubtype', 'tags', 'valueTags'
"cat-links"	'links', 'self', 'edit', 'alternate', 'via', 'viaMap', 'viaExternal', 'related','sourceId', 'id', 'href' (note that 'next' and 'subscription' appear when, and only when, required and are thus not subject to query parameters)
"cat-ui"	'displayName', 'displayNameForWriting', 'description', 'comment', 'namedValues', 'namedBits'
"cat-doc"	'documentation'
"cat-restrictions"	'units', 'unitsText', 'minimum', 'maximum', 'minimumForWriting', 'maximumForWriting', 'resolution', 'minimumLength', 'maximumLength', 'minimumEncodedLength', 'maximumEncodedLength', 'minimumLengthForWriting', 'maximumLengthForWriting', 'minimumEncodedLengthForWriting', 'maximumEncodedLengthForWriting', 'minimumSize', 'maximumSize', 'associatedWith', ' requiredWith',' requiredWithout',' notPresentWith',' notForReading', ' notForWriting', 'isMultiline','writableWhen', 'writableWhenText', 'requiredWhen', 'requiredWhenText',
"cat-encoding"	'contextTag', 'propertyIdentifier', 'bit'
"cat-mutability"	'writable', 'readable', 'volatility', 'variability', 'commandable', 'writeEffective', 'authRead', 'authWrite', 'authVisible',
"cat-change"	'published', 'updated', 'author'
"cat-extensions"	Any metadata that is not defined by this standard.
"cat-value"	The value of primitive data items and the following metadata when available: 'error', 'errorText', 'unspecifiedValue', 'length', 'mediaType', 'fault', 'inAlarm', 'outOfService', 'overridden','priorityArray','relinquishDefault', 'etag', 'failures'

Metadata not included in the above table is available only by using its name explicitly or with "cat-all".

W.15.3 Implied Content

To prevent redundancy and reduce the size of encoded responses, information that is derivable by some means can be elided from the response. This includes:

(a) Metadata whose values are equal to their base type's default value.
(b) Metadata values that are equal to their defined values, for data items that have a 'type' metadata.
(c) Metadata values that are equal to their "effective values". See Clause W.31.

The server can only return metadata that has a value that is different from the data item's definition. If the server has no definition for the data (no 'type' metadata is known), then all metadata values that are different from their base type's default value are considered to be different and will be returned if requested. It is therefore recommended that server devices make use of 'type' definitions, even if those definitions are local to the server device.

If parentally inheritable metadata, like 'readable', 'writable', 'volatility', 'variability', 'authRead', 'authWrite', 'authVisible', are selected and the effective values for those metadata that are parentally inherited by the top level

data item in a response are different from their default values, then the server shall provide the effective values for those metadata on that top level data item. See Clause W.31 for definition of "effective value".

In order to accommodate a variety of implementation styles with different levels of capabilities for understanding their data definitions, it is not considered a violation of this standard to return metadata values that are, in fact, equal to their defined values. Thus, a server that allocates storage for metadata data upon first write is not required to notice when the value has actually returned to its default value.

W.15.4 The 'type' Metadata

Since the server returns only the metadata that is different from its definition, the client needs to know that definition in order to infer the elided metadata values. If the top level response data is a direct instance of a named type, then that is easy and the 'type' metadata directly specifies that named type. But if the client has "drilled down" into an instance using a URI path that references data that is only a part of a larger named type, then the server needs to reflect that information in the 'type' metadata so the client can know what higher level named definition the referenced data was instantiated from. Therefore, for the top level data item in a response:

(a) If the data item is a direct instance of a defined type, then the 'type' metadata on the data item shall simply be the name of that type definition. This includes members of collections that use 'memberType' since no other metadata can be defined for the members.
(b) If the data item is a part of an instance of larger data definition (it has an anonymous type), then the 'type' metadata shall be the name of the outermost applicable definition plus a slash-separated path down to that data item. This includes members of collections that use 'memberTypeDefinition', since additional metadata can be defined by that construct.

Therefore, given a definition of:

```
<Definitions>
  <Composition name="A" comment="This is an A">
    <String name="a1" />
    <String name="a2" />
    <String name="a3" comment="comment on A/a3"/>
  </Composition>
  <Composition name="B" extends="A" comment="This is a B">
    <String name="a1" comment="comment on B/a1"/>
    <Composition name="b1" type="A">
      <String name="a2" comment="comment on B/b1/a2"/>
    </Composition>
    <List name="b2" memberType="A"/>
    <List name="b3" >
      <MemberTypeDefinition>
        <Composition type="A" comment="an anon ext of A">
          <String name="a2" comment="comment on B/b3/?/a2" />
        </Composition>
      </MemberTypeDefinition>
    </List>
    <Choice name="b4">
      <Choices>
        <Real name="r" comment="comment on B/b4/r"/>
        <Unsigned name="u"/>
      </Choices>
    </Choice>
  </Composition>
</Definitions>
```

For an instance of "B" at /b, the server shall return the 'type' metadata, and the client shall infer the effective value of the 'comment' metadata, for the following paths:

/b/$type = "B"	/b/$comment = "This is a B"
/b/a1/$type = "B/a1"	/b/a1/$comment = "comment on B/a1"
/b/a2/$type = "B/a2"	/b/a2/$comment = ""
/b/a3/$type = "B/a3"	/b/a3/$comment = "comment on A/a3"
/b/b1/$type = "B/b1"	/b/b1/$comment = ""
/b/b1/a1$type = "B/b1/a1"	/b/b1/a1/$comment = "comment on B/a1"
/b/b1/a2$type = "B/b1/a2"	/b/b1/a2/$comment = "comment on B/b1/a2"
/b/b1/a3$type = "B/b1/a3"	/b/b1/a3/$comment = "comment on A/a3"
/b/b2/$type = "B/b2"	/b/b2/$comment = ""
/b/b2/{n}/$type = "A"	/b/b2/{n}/$comment = "This is an A"
/b/b2/{n}/a1/$type = "A/a1"	/b/b2/{n}/a1/$comment = ""
/b/b2/{n}/a2/$type = "A/a2"	/b/b2/{n}/a2/$comment = ""
/b/b2/{n}/a3/$type = "A/a3"	/b/b2/{n}/a3/$comment = "comment on A/a3"
/b/b3/$type = "B/b3"	/b/b3/$comment = ""
/b/b3/{n}/$type = "B/b3/$memberTypeDefinition/1"	/b/b3/{n}/$comment = "an anon extension of A"
/b/b3/{n}/a1/$type = "B/b3/$memberTypeDefinition/1/a1"	/b/b2/{n}/a1/$comment = ""
/b/b3/{n}/a2/$type = "B/b3/$memberTypeDefinition/1/a2"	/b/b3/{n}/a2/$comment ="comment on B/b3/?/a2"
/b/b3/{n}/a3/$type = "B/b3/$memberTypeDefinition/1/a3"	/b/b2/{n}/a3/$comment = "comment on A/a3"
/b/b4/$type = "B/b4"	/b/b4/$comment = ""
/b/b4/r/$type = "B/b4/$choices/r"	/b/b4/r/$comment = "comment on B/b4/r"

W.15.5 Requesting Definitions

Since the metadata that is returned is only that which is different from its definition, it is important for clients to know the definitions to be able to have a complete set of metadata for a given instance. For all the definitions in use in the server, the server shall make the definitions with names that do not begin with "0-" available in the "/.defs" collection. The client is required, therefore, to have a priori knowledge of definitions beginning with "0-", because it cannot rely on the server to contain them.

For definitions that the client does not already know, the client can ask for the definitions to be returned along with instance data. This will give the client the instance differences and the entire definition chain in one response.

If the string "defs" is included in the 'metadata' query parameter, then the server shall include one or more "definition contexts" in the response that shall hold the definitions for each 'type', 'memberType', and 'extends' reference that is included in the response. This shall apply also to 'type', 'memberType', and 'extends' references that are made inside the included definitions. Therefore, the client receives all needed definitions in one response.

Because forward references are allowed in definitions, it is not possible to require a strict order for the included definitions. Therefore, the arrangement of the definitions into one or more definition contexts and their placement within the response is a local matter.

Only top level named definitions shall be returned. Even though the 'type' metadata on an instance can contain a type name and a path, e.g., type="555-Floating-Motor-Object/present-value", the definitions included in the response shall always be the top level named definition, e.g., the definition for "555-Floating-Motor-Object".

See example W.41.29

W.16 Specifying Ranges

Constructed data types can be treated as collections of items that can be accessed in ranges.

W.16.1 Specifying a Range of a List, Array, and SequenceOf

A server shall allow reading the members of a List, Array, and SequenceOf by position with 'skip', and 'max-results'. Either 'skip' or 'max-results' can be omitted. If all of the requested range cannot be returned, the server shall return a 'next' metadata for the next portion. There is no defined way to write a range to a List, Array, or SequenceOf.

Note that 'skip' is the number of items to skip over, not an array index number, and thus is not to be confused with the names of Array members. A skip=1 query for an Array will return the members named (indexed) "2" and above. Array members can also be read individually by addressing them directly in a path by their index number, which is also their name. e.g., /path/to/array/3 accesses the third member of the array.

W.16.2 Specifying a Range of a Sequence, Composition, Collection, or Object

Even though the children of Sequence, Composition, Collection, and Object base types are normally accessed by name, they can nonetheless be read as a partial listing of children. This might be helpful if the number of children is larger than the client or server can handle all at once.

A server shall allow reading by position with 'skip' and 'max-results'. Either 'skip' or 'max-results' can be omitted. If all of the requested range cannot be returned, the server shall return a 'next' metadata for the next portion. There is no defined way to write a range of these base types.

W.16.3 Specifying a Range of a String, OctetString, or Raw

The values of some String, OctetString, or Raw base types might be quite large, possibly representing the entire contents of a file. A range of the value of these base types can be read or written in plain text.

The server shall support the use of 'skip' and 'max-results' to read or write a range of the value for String, OctetString, or Raw base types. The 'skip' and 'max-results' shall specify the number of characters for a String and the number of octets for an OctetString or Raw.

For writing to a String, OctetString, or Raw, if 'skip' is -1, then the starting point is the current end of the data (an "append" operation).

For reading, since the result is in plain text, no server-driven chaining is possible and the server shall either return the requested range or return a WS_ERR_TOO_LARGE error response.

For OctetString and Raw, 'skip' and 'max-results' refer to the underlying binary octet string, not to its textual representation as hexadecimal or base64 characters.

For CharacterString, 'skip' and 'max-results' refer to the underlying characters, not to the octets used to represent the resulting range in UTF-8.

W.16.4 Reading a Range of a Time Series List

A client can read a range of any timestamped List by using an appropriate combination of the query parameters 'published-gt', 'published-ge', 'published-lt', and 'published-le'.

In all cases, the client can also specify 'max-results' to limit the size of a single response.

If a start-time parameter, either 'published-gt' or 'published-ge', is given without an end-time parameter, then the selected range includes the record specified by the start-time parameter up to and including the newest record. If an end-time parameter, either 'published-lt' or 'published-le', is given without a start-time parameter, then the selected range includes the oldest record up to and including the record specified by the end-time parameter.

The server shall return the selected records in the order specified by the 'reverse' query parameter.

If 'reverse' is absent or false, then the selected records shall be returned in increasing time order where the oldest selected record is returned first, followed by records forward in time until either 'max-results' records, the server's size limit, or the most recent selected record is reached. If more records are available than could be returned, the server shall provide a 'next' metadata to the next portion.

If 'reverse' is present and true, then the selected records shall be returned in reverse order where the newest selected record is returned first, followed by records backward in time until either 'max-results' records, the server's size limit, or the earliest selected record is reached. If more records are available than could be returned, the server shall

provide a 'next' metadata to the next portion. Note that even though the link is called "next", in this case, it represents the next portion backward in time.

W.16.5 Reading a Range of a Sequenced List

Some Lists, like Log Buffers, have both timestamps and an inherent sequence number. The client can use the sequence number of a previous request as a parameter for a subsequent request and thus retrieve records by sequence order rather than by timestamp order. The sequence number, if available, can be obtained from a previous time-based response in the name of the record.

A client can read a range of any sequenced List by using an appropriate combination of the query parameters 'sequence-gt', 'sequence-ge', 'sequence-lt', and 'sequence-le'.

An all cases, the client can also specify 'max-results' to limit the size of a single response.

If a start-sequence parameter, either 'sequence-gt' or 'sequence-ge', is given without an end-sequence parameter, then the selected range includes the record specified by the start-sequence parameter up to and including the highest sequence record .If an end-sequence parameter, either 'sequence-lt' or 'sequence-le', is given without a start-sequence parameter, then the selected range includes the lowest sequenced record, up to and including the record specified by the end-sequence parameter.

The server shall return the selected records in the order specified by the 'reverse' query parameter.

If 'reverse' is absent or false, then the selected records shall be returned in increasing sequence order where the lowest sequenced selected record is returned first, followed by records forward in sequence until either 'max-results' records, the server's size limit, or the highest sequenced selected record is reached. If more records are available than could be returned, the server shall provide a 'next' metadata to the next portion.

If 'reverse' is present and true, then the selected records shall be returned in reverse order where the highest sequenced selected record is returned first, followed by records backward in sequence order until either 'max-results' records, the server's size limit, or the lowest sequenced selected record is reached. If more records are available than could be returned, the server shall provide a 'next' metadata to the next portion. Note that even though the link is called "next", in this case, it represents the next portion backward in the sequence order.

W.17 Localized Values

If the server supports localized data, then String data can have multiple values for the different locales. The server declares its support for multiple locales in /.info/supported-locales and indicates which of those is the system default in /.info/default-locale.

All data of base type "String" can have localized values. See the encoding (XML/JSON) clauses for the methods to encode values for different locales.

The 'locale' query parameter controls the locale of the value of String data when accessed in plain text (alt=plain). If provided for other representations or other base types, it shall be ignored.

For getting data, the server shall follow the following rules, in order:

(1) If there is a value with a locale matching exactly the 'locale' parameter, then the server shall return that value.
(2) If there are one or more values with a locale matching the language portion of the 'locale' parameter, then the server shall return one of those values, the selection of which is a local mater.
(3) If there is a value with a locale matching exactly the system default locale, then the server shall return that value.
(4) If there are one or more values with a locale matching the language portion of the system default locale, then the server shall return one of those values, the selection of which is a local mater.
(5) The server shall return any value, the selection of which is a local matter.

For setting data, the server shall follow the following rules, in order:

(1) If the 'locale' parameter is not provided, the server shall set the value in the system default locale and delete all values in other locales.
(2) If the server supports the given locale, then the server shall set the value for that locale, preserving values for other locales.
(3) If the server does not support the given locale, then the server shall return a WS_ERR_LOCALE_NOT_SUPPORTED error response.

See Examples in Clause W.41.27 and W.41.28.

W.18 Accessing Individual Tags and Bits

The BitString and StringSet base types are primitives with a value that is usually accessed and represented as a single quantity. But this value is actually a concatenation of constituent components and in some cases it can be useful to access a single component, for example, to test or set a single bit or a single tag.

Querying a single bit or string in a set is accomplished with the 'contains' function in either a GET operation or in a filter expression. For example,

```
GET /some/bitstringdata/contains(fault)
```

```
GET /some/taggeddata/$tags/contains(ahu)
```

```
GET /some/collection?filter=$tags/contains(ahu)
```

To set or clear an individual bit or add or remove an individual string in a set, the individual component shall be prefixed with a "+" or "-" character. Any number of prefixed items can be specified together, separated by semicolon characters, however, prefixed items cannot be mixed with non-prefixed items. See examples in Clause W.41.12.

If prefixed items are specified along with non-prefixed items, the server shall return a WS_ERR_VALUE_FORMAT and the value shall be modified. If a specified bit does not exist, the server shall return a WS_ERR_VALUE_OUT_OF_RANGE, and the value shall not be modified.

W.19 Semantics

A client's discovery of the "meaning" of data is greatly enhanced by the presence of "semantic tags" on the data, which allow clients to make presentation, reporting, grouping, and operational decisions based on the meaning of the data.

There are three kinds of semantic information. The first is a simple enumeration that acts as a broad categorization, perhaps indicating enough information for a client to pick a display icon, perform grouping, etc. This information is available in the 'nodeType' metadata. Additional semantic information is available in 'nodeSubtype' and as a collections of tags, represented as members of the 'tags' metadata, as described in the Data Model in Annex Y. Finally, semantic information that is "parameterized" (name/value pairs) is represented by the metadata 'valueTags'. See Clause Y.1.4.

For example, this query returns all objects with semantic tag of "meter" from the tag scheme maintained by example.org.

```
GET /.data/objects?filter=$target/$tags/contains(org.example.tags.meter)
```

W.20 Links and Relationships

The data model in Annex Y defines the modeling of relationships between data items by use of link metadata which define a relationship type and provide a pointer to the other data item. See Clause Y.16.

W.21 Foreign XML and Other Media Types

XML data that is not represented in CSML is considered "Foreign XML" and is hosted inside the value of a String. The 'mediaType' metadata of this String shall be set to "application/xml" if no other, more specific, content type is known by the server or provided by the client.

Foreign XML is not required to be understood by the server, and thus the server is not required to be able to descend into foreign XML as if it were composed of data as defined in this annex. For example, if /foo/bar refers to a foreign XML resource, the server is not required to be able to process a request for /foo/bar/baz, even though there may indeed be a "baz" in the foreign XML. Likewise, the server is also not required to support filtering, selecting, limiting, etc., for the foreign XML. Note that the preceding is only a non-requirement, not a prohibition. Some servers may in fact understand certain foreign XML dialects, so support for features like hierarchical descent or filtering into foreign XML remains a local matter.

Other media types, like documents, graphics, etc., shall be hosted inside the value of a String or OctetString, depending on its content type. If the 'mediaType' metadata begins with "text/" then the data shall be contained in a String. Otherwise, it shall be contained in an OctetString. If the content type is unknown, then it shall be contained in an OctetString and the 'mediaType' shall be set to "application/octet-stream".

W.21.1 Direct Media Access: alt=media

Foreign XML and other media types can be accessed without a CSML wrapper using the "alt=media" query parameter. This parameter value shall not be used in conjunction with any other standard query parameters, with the exception of 'locale'. When "alt=media" is specified, the following changes to normal operations occur:

For a GET operation, the server shall return the value directly in the HTTP body with an HTTP Content-Type header equal to the 'mediaType' metadata of the data item. If the 'mediaType' metadata is absent or unknown, then a WS_ERR_NOT_REPRESENTABLE error shall be returned.

For a PUT operation, the data shall have its value updated directly from the body of the HTTP request, and the 'mediaType' metadata shall be updated from the Content-Type header. If the data's base type is not appropriate for the Content-Type, the server shall return a WS_ERR_VALUE_FORMAT error response.

For a POST operation, the server shall examine the HTTP Content-Type header and create an appropriate host data item. The data's value shall be set directly from the body of the HTTP request, and the 'mediaType' metadata shall be set from the Content-Type header.

W.22 Logical Modeling

The data model described in Clause Y.1 is abstract and can be used to represent a wide variety of data from different sources. In a server device, these are primarily split into two types: logical and mapped. Examples of logical data are the normalized points described in Clause Y.1.6 and hierarchical tree data with broad nodeType metadata of "area", "equipment", etc. Mapped data items reflect data that originates in a physical device, for example, BACnet objects, MODBUS registers, etc. Another distinction is that logical data is usually the result of human configuration, either directly in the server or by import from other databases, whereas mapped data might be the result of an automated discovery process.

Support for logical data and/or mapped data is a local matter. Therefore, some servers may only be capable of supporting automatically discovered mapped data from physical devices and objects, and some may only support logical and normalized data. For those servers that support both, this standard provides a method of relating logical data to corresponding mapped data so that clients can move between the two worlds.

As described in Clause Y.1.6, one of the main logical abstractions defined in this standard is that of a "point". Several of the metadata items are dedicated to these logical normalized points, regardless of their origin. In some cases, however, those normalized points might have originated in a physical device, and the following clauses define how to make that association.

W.22.1 Associating Logical and Mapped Points

Points are a common example of data that might exist in both the logical and mapped models. The following shows two paths, one logical and one mapped, that refer to the same point from two different perspectives. The current value of the point in the logical case is the value of the normalized data item directly, whereas in the mapped case, it is in the Present_Value property.

/henry-building/floor-5/AHU-5A/mixed-air-temp <==> /.bacnet/.local/1234/analog-input,2/present-value

The connection between logical and mapped is exposed with the 'viaMap' metadata.

If the server is configured to know a relationship between a logical data item and a mapped object, it shall expose a 'viaMap' metadata on the logical data item.

See example in Clause W.41.15

W.23 Mapped Modeling

The modeling of the communication addresses and geographical location information for mapping data from physical devices shall be accomplished by using a hierarchy of data under a standardized data item corresponding to the protocol that is used by the device. The organization responsible for the protocol shall obtain a registered dot-prefixed name that shall be used as a peer to "/.info". The 'nodeType' metadata of the protocol data item shall be "protocol". The data below the registered protocol name are defined by the registering organization and are beyond the scope of this standard. The path format is:

/.{registered-protocol}/{protocol-specific-parts...}

For example, for the BACnet protocol, mapped data from physical devices would be accessed at

/.bacnet/...

The list of registered protocol prefixes is administrated by ASHRAE and the current list of prefixes, as well as instructions for registering a new prefix, can be obtained from the ASHRAE Manager of Standards.

W.24 Commandability

In addition to data that can be designated as 'writable', data can also be designated as 'commandable'. Commandable data has an associated 'priorityArray' metadata that is a 16 slot array that is compatible with the BACnet command prioritization mechanism defined by Clause 19.2. This is not limited to data originating from BACnet devices. Any data in the logical trees can be designated as commandable and additionally, so other protocols support a BACnet-compatible priority scheme and can thus be designated as 'commandable' in the data model.

For data that originates from a BACnet device or from a protocol that supports a BACnet-compatible priority scheme, the 'priorityArray' and 'relinquishDefault' metadata items shall map bidirectionally to the respective data in the source protocol.

For data that does not originate from a BACnet device or from a protocol that supports a BACnet-compatible priority scheme, the 'priorityArray' and 'reliquishDefault' metadata items shall be local to the BACnet/WS server device. The server device shall operate the priority mechanism as defined in Clause 19.2 with the results being used as the value of the commandable data item.

To command a value at a priority, the client shall write a non-Null value and optionally provide the "priority" query parameter to specify the intended priority. If the "priority" parameter is missing then it defaults to 16.

To relinquish a value at a specific priority, the client shall write a Null base type and shall provide the "priority" query parameter for the intended priority. There is no default priority for the relinquish case because otherwise it would appear that the client is attempting to write a Null to a non-Null data item.

For non-commandable data, if the "priority" query parameter is provided with a non-Null base type, the parameter shall be ignored and the write shall proceed as normal with the provided value. If the "priority" query parameter is provided with a Null base type, the operation shall succeed but no write shall occur.

W.25 Writability and Visibility

The server defines which data items are writable or otherwise mutable, creatable, or deletable, and which are visible for the given authorization context that the client used.

The 'writable' metadata indicates which data items are writable, given a proper authorization context. If the data is writable, but it is not writable at its "self" path, or if an alternate TLS port is required, the server shall make available an 'edit' link metadata that shall indicate an appropriate method to perform the operation. Without such a link metadata, the presence of an 'authWrite' metadata implies that TLS on the default port shall be used.

For data items that require authorization to write, the client shall make use of the 'authWrite' metadata to determine the scope(s) required to perform the modification.

Unless the client has a priori knowledge of the security requirements by some other means, the client shall always read potentially writable data before writing in order to check for 'authWrite' and 'edit' link information. This is particularly significant for highly secure data that should not be sent without first knowing which security mechanism should be used to wrap it safely.

The following scenarios show possible combinations of secured and unsecured writability.

Use Case	Example Read (with query parameter metadata=links,writable,auth)
Read only item	<Real writable="false" value="0.0" />
Writable using plain http on the default http port (i.e. writable at "self")	<Real writable="true" value="0.0" />
Writable with TLS, using a specified scope	<Real writable="true" value="0.0" authWrite="555-xyz" />
Writable with TLS, using an alternate port	<Real writable="true" value="0.0" authWrite="555-alt-sec-xyz" edit="https://theserver:1443/path-to-resource"/>

If the server allows the presence of secured data to be made known on an unsecured channel ('authVisible' is true), then only the name of the unreadable data items shall be included, and the 'authRead' scope needed to read the rest shall be provided. If a different HTTP access method is required, an 'alternate' metadata shall be provided to indicate how to read more of the data in a secure manner. Without such metadata, the presence of an 'authRead' metadata implies that TLS on the default port shall be used. If the server does not allow the presence of a top level data to be made known, then a "404 Not Found" HTTP response is returned. If the server does not allow the presence of lower level data to be made known, then the secured data is simply omitted.

Use Case	Example Data
Top level secured resource on unsecured connection or on secure connection with no read authorization - knowledge of presence allowed. Full access is with TLS on default port using given scope.	<Real name="foo" authRead="555-xyz" />

Top level secured resource on unsecured connection or secure connection with no read authorization - knowledge of presence allowed. Full access is with TLS on an alternate port, using given scope.	<Real name="foo" authRead="555-alt-sec-xyz" alternate="https://theserver:1443/path-to-resource"/> </Real>
Top level secured resource on unsecured connection - knowledge of presence disallowed	(404 error response)
Lower level secured resource on unsecured connection - knowledge of presence allowed	<Sequence name="unsecured-resource" > <Real name="unsecured-child" value="75.0" /> <Real name="secured-child" authRead="xyz" /> </Sequence>
Lower level secured resource on unsecured connection - knowledge of presence disallowed	<Sequence name="unsecured-resource" > <Real name="unsecured-child" value="75.0" /> <!-- "secured-child" is omitted --> </Sequence>

When reading the 'descendants' and 'children' metadata, only the items that are currently visible in the authorization context that the client used shall be included in the list.

The 'edit', 'alternate', 'authRead', and 'authWrite' metadata shall be included in responses when appropriate and required by the above cases, regardless of the presence or value of a client-provided 'metadata' query parameter.

W.26 Working with Optional Data

Some data structures define optional children. The methods for working with optional data are detailed in this clause.

When performing a GET, PUT, or POST method that addresses the parent of an absent data item, the absent data item is simply not present in the representation of the parent structure, just as it would be absent when using ASN.1-based services.

When performing a GET method that directly addresses an absent data item, a WS_ERR_DATA_NOT_FOUND error response shall be returned.

When performing a PUT method that directly addresses an absent data item, the new value is assigned to the item and the item is thus no longer absent.

When performing a DELETE method that directly addresses an optional item: if that item is present, it shall be made absent, and the server shall return an HTTP status code of 204 No Content.

The POST method shall not be used to assign values to absent items directly.

See example W.41.9.

W.27 Working with Optional Metadata

All base types have an applicable set of optional standard metadata. Additionally, a server can optionally allow the creation of proprietary metadata for any data item. The methods for working with metadata are detailed in this clause.

When performing a GET method that addresses a data item, the absent metadata items for that data item are simply not present in the representation of the data item.

When performing a PUT method that addresses a data item:

(a) metadata items that are missing from the representation are not deleted from the target data item.
(b) if metadata items are present in the representation but are not writable or not creatable, they shall be ignored.

When performing a POST method for a new data item:

(a) metadata items that are missing from the representation are either not created or are assigned default values by the server.
(b) if metadata items are present in the representation but are not creatable, they shall be ignored.

When performing a GET method that directly addresses an absent metadata item, the server shall return a WS_ERR_METADATA_NOT_FOUND error response.

When performing a PUT method that directly addresses an absent metadata item:

(a) if the metadata is standard and is not appropriate to the base type of the containing data item, the server shall return a WS_ERR_ILLEGAL_METADATA error response.
(b) if the server does not allow the creation of the metadata item, the server shall return a WS_ERR_CANNOT_CREATE error response.
(c) if the server does not know the base type for a proprietary metadata item and a plain text representation is provided, the server shall return a WS_ERR_NOT_REPRESENTABLE error response.

When performing a DELETE operation that directly addresses a metadata item:

(a) if the metadata is not present, the server shall return a WS_ERR_METADATA_NOT_FOUND error response.
(b) if the metadata is present and not deletable, the sever shall return a WS_ERR_CANNOT_DELETE error response

W.28 Creating Data

If the server allows it, new members of the collection types Collection, List, SequenceOf, and Array shall be creatable by POSTing a fully formed data to the path of the collection. When POSTing to an Array or SequenceOf, the newly created resource is always added to the end of the collection, increasing the size of the collection by one. When POSTing to a List or Collection, the resultant order is a local matter and might cause other members to be rearranged. If an underlying semantic for the container or a limitation of a downstream protocol prevents duplication of members (e.g., BACnet Lists), then server shall respond with WS_ERR_DUPLICATES_NOT_ALLOWED if a duplicate data is POSTed.

The names of members of a List, Array, and SequenceOf are equal to their position in the collection and the client has no control over this. If the client provides a name for the POSTed data, the name shall be ignored by the server.

For the Collection base type, the client can optionally provide a suggested name for the resource in the 'name' of the POSTed data. The server shall accept that suggested name unless it violates the server's restrictions or an existing member already has that name, in which case, the server shall create a suitable name, based on the given name if possible. For example, the server might truncate the provided name, or append a suffix to maintain uniqueness among peers.

The server shall return the location of the newly created resource in the HTTP "Location" header. Since the client does not directly control the path name of the created resource, the client shall note the returned Location header to know the URL of the created resource.

On success, the server shall return a "201 Created" status code. The server is not required to return a body with this status code unless required by the definition of a specific resource. e.g., see the definition of ".multi". Inclusion of a body in other cases is a local matter and clients shall not expect specific contents.

Since a client might be unaware of any atomic-access boundaries imposed by downstream protocols, the server device shall allow creatable data to be created at any depth. The method of creating the provided data into larger atomically-accessed data structures in downstream protocols is a local matter. For example, a gateway to BACnet Clause 15 services can do a read-modify-write cycle on the remote data to effect a POST of an internal portion of a property with a constructed datatype. See Clause W.32 for concurrency control methods that can optionally be employed in such situations.

See example W.41.10.

W.29 Setting Data

Data items are set by using PUT to specify a single resource URI, or by using POST to specify multiple resource URIs using the mechanism defined in W.38. The use of POST for updating a single URI directly is not supported.

The PUT operation is most often used to set the "value" of the target data. In this case, "value" means the value of primitives and atomically associated metadata like 'length' (and, in special cases, 'mediaType' also), the chosen member of a Choice, and the children of collections. The presence of metadata along with the value shall cause the server to set the values for that metadata, which might result in errors if the metadata is not changeable. Therefore, the client shall not provide metadata unless it is intending for the metadata to be written. See Example W.41.11.

The client can PUT a new value directly to a single metadata item, if the server allows it, by using a URI that specifies the metadata directly (e.g., /path/to/data/$description). When accessing metadata items directly in this manner, the client can then provide a value and possibly metadata-of-metadata. See Example W.41.13.

Some primitive values are tightly associated with metadata, e.g., 'length' or 'mediaType". When using plain text (alt=plain), it is not possible to change these metadata at the same time that the value is changed. Therefore, to change something like 'length', either a structured syntax (e.g., XML, JSON) specifying 'length' and 'value' together can be used, or a separate plain text PUT to $length can be used, if allowed by the server.

Since a client might be unaware of any atomic-access boundaries imposed by downstream protocols, the server device shall allow writable data to be written at any depth. The method of inserting the provided data into larger atomically-accessed data structures in downstream protocols is a local matter. For example, a gateway to BACnet Clause 15 services can do a read-modify-write cycle on the remote data to effect a PUT of an internal portion of a property with a constructed datatype. See Clause W.32 for concurrency control methods that can optionally be employed in such situations.

Putting data that is marked as 'partial' is different from putting a complete set of data. In the 'partial' case, existing children that are not included in the provided set of children are not affected because the client is explicitly stating that it is providing only a partial set of children. However, if 'partial' is not true, then the client is intending to provide a complete set of children and any existing children that are not provided in the new set are to be removed.

Upon receipt of a properly authorized PUT, the server shall recursively update the values, metadata, and children in the target data with the values, metadata, and children in the provided data, according to the rules in the following clause. When a rule says to "recurse on" an item, that item becomes the new "target" and the rules are followed again, descending as needed to process all the provided data.

When the rules say to "indicate an error", if that error occurs before any modifications have been made, then the appropriate error response shall be returned by the server. However, if that error occurs after a modification has been made, then an entry shall be made in a 'failures' metadata item that will be returned in a data response. The reason for the failure shall be recorded in the 'error' metadata, and optionally the 'errorText' metadata, on that entry.

If any failure entries have been created, then the server shall return a body containing a single Null base type with the 'failures' metadata present, otherwise the server shall return no body. In both cases, the HTTP status code shall be 200 OK. Therefore, upon receipt of a 200 OK, clients shall look for the presence of 'failures' metadata to know if the PUT operation was completely successful.

W.29.1 Data Updating Rules

The server shall first check that the base type of the provided data matches the target's base type and that none of the server-computed metadata, 'count', 'children', 'descendants', 'truncated', 'history', 'valueAge', 'etag', 'next', 'self', 'edit', 'failures', 'subscription', or 'id', are present. If these basic conditions are not met, a WS_ERR_VALUE_FORMAT error shall be indicated.

For all primitive base types: The server shall update the target value with the value that is provided.

For BitString base types, if the target data length is fixed and the provided 'length' metadata does not match that length, then a WS_ERR_VALUE_FORMAT error shall be indicated.

For all base types: For each metadata item that is provided:

(a) If the corresponding metadata exists, then the server shall descend recursively on that metadata, using the rules in this clause.
(b) If the corresponding metadata does not exist and the provided metadata is not appropriate for the target base type, then a WS_ERR_ILLEGAL_METADATA error shall be indicated.
(c) If the corresponding metadata does not exist and the server does not support the creation of that metadata for the target data, then a WS_ERR_CONNOT_CREATE error shall be indicated.
(d) If the corresponding metadata does not exist and the server supports the creation of the metadata, the server shall create the metadata and then the server shall descend recursively on that metadata, using the rules in this clause.

For Sequence, Composition, and Object base types: For each provided child:

(a) If the corresponding child exists, then the server shall descend recursively on that child using the rules in this clause.
(b) If the corresponding child does not exist but is defined by the target's type to be 'optional' then the child shall be made present and the server shall descend recursively on that child using the rules in this clause.
(c) If the corresponding child does not exist and the server has no definition for the target, then a child shall be created with the provided child's name and base type and the server shall descend recursively on that child using the rules in this clause.

After processing the provided children: If the provided data does not have 'partial' set to true, then for each pre-existing child that was not processed above:

(a) If the target has no definition, then the child shall be deleted.
(b) If the target has a definition and the child is defined to be 'optional' then it shall be made absent.
(c) If the target has a definition and the child is not defined to be 'optional' then a WS_ERR_CANNOT_DELETE error shall be indicated.

For Choice base type: If there is more than one provided child, or if there is no provided child and 'partial' is set not true, then a WS_ERR_VALUE_FORMAT error shall be indicated. If there is a provided child:

(a) If the corresponding child exists, then the server shall descend recursively on that child using the rules in this clause.
(b) If the corresponding child does not exist but is defined by the target's type to be in the 'choices' metadata, then the child shall be instantiated from the definition and the server shall descend recursively on that child using the rules in this clause.
(c) If the corresponding child does not exist and the server has no definition for the target, then a child shall be created with the provided child's name and base type and the server shall descend recursively on that child using the rules in this clause.

If the provided data does not have 'partial' set to true and the pre-existing child was not processed above, the pre-existing child shall be deleted.

For List, SequenceOf, Collection, and Array base types: If the provided data does not have 'partial' set to true, then all children shall be deleted, or, in the case of fixed-sized Arrays, all children shall be reset to their default values. Then, for each provided child:

(a) If the corresponding child exists, then the server shall descend recursively on that child, using the rules in this clause.
(b) If the corresponding child does not exist and the target has a defined member type, then an instance of that type shall be created and the server shall descend recursively on that new member using the rules in this clause.
(c) If the corresponding child does not exist and the target does not have a defined member type, then a new member shall be created with the provided name and base type and the server shall descend recursively on that new member using the rules in this clause.

W.30 Deleting Data

If the server allows it, members of the collection types Collection, List, SequenceOf, and Array may be deleted by a DELETE operation to the path of the member. How the server handles deleting members of an Array is a local matter with the exception that deleting a member of an array other than the last member shall not cause the array to be resized or any members to shift position.

Since a client might be unaware of any atomic-access boundaries imposed by downstream protocols, the server device shall allow deletable data to be deletable at any depth. Note that 'writable" and "deletable" are separate concepts. Just because a data item is deletable does not mean that individual members of that item are necessarily deletable. e.g., in a writable List of Sequence, each List member can be deleted, but that does not mean that the individual members of the Sequence items can be deleted, even though they can be written. The method of deleting the identified data from larger atomically-accessed data structures in downstream protocols is a local matter. For example, a gateway to BACnet Clause 15 services can do a read-modify-write cycle on the remote data to effect a DELETE of an internal portion of a property with a constructed datatype. See Clause W.32 for concurrency control methods that can optionally be employed in such situations.

See example W.41.14.

W.31 Parentally Inherited Values

The metadata 'authRead', 'authWrite', 'authVisible', 'readable', 'writable', 'volatility', and 'variability' apply to all of the data's descendants until overridden by a descendant's local value. The "effective value" is the value of the metadata on the data item itself, if present, otherwise, the "effective value" is the value inherited from the nearest ancestor that has the metadata item present.

To make this work in a simple way for most client scenarios, the reading and writing of these metadata are defined to be asymmetric:

(a) When reading these metadata, the server shall respond with the effective value.
(b) When writing these metadata, the server shall attempt to write a local metadata item to the target data item, thus overriding the inherited value. If the server cannot write the local metadata, it shall return a WS_ERR_CANNOT_CREATE error response.

Note that this inheritance is determined by the single parent/child hierarchy and is not related to arrangements that are made using Links. Therefore, user-interfaces that display trees constructed from Links shall take precautions to not mislead the users as to the inheritance of these metadata, noting particularly the configuration of security information like 'authWrite'.

W.32 Concurrency Control

This protocol uses HTTP ETags, as defined in RFC 2616, for optimistic concurrency control. The use of ETags is optional and is controlled by the server device based on the concurrency requirements of the data that it contains. The server decides which data items require concurrency control and which do not. The server indicates this requirement by the inclusion of an ETag header in an HTTP response. The server shall use weak Etags so that the

data can be written in a different format than that in which it was read (i.e. it is possible to read data in XML and write it back in JSON).

If the server requires an ETag for PUT or POST operations for a particular data item, it shall provide an ETag header in the HTTP response to a GET, POST, or PUT operation on that data item. This Etag header applies only to the top level item. It is not required, nor is it forbidden, that the server also provides that same ETag in the 'etag' metadata for the top level data item. If ETags for lower level data items are required, they shall be encoded in the 'etag' metadata for that data item.

When issuing a PUT, POST, or DELETE request, clients shall indicate an ETag in the If-Match HTTP request header. If, for a given client and/or given authorization, the client is allowed to overwrite any version of the data item in the server, then the value "*" can be used instead.

If a given data item requires an ETag and a client attempts to modify or delete the data item without a matching If-Match header, the server device shall return a 412 Precondition Failed response.

W.33 Server Support for Data Definitions

To interact with the data served by a BACnet/WS server device, the client often needs information about the definition of the data. Therefore, the following requirements are made of all servers so that clients can rely on these capabilities.

(a) If the data is known to conform to a standard BACnet type, the server shall make the metadata 'type' available and, if requested, shall indicate the type name on the top level returned item, or anywhere where the type is ambiguous or unknown (e.g., where "ABSTRACT-SYNTAX.&Type" is used).
(b) If the data is known to conform to another standard (non-BACnet) type, the server device shall make the 'type' metadata available and, if requested, shall indicate the type name on the top level returned item, or anywhere where the type is ambiguous or unknown.
(c) If the data contains Enumerated or BitString instances that use a textual representation instead of the numerical form, then those instances shall either have a 'type' metadata available or shall have namedValues or namedBits metadata available on the instance.
(d) If the data contains Choice instances, then those instances shall either have a 'type' metadata available or shall have 'choices' metadata available on the instance.
(e) In general, if instances are known by the server, either by hard-coding or by discovery, to have any restrictions, descriptions, etc., that can be represented by the metadata defined by this standard, the server device shall either provide a 'type' name referring to a definition where those metadata are defined, or shall make that metadata available on the instance itself. To prevent an explosion of repeated metadata in collections, is it recommended that 'type' be used when possible.

Note that 'type' names might only occur in a particular server device and do not need to be "published" or "standard", although they are still required to be globally unique. For example, a server device might use type "555-RestrictedInt" for a vendor specific type, or even "555-local-3F2504E0-4F89-41D3-9A0C-0305E82C3301-T23" for a server-specific type.

For all 'type' names that do not start with "0-", the server device shall make definitions for those types available in its "/.defs" collection.

W.34 Server Support for Metadata

To allow for minimal implementations, a server is not required to have knowledge of all the metadata provided by the definitions of the type of data it contains, nor is it required to support and retain all metadata provided to it by a client when creating or modifying data. However, it is highly recommended that the server at least knows the 'type' and 'memberType', or 'memberTypeDefinition', of the data it returns.

The required support for 'published' and 'updated' metadata varies according to usage. Where clauses in this standard require a specific meaning of these for their proper operation (e.g., change of value notifications require 'updated'), then that becomes a requirement if the server supports that operation, else support for these is a local matter.

Support for, and retention of, other metadata, e.g., the 'nodeType', 'nodeSubtype', 'displayName', and tagging and other linking metadata, is a local matter and may vary based on context or usage.

However, it is recommended that servers that support logical modeling retain semantic and relationship information ('nodeType', 'tags', and 'links' metadata) when possible.

W.34.1 Server Support for 'href'

When the server encounters an 'href' metadata on a data item during the evaluation of a path component of a URI, and any additional path components follow that data item, then the server shall consider the rules in Clause Y.4.39 to determine whether the additional path component can be evaluated relative to the current data item itself or whether the server needs to traverse the 'href' to the referenced data. If 'href' needs to be traversed, then

(a) If the 'href' refers to a data item on the same server, then the server shall continue its path evaluation from the location referenced by the 'href'. Such traversal shall be done in a manner that does not violate any security requirements of the new location. i.e., the security requirements shall be evaluated as if the URI had referenced the new location directly.
(b) If the 'href' refers to a data item on another server, then, if the evaluation was during an internal operation, then the server shall fail the evaluation with a WS_ERR_CANNOT_FOLLOW error code. If the evaluation was during the processing of a GET, PUT, POST, or DELETE operation, then the server shall either return a properly formed 302 or 307 redirection or shall return a WS_ERR_CANNOT_FOLLOW error response.
(c) If the 'href' uses a scheme other than "http" or "https", then, if the evaluation was during an internal operation, then the server shall fail the evaluation with a WS_ERR_CANNOT_FOLLOW error code. If the evaluation was during the processing of a GET, PUT, POST, or DELETE operation, then the server shall return a WS_ERR_CANNOT_FOLLOW error response.

W.35 Client Implementation Guidelines

This clause provides guidance on the creation and deployment of clients of this protocol.

W.35.1 Client Support for Metadata

Because of the allowance of limited server support for metadata described in Clause W.34, clients shall not make assumptions about what metadata is available and shall be prepared for variations among servers as to what metadata is available for filtering and other queries.

For example, a gateway server might not know the writability or range restrictions of its proxied data, and in this case, the 'writable' metadata would be absent rather than 'true' or 'false. This possibility needs to be considered by the client for operations like filtering.

Also, a server might not have a full complement of UI metadata, like descriptions and documentation, or even display names. If this information is critical to a particular client, then that client needs to be prepared to get the information from elsewhere, for example, by finding or using the data definitions indicated by the 'type' metadata.

W.35.2 Client Bandwidth Consideration

Since web services generally operate over high speed networks, serious consideration shall be given to the downstream impact of repeated client requests. Clients shall be designed so that their requests can be configured to be throttled as appropriate for the ultimate source of the data. This will allow an installer or end user to tailor the client for appropriate behavior so as not to cause an unsustainable load on the rest of the system.

W.35.3 Server Response Size limitations

The client can control the size of a deeply nested response with the 'depth' query parameter; however, the server device is allowed to truncate responses at levels less than the specified 'depth' if the server is unable to construct or return the data for some reason. The presence of 'truncated' at a level less than the specified 'depth' alerts the client that some of the requested data is missing and that follow-up requests may be required.

Additionally, for any top level constructed data types, the server is allowed to provide only a partial set of children and include a 'next' metadata link to instruct the client how to obtain the rest.

W.36 Subscriptions

Subscriptions are created and cancelled by client manipulations of resources on the server. This allows clients to create a subscription and then to have a network visible persistent resource that represents that subscription for the purpose of adding, modifying, refreshing, or cancelling a subscription.

W.36.1 Subscription Resource

Subscriptions are represented by the members of a list at /.subs.

Table W-10. ".subs" Data Items

Path	Type	Description
{prefix}/.subs	Collection	The list of active subscriptions
{prefix}/.subs/{id}	Composition	A single subscription
{prefix}/.subs/{id}/label	String	Client-provided description string whose content is not restricted.
{prefix}/.subs/{id}/callback	String	The URI of the client endpoint to receive the callback POSTs
{prefix}/.subs/{id}/lifetime	Unsigned	The remaining lifetime for this subscription.
{prefix}/.subs/{id}/status	Enumerated	The status of callback success
{prefix}/.subs/{id}/error	String	The error description for callback failure
{prefix}/.subs/{id}/covs	List	The list of COV specifications
{prefix}/.subs/{id}/covs/{n}	Composition	A single COV specification
{prefix}/.subs/{id}/covs/{n}/path	String	The path to the data
{prefix}/.subs/{id}/covs/{n}/increment	Real	The optional threshold for significant change of numeric value types
{prefix}/.subs/{id}/logs	List	The list of log specifications
{prefix}/.subs/{id}/logs/{n}	Composition	A single log specification
{prefix}/.subs/{id}/logs/{n}/path	String	The path to the log buffer
{prefix}/.subs/{id}/logs/{n}/frequency	Enumerated	How often to receive notifications

The "lifetime" value is the remaining lifetime of the subscription, in seconds. The client provides this value when the subscription is created. During operation, it will time down. When it reaches zero, the subscription record shall be deleted. The server is allowed to modify the value at any time to meet its policies.

The "status" enumeration shall be one of the literal values "initializing", "success" or "failure" to indicate the server's ability to reach the callback endpoint. If the "status" value is "failure" then the "error" string shall contain a human readable reason to assist with problem resolution, else it shall be empty.

A "path" shall be the absolute path of the data being subscribed to. Query parameters are not allowed.

The optional "increment" provides the absolute value that constitutes a significant change for numeric values. The behavior in the absence of this value is a local matter. Server support is required. Client inclusion is optional.

The "frequency" for log notifications shall be one of the literal values "instant", "hourly, or "daily".

The "stagger" value allows the client to specify the width, in seconds, of window of time after which a notification would normally be delivered and when it is actually delivered. The server shall pick a random delay within this window to perform the actual notification. This allows clients with a large number of notifications that would otherwise be aligned (e.g., "daily") to request that they be staggered over a period of time to relieve processing and bandwidth requirements.

W.36.2 Creating, Refreshing, Modifying, and Cancelling

To create a subscription, the client shall POST a properly constructed Composition construct to the URI "/.subs". The "covs" and "logs" lists are not allowed to both be empty. If the client plans to make subsequent modifications, it shall note the location of the created resource from the Location HTTP header of the response.

To refresh a subscription, the client shall write directly to the "lifetime" value.

After either creation or a refresh, the server shall send a callback notification with all cov items.

The client shall be allowed to modify the subscription by direct manipulation of the members of the "covs" and/or "logs" lists. Modification of the "covs" list shall cause the server to send notifications for the added items. If both of these lists become empty, the subscription is cancelled.

To cancel a subscription, the client either writes a value of zero to the "lifetime", clears both "covs" and "logs" lists, or uses HTTP DELETE to directly delete the subscription resource.

W.36.3 Callback Notifications

When the conditions are met for the server to send notifications to a subscriber, the server shall POST a new notification record at the callback URI. Servers shall always POST their notifications wrapped in a List which allows for efficiency in cases where multiple notifications are batched and delivered at the same time. This wrapper shall contain a 'subscription' metadata that is the URI of the subscription resource. This allows the client to match a notification with a subscription and to cancel unwanted or forgotten subscriptions.

Each data item in the wrapper shall contain a 'via' metadata indicating the URI as the source of the data. See Examples W.41.25 and W.41.26.

Subscribers shall respond to this POST with a positive HTTP response code (2xx), and the server shall set "status" to "success". Any other response code shall be considered an error and the "status" shall be set to "failure" and the "error" string shall be updated to aid human troubleshooting.

Clients that require secure callback notifications can use an "https:" callback URI and then perform a TLS client certificate validation to ensure that the notification is from the expected source.

Table W-11. Callback Format

Path	Type	Description
/	Composition	The POSTed callback data container
/$subscription	Link	A Link to the subscription
/covs	List	A list of COV values
/covs/{n}	Composition	A single COV value
/covs/{n}/path	String	The path to the data
/covs/{n}/value	Any	The value of the data
/logs	List	The list of log specifications
/logs/{n}	Composition	A single log specification
/logs/{n}/path	String	The path to the log buffer
/logs/{n}/records	List	The list of trend records
/logs/{n}/records/{x}	Sequence	a record of type "0-BACnetLogRecord"

W.37 Reading Multiple Resources

In many cases, the client knows that it wants to read a set of disjoint data and would like an efficient way to retrieve the values of that data in a single batch operation. It can do that with the ".multi" resource, if the server supports it. Additionally if the client wants to read that collection repeatedly, it can construct persistent records under ".multi", if the server allows, and repeatedly query that for updated values. This is similar to the subscriptions mechanism but can be lighter weight, especially when non-persistent, on-the-fly, response is desired.

Table W-12. ".multi" Data Items

Path	Type	Description
{prefix}/.multi	Collection	The list of active subscriptions
{prefix}/.multi/{id}	Composition	A single subscription
{prefix}/.multi/{id}/lifetime	Unsigned	Optional number of seconds that the record will continue to exist before being purged. If absent, a persistent record is not created.
{prefix}/.multi/{id}/values	List	A list of resources to query
{prefix}/.multi/{id}/values/{n}	(varies)	A single resource

The "lifetime" value is the remaining lifetime of the record, in seconds. The client provides this value when the record is created. During operation, it will time down. When it reaches zero, the record shall be deleted. The server is allowed to modify the value at any time to meet its policies.

The "values" list is provided by the client to specify which resources it wants to be a part of the record. Each member of the list shall have the 'via' metadata line present that shall point to the resource.

The 'via' link cannot have any query parameters appended. The 'via' link shall not refer to another ".multi" record.

When a ".multi" record is read, the resource at the 'via' link is evaluated with respect to the current authorization context of the read request and data that is not readable with the given credentials shall have an appropriate 'error' and/or 'errorText' metadata present in the response.

W.37.1 Creating, Refreshing, Modifying, and Cancelling

To initiate a multi request, the client shall POST a properly constructed Composition to the URI "/.multi" resource. The successful response body shall contain that Composition, modified as described below.

The client shall specify each member of the "values" list as an "Any" base type. The presence of non-Any base types will indicate that this is a write operation (see W.38). A mixture of Any and non-Any is ambiguous and shall result in a WS_ERR_INVALID_DATATYPE response.

To create a persistent multi record, the client shall specify a nonzero "lifetime". The client shall note the location of the created resource from the Location HTTP header of the response. The client can then repeatedly read the created record until it is purged by the lifetime reaching zero. If the client wants to persist the record longer, it can write directly to the "lifetime", subject to server restrictions.

Upon receipt of the POST, the server shall replace the "Any" items with the appropriate base type if the 'via' path resolves successfully. The server shall attempt to obtain fresh values for each member of the list and return the updated record in the response to the POST. If the 'via' path does not resolve successfully or any other read error occurs, then the "Any" base type shall remain and an 'error' metadata assigned to it indicate the reason for the error and an entry shall be made into the 'failures' metadata on the top level returned data item.

If the client specifies a nonzero "lifetime" and the server supports creation of persistent records, the server shall respond with the location of the created record in the 'Location' HTTP header and a "201 Created" response with the updated record in the body of the POST response. The server shall re-evaluate the values, subject to its normal data-freshness and caching policies, for each subsequent GET of the record.

To modify a multi record's lifetime, the client shall be able to write directly to the "lifetime" value, however, the client shall not be allowed to modify the "values" list.

If the client specifies a "lifetime" value of zero, or leaves the optional "lifetime" absent, or the server does not support the creation of persistent records, then the server shall return the updated record in the body of the POST response but shall not create a persistent record and shall use a "200 OK" response.

To delete a record, the client either writes a value of zero to the "lifetime", or uses HTTP DELETE to directly delete the record resource.

W.38 Writing Multiple Resources

In many cases, the client knows that it wants to write to a set of disjoint data and would like an efficient way to set the values of those data items in a single batch operation. It can do that with the ".multi" resource, if the server supports it. The structure of the ".multi" resource is described in Clause W.37.

To initiate a multi write request, the client shall POST a properly constructed Composition to the URI "/.multi" resource. This Composition has the same format as the one used for reading multiple values that is defined in Clause W.37. The response body shall contain that Composition, modified as described below.

The client shall specify each member of the "values" list as the specific base type that matches the data item specified by the 'via' link. The "lifetime" shall be absent. The use of the Any base type for all entries signals that this is a read operation (see Clause W.37). A mixture of Any and non-Any is ambiguous and shall result in a WS_ERR_INVALID_DATATYPE response.

Upon receipt of the POST, the server shall attempt to write each provided member of the "values" list to the data item indicated by that member's 'via' link. All of these resources shall be local to the server. To avoid security problems, servers are not allowed to write to resources on other servers.

The 'via' link can have an optional "priority" query parameter appended that will be used during the write process. No other query parameters are allowed. The 'via' link shall not refer to another ".multi" record.

If the 'via' path does not resolve successfully or if a value write fails, at any level and for any reason including security reasons, an 'error' and 'errorText' metadata can be provided on the failed data item and an entry shall be made to the 'failures' metadata that is returned on the top level data item in the response to the POST response. The server shall return the Composition in the body of the POST response with a status code of "200 OK". If no errors were found, the server can return the Composition with 'truncated' true to save bandwidth.

W.39 Mapping of BACnet Systems

The mapping of a BACnet system is represented by data arranged under the "/.bacnet" path.

W.39.1 Accessing BACnet Properties

The format for accessing BACnet properties is:

/.bacnet/{scope}/{device-inst}/{object-type},{object-inst}/{property-id}[/{property-path}]

The {property-path} is defined in the "Accessing BACnet Property Members" clause below.

These path segments and the types of data they represent are summarized in the following table and defined further in the following clauses. Some of these paths are aliases for other paths. These aliases are required to be supported whenever their defined target path is also present.

Table W-13. Accessing BACnet Properties

Path	Type	Children Type
{prefix}/.bacnet	Collection	Collection
{prefix}/.bacnet/{scope}	Collection	Collection
{prefix}/.bacnet/{scope}/.this	Collection	Object
{prefix}/.bacnet/{scope}/{device-inst}	Collection	Object
{prefix}/.bacnet/{scope}/{device-inst}/.device	Object	varies
{prefix}/.bacnet/{scope}/{device-inst}/{object-type},{object-inst}	Object	varies
{prefix}/.bacnet/{scope}/{device-inst}/{object-type},{object-inst}/{property-id}	varies	varies or none
{prefix}/.bacnet/{scope}/{device-inst}/file,{object-inst}/.contents	OctetString or Array of OctetString	none or OctetString
{prefix}/.blt	Collection	Object
{prefix}/.bltd	Object	varies

W.39.1.1 ".bacnet"

The data at "/.bacnet", is a Collection, with 'nodeType' metadata of "protocol", containing the addressing scopes modeled by the server. If a server device is acting as a gateway for multiple addressing scopes, then there will be more than one entry in this list, else there will only be one entry named ".local".

An "addressing scope" is a numbering space that encompasses all the "global" identifiers in a "BACnet internetwork". This provides a context for device identifiers, as defined in Clause 12.11.1, device names, as defined in Clause 12.11.2, access zone identifiers, as defined in 12.32.5, and any other values that are defined to be unique "internetwork-wide".

W.39.1.2 Scope

The data at "/.bacnet/{scope}", is a Collection of devices modeled by the server device for a given addressing scope, including the server device itself. If a server device is acting as a gateway for other BACnet devices in the addressing scope, then there will be more than one entry in this list, else there will only be the server device itself and the ".this" alias.

The choice of names for {scope} is a local matter except that the name ".local" is defined for when the server is not configured to support multiple addressing scopes.

W.39.1.3 ".this"

The URI "/.bacnet/{scope}/.this", is an alias for the device data in the "/.bacnet/{scope}" collection that represents the server device itself in that addressing scope. All operations on the "/.bacnet/{scope}/.this" alias are the same as if performed on the specific device path. It is a URI alias only. Accessing it is the same as accessing the device resource directly. Since it is not an actual data item, it is not included in 'descendants', 'children', etc.

W.39.1.4 Device

Each child of "./bacnet/{scope}" is data that that represents a BACnet device. Each of these is a Collection, with 'nodeType' metadata of "device", containing the object instances in that device. The 'displayName' metadata for the device collection, for all locales, shall be the Object_Name property of the device's Device object without modification.

W.39.1.5 ".device"

The URI "/.bacnet/{scope}/{device-instance}/.device", is an alias for the Device Object in the specified device instance. All operations on the "/.bacnet/{scope}/{device-instance}/.device" alias are the same as if performed on the path including the specific Device Object's name. It is a URI alias only. Accessing it is the same as accessing the Device Object resource directly. Since it is not an actual data item, it is not included in 'descendants', 'children', etc.

W.39.1.6 Object

Each child of a device is data that represents a single object instance. Each of these is an Object base type, each containing the properties for that object. The 'displayName' metadata for the Object, for all locales, shall be the Object_Name property of that object without modification.

The name for object data shall be either the decimal representation of the object type or the enumeration member name from the BACnetObjectType production in Clause 21, followed by a comma, followed by the decimal representation of the object instance number. There shall be no whitespace characters in this concatenation.

The server shall use the name format for all types that were defined for the protocol revision that the server device supports. If the server can contain data from other devices, it shall be configurable by some means to know the names for the object types that it can contain or contain references to. The server shall use numbers in place of names only for data that it has not yet been configured to understand.

W.39.1.7 Property

Each member of an object instance is data that represents a property of the object with 'nodeType' metadata of "property". The base type of the property data, Real, Sequence, etc., varies. The 'displayName' metadata for the property shall be a local matter, however, it is recommended that the server uses the BACnet property names, as appropriate to a supported locale, that were defined for the protocol revision that the server supports, e.g., "Present_Value".

The name for property data shall be either the decimal representation of the property identifier number or the enumeration member name from the BACnetPropertyIdentifier production in Clause 21.

The Property_List property shall not be included in the mapping because that information is available in the 'children' metadata of the Object.

The server shall use the name format for all properties that were defined for the protocol revision that the server device supports. If the server can contain data from other devices, it shall be configurable by some means to know the names for the properties that it can contain or contain references to. The server shall use numbers in place of names only for data that it has not yet been configured to understand. The server shall use Unknown and Raw base types containing the partially parsed ASN.1 encoding for properties that it has not yet been configured to understand. See Clauses Y.12.8 and Y.13.4.

W.39.1.8 Unknown Property Data

Rather than using a single Raw, BACnet property values that are not known to the gateway shall be parsed to the extent possible:

a) open/close tags are represented as an Unknown container
b) application tagged data uses the specific base type
c) context tagged data uses Raw

The whole property shall always be wrapped in an Unknown even if it appears to be a single application or context tagged value. The reason for always wrapping it is that it might actually be a Sequence where some optional items are absent at the moment, or it might be a SequenceOf with one item, or it might be a Choice. Therefore, the only safe thing to do is assume that it might be a constructed type and wrap it in an Unknown so that it does not change types when a second item subsequently appears or the one item changes type or context number.

Example for an unknown property that starts with an open tag:

```
<Unknown  contextTag="1" >                                  open tag
    <Real name="1" value="75.5"/>                           application tag
    <Raw name="2" contextTag="1" value="DEADBEEF" />        context tag
   <Unknown name="3" contextTag="2">                        open tag
      <Real name="1" value="100.0"/>                        application tag
  </Unknown>                                                close tag
</Unknown>                                                  close tag
```

Example for an unknown property that starts with an application tag:

```
<Unknown>
  <Real name="1" value="1.0" />
</Unknown>
```

Example for an unknown property that starts with a context tag:

```
<Unknown>
  <Raw name="1" value="55AA" contextTag="1"/>
</Unknown>
```

Representing date and time types shall be dependent on the definition of the property data elsewhere in this standard. For property data that is defined by the standard to be a "specific time", "specific date", or "specific datetime", the Time, Date, and DateTime base types shall be used. Otherwise, the TimePattern, DatePattern, and DateTimePattern shall be used.

W.39.1.9 ".blt"

The URI "/.blt" is an alias for the URI "/.bacnet/.local/.this". All operations on this alias are the same as if performed on the full path. This is a URI alias only. Since it is not actually a data item, it is not included in 'descendants', 'children', etc.

W.39.1.10 ".bltd"

The URI "/.bltd" is an alias for the path "/.bacnet/.local/.this/.device". All operations on this alias are the same as if performed on the full path. This is a URI alias only. Since it is not actually a data item, it is not included in 'descendants', 'children', etc.

W.39.2 Accessing BACnet File Contents

BACnet File objects shall make a property named ".contents" available that can be used to access the contents of the file. If the file is stream based, the ".contents" property shall be a single OctetString containing the contents of the file. Note that OctetString base types can be read and written partially when using the "alt=plain" representation. See Clause W.16.3. Therefore, when reading with a GET, the "skip" query parameter can be used as the BACnet "file-start-position" and the "max-results" can be used to specify the "requested-octet-count". When writing with a PUT, the "skip" query parameter can be used as the BACnet "file-start-position" and the length of the data is determined by the data value provided in the body. Specifying -1 for 'skip' will append the given data to the end of the OctetString, and therefore to the end of the file.

If the file is record based, then the ".contents" property shall be an Array of OctetString, each member containing a record of the file. Multiple records (Array members) can be read by specifying the "skip" and "max-results" query parameters when accessing the Array with a GET. This would be equivalent to the BACnet "file-start-record" and "requested-record-count". For writing with a PUT, if 'partial' is true, then multiple records can be updated individually. However, if 'partial' is false, then the contents of the file are replaced entirely by the provided records. Note that subsequent records can be appended with a POST.

W.39.3 Accessing BACnet Property Members

Constructed properties have children that are accessed either by name or by decimal position in a hierarchical fashion to any depth.

The server shall use the name format for the fields of all data types that were defined for the protocol revision that the server device supports. If the server can contain data from other devices, it shall be configurable by some means to know the names and base types for the fields of the data types that it can contain. The server shall use numbers in place of names only for data that it has not yet been configured to understand. The server shall use Unknown and Raw base types containing the partially parsed raw ASN.1 encoding for fields that it has not yet been configured to understand.

For Array, List, and SequenceOf data, the path identifier of the children is the numeric position, starting with "1".

For Sequence data, the path identifiers for the children shall be either the string names from the appropriate production in Clause 21, e.g., "change-of-value", for all data types that were defined for the protocol revision that the server supports, or proprietary names for proprietary datatypes.

This process of appending these identifiers can be repeated to any depth as appropriate for the data structure, e.g., "..{property}/change-of-value/cov-criteria/bitmask".

The 'displayName' metadata for fields and members of properties shall be a local matter; however, it is recommended that the server create BACnet property names, as appropriate to a supported locale, for fields that were defined for the protocol revision that the server supports, e.g., "Change of Value".

W.39.4 Creating Objects

To create a BACnet object, the client shall POST an Object base type to the appropriate device path, e.g., POST to "/.bacnet/.local/1234". The Object data shall contain, at a minimum, either the "object-type" property or the "object-identifier" property. If the "object-type" is present and "object-identifier" is absent, then the server shall attempt to create the object in the BACnet device using only the object type information. If the "object-identifier" property is provided, then the server shall attempt to create the object in the BACnet device using the complete object type and instance. If both "object-type" and "object-identifier" are present and inconsistent, then a WS_ERR_INCONSISTENT_VALUES shall be returned. If the creation fails for other reasons, a WS_ERR_CANNOT_CREATE shall be returned.

If the creation was successful, the new resource location shall be indicated with the Location header, and the response will be status code 201 with body if there were any failures or without a body if no failures were encountered.

If other properties are provided, then the server shall attempt to set those properties in the created object, either during initial creation or subsequent write operations, including mapping non-Null priority array slots to the commanded property with the appropriate priority. However, because the support for initial properties is limited by both the capabilities of the server and a possibly remote target BACnet device, the method for setting the initial properties is a local matter and the client shall not expect success in all cases. Individual failure reasons shall be placed on the properties in the returned data. Every failed property shall have an entry in the 'failures' metadata on the Object data that is returned in the POST response.

W.39.5 Deleting Objects

To delete a BACnet object, the client shall DELETE the object resource directly, e.g., DELETE "/.bacnet/.local/1234/analog-value,22". If the deletion succeeds, the server shall return status code "204 No Content". Otherwise, the server shall return a WS_ERR_CANNOT_DELETE error response.

W.40 Errors

To report errors to the client, the server shall use an appropriate HTTP status code with a Content-Type of "text/plain" and a response body containing text that is both machine readable and human readable.

The body of an error response shall consist of at least one line of text. Other lines of text can be provided at the server's option. The format of the first line of text shall normally consist of a question mark followed by a space followed by a decimal error number, defined by the table in this clause, followed by a human readable text to provide details. The content of the human readable text is a local matter; however, it is recommended that it be appropriate to the locale into which the server device is deployed.

The "error-prefix" query parameter shall override the question mark, and the "error-string" shall replace the entire error text including any extra lines. For example, if the normal text is "? 9 The data named 'foo' does not exist", then providing an "error-prefix" of "ERR" will produce a response of "ERR 9 The data named 'foo' does not exist" and providing an "error-string" of "Abc xyz" will produce "Abc xyz" for the entire response body.

Table W-14. Error Numbers

Error Name	Error Number	HTTP Status Code	Example Error Text
WS_ERR_OTHER	0	500	"Other error"
WS_ERR_NOT_AUTHENTICATED	1	401	"Not authenticated"
WS_ERR_NOT_AUTHORIZED	2	401	"Not authorized"
WS_ERR_PARAM_SYNTAX	3	400	"Bad parameter syntax"
WS_ERR_PARAM_NOT_SUPPORTED	4	403	"Parameter not supported"
WS_ERR_PARAM_VALUE_FORMAT	5	400	"Bad parameter value format"
WS_ERR_PARAM_OUT_OF_RANGE	6	403	"Parameter out of range"
WS_ERR_LOCALE_NOT_SUPPORTED	7	403	"Locale Not Supported"
WS_ERR_PATH_SYNTAX	8	400	"Bad path syntax"
WS_ERR_DATA_NOT_FOUND	9	404	"Data not found"
WS_ERR_METADATA_NOT_FOUND	10	404	"Metadata not found"
WS_ERR_ILLEGAL_METADATA	11	404	"Illegal metadata"
WS_ERR_VALUE_FORMAT	12	403	"Bad value format"
WS_ERR_VALUE_OUT_OF_RANGE	13	403	"Value out of range"
WS_ERR_INDEX_OUT_OF_RANGE	14	403	"Index out of range"
WS_ERR_NOT_WRITABLE	15	403	"Not writable"
WS_ERR_WRITE_FAILED	16	403	"Write failed"
WS_ERR_LIST_OF_PATHS_IS_EMPTY	17	403	"No paths provided"
WS_ERR_COUNT_IS_ZERO	18	403	"Requested count is zero"
WS_ERR_INTERVAL_IS_ZERO	19	403	"Requested interval is zero"
WS_ERR_NO_HISTORY	20	403	"No history"
WS_ERR_NO_DATA_AVAILABLE	21	403	"No data available"
(not used)	22	n/a	n/a
WS_ERR_NOT_AN_ARRAY	23	403	"Not an array"
WS_ERR_COMMUNICATION_FAILED	24	403	"Comm with the remote device failed"
(not used)	25	n/a	n/a
WS_ERR_TLS_CONFIG	26	403	"Inconsistent TLS settings"
WS_ERR_NOT_REPRESENTABLE	27	403	"Data not representable in requested format"
WS_ERR_BAD_METHOD	28	405	"Method not allowed for this data"
WS_ERR_TOO_LARGE	29	403	"Requested data or range is too large"
WS_ERR_TOO_DEEP	30	403	"The requested data is too deep to process"
WS_ERR_CANNOT_CREATE	31	403	"Creation of data or metadata not possible"
WS_ERR_CANNOT_DELETE	32	403	"Deletion of data or metadata not possible"
WS_ERR_AUTH_EXPIRED	33	401	"Authorization Expired"
WS_ERR_AUTH_INVALID	34	401	"Invalid Authorization Information"
WS_ERR_MISSING_PARAMETER	35	403	"Missing required parameter"
WS_ERR_UNSUPPORTED_MEDIA_TYPE	36	415	"Unsupported media type"
WS_ERR_UNSUPPORTED_DATATYPE	37	403	"Unsupported data type"
WS_ERR_INVALID_DATATYPE	38	403	"Invalid data type"
WS_ERR_INCONSISTENT_VALUES	39	403	"Inconsistent data or parameter values"
WS_ERR_EXPIRED_LINK	40	404	"Expired link or link context"
WS_ERR_NOT_READABLE	41	403	"Value not readable"
WS_ERR_DUPLICATES_NOT_ALLOWED	42	403	"Duplicates data items are not allowed"
WS_ERR_UNINITIALIZED	43	403	"Data is uninitialized and has no value"
WS_ERR_EXPIRED_CONTEXT	44	403	"Expired Context"
WS_ERR_NOT_ATOMIC	45	403	"Cannot process the data atomically"
WS_ERR_CANNOT_FOLLOW	46	404	"Cannot follow reference to data"

W.41 Examples

Note: these examples are written for human understanding purposes. In most cases, they are not the literal character-for-character exchanges. In all of these examples, the "HTTP/1.1" is left off of requests and responses, most HTTP headers are not shown, and all URLs are left unencoded. This concession for readability is not to be taken as a requirement of this specification. All operations shall conform to relevant standards.

W.41.1 Getting the {prefix} to Find the Server Root

Given the following exchange, the client will substitute "/myprefix" for "{prefix}" where indicated elsewhere in this specification. (note that in all the examples following this, the {prefix} is assumed to be the empty string). The client interprets this example as a single BACnet/WS server at prefix "/myprefix" using "http" on the standard port 80 and "https", if the server uses it, on the standard port 443.

```
GET /.well-known/ashrae
```

```
200 OK
Content-Type: text/plain

Link: </myprefix>; rel="http://bacnet.org/csml/rel#server-root"; title="A really great server"
```

In this complex example, the client finds several BACnet/WS servers when reading the /well-known/ashrae file on host.example.com. Server A is on host.example.com rooted at "/a" and uses standard ports. Server B is also host.example.com, rooted at "/b", and also uses standard ports. Server C is on host.example.com but uses nonstnadard ports. Server C is on another host, and uses standard ports. Server E is on another host and uses non-standard ports.

```
GET /.well-known/ashrae
```

```
200 OK
Content-Type: text/plain

Link: </a>; rel="http://bacnet.org/csml/rel#server-root"; title="Server A"
Link: </b>; rel="http://bacnet.org/csml/rel#server-root"; title="Server B"
Link: <http://host.example.com:8080/c>; rel="http://bacnet.org/csml/rel#server-root"; title="Server C"
Link: <https://host.example.com:1443/c>; rel="http://bacnet.org/csml/rel#server-root"; title="Server C"
Link: <http://another.example.com/d>; rel="http://bacnet.org/csml/rel#server-root"; title="Server D"
Link: <http://another.example.com:8080/e>; rel="http://bacnet.org/csml/rel#server-root"; title="Server E"
Link: <http://another.example.com:1443/e>; rel="http://bacnet.org/csml/rel#server-root"; title="Server E"
```

W.41.2 Getting Metadata

In this example, the client is requesting a single metadata item by referencing it directly.

Plain Text:

```
GET /path/to/primitive-data/$maximum?alt=plain
```

```
200 OK
Content-Type: text/plain

100.0
```

XML:

```
GET /path/to/primitive-data/$maximum?alt=xml
```

```
200 OK
Content-Type: application/xml

<?xml version='1.0' encoding='utf-8'?>
<Real value="100.0" xmlns="http://bacnet.org/csml/1.2"/>
```

JSON:

```
GET /path/to/primitive-data/$maximum
```

```
200 OK
Content-Type: application/json

{"$value":100.0}
```

W.41.3 Getting Primitive Data

Plain Text:

```
GET /path/to/primitive?alt=plain
```

```
200 OK
Content-Type: text/plain

75.0
```

XML:

```
GET /path/to/primitive?alt=xml
```

```
200 OK
Content-Type: application/xml

<?xml version='1.0' encoding='utf-8'?>
<Real value="75.0" xmlns="http://bacnet.org/csml/1.2"/>
```

JSON:

```
GET /path/to/primitive
```

```
200 OK
Content-Type: application/json

{"$value":75.0}
```

W.41.4 Getting Constructed Data

XML:

```
GET /path/to/array?alt=xml
```

```
200 OK
Content-Type: application/xml

<?xml version='1.0' encoding='utf-8'?>
<Array xmlns="http://bacnet.org/csml/1.2">
   <Real name="1" value="75.0"/>
   <Real name="2" value="73.5"/>
   <Real name="3" value="77.5"/>
</Array>
```

JSON:

```
GET /path/to/array
```

```
200 OK
Content-Type: application/json

{
   "1":75.0,
   "2":73.5,
   "3":77.5
}
```

W.41.5 Limiting the Response Size

However, in this example, the client cannot read the entire contents of an array at once, so it limits the size of the response using the 'max-results'. Because only some of the entries are returned, the server provides a 'next' link for the client to be able to chain to the rest of the members. Note that the format of the 'next' pointer is completely server-defined. This example shows the usage of the original resource path with a 'cursor' query parameter, but the server could also have returned "http://theserver.example.com/next?context=38yr9w8" or any other valid URI.

XML:

```
GET /path/to/array?max-results=2 &alt=xml
```

```
200 OK
Content-Type: application/xml

<?xml version='1.0' encoding='utf-8'?>
<Array xmlns="http://bacnet.org/csml/1.2"
          next="http://theserver.example.com/path/to/array-data?cursor=EF38C2AF">
   <Real name="1" value="75.0"/>
   <Real name="2" value="73.5"/>
</Array>
```

JSON:

```
GET /path/to/array?max-results=2
```

```
200 OK
Content-Type: application/json

{
   "$next":"http://theserver.example.com/path/to/array-data?cursor=EF38C2AF",
   "1":75.0,
   "2":73.5
}
```

W.41.6 Getting Time Series Records from a BACnet Trend Log

The following example shows a log buffer being read as a list of individual CSML records.

XML:

```
GET /path/to/primitive/.history?published-ge=2012-04-02T09:00:00Z&published-lt=2012-04-
02T09:02:00Z&alt=xml
```

```
200 OK
Content-Type: application/xml

<?xml version='1.0' encoding='utf-8'?>
<List xmlns="http://bacnet.org/csml/1.2" >
   <Sequence name="24216">
      <DateTime name="timestamp" value="2012-04-02T09:00:00Z"/>
      <Choice name="log-datum" >
         <Real name="real-value" value="75.5">
      </Choice>
   </Sequence>
   <Sequence name="24217">
      <DateTime name="timestamp" value="2012-04-02T09:01:00Z"/>
      <Choice name="log-datum" >
         <Real name="real-value" value="76.0"/>
      </Choice>
   </Sequence>
</List>
```

JSON:

```
GET /path/to/primitive/.history?published-ge=2012-04-02T09:00:00Z&published-lt=2012-04-
02T09:02:00Z
```

```
200 OK
Content-Type: application/json

{
   "24216": {
      "timestamp":"2012-04-02T09:00:00Z",
      "log-datum": {
         "real-value":75.5
      }
   },
   "24217": {
      "timestamp":"2012-04-02T09:01:00Z",
      "log-datum": {
         "real-value":76.0
      }
   }
}
```

W.41.7 Controlling CMSL Metadata with the 'metadata' Parameter

Without the 'metadata' query parameter, the server would return only 'name' and 'value' where appropriate, as specified in W.15.1:

XML:

```
GET /path/to/great-thing?alt=xml
```

```
200 OK
Content-Type: application/xml

<?xml version='1.0' encoding='utf-8'?>
<Sequence  xmlns="http://bacnet.org/csml/1.2">
   <Real name="foo" value="75.0"/>
   <Array name="bar">
      <Real name="1" value="123.0"/>
      <Real name="2" value="456.0"/>
   </Array>
   <Enumerated name="baz" value="medium"/>
</Sequence>
```

JSON:

```
GET /path/to/great-thing
```

```
200 OK
Content-Type: application/json

{
  "foo":75.0,
  "bar": {
    "1":123.0,
    "2":456.0
  },
  "baz":"medium"
}
```

Setting the 'metadata' parameter to "cat-types" causes the server to return all the type data it knows.

XML:

```
GET /path/to/great-thing?metadata=cat-types&alt=xml
```

```
200 OK
Content-Type: application/xml

<?xml version='1.0' encoding='utf-8'?>
<Sequence type="555-GreatType" xmlns="http://bacnet.org/csml/1.2">
   <Real name="foo"  />
   <Array name="bar" memberType="Real">
      <Real name="1"/>
      <Real name="2" />
   </Array>
   <Enumerated name="baz" type="555-EnumLMH">
</Sequence>
```

JSON:

```
GET /path/to/great-thing?metadata=cat-types
```

```
200 OK
Content-Type: application/json

{
   "$base":"Sequence",
   "$type":"555-GreatType",
   "foo": { "$base":"Real" },
   "bar": {
      "$base":"Array",
      "$memberType":"Real",
      "1": { "$base":"Real" },
      "2": { "$base":"Real" }
   },
  "baz"{  "$base":"Enumerated", "$type":"555-EnumLMH" }
}
```

Setting the 'metadata' parameter to "cat-all" causes the server to return all the metadata that is different from the type's definition, in addition to all type information.

XML:

GET /path/to/great-thing?**metadata=cat-all**&alt=xml

```
200 OK
Content-Type: application/xml

<Sequence displayName="A Really Great Thing" type="555-GreatType"
   id="urn:uuid:1225c695-cfb8-4ebb-aaaa-80da344edead" description="blah blah blah"
   writable="true" xmlns="http://bacnet.org/csml/1.2">
   <Real name="foo" value="75.0" displayName="Foo, James Foo"
         units="watts-per-square-meter-degree-kelvin"/>
   <Array name="bar" maximumSize="20">
      <Real name="1" value="23.0" maximum="100.0"/>
      <Real name="2" value="56.0" maximum="100.0"/>
   </Array>
   <Enumerated name="baz" value="medium" type="555-EnumLMH"/>
</Sequence>
```

JSON:

GET /path/to/great-thing?**metadata=cat-all**

```
200 OK
Content-Type: application/json

{
   "$base":"Sequence",
   "$type":"555-GreatType",
   "$id":"urn:uuid:1225c695-cfb8-4ebb-aaaa-80da344edead"
   "$displayName":"A Really Great Thing",
   "$description":"blah blah blah",
   "$writable":true,
   "foo": {
      "$base":"Real",
      "$value":75.0,
      "$displayName":"Foo, James Foo",
      "$units":"watts-per-square-meter-degree-kelvin"
   },
   "bar": {
      "$base":"Array",
      "$memberType":"Real",
      "$maximumSize":20,
       "1": { "$base":"Real", "$value":23.0, "$maximum":100.0 },
       "2": { "$base":"Real", "$value":56.0, "$maximum":100.0 }
   },
   "baz": {
      "$base":"Enumerated",
      "$type":"555-EnumLMH",
      "$value":"medium"
   }
}
```

W.41.8 Getting a Filtered List of Objects and Properties

The /.data/objects list contains links to all objects known to the server; therefore, some kind of filtering with 'filter' is usually applied.

XML:

```
GET /.data/objects?filter=$target/object-type eq analog-input or $target/object-type eq analog-value&alt=xml
```

```
200 OK
Content-Type: application/xml

<?xml version='1.0' encoding='utf-8'?>
<List xmlns="http://bacnet.org/csml/1.2">
   <Link name="1" value="/.bacnet/.local/1234/analog-input,14"/>
   <Link name="2" value="/.bacnet/.local/4321/analog-input,2"/>
   <Link name="3" value="/.bacnet/.local/4321/analog-value,1"/>
</List>
```

JSON:

```
GET /.data/objects?filter=$target/object-type eq analog-input or $target/object-type eq analog-value
```

```
200 OK
Content-Type: application/json

{
   "1":"/.bacnet/.local/1234/analog-input,14",
   "2":"/.bacnet/.local/4321/analog-input,2",
   "3":"/.bacnet/.local/4321/analog-value,1"
}
```

W.41.9 Working with Optional Data

When the parent of an optional item is addressed, the missing fields (here, "deviceIdentifier") are simply missing from the serialization.

XML:

```
GET /path/to/some-address?alt=xml
```

```
200 OK
Content-Type: application/xml

<?xml version='1.0' encoding='utf-8'?>
<Sequence xmlns="http://bacnet.org/csml/1.2">
   <ObjectIdentifier name="objectIdentifier" value="analog-input,1"/>
</Sequence>
```

JSON:

```
GET /path/to/some-address
```

```
200 OK
Content-Type: application/json

{
   "objectIdentifier":"analog-input,1"
}
```

When the missing item is addressed directly, an error is generated.

```
GET /path/to/some-address/deviceIdentifier
```

```
404 Not Found

9 The child 'deviceIdentifier' does not exist
```

When the missing item is addressed directly with a PUT, the new value is set and shall appear in subsequent GETs.

```
PUT /path/to/some-address/deviceIdentifier?alt=plain
Content-Type: text/plain

device,1234
```

```
204 No Content
```

When read back, the parent now shows the new value.

XML:

```
GET /path/to/some-address?alt=xml
```

```
200 OK
Content-Type: application/xml

<?xml version='1.0' encoding='utf-8'?>
<Sequence xmlns="http://bacnet.org/csml/1.2">
   <ObjectIdentifier name="objectIdentifier" value="analog-input,1"/>
   <ObjectIdentifier name="deviceIdentifier" value="device,1234"/>
</Sequence>
```

JSON:

```
GET /path/to/some-address
```

```
200 OK
Content-Type: application/json

{
  "objectIdentifier":"analog-input,1",
  "deviceIdentifier":"device,1234"
}
```

W.41.10 Creating Data

New resources are added to collection with a POST operation. Note that not all metadata is retained for this object and that the suggested resource name was modified to fit the server's naming rules.

XML:

```
POST /foo/biglist?alt=xml
Content-Type: application/xml

<?xml version='1.0' encoding='utf-8'?>
<Object name="EC:b41:meter" xmlns="http://bacnet.org/csml/1.2"
        displayName="East Campus, Building 41 Meter"
        description="MeterCo model FS342" >
        tags="org.example.tags.meter" >
   ...properties...
</Object>
```

```
201 CREATED
Location: http://server/foo/biglist/EC_b41_meter
...
```

JSON:

```
POST /foo/biglist
Content-Type: application/json

{
   "$name":"EC:b41:meter",
   "$displayName":"East Campus, Building 41 Meter",
   "$description": "MeterCo model FS342",
   "$tags": "com.example.tags.meter",
   ...properties...
}
```

```
201 CREATED
Location: http://server/foo/biglist/EC_b41_meter
...
```

W.41.11 Putting Data

This example shows setting a new value for a data item.

XML:

```
PUT /path/to/primitive-data?alt=xml
Content-Type: application/xml

<?xml version='1.0' encoding='utf-8'?>
<Real xmlns="http://bacnet.org/csml/1.2" value="99.9"/>
```

```
204 No Content
```

JSON:

```
PUT /path/to/primitive-data
Content-Type: application/json

{"$value":99.9}
```

```
204 No Content
```

Plain Text:

```
PUT /path/to/primitive-data?alt=plain
Content-Type: text/plain

99.9
```

```
204 No Content
```

W.41.12 Putting Individual Bits and Tags

This example shows setting the 'foo' bit and clearing the 'bar' bit:

XML:

```
PUT /some/bitstring?alt=xml
Content-Type: application/xml

<?xml version='1.0' encoding='utf-8'?>
<BitString xmlns="http://bacnet.org/csml/1.2" value="+foo;-bar" />
```

```
204 No Content
```

JSON:

```
PUT /some/bitstring
Content-Type: application/json

{"$value":"+foo;-bar"}
```

```
204 No Content
```

Plain Text:

```
PUT /some/bitstring?alt=plain

+foo;-bar
```

```
204 No Content
```

This example adds the 'newTag' tag and removes the 'oldTag' tag:

XML:

```
PUT /some/taggeddata/$tags?alt=xml
Content-Type: application/xml

<?xml version='1.0' encoding='utf-8'?>
<StringSet xmlns="http://bacnet.org/csml/1.2" value="+newTag;-oldTag" />
```

```
204 No Content
```

JSON:

```
PUT /some/taggeddata/$tags
Content-Type: application/json

{"$value":"+newTag;-oldTag"}
```

```
204 No Content
```

Plain Text:

```
PUT /some/taggeddata/$tags?alt=plain

+newTag;-oldTag
```

```
204 No Content
```

W.41.13 Putting Metadata

This example shows setting a new value for a metadata item (if allowed by the server). Since accessing a metadata item directly promotes it to the top level, the metadata item is represented as its appropriate base type and thus exposes its value for modification.

XML:

```
PUT /path/to/primitive/$description?alt=xml
Content-Type: application/xml

<?xml version='1.0' encoding='utf-8'?>
<String  value="a new description!"  xmlns="http://bacnet.org/csml/1.2" />
```

```
204 No Content
```

JSON:

```
PUT /path/to/primitive/$description
Content-Type: application/json

{"$value":"a new description!"}
```

```
204 No Content
```

Plain Text:

```
PUT /path/to/primitive/$description?alt=plain
Content-Type: text/plain

a new description!
```

```
204 No Content
```

W.41.14 Deleting Data

Data in collections and optional data items are removed by directly addressing the resource with a DELETE operation.

Removing a previously created list member:

```
DELETE /foo/biglist/EC_b41_meter
```

```
204 No Content
```

Removing an optional data field:

```
DELETE /foo/some-address/deviceIdentifier
```

```
204 No content
```

W.41.15 Logical Tree Data Associated with a Mapped Object

In this example, a mixed air temperature is logically modeled under the air handler as a normalized "Point" child of the air handler. Logical, normalized points are not necessarily tied to a mapped object, but in this example, the logical mixed air temperature data is associated with a mapped BACnet Analog Input object. This relationship is expressed with a link metadata, so retrieving the mixed air temperature including the links metadata would return:

XML:

```
GET /henry-building/floor-5/AHU-5A/mixed-air-temp?metadata=links&alt=xml
```

```
200 OK
Content-Type: application/xml

<?xml version='1.0' encoding='utf-8'?>
<Real  xmlns="http://bacnet.org/csml/1.2"
        viaMap="http://theserver/.bacnet/.local/123/analog-input,2" value="75.0"/>
```

JSON:

```
GET /henry-building/floor-5/AHU-5A/mixed-air-temp?metadata=links
```

```
200 OK
Content-Type: application/json

{
   "$viaMap":"http://theserver/.bacnet/.local/123/analog-input,2",
   "$value":75.0
}
```

Note that the name of associated mapped object (e.g., "HEN:AHU-5A:MAT") is not necessarily the same as the logical name (e.g., "mixed-air-temp").

W.41.16 Logical Tree Data without a Declared Definition

In this example, the fifth floor air handler has points and settings as children, but does not conform to any particular definition. So the 'type' metadata in XML is empty or absent.

GET /henry-building/floor-5/AHU-5A/$type ==> 404 Not Found (no declared definition)

Since no definition is declared, the client discovers the children either by reading the 'children' metadata, or by reading the entire data item in CSML.

XML:

```
GET /henry-building/floor-5/AHU-5A?metadata=cat-types&alt=xml
```

```
200 OK
Content-Type: application/xml

<?xml version='1.0' encoding='utf-8'?>
<Composition nodeType="equipment"                                    (no type="..." )
            xmlns="http://bacnet.org/csml/1.2" >
   <Real name="mixed-air-temp" value="65.0" />
   <Real name="min-outside-air"  value="10.0" />
    ...
</Composition >
```

JSON:

```
GET /henry-building/floor-5/AHU-5A?metadata=cat-types
```

```
200 OK
Content-Type: application/json

{
  "$base":"Composition ",                                    (no "$type":"...")
  "$nodeType":"equipment",
  "mixed-air-temp": { "$base":"Real", "$value":65.0 },
  "min-outside-air": { "$base":"Real",  "$value":10.0 },
    ...
}
```

W.41.17 Logical Tree Data with a Declared Definition

In this example, the fifth floor air handler declares that it conforms to an identified definition, which defines the names and meanings of the children for the client.

XML:

```
GET /henry-building/floor-5/AHU-5A?metadata=cat-types&alt=xml
```

```
200 OK
Content-Type: application/xml

<?xml version='1.0' encoding='utf-8'?>
<Composition type="555-AHU-2"  xmlns="http://bacnet.org/csml/1.2"  >
   <Real name="mixed-air-temp" value="65.0" />
   <Real name="min-outside-air"  value="10.0" />
    ...
</Composition >
```

JSON:

```
GET /henry-building/floor-5/AHU-5A?metadata=cat-types,base
```

```
200 OK
Content-Type: application/json

{
   "$base":"Composition ",
   "$type":"555-AHU-2",
   "mixed-Air-temp": { "$base":"Real",  "$value":65.0 },
   "min-outside-air": { "$base":"Real", "$value":10.0 },
  ...
}
```

W.41.18 Finding the Definition for a Declared Definition

In the above example, the air handler declares that it conforms to an identified definition, however, the client does not already have knowledge of that definition. The server has been configured to store the definitions for all the data it models, so the client retrieves the definition from the /.defs list.

XML:

```
GET /.defs/555-AHU-2?metadata=all&alt=xml
```

```
200 OK
Content-Type: application/xml

<?xml version='1.0' encoding='utf-8'?>
<Composition xmlns="http://bacnet.org/csml/1.2"
             nodeType="equipment"
              displayName="Example Co. Air Handler, Single Duct with AC2 Mixed" >
    <Real name="mixed-air-temp"
              variability="status"
              displayName="Mixed Air Temperature"
              units="degrees-Fahrenheit"/>
    <Real name="min-outside-air"
              units="percent"
              writable="true"
              variability="config"
              volatility="nonvolatile"
              displayName="Minimum Outside Air Setting" />
       ... much more ...
</Composition >
```

JSON:

```
GET /.defs/555-AHU-2?metadata=all
```

```
200 OK
Content-Type: application/json

{
  "$base":"Composition",
  "$nodeType":"equipment",
  "$displayName":"Example Co. Air Handler, Single Duct with AC2 Mixed",
  "Mixed_Air_Temp": {
    "$base":"Real",
    "$variability":"status",
    "$displayName":"Mixed Air Temperature",
    "$units":"degrees-Fahrenheit"
  },
  "Min_Outside_Air": {
    "$base":"Real",
    "$units":"percent",
    "$writable":true,
    "$variability":"config",
    "$volatility":"nonvolatile",
    "$displayName":"Minimum Outside Air Setting"
  },
   ... much more ...
}
```

W.41.19 Logical Tree Data with a Declared Definition and a Protocol Mapping

In this example, the fifth floor air handler not only declares that it conforms to an identified definition, but also associates it members with mapped objects. The connection to the mapped objects is done with the 'viaMap' links on the members:

XML:

```
GET /henry-building/floor-5/AHU-5A?metadata=links,value&alt=xml
```

```
200 OK
Content-Type: application/xml

<?xml version='1.0' encoding='utf-8'?>
<Composition   xmlns="http://bacnet.org/csml/1.2" type="555-AHU-2" >
   <Real name="mixed-air-temp" value="65.0" viaMap="/.bacnet/.local/1234/analog-input,12" />
   <Real name="min-outside-air" value="10.0" viaMap="/.bacnet/.local/1234/analog-value,2" />
    ...
</Composition >
```

JSON:

```
GET /henry-building/floor-5/AHU-5A?metadata=links,value
```

```
200 OK
Content-Type: application/json

{
   "Mixed_Air_Temp": {
      "$value":65.0,
      "$viaMap":"/.bacnet/.local/1234/analog-input,12"
   },
   "Min_Outside_Air": {
      "$value":10.0,
      "$viaMap":"/.bacnet/.local/1234/analog-value,2"
  },
  ...
}
```

W.41.20 Example .info

In this example, the client asks for the .info structure to discover server information.

XML:

```
GET /.info?alt=xml
```

```
200 OK
Content-Type: application/xml

<?xml version='1.0' encoding='utf-8'?>
<Composition  xmlns="http://bacnet.org/csml/1.2">
   <String name="vendor-name" value="Example Controls, Inc."/>
   <String name="model-name" value="Big Server Mark IV"/>
   <String name="software-version" value="1.0"/>
   <String name="standard-version" value="15"/>
</Composition>
```

JSON:

```
GET /.info
```

```
200 OK
Content-Type: application/json

{
   "vendor-name": "Example Controls, Inc.",
   "model-name":"Big Server Mark IV",
   "software-version":"1.0",
   "standard-version":"15"
}
```

W.41.21 Tree Discovery

In this example, the client asks for the roots of the logical trees. It uses the query parameter "metadata=tags" to discover if the "other" trees have any semantic tags that might describe their purpose.

XML:

```
GET /.trees?metadata=tags&depth=1&alt=xml
```

```
200 OK
Content-Type: application/xml

<?xml version='1.0' encoding='utf-8'?>
<Collection nodeType="Collection" xmlns="http://bacnet.org/csml/1.2">
   <Collection name=".geo" truncated="true" nodeType="tree"/>
   <Collection name="power" truncated="true" nodeType="tree" tags="org.example.tags.treesEpwr" />
   <Collection name="steam" truncated="true" nodeType="tree" tags="steamTree" />
</Collection>
```

JSON:

```
GET /.trees?metadata=tags&depth=1
```

```
200 OK
Content-Type: application/json

{
   ".geo":{"$nodeType":"tree","$truncated":true},
   "power":{"$nodeType":"tree","$truncated":true,"$tags":"org.example.trees.epwr"},
   "steam":{"$nodeType":"tree","$truncated":true, "$tags":"steamTree"}
}
```

W.41.22 Example 'multi'

XML:

```
POST /.multi?alt=xml

<?xml version='1.0' encoding='utf-8'?>
<Composition xmlns="http://bacnet.org/csml/1.2">
   <Unsigned name="lifetime" value="60" />    <!--optional; if absent, no persistent record is created -->
   <List name="values">
      <Any name="1" via="/path/to/first/item"/>
      <Any name="2" via="/path/to/second/item"/>
      <Any name="3" via="/path/to/nonexistent/item"/>
   </List>
</Composition>
```

200 OK	-or-	**201 Created** **Location: http://theserver/.multi/slkdjflwekjf**

```
<?xml version='1.0' encoding='utf-8'?>
<Composition  xmlns="http://bacnet.org/csml/1.2">
   <Unsigned name="lifetime" value="60" />
   <List name="values">
      <Real name="1" via="/path/to/first/item" value="75.5" >
      <Sequence name="2" via="/path/to/second/item">
         <Boolean name="bar" value="true"/>
         <Real name="baz" value="100.0"/>
      </Sequence>
      <Any name="3" via="/path/to/nonexistent/item" error="9"/>
   </List>
</Composition>
```

JSON:

POST **/.multi**

```
{
  "lifetime"60,
  "values":{
    "1": { "$base":"Any", "$via":"/path/to/first/item" },
    "2": { "$base":"Any", "$via":"/path/to/second/item" },
    "3": { "$base":"Any", "$via":"/path/to/nonexistent/item" }
  }
}
```

200 OK	-or-	**201 Created** **Location: http://theserver/.multi/slkdjflwekjf**

```
{
  "lifetime":60,
  "values":{
    "1":{"$base":"Real","$via":"/path/to/first/item","$value":75.5},
    "2":{
      "$base":"Sequence",
      "$via":"/path/to/second/item",
       "bar":{"$base":"Boolean","$value":true},
       "baz":{"$base":"Real","$value":100.0}
    },
    "3":{"$base":"Any","$via":"/path/to/nonexistent/item","$error":"9"}
  }
}
```

W.41.23 Subscribing for COV

The client requests to create a subscription for two data items with a large lifetime. The server responds by assigning it a resource identifier and limits the lifetime to match its policies.

XML:

```
POST /.subs?alt=xml
Content-Type: application/xml
...

<?xml version='1.0' encoding='utf-8'?>
<Composition xmlns="http://bacnet.org/csml/1.2">
   <String name="label" value="A Great Client"/>
   <String name="callback" value="http://theclient/callback/path"/>
   <Unsigned name="lifetime" value="86400"/>
   <List name="covs">
      <Composition name="c342">
         <String name="path" value="/path/to/firstvalue"/>
         <Real name="threshold" value="1.0"/>
      </Composition>
      <Composition name="c343">
         <String name="path" value="/path/to/secondvalue"/>
      </Composition>
   </List>
</Composition>
```

```
201 CREATED
Location: http://theserver/.subs/1322
...
```

JSON:

```
POST /.subs
Content-Type: application/json
...

{
   "label":"A Great Client",
   "callback":"http://theclient/callback/path",
   "lifetime":86400,
   "covs":{
      "c342": {
          "path":"/path/to/firstvalue",
          "threshold":1.0
      },
      "c343": {
         "path":"/path/to/secondvalue"
      }
   }
}
```

```
201 CREATED
Location: http://theserver/.subs/1322
...
```

W.41.24 Subscribing to Log Buffers

The client requests to create a subscription for two Log Buffers. The server responds by assigning it a resource identifier ("1323").

XML

```
POST /.subs?alt=xml
Content-Type: application/xml
...

<Composition xmlns="http://bacnet.org/csml/1.2">
   <String name="label" value="A Great Client"/>
   <String name="callback" value="http://theclient/callback/path"/>
   <Unsigned name="lifetime" value="3600"/>
   <List name="logs">
      <Composition name="1">
         <String name="path" value="/path/to/logical/data/$history"/>
         <Enumerated name="frequency" value="instant"/>
      </Composition>
      <Composition name="2">
         <String name="path" value="/path/to/mapped/logobject/buffer"/>
         <Enumerated name="frequency" value="daily"/>
         <Unsigned name="stagger" value="3600"/>
      </Composition>
   </List>
</Composition>
```

```
201 CREATED
Location: http://theserver/.subs/1323
...
```

JSON:

```
POST /.subs
Content-Type: application/json
...

{
   "label":"A Great Client",
   "callback":"http://theclient/callback/path",
   "lifetime":3600,
   "logs":{
      "1":{
         "path":"/path/to/logical/data/$history",
         "frequency":"instant"
      },
      "2":{
         "path":"/path/to/mapped/logobject/buffer" },
         "frequency":"daily"
         "stagger":3600
      }
   }
}
```

```
201 CREATED
Location: http://theserver/.subs/1323
...
```

W.41.25 Receiving a Subscription COV Callback

In this example, if the client has subscribed to change of value for a data item, when the data changes by the client-defined "increment" value, the server would POST something like this:

XML:

```
POST /subscriber/callback/uri?alt=xml
Content-Type: application/xml
...

<List xmlns="http://bacnet.org/csml/1.2" subscription="http://theserver/.subs/1232" >
   <Real name="1" value="75.6"
         updated="2012-04-09T12:45:53Z" via="http://theserver/path/to/data"/>
   ...possibly others from this same subscription...
</List>
```

JSON:

```
POST /subscriber/callback/uri
Content-Type: application/json
...

{
   "$subscription":"http://theserver/.subs/1232",
   "1": { "$base":"Real", "$value":75.6,
          "$updated":"2012-04-09T12:45:53Z", "$via":"http://theserver/path/to/data"
   }
   ...possibly others from this same subscription...
}
```

Note that the callback is defined to be a List of Any, therefore either "$type" or "$base" information is needed in the JSON representation of the members of the List.

W.41.26 Receiving a Subscription Log Callback

In this example, if the client has subscribed to a Log Buffer, when new records are added to the buffer, subject to the "frequency" value (here set to "hourly"), the server would POST something like this:

XML:

```
POST /subscriber/callback/uri?alt=xml
Content-Type: application/xml
...

<List xmlns="http://bacnet.org/csml/1.2"   subscription="http://theserver/.subs/4223" >
   <List name="1" via="http://theserver/path/to/data/$history">
      <Sequence name="543123">
         <DateTime name="timestamp" value="2012-04-09T12:00:00Z"/>
         <Choice name="logDatum">
            <Real name="realValue" value="74.0" />
         </Choice>
      </Sequence>
      <Sequence name="543124">
         <DateTime name="timestamp" value="2012-04-09T12:15:00Z"/>
         <Choice name="logDatum">
            <Real name="realValue" value="75.5" />
         </Choice>
      </Sequence>
      <Sequence name="543125">
         <DateTime name="timestamp" value="2012-04-09T12:30:00Z"/>
         <Choice name="logDatum">
            <Real name="realValue" value="76.0" />
         </Choice>
      </Sequence>
   </List>
   ...possibly others from this same subscription...
</List>
```

JSON:

```
POST /subscriber/callback/uri
Content-Type: application/json
...

{
   "$subscription": "http://theserver/.subs/4223",
   "1": {
      "$base":"List",
      "$via":"http://theserver/path/to/data/$history",
      "$memberType":"0-BACnetTrendLogRecord",
      "543123": {
         "timestamp":"2012-04-09T12:00:00Z",
         "log-datum":{
            "real-value":74.0
         }
      },
      "543124": {
         "timestamp":"2012-04-09T12:15:00Z",
         "log-datum": {
            "real-value":75.5
         }
      },
     "543125": {
         "timestamp":"2012-04-09T12:30:00Z",
         "log-datum":{
            "real-value":76.0
         }
      }
   }
   ...possibly others from this same subscription...
}
```

Note that the callback is defined to be a List of Any, therefore, the "$base" information is included in the JSON representation of the members of the List.

W.41.27 Getting Localized String Data

Given a data item with localized display names (where the system default locale is "de-DE"),

Get all locales in XML:

```
GET /some-data?metadata=ui&alt=xml
```

```
200 OK
Content-Type: application/xml

<?xml version='1.0' encoding='utf-8'?>
<Real  displayName="Farbe" xmlns="http://bacnet.org/csml/1.2">
  <DisplayName locale="en-US">Color</DisplayName>
  <DisplayName locale="en-CA">Colour</DisplayName>
</Real>
```

Get the display name in the system default locale:

```
GET /some-data/$displayName?alt=plain
```

```
200 OK
Content-Type: text/plain

Farbe
```

Get the display name in the "en-US" locale:

```
GET /some-data/$displayName?alt=plain&locale=en-US
```

```
200 OK
Content-Type: text/plain

Color
```

Request the display name in any "en" locale:

```
GET /some-data/$displayName?alt=plain&locale=en
```

```
200 OK
Content-Type: text/plain

Colour
```

- or -

```
200 OK
Content-Type: text/plain

Color
```

Request the display name in another "en-XX" locale:

```
GET /some-data/$displayName?alt=plain&locale=en-GB
```

```
200 OK
Content-Type: text/plain

Colour
```

- or -

```
200 OK
Content-Type: text/plain

Color
```

Request the display name for a locale that does not have a value:

```
GET /some-data/$displayName?alt=plain&locale=es
```

```
200 OK
Content-Type: text/plain

Farbe
```

W.41.28 Setting Localized String Data

Setting localized data is shown with a subsequent read back of all the localized values to see the result.

Starting with an empty but writable display name:

```
GET /some-data?metadata=ui&alt=xml
```

```
200 OK
Content-Type: application/xml

<?xml version='1.0' encoding='utf-8'?>
<Real  value="0.0" xmlns="http://bacnet.org/csml/1.2"/>
```

Set the display name with no 'locale' parameter (the system default locale is "de-DE" in this example):

```
PUT /some-data/$displayName?alt=plain

Farbe
```

```
204 No Content
```

```
GET /some-data?metadata=ui
```

```
200 OK
Content-Type: application/xml

<?xml version='1.0' encoding='utf-8'?>
<Real  displayName="Farbe" value="0.0" xmlns="http://bacnet.org/csml/1.2"/>
```

Set the display name in another locale:

```
PUT /some-data/$displayName?alt=plain&locale=en-CA

Colour
```

```
204 No Content
```

```
GET /some-data?metadata=ui
```

```
200 OK
Content-Type: application/xml

<?xml version='1.0' encoding='utf-8'?>
<Real  displayName="Farbe" value="0.0" xmlns="http://bacnet.org/csml/1.2">
   <DisplayName locale="en-CA">Colour</DisplayName>
</Real>
```

Set the display name with no 'locale' parameter again (removes others):

```
PUT /some-data/$displayName?alt=plain

Farbe
```

```
204 No Content
```

```
GET /some-data?metadata=ui
```

```
200 OK
Content-Type: application/xml

<?xml version='1.0' encoding='utf-8'?>
<Real  displayName="Farbe" value="0.0" xmlns="http://bacnet.org/csml/1.2"/>
```

Set the display name with an unsupported 'locale' parameter:

```
PUT /some-data/$displayName?alt=plain&locale=eo

Koloro
```

```
400 Bad Request

Unsupported 'locale'.
```

W.41.29 Getting Definitions Along with Instance Data

The client asks for the definitions to be returned along with instance data. In this example, all type names start with something other than "0-" and are therefore included in the response.

Things to note in the examples below:

(a) The server includes definitions in a standard "definition context" container, using <Definitions> in XML and "$$definitions" in JSON. While these are normally used as "first level" items under <CSML> they can in fact occur at any level.

(b) The server includes a definition only once in a single response, so "setpoint" does not repeat the definition of "555-BoundedPercent".

(c) The definitions can be included "as encountered" by the server. This means that the server does not have to pre-scan the result set, or can be included based on the server's pre-knowledge of which ones are required for a given instance.

(d) The server will notice or know that a definition uses 'type', 'extends', and 'memberType', and will include those definitions as well. Note that an anonymous type defined by 'memberTypeDefinition' is just normal metadata and does not need a "definition context" to contain it.

(e) The metadata filtering can continue in force even for the definition, so only the parts of the definition that the client is interested in are returned in the first example below. However, this behavior is optional, and

the server is allowed to return more than was asked for if, perhaps, it is returning definitions out of storage without processing.

There are two responses shown here, from two different servers to show that the arrangement of the definitions and their contents are a local matter.

This first server "understands" the definitions and can serve them up as any other data item. It therefore can nest the definitions as they are encountered and can adapt to the 'metadata' query parameter to return only what is asked for by the client. So with these query parameters, things like 'minimum' and 'writable' are returned, but 'displayName' is not.

XML:

```
GET /path/to/a-damper?metadata=type,cat-restrictions,cat-mutability,defs &alt=xml
```

```
200 OK
Content-Type: application/xml

<?xml version='1.0' encoding='utf-8'?>
<Object name="a-damper" type="555-DamperObject" xmlns="http://bacnet.org/csml/1.2">
   <Definitions>
      <Object name="555-DamperObject">
         <Real name="position" type="555-BoundedPercent">
            <Definitions>
               <Real name="555-BoundedPercent" minimum="0.0" maximum="100.0" extends="555-Percent">
                  <Definitions>
                     <Real name="555-Percent" units="percent" extends="555-Numeric">
                        <Definitions>
                           <Real name="555-Numeric"/> <!-- there could be some metadata here, but it was not asked for -->
                        </Definitions>
                     </Real>
                  </Definitions>
               </Real>
            </Definitions>
         </Real>
         <Real name="setpoint" type="555-BoundedPercent" writable="true"/>
         <Enumerated name="motion" type="555-MotionIndicator">
            <Definitions>
               <Enumerated name="555-MotionIndicator">
                  <NamedValues>
                     <Unsigned name="idle" value="0"/>
                     <Unsigned name="opening" value="1"/>
                     <Unsigned name="closing" value="3"/>
                  </NamedValues>
               </Enumerated>
            </Definitions>
         </Enumerated>
         <List name="faults" memberType="555-FaultTime">
            <Definitions>
               <Composition name="555-FaultTime">
                  <Time name="timestamp"/>
                  <Unsigned name="code"/>
               </Composition>
            </Definitions>
         </List>
      </Object>
   </Definitions>
   <Real name="position" value="76.0"/>
   <Real name="setpoint" value="100.0" minimum="10"/>
   <Enumerated name="motion" value="opening"/>
   <List name="faults"/>
</Object>
```

JSON:

```
GET /path/to/a-damper?metadata=type,cat-restrictions,cat-mutability,defs
```

```
200 OK
Content-Type: application/json

{"$type":"555-DamperObject",
  "$$definitions":{
    "555-DamperObject":{"$base":"Object",
      "position":{ "$base":"Real", "$type":"555-BoundedPercent",
        "$$definitions":{
          "555-BoundedPercent":{ "$base":"Real", "$minimum":"0.0",  "$maximum":"100.0",  "$extends":"555-Percent",
            "$$definitions":{
              "555-Percent":{ "$base":"Real", "$units":"percent", "$extends":"555-Numeric",
                "$$definitions":{
                  "555-Numeric":{ "$base":"Real"}
                }
              }
            }
          }
        }
      },
      "setpoint":{"$base":"Real","$type":"555-BoundedPercent","$writable":"true"},
      "motion":{"$base":"Enumerated","$type":"555-MotionIndicator",
        "$$definitions":{
          "555-MotionIndicator":{ "$base":"Enumerated",
            "$namedValues":{
              "idle":{"$base":"Unsigned","$value":"0"},
              "opening":{"$base":"Unsigned","$value":"1"},
              "closing":{"$base":"Unsigned","$value":"2"}
            }
          }
        }
      },
      "faults":{"$base":"List","$memberType":"555-FaultTime",
        "$$definitions":{
          "555-FaultTime":{"$base":"Composition",
            "timestamp":{"$base":"Time"},
            "code":{"$base":"Unsigned"}
          }
        }
      }
    }
  },
  "position":{"$base":"Real", "$value":"76.0"},
  "setpoint":{"$base":"Real", "$value":"100.0"},
  "motion":{"$base":"Enumerated", "$value":"opening"},
  "faults":{"$base":"List"}
}
```

This second server only "holds" the definitions and cannot process them. It therefore does not process them "as encountered" or adapt them to the client's request based on the 'metadata' query parameter. It is a local matter how the server knows which definitions are needed based on the data instance that is requested.

XML:

```
GET /path/to/a-damper?metadata=type,cat-restrictions,cat-mutability,defs &alt=xml
```

```
200 OK
Content-Type: application/xml

<?xml version='1.0' encoding='utf-8'?>
<Object name="a-damper" type="555-DamperObject" xmlns="http://bacnet.org/csml/1.2">
   <Real name="position" value="76.0" displayName="Damper Position"/>
   <Real name="setpoint" value="100.0" minimum="10" displayName="Damper Setpoint"/>
   <Enumerated name="motion" value="opening" displayName="Damper Motion"/>
   <List name="faults" displayName="Damper Fault Codes"/>
   <Definitions>
      <Object name="555-DamperObject">
         <Real name="position" type="555-BoundedPercent"/>
         <Real name="setpoint" type="555-BoundedPercent" writable="true"/>
         <Enumerated name="motion" type="555-MotionIndicator"/>
         <List name="faults" memberType="555-FaultTime"/>
      </Object>
      <Enumerated name="555-MotionIndicator">
         <NamedValues>
            <Unsigned name="idle" value="0"/>
            <Unsigned name="opening" value="1"/>
            <Unsigned name="closing" value="3"/>
         </NamedValues>
      </Enumerated>
      <Real name="555-BoundedPercent" minimum="0.0" maximum="100.0" extends="555-Percent"/>
      <Real name="555-Percent" units="percent" extends="555-Numeric"/>
      <Real name="555-Numeric"/>
      <Composition name="555-FaultTime">
         <Time name="timestamp"/>
         <Unsigned name="code"/>
      </Composition>
   </Definitions>
</Object>
```

JSON:

```
GET /path/to/a-damper?metadata=type,cat-restrictions,cat-mutability,defs
```

```
200 OK
Content-Type: application/json

{"$type":"555-DamperObject",
   "position":{"$base":"Real", "$value":"76.0"},
   "setpoint":{"$base":"Real", "$value":"100.0"},
   "motion":{"$base":"Enumerated", "$value":"opening"},
   "faults":{"$base":"List"},
   "$$definitions":{
     "555-DamperObject":{"$base":"Object",
        "position":{ "$base":"Real", "$type":"555-BoundedPercent"},
        "setpoint":{"$base":"Real","$type":"555-BoundedPercent","$writable":"true"},
        "motion":{"$base":"Enumerated","$type":"555-MotionIndicator"},
        "faults":{"$base":"List","$memberType":"555-FaultTime"}
      },
      "555-BoundedPercent":{ "$base":"Real", "$minimum":"0.0",  "$maximum":"100.0",  "$extends":"555-Percent"},
      "555-Percent":{ "$base":"Real", "$units":"percent", "$extends":"555-Numeric"},
      "555-Numeric":{ "$base":"Real"},
      "555-FaultTime":{"$base":"Composition",
        "timestamp":{"$base":"Time"},
        "code":{"$base":"Unsigned"}
      },
     "555-MotionIndicator":{ "$base":"Enumerated",
       "$namedValues":{
          "idle":{"$base":"Unsigned","$value":"0"},
          "opening":{"$base":"Unsigned","$value":"1"},
          "closing":{"$base":"Unsigned","$value":"2"}
       }
     }
   }
}
```

135-2012*am*-2. Extract data model from Annex Q into separate sharable model.

Rationale

The data model defined by the existing Annex Q is currently tied closely to the definition of the XML syntax. Also, the data from Annex N is similar but not identical. This annex extracts the data model from Annex Q into an abstract model that can be shared by other annexes for a variety of purposes.

ANNEX Y - ABSTRACT DATA MODEL (NORMATIVE)

(This annex is part of this standard and is required for its use.)

This annex defines a data model for defining, transmitting, and storing facility data from disparate data sources for a variety of building management and business applications. The data model is abstract and protocol independent and can therefore be used to model data from any source, whether generated locally or as a gateway to other standard or proprietary protocols.

Y.1 Model Components

The data model fundamentally consists of "data" and "metadata", with individual instances of these referred to as "data items" and "metadata items". Data items hold all the "value" of the model. Data items can be further differentiated into "points", "objects", and "properties". And useful types of metadata include "tags" and "links". All of these are defined in the following clauses.

Y.1.1 Data

Data items are the primary building blocks for modeling data. They are ultimately the holders of all the "value" in the model. Additional (non-value) information is modeled by metadata.

Data items have a single parent and optional children data, and can thus be arranged to form hierarchies. For a given context (file, web services database, etc.), all hierarchies will start at a common root, which uniquely has no parent. These hierarchies can represent geographic location, mapped network/device/point organizations, logical subsystem/equipment/terminal structure, energy distribution hierarchies, or any other arrangement appropriate to the data being modeled.

All data items have a name. Most names will be installation-specific and will be appropriate to their location or usage, like "B41-AHU-5A", or "campus-meter". Some names are predefined by this standard, like ".info", and some names are restricted by their container, like "1" for the first array member.

Data items can have a declared "type" which can be defined outside of this annex and can be either standardized or proprietary. In contrast, standard metadata items do not declare a "type" since their types are all fixed by this standard.

Y.1.2 Value

Data items hold all the "value" of the model. A primitive data item holds its value in a single quantity of some simple data type, and constructed data items contain their value in the values of their children. A primitive data item's value is composed of an indivisible set of related information, collectively called "value information". This set includes the simple value itself along with indicators for errors, age, character set, etc. See Clause Y.7.

Y.1.3 Metadata

Metadata is data that describes other data. Metadata can represent characteristics, restrictions, relationships, and semantics for an associated data item. Relationships describe how the data is related to other data beyond the implicit parent/child relationship of the data hierarchy. Semantics describe the meaning of the data. Metadata are made of a variety of base types and are not limited to primitive types.

Simple metadata names, e.g., 'maximum', are reserved for definition by ASHRAE. Metadata names defined by organizations other than ASHRAE, referred to as "extended metadata", shall use a prefix to ensure uniqueness. This prefix shall be either:

1) A reversed registered DNS name, followed by a period character. e.g., "com.example.", or
2) A BACnet vendor identifier in decimal, followed by a dash character. e.g., "555-"

Y.1.4 Tags

Discovery of the "meaning" of data is greatly enhanced by the presence of semantic information on the data, which can allow data model consumers to make presentation, reporting, grouping, and operational decisions based on the meaning of the data. Semantic information is generally not expected to change during the normal course of operation since it is primarily related to the meaning, associations, or identity of the data and not to its current value.

There are two kinds of semantic tagging information available. One is a simple enumeration that acts as a broad categorization, perhaps indicating enough information for a client to pick a display icon, perform grouping, etc. This information is available in the 'nodeType' metadata. Example values are "area", "equipment", "point", etc.

Additional semantic information is modeled as a collection of "tags". Some tags convey their semantic meaning merely by their presence; they do not have values. These are referred to as "semantic tags". See Clause Y.4.34. Other tags are "parameterized"; they convey their information both by their presence and with a value. These are referred to as "value tags", and are often used to convey "user-applied" kinds of information. See Clause Y.4.35.

A data item can have multiple tags from multiple tagging schemes applied to it. Each tag has a name that is a selection from a set of names defined by some organization. It is expected that multiple organizations will define tagging schemes, and reference to any in particular scheme is beyond the scope of this specification.

Tag names shall be differentiated between organizations that are defining tag lists. To accomplish this, tag names shall conform to the following rules.

Simple tag names, e.g., 'exhaust', are reserved for definition by ASHRAE. Tag names defined by organizations other than ASHRAE shall use a prefix to ensure uniqueness. This prefix shall be either:

1) A reversed registered DNS name, followed by a period character. e.g., "com.example.", or
2) A BACnet vendor identifier in decimal, followed by a dash character. e.g., "555-"

To allow tag names to be concatenated, tag names shall not contain semicolon characters.

Example of tags:

```
<Real name="exhaust-flow" value="1030.0" tags="exhaust;com.example.tags.foo" />
```

The tag names can be formally defined in a machine readable way using the mechanism provided by the serialization context, e.g., the <TagsDefinition> in XML. This allows the controlling organization to provide descriptive metadata to aid the understanding of the tag's meaning. The location of the CSML file containing these definitions is determined by the controlling organization and is therefore outside the scope of this specification.

Y.1.5 Links

Links are used to represent relationships between data and also relationships to external data. Some links are common enough to be built in as standard metadata, e.g., the 'next' link, while others may have non-standard meanings and can be user-applied.

Links that are not built in as standard metadata do not have standardized names, however, the 'displayName' metadata can be used to give an indication to a human as to the link's purpose and tags on the Link can possibly aid an HMI in knowing the purpose of the link.

For example:

```
<Composition name="UV-5A" type="555-UnitVent-1">
  <Links>
    <Link name="maintnotes" displayName="Maintenance Notes"
          value="/path/to/notes/for/UV-5A" />
    <Link name="manufacturer" displayName="Manufacture's Web Site" mediaType="text/html"
          value="http://www.bigmfg.com" />
    <Link name="uman" displayName="User's Manual" mediaType="application/pdf"
          value="http://someserver/docs/model1600.pdf" tags="documentation" />
  </Links>
</Composition >
```

See Clause Y.16 for the definition of the Link base type.

Y.1.6 Points

As an abstract data model, it is possible that some data is integrated from multiple sources. Most building automation protocols, both standard and proprietary, have the concept of organizing data into "points" that have "values." In addition to their values, points often contain data such as "point description" or "point is in fault." But these data may be named, structured, and/or accessed differently in different protocols.

To ensure that a user of the data model can interpret data without knowing these naming and access-method details of underlying protocols, this standard defines "normalized points." This means that the common metadata of points available in the majority of building data models are exposed using a common set of names.

Data with a 'nodeType' of "point" can be assumed to have a value, and metadata for 'units' (for analog points), 'fault', 'overridden', 'inAlarm', 'outOfService', and 'description'. Some of this metadata may not be available in some protocols, in which case a reasonable default value shall be provided.

Y.1.7 Objects

In addition to the logical concept of "points", several building automation protocols have the concept of organizing data into "objects". These are generally protocol-addressable entities, like a BACnet Object. This data is modeled with a base type of "Object". See Clause Y.13.9.

Y.1.8 Properties

To complement to the set of build-in metadata, additional related properties, qualities, parameters, or status values, such as "enable" or "start-time" for an Object, or "height" or "color" for a piece of equipment, are modeled with child data designated with a 'nodeType' of "property".

An example of data representing a mapped BACnet object would be:

```
<Object name="A Great Object">
  <Enumerated name="object-type" value="analog-input" nodeType="property"/>
  <ObjectIdentifier name="object-identifier" value="analog-input,2" nodeType="property"/>
  <Real name="present-value" value="75.5"/>
  ...
</Object>
```

The following logically modeled equipment has two proprietary properties added to it, one of which uses a standard tag to indicate what its meaning is. In addition to these properties, it has a collection of points, sections, etc. These are all given explicit nodeType values like "point", "section", etc.

```
<Composition name="boiler" displayName="Main Boiler" nodeType="equipment" >
   <String name="com.example.asset-id" value="EC345-2" nodeType="property"/>
   <Enumerated name="com.example.color" value="blue" tags="color" nodeType="property"/>
   <Real name="pressure" value="15" units="psi" nodeType="point"/>
   ...
</Composition >
```

Y.2 Trees

Data can be modeled and arranged into trees for a variety of purposes. For example, data could be arranged geographically, or by distribution of air, water, or energy. Since a particular data item can be logically arranged into multiple places, a single parent/child relationship is not sufficient.

Trees generally either "contain" data or "reference" data.

In the case of "containing" data, a few data items are used to provide a hierarchical "home" for the data which is then directly modeled under the last data item down a particular branch. An example would be a desired hierarchy of region/site/building/floor/equipment/point, where region, site, building, and floor are generic collections, but equipment and point are directly modeled as appropriate data types like Real.

```
<Collection name=".trees" nodeType="collection" >
   <Collection name=".geo" displayName="Geographic" nodeType="tree" >
      <Collection name="east" displayName="East Campus" nodeType="area">
         <Collection name="b41" displayName="Chemistry Building" nodeType="building" >
            <Composition name="ahu-2" displayName="AHU #2" nodeType="equipment"
                              type="555-AHU-type-1" >
               <Real name="sat" displayName="Supply Air Temperature" value="57.5" nodeType="point" />
               … more points …
            </Composition>
            <Composition name="vav-2-1" displayName="Room 332B VAV" nodeType="equipment"
                              type="555-VAV-type-1">
               <Real name="damper" displayName="Damper Position" value="100.0" nodeType="point" />
               … more points …
            </Composition>
            ... more AHUs, VAVs, and other equipment in this building ...
         </Collection>
      </Collection>
   </Collection>
</Collection>
```

In the case of "referencing" data, the data items are used to provide a hierarchical "view" on data that is modeled elsewhere.

In this example, a tree creates different "children" for an air handler. Normally, the "children" of an air handler are its points, settings, and status values. However, in a logical air distribution tree, the children of an air handler are its terminal boxes.

For this example, then, the air distribution tree could look like the following:

```
<Collection name=".trees" nodeType="collection" >
   <Collection name="air" displayName="Air Distribution" nodeType="tree">
      <Collection name="ahu-5a" represents="/path/to/real/ahu-5a" memberRelationship="supplies-air" >
         <Composition name="zone-5a-1" href="/path/to/real/zone-5a-1"/>
         <Composition name="zone-5a-2" href="/path/to/real/zone-5a-2"/>
      </Collection>
   </Collection>
</Collection>
```

Generally, these kinds of trees are constructed with the branches modeled as Collection data with the 'represents' metadata pointing to data being represented as the "logical parent", and the leaves are modeled as the appropriate base type with the 'href' pointing to the data being represented as the "logical child".

Other modeling methods are possible, however. The logical children can be modeled with Link data:

```
<Link name="zone-5a-1" value="/path/to/real/zone-5a-1"/>
<Link name="zone-5a-2" value="/path/to/real/zone-5a-2"/>
```

or they can use 'represents' to point to the logical child:

```
<Null name="zone-5a-1" represents="/path/to/real/zone-5a-1"/>
<Null name="zone-5a-2" represents="/path/to/real/zone-5a-2"/>
```

or any mixture of styles:

```
<Link name="zone-5a-1" value="/path/to/real/zone-5a-1"/>
<Null name="zone-5a-2" represents="/path/to/real/zone-5a-2"/>
<Composition name="zone-5a-3" href="/path/to/real/zone-5a-3"/>
```

In most cases, the relationship to the children will be defined by the Collection using 'memberRelationship', which will then apply to all its children. However, the relationship can also be overridden by each child for special modeling needs or for mixed relationship cases.

```
<Collection name="ahu-5a" represents="/path/to/real/ahu-5a" >
   <Composition name="zone-5a-1" href="/path/to/real/zone-5a-1" relationship="supplies-air" />
   <Composition name="zone-5a-2" href="/path/to/real/zone-5a-2" relationship="supplies-air"/>
   <Composition name="boiler" href="/path/to/real/boiler1" relationship="receives-hot-water"/>
</Collection>
```

The selection of the modeling method by the server is a local matter and can vary between, and within, trees. Therefore, user interface clients that consume trees shall not make assumptions as to a specific method of modeling.

Y.3 Base Types

All data is based on a common set of underlying data structures called "base types".

A base type is the lowest indivisible unit of the data model, sometimes called "built-in types". Some base types, like 'String', have a single primitive value, while some, like 'List', can have children.

A data item has only one base type but could have several "derived types" that are layered on top of its base type using the 'extends' mechanism. Derived types are declared with the 'type' metadata.

Other than the distinctions of function, arrangement, identification, and derived type capability defined above, data and metadata are structurally identical since they are both based on a common set of base types. The base types are summarized in the following table and discussed in more detail in subsequent clauses.

Table Y-1. Base Type Summary

Base Type	Description
BitString, Boolean, Date, DatePattern, DateTime, DateTimePattern, Double, Enumerated, Integer, Link, ObjectIdentifier, ObjectIdentifierPattern, OctetString, Raw, Real, String, StringSet, Time, TimePattern, Unsigned, WeekNDay	A single primitive value
Sequence	An ordered collection of fixed-named children
Composition, Object	An unordered collection of fixed-named children
List	An unordered collection of unnamed children
Array, Unknown	An ordered/indexable collection of unnamed children
SequenceOf	An ordered collection of unnamed children
Collection	An unordered collection of arbitrarily named children.
Choice	A single selection from multiple fixed-named children
Null	A data item with no value and no children
Any	A place-holder for an unknown base type

Y.4 Common Metadata

All base types share a common set of metadata. These metadata items are either optional or required based on the context where and how the base type is used, as defined elsewhere. In addition to the common metadata described here, each base type may also define a specific additional set of metadata of its own. This is done in individual clauses that define those base types.

Y.4.1 'name'

This metadata, of type String, provides a name for the data. All data in the data model has a name, and the name shall be unique among the data's siblings. However, a name may or may not be required in serializations, based on the data's context. For example, a name is required for definitions, and for Sequence, Choice, Composition, Collection, and Object members, but not for Array, List, and SequenceOf members, where missing names will be provided by the consumer of the serialization.

The allowed string values for the name are restricted. Only printable characters may be used to construct names, and, as an additional restriction, all characters equivalent to the ANSI X3.4 "control characters" (those less than X'20') are not allowed, and neither are any characters equivalent to the following ANSIX3.4 characters:

/ \ : ; | < > * ? " [ ] { }

Names shall not begin with a dollar sign ("$") character, and shall not contain a double dollar sign ("$$") character sequence. Names beginning with a period (".") character are reserved for use by ASHRAE. This restriction separates data names that are defined by this standard from those that are defined by the data model holder, perhaps based on user input. Space characters are allowed and are significant in names; however, it is recommended that names should not begin or end with space characters. The semicolon character shall be used to delimit names in a list, such as in the 'requiredWith' metadata.

When required by a serialization context, the name of a member of an Array, List, or SequenceOf base type shall be equal to its index/position as a decimal number, "1", "2", etc.

Because the members of a List base type can be reordered at runtime when the list is changed, it is possible for the positional 'name' for a member to change and consumers shall be designed to expect that.

Y.4.1.1 Definition Names

When used in a definition context, the name provides a globally unique name for the defined type, which other data may refer to by using the 'type', 'extends', or 'overlays' metadata.

As systems change over time, it is expected that the definitions of types will change. Versioning of a defined type can be accomplished within the name of the defined data. The name should be prefixed with the vendor ID of the defining organization followed by a hyphen character. A suffix may be also added to indicate newer definitions. The content of the suffix is a local matter and clients cannot make any assumptions about format of names or suffixes.

For example, the name is used below to define the type name "0-BACnetDeviceObjectReference" and also to define the names of the two members. Note that in this example the name uses the ASHRAE vendor identifier as a prefix.

```
<Definitions>
    <Sequence name="0-BACnetDeviceObjectReference">
        <ObjectIdentifier name="deviceIdentifier" contextTag="0" optional="true" />
        <ObjectIdentifier name="objectIdentifier" contextTag="1" />
    </Sequence>
</Definitions>
```

An XML representation of an instance of that type assigns values to the members by identifying the member by its name. Optional elements that do not have a value are simply omitted from the XML.

```
<Sequence type="0-BACnetDeviceObjectReference" >
    <ObjectIdentifier name="objectIdentifier" value="analog-input,0" />
</Sequence>
```

Y.4.1.2 Tag Names

When used for a tag definition, the name provides a globally unique name for the defined tag, which other data may refer to by using the 'tags' and 'valueTags' metadata.

Y.4.2 'id'

This metadata item, of type String, provides a globally unique identifier for a data item that is not dependent on the data item's location. It shall be permanent and shall move with the data if the data changes its location.

Y.4.3 'type'

This optional metadata item, of type String, indicates the type of a data item when that item is an instance of a previously defined type. If the data item has a defined type, then the type of that data can only be changed to a type that is an extension of the defined type, unless the defined type is Any, in which case the new type is limited to the types, or extensions thereof , allowed by the definition of the Any. See Clause W.15.4 for more information on the type metadata item.

Y.4.4 'base'

This required metadata, of type Enumerated, provides the identification of the base type of a data item. The allowable enumerated values for this metadata are the literal strings:

{ "Boolean", "Integer", "Unsigned", "Real", "Double", "String", "OctetString", "Raw", "BitString", "Enumerated", "Date", "DatePattern", "DateTime", "DateTimePattern", "Time", "TimePattern", "StringSet", "Object", "Composition", "List", "Sequence", "Array", "SequenceOf", "Choice", "Null", "Bit", "Any", "Link", "Collection", "Unknown" }

For modeling BACnet-specific data, this set is extended to include:

{ "ObjectIdentifier", "ObjectIdentifierPattern", "WeekNDay" }.

Y.4.5 'extends'

This optional metadata, of type String, indicates the name of the existing defined type that is being extended by or within a new definition. If the new definition is not making any structural changes, then the 'type' metadata shall be used rather than the 'extends' metadata. See the description of the 'type' metadata for more information on this distinction.

When extending a standard type to add new members to the 'namedValues' or 'namedBits' metadata, the 'name' of the new members shall use a vendor specific prefix to prevent conflict with future standard additions with the same name.

The base type of the existing definition shall match the new definition with the exception that, if the existing definition is Any, then the new definition can be of any base type.

Y.4.6 'overlays'

This optional metadata, of type String, indicates the name of an existing type that is being augmented with extra metadata. It is used instead of the 'type' metadata to identify the existing type. It is used for data in a definition context but does not create a new definition and cannot make any structural changes; therefore, the 'name' and 'extends' metadata are not used either.

The base type of the existing definition shall match the overlay base type.

A likely use for the "overlays" metadata could be to provide additional localization information to existing type definitions (e.g., translation information made available in a separate "language pack" file).

For example, the following provides Spanish display names for the members of the 0-BACnetDeviceObjectReference type, which was defined elsewhere.

```
<Definitions>
    <Sequence overlays="0-BACnetDeviceObjectReference">
        <ObjectIdentifier name="deviceIdentifier">
            <DisplayName locale="es">Identificador del Dispositivo</DisplayName>
        </ObjectIdentifier>
        <ObjectIdentifier name="objectIdentifier">
            <DisplayName locale="es">Identificador del Objeto</DisplayName>
        </ObjectIdentifier>
    </Sequence>
</Definitions>
```

Y.4.7 'nodeType'

This optional metadata, of type Enumerated, indicates the general classification of a data item, if known. This Enumerated data item shall have a 'type' metadata of "0-BACnetNodeType" and shall be derived from the definitions of the BACnetNodeType production in Clause 21. It is intended as a rough categorization for client applications about the contents or function of a data item and is not intended to convey an exact definition. The list of values for this metadata is not extensible. Further refinement of classification is provided by the 'tags' metadata. The allowable values for this metadata are described in Clause 12.29.5.

Y.4.8 'nodeSubtype'

This optional localizable metadata, of type String, indicates a further refinement of the classification of a data item. It provides a more specific and flexible or site-specific classification of the data than is provided by the general classification indicated by 'nodeType'. For example, when the 'nodeType' metadata has a value of "device", the nodeSubtype metadata could have a value such as "Controller", "Router", or "Gateway".

Y.4.9 'displayName'

This optional localizable metadata, of type String, provides a brief human-readable text to associate with the value of a data item. This is intended to be a short descriptive identifier (approximately 30 characters or less) usable for human interface displays like dialog boxes and menus. The text consists of a single line of plain printable characters with no formatting markup (limited to mediaType="text/plain"). Because a textual serialization may wrap and indent metadata values, all contiguous whitespace shall be collapsed into a single space for display.

Y.4.10 'description'

This optional localizable metadata, of type String, provides a human readable description of a data item. This is intended to be a reasonably complete description of the purpose or use of the data, but does not provide for any "rich text" formatting capabilities. It could be usable as "hover text", "tool tip" or "pop-up help". The text consists of plain printable characters with no formatting markup or line breaks (limited to mediaType="text/plain"). Full "rich text" formatted documentation is provided by the 'documentation' metadata.

Y.4.11 'documentation'

This optional localizable metadata, of type String is used to provide formatted "rich text" documentation on the purpose and use of the data. If the media type of the text value is other than "text/plain", then the 'mediaType' metadata shall be used to indicate the type of the formatting.

The following example shows some formatted text for a 'documentation' metadata (which is encoded in XML as a <Documentation> element). Note that the "<![CDATA[" and "]]> is an XML-specific syntax and is not part of the value of the String data item.

```
<Definitions>
    <Object name="555-ExampleObject">
        <Real name="a-good-property" … >
            <Documentation mediaType="text/xhtml"><![CDATA[
             <html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en"><body>This property documentation
             contains <b>bold</b> words
             and is spread over several lines (all <i>white space</i> in HTML is collapsed to a
             single space)</body></html>]]></Documentation>
        </Real>
    </Object >
</Definitions>
```

Y.4.12 'comment'

This optional localizable metadata, of type String, provides a human-readable comment for a data item. This is usually a technical note intended for human readers of a serialization, rather than users of the data, as 'displayName', 'description', and 'documentation' are intended to be.

For example, in the following, an internal comment copied from Clause 21 is intended for readers of the XML, not user interfaces.

```
<Definitions>
    <Sequence name="0-BACnetPropertyReference">
        <Enumerated name="property-identifier"      contextTag="0"   type="0-BACnetPropertyIdentifier" />
        <Unsigned    name="property-array-index"   contextTag="1"  optional="true"
                     comment="Used only with array datatype. If omitted, the entire array is referenced."/>
    </Sequence>
</Definitions>
```

Y.4.13 'writable'

This optional metadata, of type Boolean, specifies whether the data value is generally expected to be writable. This applies to all of the data's descendants until overridden with another 'writable'. Security concerns or temporary

modes of operations may make the data value not writable at any given time, but this metadata represents the general case.

Absence of this metadata indicates that the writability is unknown.

The following example declares a property of the File Object to be writable.

```
<Definitions>
    <Object name="0-FileObject">
        …
        <Boolean name="archive" writable="true" … />
        …
    </Object >
</Definitions>
```

Y.4.14 'commandable'

This optional metadata, of type Boolean, specifies whether the data value is commandable using the command prioritization mechanism described in Clause W.24. While "commandable" often implies "writable", the two metadata nonetheless have independent values. It is possible for a definition to declare that commandable="true" and writable="false", meaning that, by default, the property is not externally writable, at any priority, but is nevertheless commandable in nature.

Absence of this metadata indicates that the commandability is unknown.

The following example declares that the Present_Value of an Analog Output Object is writable and commandable.

```
<Definitions>
    <Object name="0-AnalogOutputObject">
        …
        <Real name="present-value" writable="true" commandable="true" … />
        …
    </Object >
</Definitions>
```

The following example shows a present value that is not externally writable but is nevertheless commandable and, as such, has a priority array and default value.

```
<Definitions>
    <Object name="555-InternalScheduleResult" >
        …
          <Real name="present-value" writable="false" commandable="true" … />
          <Array name="priority-array" … />
          <Real name="relinquish-default" … />
        …
    </Object >
</Definitions>
```

Y.4.15 'priorityArray'

This optional metadata, of type Array of Choice, is the priority array associated with a commandable data item. See Clause W.24. Each member is a Choice base type and conforms to definition of BACnetPriorityValue in Clause 21.

Y.4.16 'relinquishDefault'

This optional metadata, of the same type as the data's value, is the value that will be used when the priorityArray contains all nulls. See Clause W.24.

Y.4.17 'failures'

This optional metadata, of type List of Link, can be present on data that represents the results of an attempted read or write operation of some kind. Each entry in the list is a Link pointing to a descendant data item that failed its read or write operation. The 'error' and 'errorText' metadata on the failed data item can be used to provide a reason for the failure.

Y.4.18 'readable'

This optional metadata, of type Boolean, specifies whether the data's value is generally expected to be readable using simple value reading services (e.g., BACnet ReadProperty or RESTful GET). This applies to all of the data's descendants until overridden with another 'readable'. Security concerns or temporary modes of operations may make the data value not readable at any given time, but this metadata represents the general case. An example where this is "false" is the Log_Buffer property of the Trend Log object.

This example shows that, while rare, some properties are not readable using the simple-value reading services.

```
<Definitions>
    <Object name="0-TrendLogObject">
        …
        <List name="log-buffer" readable="false" … />
        …
    </Object >
</Definitions>
```

Y.4.19 'associatedWith'

This optional metadata, of type String, indicates a peer data item that this data item is associated with. The value of this metadata is equal to the value of the name of the referenced peer. This is primarily for human user interface purposes, to define hints for grouping related data or to form a display hierarchy from an otherwise flat list of peers. Only one such relationship can be formed for a given data item, so that it is not possible to define multiple associations that could result in a grouping conflict or the display of the data in more than one place.

This metadata appears on the dependent or subservient data in a relationship, if such a relationship exists. For example, if there is a many-to-one relationship, then the 'associatedWith' metadata is present on the "many" data items and contains the name of the "one". If only two items are involved, the one that is seen as secondary or dependent is given the 'associatedWith' metadata, which refers to the name of the primary one.

An example is a commandable property in a BACnet object. The Priority_Array and Relinquish_Default properties both have an 'associatedWith' metadata which refers to the name of the Present_Value property.

The choice of a "primary" data item may be seem arbitrary in some groups of peers that have no clear hierarchy or dependency relationship. However, the choice of a primary data item is nonetheless important because it may influence a user interface to put that selected data item at the top of the list of associated peers. For example, all of the properties associated with BACnet intrinsic alarming are equal peers, but they may wish to be "associated with" the Event_Enable property as the "primary" since it exists for all algorithms.

Since data exchanged between systems is often dynamic and thus not certifiably correct ahead of time, consumers of this metadata should be designed defensively to deal with malformed or circular relationships.

The association created by 'associatedWith' is distinct from 'requiredWith'. Data that are "associated" with each other are nonetheless still independently optional, whereas 'requiredWith' defines constraints to optionality.

The following example associates the Priority_Array property with the commandable Present_Value property.

```
<Definitions>
    <Object name="0-AnalogOutputObject">
        …
        <Array name="priority-array" associatedWith="present-value" … />
        …
    </Object>
</Definitions>
```

Y.4.20 'requiredWith'

This optional metadata, of type StringSet, indicates a list of peer data items that an optional data item's presence is tied to. When any of the named peers is present, then the current data will be present as well. No implication is made about the reverse situation - if all of the peers are absent, the current data may be present or absent for other reasons. The value of this metadata is equal to a semicolon-separated concatenation of the values of the names of the referenced peers.

One of the purposes of this metadata is to allow consumers to avoid attempts to read the "dependent" data if the "primary" data is known to be absent.

An example is the Inactive_Text property indicating that it is 'requiredWith' the Active_Text property.

Since data exchanged between systems is often dynamic and thus not certifiably correct ahead of time, consumers of this metadata should be designed defensively to deal with malformed or circular relationships.

The following example connects the presence of two optional properties so that if either is present then they are both present.

```
<Definitions>
    <Object name="0-BinaryInputObject">
        …
        <String name="inactive-text" optional="true" requiredWith="active-text" … />
        <String name="active-text"    optional="true" requiredWith="inactive-text"  … />
        …

    </Object>
</Definitions>
```

The following example connects the presence of three optional properties so that if any is present, then they are all present.

```
<Definitions>
    <Object name="0-BinaryInputObject">
        …
        <DateTime name="change-of-state-time"    optional="true"
                requiredWith="change-of-state-count;time-of-state-count-reset" … />
        <Unsigned name="change-of-state-count"  optional="true"
                requiredWith="change-of-state-time;time-of-state-count-reset" … />
       <DateTime name="time-of-state-count-reset"  optional="true"
                requiredWith="change-of-state-time;change-of-state-count" … />
        …
    </Object>
</Definitions>
```

Y.4.21 'requiredWithout'

This optional metadata, of type StringSet, indicates a list of peer data items that an optional data item's presence is tied to. When any of the named peers is absent, then the current data will be present. No implication is made about the reverse situation - if all of the peers are present, the current data may be present or absent for other reasons. The

value of this metadata is equal to a semicolon-separated concatenation of the values of the names of the referenced peers.

One of the purposes of this metadata is to allow clients to know that, if an optional data item is absent, then another is available, often as an alternative for a related purpose.

Since data exchanged between systems is often dynamic and thus not certifiably correct ahead of time, consumers of this metadata should be designed defensively to deal with malformed or circular relationships.

The following example connects the presence of two optional properties so that if either is absent then the other shall be present.

```
<Definitions>
    <Object name="0-ScheduleObject">
        …
        <String name="weekly-schedule" optional="true" requiredWithout="exception-schedule" … />
        <String name="exception-schedule" optional="true" requiredWithout="weekly-schedule"  … />
        …

    </Object>
</Definitions>
```

Y.4.22 'notPresentWith'

This optional metadata, of type String, indicates a list of peer data items that an optional data item's presence is tied to in a negative way. When any of the named peers is present, then the current data will be absent. No implication is made about the reverse situation. If all of the peers are absent, the current data may be present or absent for other reasons. The value of this metadata is equal to a semicolon-separated concatenation of the names of the referenced peers.

This metadata usually appears on the dependent data item(s) in a relationship. For example, if there is a many-to-one relationship, then the 'notPresentWith' metadata is present on the "many" data items and contains the name of the "one". If only two items are involved, then the one that is seen as dependent is given the 'notPresentWith' metadata, which refers to the name of the primary one. If neither is dependent, then the choice of primary is arbitrary, or they may each refer to each other.

One of the purposes of this metadata is to allow clients to know which sets of properties are mutually exclusive and to thus avoid attempts to read the "dependent" optional data if the "primary" optional data is known to be present.

Since data exchanged between systems is often of dynamic in origin and thus not certifiably correct ahead of time, consumers of this metadata should be designed defensively to deal with malformed or circular relationships.

The following example connects the presence of three optional properties where two are present as a pair but are mutually exclusive with a third.

```
<Definitions>
    <Object name="555-ExampleObject">
        …
        <Real name="high-limit" optional="true" requiredWith="low-limit" notPresentWith="limits" … />
        <Real name="low-limit" optional="true" requiredWith="high-limit" notPresentWith="limits" …/>
        <Sequence name="limits" optional="true" notPresentWith="high-limit;low-limit" … />
        …
    </Object>
</Definitions>
```

Y.4.23 'writeEffective'

This optional metadata, of type Enumerated, is an indication of when a write to this value will be effective. The enumerated value choices are the literal strings:
{ "immediately", "delayed", "on-program-restart", "on-device-restart" }

The actual time delay associated with the "delayed" case is not specified, but it is nonetheless an indication that the effect of the write should not be expected to be immediate.

This example shows that a setting controlling how much memory is allocated to audit logs is effective only after the next device restart.

```
<Definitions>
    <Object name="555-MemoryControlObject ">
        …
        <Unsigned name="max-audit-log-space" units="percent" writeEffective="on-device-restart" … />
        …
    </Object>
</Definitions>
```

Y.4.24 'optional'

This optional metadata, of type Boolean, used only in definitions, indicates that this data item may not be present in an instance of this definition. This metadata can only be set to "true" when a data item is initially defined. Subsequent definitions that inherit the data may set the value to "false" if the data will always be present in instances of that new definition, or they may set the 'absent' metadata to "true" to indicate that the data will never be present in an instance of that new definition.

An example case for this is where the standard definition of a BACnet Analog Input declares the Description property to be optional by setting the 'optional' metadata to "true", but a new derived type for a specific vendor's extension to the standard type declares that every instance will have a Description property present by setting the 'optional' metadata in the derived type to "false", or it declares that every instance will never have a Description property present by setting the 'absent' metadata to "true". Note that these changes only describe the vendor's specific implementations of objects; this derived type does not change the meaning of Clause 12 definitions of the base object type and therefore has no effect on the ReadPropertyMultiple service's use of "REQUIRED", and "OPTIONAL".

See the description of the 'absent' metadata for an example of the interaction between the 'optional' and 'absent' metadata.

Y.4.25 'absent'

This optional metadata, of type Boolean, used only in definitions, indicates that an optional data item will not be present in instances of that definition.

An example of the use of this metadata is where the standard definition for BACnetDeviceObjectReference has the 'deviceIdentifier' field marked as optional, but a specific vendor's device does not support references outside the device, so it can derive a new definition from the standard definition and set the 'absent' metadata on the 'deviceIdentifier' field to be "true" so that clients of that device do not try to write a deviceIdentifier to it.

Y.4.26 'variability'

This optional metadata, of type Enumerated, indicates when and how the value of the data is expected to change over time. This applies to all of the data's descendants until overridden with another 'variability'. The enumerated value choices are the literal strings:
{ "constant", "configuration-setting", "operation-setting", "control", "status" }

A value marked as "constant" is expected to not change, so clients can just consume it once. Values marked as "configuration-setting" are expected to be non-volatile settings that are made only during configuration or commissioning. Values marked as "operation-setting" are user settings like setpoints, alarm limit, etc. that are expected to change relatively infrequently, whether by operator or programmed control events. Values marked as "control" are potentially continuously variable values that accept updates from a control algorithm of some kind. Values marked "status" are potentially continuously variable values representing the live status of calculated or measured quantities.

Absence of this metadata means that the variability of the data item is unknown.

In this example, the definition of an object indicates that the "max-audit-log-space" property is a value that is intended to be set when the device is commissioned, not as an on-going part of its operation, and therefore changes infrequently. It also indicates that the "audit-log-alarm-limit" is expected to be changed by an outside entity occasionally during the course of operation of the device, and that the "audit-log-space-used" is a status value that changes by itself at any time.

```
<Definitions>
    <Object name="555-MemoryControlObject ">
        …
        <Unsigned name="max-audit-log-space"   variability="configuration-setting" … />
        <Unsigned name="audit-log-alarm-limit"  variability="operation-setting" … />
        <Unsigned name="audit-log-space-used"  variability="status" … />
        …
    </Object>
</Definitions>
```

Y.4.27 'volatility'

This optional metadata, of type Enumerated, indicates how values that are written are retained. This applies to all of the data's descendants until overridden with another 'volatility'. The enumerated value choices are the literal strings:
{"volatile", "nonvolatile", "nonvolatile-limited-writes"}

The "volatile" case indicates that a written value may be forgotten over device resets and power failures. The "nonvolatile" case indicates that values are intended to survive device resets and power failures. And the "nonvolatile-limited-writes" is an extension to "nonvolatile" that indicates that the value is written to a form of memory that has a limited number of write cycles before wearing out, indicating to clients that this value should not be continuously changed.

Absence of this metadata means that the volatility of the data is unknown.

In this example, the definition of an object indicates that the "output-percent" property is a volatile commanded value that will likely not survive a device reset or power failure and should therefore be checked or refreshed periodically as needed. It also indicates that the "alarm-threshold" should not be continuously written to as a part of normal operation.

```
<Definitions>
    <Object name="555-FanControlObject ">
        …
        <Unsigned name="output-percent" volatility="volatile" … />
        <Unsigned name="alarm-threshold" volatility="nonvolatile-limited-writes" … />
        …
    </Object>
</Definitions>
```

Y.4.28 'isMultiLine'

This optional metadata, of type Boolean, indicates that a String value is intended to be capable of containing multiple lines of text. The value might not actually contain multiple lines at any given time, and it is not intended that isMultiLine change dynamically based on the contents of the value. This metadata is primarily used as a hint to a user interface to display or edit the text in a manner capable of supporting multiple lines.

If the value contains multiple lines, the lines are separated by the character equivalent to the ANSI X3.4 control character known as "new line" or "line feed" (X'0A').

If isMultiLine is missing or false, the presence of, acceptance of, or rejection of "new line" characters in the value is a local matter.

This metadata applies only to String base types.

Y.4.29 'inAlarm'

This optional metadata, of type Boolean, indicates that the value of the data item is "in alarm". The definition of "in alarm" is a local matter. If the concept of "in alarm" is not appropriate for the data item, then this metadata shall not be present.

Y.4.30 'overridden'

This optional metadata, of type Boolean, indicates that the value of the data was overridden by some means. For models of physical inputs or outputs, this shall mean that the value is no longer tracking changes to the physical input or that the physical output is no longer reflecting changes made to the value. If the concept of "overridden" is not appropriate for the data, then this metadata shall not be present.

Y.4.31 'fault'

This optional metadata, of type Boolean, indicates that the value of the data is "in fault", and generally should not be trusted to be accurate or reliable. The definition of "in fault" is a local matter. If the concept of "in fault" is not appropriate for the data, then this metadata shall not be present.

Y.4.32 'outOfService'

This optional metadata, of type Boolean, indicates that the source or destination of the value of the data has been taken "out of service" by some means. The definition of "out of service" is a local matter. If the concept of "out of service" is not appropriate for the data, then this metadata shall not be present.

Y.4.33 'links'

This optional metadata, of type Collection of Link, can be applied to any data and is used to provide additional relationships and links to external data. Common links like 'next' are built in as standard metadata. See Clause Y.16.

Y.4.34 'tags'

This optional metadata, of type StringSet, can be applied to any data and is used to provide semantic information about the data beyond the basic categorization provided by the 'nodeType' and 'nodeSubtype' metadata. The string value is a concatenation of all the tag names separated by a semicolon character. For example, this data item has two semantic tags applied:

<Real name="exhaust-flow" value="1030.0" tags="exhaust;com.example.tags.foo" />

See Clause Y.1.4 for a description of usage and naming restrictions.

Y.4.35 'valueTags'

This optional metadata, of type List of Any, can be applied to any data and is used to provide additional, often user-applied, information about the data. The base types that can replace the Any are limited to the primitive base types plus DateTime and DateTimePattern .

For example, this data item has both a semantic tag and a value tag:

```
<Real  tags="org.example.tags-foo" >
   <ValueTags>
      <String  name="org.example.tags-baz" value="Glorp"/>
   </ValueTags>
</Real>
```

See Clause Y.1.4 for a description of usage and naming restrictions on the members of this List.

Y.4.36 'authRead'

This optional metadata, of type String, is used to provide the identification of the authorization scope(s) that is(are) needed to read the value of the data. If multiple scope identifiers are present, they shall be separated by a single space character. See Clause W.3.5.3. This applies to all of the data's descendants until overridden with another 'authRead'.

Y.4.37 'authWrite'

This optional metadata, of type String, is used to provide the identification of the authorization scope(s) that is(are) needed to write the value of data. If multiple scope identifiers are present, they shall be separated by a single space character. See Clause W.3.5.3. This applies to all of the data's descendants until overridden with another 'authWrite'.

Y.4.38 'authVisible'

This optional metadata, of type Boolean, is used to specify whether the knowledge of the presence of the data is allowed on an unsecured connection or when authorization for the scope specified by 'authRead' has not been provided. This applies to all of the data's descendants until overridden with another 'authVisible'. If allowed to be written at all, this metadata is only writable with the scope "auth". Absence of this metadata implies that the data is visible.

Y.4.39 'href'

This optional metadata, of type String, is used to provide the URI for the remainder of a data item's value and metadata. When present, it indicates that not all of the value and metadata is present in this location and it instructs the consumer that is has to take action to fetch the remainder of the data from the location/protocol indicated by the 'href' URI.

Any metadata local to the data item containing the 'href' are considered to logically override those at the 'href' target, but all the remaining values and children are to be attained from the target and are not copied by the server into the local data item. This is in contrast with a 'via' metadata, which indicates the source of the data that the server has copied and made present locally.

It is highly recommended that servers manage their data to prevent circular references so that clients or user interfaces doing hierarchical descent do not get caught in a loop of hrefs.

Example 1: Tree data items that "glue" other trees into a larger tree. In this case, the "east" item provides a 'displayName' that logically overrides the one in the target, while "west" logically uses the 'displayName' of the target. In both cases, the consumer of this upper tree needs to connect to the lower remote trees to continue descending.

```
<Collection name="main" displayName="Luna U" nodeType="area">
   <Collection name="east" displayName="East Campus" href="http://east.bas.luna.edu/.trees/.geo"/>
   <Collection name="west" href="http://west.bas.luna.edu/.trees/.geo"/>
</Collection>
```

Example 2: A data item that "hard links" data from one location to another location so that it appears that the data is in more than one place. For example, a shared sensor, like outside air temperature, appears in every building, even though it really only modeled in one. All the other appearances are represented with an href to the data's "real" home.

```
<Collection name=".trees" nodeType="collection">
   <Collection name=".geo" nodeType="tree">
      <Collection name="east" displayName="East Campus" nodeType="area">
         <Collection name="bldg41" displayName="Chemistry Building" nodeType="building">
            <Real name="oat" value="96.0" viaMap="/.bacnet/.local/1234/analog-input,2" nodeType="point"/>
         </Collection>
         <Collection name="bldg42" displayName="History Building" nodeType="building">
            <Real name="oat" href="/.trees/.geo/east/bldg41/oat" />
         </Collection>
         … more …
      </Collection>
   </Collection>
</Collection>
```

Example 3: A data item that augments data from another data model source. In this case, the majority of the data is in another server but this server provides a 'displayName' that the other server does not have. Consumers need to refer to the other server for the rest of the metadata and children of this Composition.

```
<Composition name="ahu-5A" displayName="Air Handler, Floor 5 East"
href="http://simple123.bas.luna.edu/.trees/.geo/b41/f5/ahu5a"/>
```

Example 4: A data item provides metadata for a value from another protocol. In this case, the remote data is in a protocol that has no metadata capabilities, so the only thing that needs to be retrieved from the 'href' is the 'value'.

```
<Real name="fan" displayName="Fan Speed" units="percent" href="modbus://somehost/3/40023"/>
```

Example 5: A data item that augments data from another protocol that has its own structured data. In this case, the logical data definition is provided with 'type', so the consumer knows that the object data has an optional "description" property. This local object augments the remote object by providing local data that the remote source does not have. The consumer merges the two to get the complete set of values and children.

```
<Object name="sat" type="0-AnalogInputObject" href="bacnet://1234/0,3/85" >
   <String name="description" value="A property that the BACnet object doesn't actually have."/>
</Object>
```

Y.4.40 'sourceId'

The 'sourceId' metadata, of type String, can be applied to any data item. The value contains the 'id' metadata of the source of the value for the data. This can be used by proxies and archives to refer to the identity of the original source of the data. A data item need not be in the same form as the source item.

The ‘sourceId’ and ‘via’ metadata items are used to indicate that the data item mirrors, or is calculated from, another data item. Where a data item is calculated from another data item, or where an aggregating server has collected data from other servers, either to provide a common place for convenience, or to provide archiving services for trends or alarms, these metadata items allow the source data item to be identified.

The "sourceId" contains the nondereferenceable "id" of the data and the "via" metadata contains the dereferenceable URL of the source of the data.

Y.4.41 'etag'

The 'etag' metadata, of type String, can be applied to any data item. See Clause W.32.

Y.4.42 'count'

The 'count' metadata, of type Unsigned, is the number of children of a constructed data item. This metadata is not normally present in serialized contexts where the children are also present since it would be redundant.

Y.4.43 'children'

The 'children' metadata, of type StringSet, contains the names of the children of constructed data. This applies to all constructed types except Array, SequenceOf, and List, where the names are simply the string form of the numbers between 1 and the value of 'count'. This metadata is not normally present in serialized contexts where the children are also present since it would be redundant.

Y.4.44 'descendants'

The 'descendants' metadata, of type List of Link, contains the relative paths to all of the descendant data (children, grandchildren, etc.) of constructed data. This applies to all constructed types. This metadata is not normally present in serialized contexts where the descendants are also present since it would be redundant. This metadata only has practical use in contexts and operations (such as web services) where the representation of the contents can be limited by filtering or depth limitations.

Y.4.45 'history'

The 'history' metadata, of type List of Sequence, is the list of all trend records available for the value of the data item. Each member of the List is of type BACnetTrendRecord. This metadata is not normally present in serialized contexts since it could be very large. This metadata only has practical use in contexts and operations (such as web services) where the representation of the contents can be limited by range selection. If the data item does not have an associated history, this metadata shall be absent, and the 'hasHistory' metadata shall be false.

Y.4.46 'target'

The 'target' metadata is an alias for the data that is being pointed at by the values of a Link. This metadata is normally not present in serialized contexts. If applied to a non-Link base type, the 'target' metadata simply refers to the data item itself, i.e. every data item implicitly "targets" itself if it is not actually a link to some other data. This metadata only has practical use in contexts and operations (such as web services) where it is desirable to "traverse" through links to the referent data, e.g., GET /path/to/link/$target/child/of/target.

Y.4.47 'targetType'

This optional metadata, of type String, is used to indicate the type of the target of a reference of some kind. The target data shall be of the type indicated by 'targetType' or a type that is extended from that type.

This metadata is most often used with the Link base type where the reference is a URL. However, it can also be applied to any data item that is usable somehow as a reference by some specific protocol, like a Sequence that is of type "0-BACnetDeviceObjectReference" for BACnet, or an OctetString whose value is an OID for SNMP.

Y.4.48 'relationship'

This optional metadata, of type Enumerated, is used to indicate the type of relationship the data item has with its parent. It shall default to be of type "0-BACnetRelationship" and can set to be any type that extends "0-BACnetRelationship". This metadata sets the individual relationship that this data item has with its parent and shall override any common relationship defined by the parent's 'memberRelationship'. See Clause Y.11.5.

Y.4.49 'virtual'

This optional metadata, of type Boolean, indicates that the data item is not physically present in a corresponding hardware device.

Y.5 Named Values

Most primitive base types can have special values that are represented by textual identifiers rather than, or in addition to, their raw value form. Some of these values may have special meanings and can actually be outside the normal restricted range of values. For example, a number normally restricted to a range of 0 to 100 may use 255 as a special value to indicate "invalid" or "unused".

In all cases, the mapping from the underlying value form to the human presentation form is done by an optional 'namedValues' metadata, the children of which provide the individual mappings. Note that the main data item's value remains appropriately formatted for its type, and, with the exception of the Enumerated type, does not become equal to the name of the named value. Rather, it simply matches the value of a named value. The Enumerated type is the exception to this because its value can be formatted to match either the string name of a named value or a named value's numeric value.

Y.5.1 'namedValues'

This metadata, of type Collection of Any, is the container for the definition of named values. The types of the children of 'namedValues' depend on the base type of data being defined, as described in the following table.

Table Y-2. Types and Meanings of the Child Members of 'namedValues'

Base Type	Member Type	Meaning of Members
Enumerated	Unsigned	The value of the Unsigned member provides the numeric value for the encoded enumeration choice, and the 'displayName' metadata can be used to provide a textual presentation of the enumeration choice. If a value is not provided, the next available value is automatically assigned, starting at 0, in the order of the children.
Boolean	Boolean	Two Boolean members, one with a value of "true" and the other with a value of "false", may be used to provide a 'displayName' metadata that can be used as an alternate textual presentation of the underlying values of "true" and "false".
BitString, Date, DatePattern, DateTime, DateTimePattern, Double, Integer, ObjectIdentifier, ObjectIdentifierPattern, OctetString, Raw, Real, String, Time, TimePattern, Unsigned, WeekNDay, StringSet	(same as enclosing data)	The members provide the definition of special values. These values may be outside the range of valid values created by 'maximum' and 'minimum' and 'resolution' metadata. The 'displayName' metadata of these special values may be used in place of the actual underlying value, if desired and appropriate, or this information may simply be used to allow the special values to be considered valid even though they are otherwise outside the valid range.

An example Enumerated shows the use of Unsigned list members to define textual names for the enumerated values states and assign the equivalent numeric value.

```
<Enumerated name="0-BACnetObjectType" minimum="128" maximum="1023" … >
    <NamedValues>
        <Unsigned name="accumulator" value="23" … />
        <Unsigned name="analog-input" value="1" … />
        …
        <Unsigned name="trend-log" value="20" … />
    </NamedValues>
</Enumerated>
```

An instance of that defined type can then use the textual names or a number from the extended range:

```
<Enumerated name="object-type" value="accumulator"/>
```

<Enumerated name="object-type" value="501"/>"

An example Boolean shows the use of Boolean list members to assign alternate text for the boolean states "true" and "false". Also shown is an example usage of the <NamedValues>.

```
<Boolean name="issue-confirmed-notifications" … >
    <NamedValues>
        <Boolean name="confirmed"     value="true"   displayName="Confirmed" … />
        <Boolean name="unconfirmed" value="false"  displayName="Unconfirmed" … />
     </NamedValues>
</Boolean>

<Boolean name="issue-confirmed-notifications" value="true"…>
```

Note that in the example of an instance value immediately above, the actual value of the 'issue-confirmed-notifications' Boolean is not "confirmed". Rather the value is "true", since its base type is Boolean. However, the true value may be mapped by a Human Interface to "Confirmed" for display purposes. The Boolean members of the 'namedValues' metadata are given names for inheritance and overlay reasons, not for use as the value of the main Boolean. The 'namedValues' metadata can only appear in a definition context because adding new named values constitutes a structural change to the data. When inheriting a 'namedValues' metadata from a definition, the newly specified members are logically added to the end of the list of existing members of the inherited 'namedValues'. The order of the children is significant since it is used for auto numbering. See Clause Y.18 for more on definitions and inheritance.

While likely rare, named values can be used for bit strings to represent specific combinations of bits. As always, named values are for human interface purposes and do not affect the value of an instance of the BitString.

```
<Definitions>
  <BitString name="555-WidgetStatusFlags" length="2">
    <NamedBits>
      <Bit bit="0" name="too-hot"/>
      <Bit bit="1" name="too-cold"/>
    </NamedBits>
    <NamedValues>
      <BitString name="ok" displayName="All is well" value=""/>
      <BitString name="error" displayName="Confused" value="too-hot;too-cold"/>
    </NamedValues>
  </BitString>
</Definitions>
```

Members of 'namedValues', have optional metadata, 'displayNameForWriting', 'notForWriting', and 'notForReading' that are available to them to provide extra information specifically for their use in the context of 'namedValues'. These metadata have no meaning outside of that context.

Y.5.2 'displayNameForWriting'

This optional localizable metadata, of type String, provides an alternate display name for use when the named value is used for writing, as opposed to when it is presented as a result of reading.

An example of this is a Boolean representing alarm states where the value zero is presented as "No Alarm" when read, and "Reset" when written.

Absence of this metadata means that the display name for writing is the same as the value of the 'displayName' metadata.

In this example, the "false" state has a different presentation when read than it does when written. This can be used to provide the "adjective for reading, verb for writing" pattern.

```
<Boolean name="tripwire">
    <NamedValues>
      <Boolean name="tripped" value="true" displayName="Tripped" … />
      <Boolean name="armed" value="false" displayName="Armed" displayNameForWriting="Reset"/>
    </NamedValues>
</Boolean>
```

Y.5.3 'notForWriting'

This optional metadata, of type Boolean, is an indicator that a special value or a mapped enumeration value is not to be used for writing. It may appear when read, but an attempt to write it will likely be unsuccessful.

Using the "tripwire" example from the description of the 'displayNameForWriting' metadata, if the "tripped" state is not allowed to be written, then that fact can be declared by using the 'notForWriting' metadata on the "true" state.

```
<Boolean name="tripwire">
    <NamedValues>
      <Boolean name="tripped" value="true" displayName="Tripped" notForWriting="true" />
      <Boolean name="armed" value="false" displayName="Armed" displayNameForWriting="Reset"/>
    </NamedValues>
</Boolean>
```

Y.5.4 'notForReading'

This optional metadata, of type Boolean, is an indicator that a special value is not to be used when displaying a value as a result of reading. It may appear as a special writable choice, but the corresponding underlying value should be presented when read.

An example of this is an Unsigned value representing the number of records collected, where zero is displayed numerically along with all other values, but the only value that is writable is a special value named "Clear" which also has the numeric value of zero but is marked 'notForReading' so that it is only used as a named choice for writing and is not used when the read value is zero.

```
<Unsigned name="record-count" minimumForWriting="0" maximumForWriting="0">
  <NamedValues>
    <!-- this is marked notForReading, so 0 will show as "0" when read -->
    <Unsigned name="clear" value="0" displayNameForWriting="Clear" notForReading="true"/>
  </NamedValues>
</Unsigned>
```

Y.5.5 Use of 'notForReading' and 'notForWriting'

The 'notForReading' and 'notForWriting' metadata are intended for UI display control only.

For example, given:

```
<Enumerated name="foo">
  <NamedValues>
    <Unsigned name="a" displayName="AA" value="0"  />
    <Unsigned name="b" displayName="BB" value="1"
                                    notForReading="true"  />
    <Unsigned name="c" displayName="CC" value="3"
                                     notForWriting="true"   />
  </NamedValues>
</Enumerated>
```

According to the rules:

(a) if the client GETs a value of "a", it displays "AA",
(b) if the client GETs a value of "b", it displays "the corresponding underlying value", which means it displays "1",
(c) if the client GETs a value of "c", it displays "CC"
(d) if the client presents a dropdown list of choices to the user to select a new value, it will only include "AA" and "BB", and then PUT "a" or "b" respectively,
(e) the rules say that a PUT of "c" will "likely be unsuccessful" so "CC" is not offered to the user.

None of this makes any requirements on the interaction between client and server across the wire or the contents of the 'value' on the wire.

Y.6 Named Bits

A Bit String base type can have a textual representation of its constituent bits. In this case, the mapping from the underlying bit position value to the human presentation form is done by an optional 'namedBits' metadata, the children of which provide the individual mappings.

Y.6.1 'namedBits'

This optional metadata, of type Collection of Bit, is the container for the definition of named bits for a BitString.

Y.6.2 Bit

This optional child of 'namedBits' provides an individual bit definition for the BitString. It is not usable by any other type. A Bit indicates a bit position with the 'bit' metadata, and a bit name with its name.

The name provides a name for use in referencing the bit by name in the value of the BitString. In addition to the normal restrictions for data names, Bit names shall not start with "+" or "-" characters.

The following example shows a definition of the BACnetLogStatus bit string. The bits are named, so they may be referenced by instances of this type. The example instance of this bit string type has two of the bits set.

```
<Definitions>
  <BitString name="0-BACnetLogStatus" length="3">
    <NamedBits>
      <Bit bit="0" name="log-disabled" displayName="Disabled"/>
      <Bit bit="1" name="buffer-purged" displayName="Purged"/>
      <Bit bit="2" name="log-interrupted" displayName="Interrupted"/>
    </NamedBits>
  </BitString>
</Definitions>

<BitString name="log-status" type="0-BACnetLogStatus" value="buffer-purged;log-interrupted" />
```

If it is not necessary or possible to name the individual bits or to refer to an existing definition, a bit string can be represented numerically in the following fashions with no definition for the bits:

```
<BitString name="referenced-bitstring" length="3" value="1;2"/>
```

Y.6.3 'bit'

This optional metadata, of type Unsigned, represents the bit position for a Bit base type. It is required in initial definitions and cannot be changed in subsequent type extensions or overlays. It is used to specify the bit position, with bit 0 being the least significant bit.

Y.7 Primitive Values

The primitive base types each have a single scalar value, with the addition that the value of the String base type can also add a collection of localized strings, each associated with a particular locale.

Y.7.1 Value

The type of the value for a given base type is specified in Table Y-3. The value of the String base type can consist of multiple localized strings, each associated with a particular locale. As an abstract data model, this Annex does not specify a serialization format for these values. See Annex Q and Z for serialization formats, including the method of representation of multiple locales for String data.

Table Y-3. Datatype for the Value

Base Type	Value Type
Boolean	boolean (true/false)
Unsigned	non-negative integer
Integer	full range integer
Real	single precision floating point
Double	double precision floating point
OctetString	binary data
Raw	unparsed/unknown data
String	unicode character string
Link	a URI character string
BitString	collection of bit identifiers concatenated into one string
StringSet	collection of strings concatenated into one string
Enumerated	string from restricted set, or number
Date	Gregorian date
DatePattern	Date with wildcards and special values
DateTime	Gregorian date and time
DateTimePattern	DateTime with wildcards and special values
Time	Time of day (<24:00:00)
TimePattern	Time with wildcards
ObjectIdentifier	Object Identifier (type, instance)
ObjectIdentifierPattern	Object Identifier with wildcards
WeekNDay	Month-Week-Day with wildcards

Y.7.2 'unspecifiedValue'

This optional metadata, of type Boolean, indicates that a value for a Date, DateTime, or Time is unspecified.

For example, in this pair of date properties, only the start date is specified.

<Date name="start-date" value="2008-06-15" />
<Date name="end-date" unspecifiedValue="true" />

This metadata applies only to the Date, DateTime, and Time base types.

The value of a data item and the 'unspecifiedValue' metadata are mutually exclusive and shall not be present in the same context. The presence of any one of them in an instance overrides any one that was inherited from a definition.

Y.7.3 'length'

This optional metadata, of type Unsigned, specifies the length of a BitString value, in bits. This is the length of the actual data bits and does not include any extra encoding overhead.

Absence of this metadata means that the length of the BitString is variable or not known. An unknown length is acceptable for definitions when a value for the BitString is not provided. However, the length of a BitString value is required to be known to properly process the value. Therefore, if a 'length' metadata is not specified on the definition of a BitString, then it shall be present on any instance that contains a value.

This metadata only applies to the BitString base type.

Y.7.4 'mediaType'

This optional metadata, of type String, indicates the content type for the value of a String or OctetString or for the target of a Link. It shall be equivalent to the HTTP Content-Type header that would be used to describe the contents. The String base type is limited to supporting mediaType values that begin with "text/".

Y.7.5 'error'

This optional metadata, of type Unsigned, indicates an error that affects the validity of the value of a data item. If the 'error' metadata is present, then the value of the data should not be trusted to be valid. The error numbers are defined in Clause W.40. When this metadata is present and equal to 0, meaning "other error", the optional 'errorText' metadata can be used to provide display text to describe the error condition.

When no known error condition exists, this metadata shall be absent. This metadata is not inherited from a definition, so its presence in a definition is meaningless.

See the description of the 'errorText' metadata for an example usage.

Y.7.6 'errorText'

This optional localizable metadata, of type String, is used to provide display text for the error condition when the 'error' metadata is present. The 'errorText' metadata shall be a single line plain text string whose content is a local matter.

For example, if an error is not caused by one of the standard conditions, or if error information from some downstream source is available, then the 'errorText' metadata can be used to provide the display text.

```
<Real name="zone-temp" error="0" >
    <ErrorText>The device is not feeling well today</ErrorText>
    <ErrorText locale="de">Das Gerät fühlt sich heute nicht gut</ErrorText>
</Real>

<Real name="zone-temp" error="21" >    <!-- 21=WS_ERR_NO_DATA_AVAILABLE -->
    <ErrorText>BACnet error-class=object, error-code=unknown-object</ErrorText>
</Real>
```

When no known error condition exists, the 'errorText' and the 'error' metadata shall be absent. The 'error' and the 'errorText' metadata are not inherited from a definition and their presence in a definition is meaningless.

If an 'errorText' metadata is present in a context without the 'error' metadata, the 'error' metadata is implicitly assigned the value "0".

Y.8 Range Restrictions

Primitive data that expresses a continuous range of values can have that range restricted by optional metadata. These metadata can be used to specify the high and low ends of the range and the minimum increment of the values.

The metadata that are used for restricting the range of primitive base types are specified in the following clauses. In the case of the ObjectIdentifier base type, the range restrictions apply to the instance portion of the value only.

The type and applicability of the range restriction metadata are summarized in the following table.

Table Y-4. Range Restriction Metadata

Base Type	Metadata Name	Metadata Type
Date	minimum maximum minimumForWriting maximumForWriting	Date
DateTime	minimum maximum minimumForWriting maximumForWriting	DateTime
	resolution	Time
Double	minimum maximum minimumForWriting maximumForWriting resolution	Double
Enumerated	minimum maximum minimumForWriting maximumForWriting	Unsigned
Integer	minimum maximum minimumForWriting maximumForWriting resolution	Integer
ObjectIdentifier	minimum maximum minimumForWriting maximumForWriting	Unsigned
Real	minimum maximum minimumForWriting maximumForWriting resolution	Real
Time	minimum maximum minimumForWriting maximumForWriting resolution	Time
Unsigned	minimum maximum minimumForWriting maximumForWriting resolution	Unsigned

Y.8.1 'minimum'

This optional metadata, of the type specified in Table Y-4, provides the inclusive lower bound on the continuous range of values.

Absence of this metadata means that the minimum for the value is unlimited or that the limit is unknown. An example of this metadata is given in the description of the 'maximumForWriting' metadata.

Y.8.2 'maximum'

This optional metadata, of the type specified in Table Y-4, provides the inclusive upper bound on the continuous range of values.

Absence of this metadata means that the maximum for the value is unlimited or that the limit is unknown. An example of this metadata is given in the description of the 'maximumForWriting' metadata.

Y.8.3 'minimumForWriting'

This optional metadata, of the type specified in Table Y-4, provides the inclusive lower bound on the continuous range of values when the value is written.

Absence of this metadata means that the minimum for writing is the same as the value of the 'minimum' metadata. An example of this metadata is given in the description of the 'maximumForWriting' metadata.

Y.8.4 'maximumForWriting'

This optional metadata, of the type specified in Table Y-4, provides the inclusive upper bound on the continuous range of values when the value is written.

Absence of this metadata means that the maximum for writing is the same as the value of the 'maximum' metadata. An example of this is a value that has separate read and write ranges. This <Unsigned> can read values up to 150%, but can't be written with a value greater than 100%.

```
<Unsigned name="motor-speed" minimum="0" maximum="150" units="percent"
        minimumForWriting="0" maximumForWriting="100" />
```

Y.8.5 'resolution'

This optional metadata, of the type specified in Table Y-4, provides the minimum increment that occurs between values. If this metadata is specified, then the value will be in increments of this metadata, starting at the value of the 'minimum' metadata if it is specified, or starting at zero if the 'minimum' metadata is not specified.

Absence of this metadata means that the resolution of the value is set by the capabilities of the underlying data type of the value.

In this example, a normally continuous Real declares that it only represents values in increments of 10, starting at -35. So the valid values are -35, -25, -15, -5, 5, 15, 25, and 35.

```
<Real name="position" minimum="-35.0" maximum="35.0" resolution="10.0" />
```

Y.9 Engineering Units

Primitive data items that express a continuous range of values often have known engineering units associated with those values. The metadata defined here only apply to the numeric data types Double, Integer, Real, and Unsigned.

Y.9.1 'units'

This optional metadata, of type Enumerated, describes the engineering units for the numeric value, if known. This Enumerated data item shall have a 'type' metadata of "0-BACnetEngineeringUnits" and shall be derived from the definitions of the BACnetEngineeringUnits production in Clause 21.

Absence of this metadata means that the metadata is implied to have the value of "no-units". See the description of the 'unitsText' metadata for an example usage.

Y.9.2 'unitsText'

This optional localizable metadata, of type String, is used to provide display text for the units. While the 'units' metadata is an enumeration of fixed strings as defined by this standard, the 'unitsText' metadata is a free-form plain text whose content is a local matter.

As an example of usage of standard engineering units, a property in a temperature sensor might be:

```
<Real name="temperature" units="degrees-celsius" >
                <UnitsText locale="en">ºC</UnitsText>
                <UnitsText locale="de">Grad Celsius</UnitsText>
</Real>
```

If the engineering unit is not one of the standard units, then the 'units' metadata shall be a decimal formatted number, in the same format as an Unsigned, and the 'unitsText' metadata can be used to provide the display text (the default locale in this example is "en").

```
<Real name="snailspeed" units="1000" >
    <UnitsText>Inches/Week</UnitsText>
    <UnitsText locale="de">Zoll/Woche</UnitsText>
</Real>
```

Y.10 Length Restrictions

Primitive base types that have variable length, String, StringSet, BitString, Raw, and OctetString, can have their length restricted by optional metadata.

The metadata that are used for restricting the length of primitive base types are specified in the following clauses.

The type and applicability of the range restriction metadata are summarized in Table Y-5.

Table Y-5. Length Restriction Metadata

Base Type	Metadata Name	Metadata Type
BitString	minimumLength maximumLength minimumLengthForWriting maximumLengthForWriting	Unsigned
OctetString, Raw	minimumLength maximumLength minimumLengthForWriting maximumLengthForWriting	Unsigned
String, StringSet	minimumLength maximumLength minimumLengthForWriting maximumLengthForWriting minimumEncodedLength maximumEncodedLength minimumEncodedLengthForWriting maximumEncodedLengthForWriting	Unsigned

Y.10.1 'minimumLength'

This optional metadata, of the type specified in Table Y-5, provides the inclusive lower bound on the length of the value. For a String and a StringSet, this indicates the length, in characters, as represented in a textual serialization. For an OctetString and Raw, this represents the length in octets of the underlying binary data, not its character representation in a textual serialization. For a BitString, this represents the length in bits of the underlying binary data, not including any binary encoding overhead, and not its character representation in a textual serialization.

Absence of this metadata means that the minimum length of the value is unknown.

Y.10.2 'maximumLength'

This optional metadata, of the type specified in Table Y-5, provides the inclusive upper bound on the length of the value. For a String and a StringSet, this indicates the length, in characters, as represented in a textual serialization. For an OctetString and Raw, this represents the length in octets of the underlying binary data, not its character

representation in a textual serialization. For a BitString, this represents the length in bits of the underlying binary data, not including any binary encoding overhead, and not its character representation in a textual serialization.

Absence of this metadata means that the maximum length of the value is unknown.

Y.10.3 'minimumLengthForWriting'

This optional metadata, of the type specified in Table Y-5, provides the inclusive lower bound on the length of the value when written. For a String and a StringSet, this indicates the length, in characters, as represented in a textual serialization. For an OctetString and Raw, this represents the length in octets of the underlying binary data, not its character representation in a textual serialization. For a BitString, this represents the length in bits of the underlying binary data, not including any binary encoding overhead, and not its character representation in a textual serialization.

Absence of this metadata means that the minimum length for writing is the same as the value of the 'minimumLength' metadata.

Y.10.4 'maximumLengthForWriting'

This optional metadata, of the type specified in Table Y-5, provides the inclusive upper bound on the length of the value when written. For a String and a StringSet, this indicates the length in characters, as represented in a textual serialization. For an OctetString and Raw, this represents the length in octets of the underlying binary data, not its character representation in a textual serialization. For a BitString, this represents the length in bits of the underlying binary data, not including any binary encoding overhead, and not its character representation in a textual serialization.

Absence of this metadata means that the maximum length for writing is the same as the value of the 'maximumLength' metadata.

Y.10.5 'minimumEncodedLength'

This optional metadata, of the type specified in Table Y-5, provides the inclusive lower bound on the length of the encoded value, in octets, when it is encoded for a binary protocol. This metadata is applicable only to Strings and StringSets.

Absence of this metadata means that the minimum encoded length of the value is unknown.

Y.10.6 'maximumEncodedLength'

This optional metadata, of the type specified in Table Y-5, provides the inclusive upper bound on the length of the encoded value, in octets, when it is encoded for a binary protocol. This metadata is applicable only to Strings and StringSets.

Absence of this metadata means that the maximum encoded length of the value is unknown.

Y.10.7 'minimumEncodedLengthForWriting'

This optional metadata, of the type specified in Table Y-5, provides the inclusive lower bound on the length of the encoded value, in octets, when it is encoded for writing with a binary protocol. This metadata is applicable only to Strings and StringSets.

Absence of this metadata means that the minimum encoded length for writing is the same as the value of the 'minimumEncodedLength' metadata.

Y.10.8 'maximumEncodedLengthForWriting'

This optional metadata, of the type specified in Table Y-5, provides the inclusive upper bound for writing on the length of the encoded value, in octets, when it is encoded for writing with a binary protocol. This metadata is applicable only to Strings and StringSets.

Absence of this metadata means that the maximum encoded length for writing is the same as the value of the 'maximumEncodedLength' metadata.

Y.11 Collections

Some constructed values are variable sized collections of data of the same type. The types of collections defined in this data model are Array, List, Collection, and SequenceOf. All types of collection have optional metadata that are specific to collections. These metadata can be applied to the Array, List, Collection, and SequenceOf base types.

Y.11.1 'minimumSize'

This optional metadata, of type Unsigned, indicates the minimum size that the collection is likely to be able to reach. Variable sized collections typically have zero as a minimum size, but fixed size collections do not. Fixed sized collections shall specify both 'minimumSize' and 'maximumSize' as the same value.

Absence of this metadata means that the minimum size is unknown.

Y.11.2 'maximumSize'

This optional metadata, of type Unsigned, indicates the maximum size that the collection is likely to be able to reach. Fixed sized collections shall specify both 'minimumSize' and 'maximumSize' as the same value.

Absence of this metadata means that the maximum size is unknown.

Y.11.3 'memberType'

This optional metadata, of type String, indicates the name of the existing defined type that is to be used as the type of the members of the collection. This can be either the name of a type defined elsewhere, or the name of one of the base types: "Any", "Array", "BitString", "Boolean", "Choice", "Collection", "Composition", "Date", "DatePattern", "DateTime", "DateTimePattern", "Double", "Enumerated", "Integer", "List", "Null", "Object", "ObjectIdentifier", "ObjectIdentifierPattern", "OctetString", "Raw", "Real", "Sequence", "SequenceOf", "String", "StringSet", "Time", "TimePattern", "Unknown", "Unsigned", or "WeekNDay".

All members of the collection shall be of the same type. If a collection of different types is desired, then a 'memberType' of "Any" can be used.

If the type of the members is not a built-in type or a previously defined type, an anonymous type can be declared using the 'memberTypeDefinition' metadata instead of the 'memberType' metadata. The 'memberType' metadata cannot be used simultaneously with the 'memberTypeDefinition' metadata.

Absence of this metadata implies that the member type is "Any", unless the 'memberTypeDefinition' metadata is present.

An inherited 'memberType' metadata can only be changed to a type that is an extension of the inherited type and a subsequent 'memberTypeDefinition' metadata cannot override it.

An example of the 'memberType' metadata is given in the description of the 'memberTypeDefinition' metadata.

Y.11.4 'memberTypeDefinition'

This optional metadata, of type List, is used to provide an anonymous in-line definition for the type of the members of a collection. The 'memberTypeDefinition' metadata has a required single child that defines the type for the members. The child may use the 'extends' metadata to refer to another type that it is extending. The name of the child is ignored and the 'type' and 'overlays' metadata are not applicable.

This example shows the three kinds of member type definitions for three different SequenceOf data items. The SequenceOf named "list-of-event-summaries" defines an anonymous type for its members using the 'memberTypeDefinition' metadata, the SequenceOf named "event-time-stamps" uses the 'memberType' metadata to refer to a previously defined type, and the SequenceOf named "event-priorities" uses the 'memberType' metadata to refer to the built-in primitive type "Unsigned".

```
<Sequence name="0-GetEventInformation-ACK">
    <SequenceOf name="list-of-event-summaries" …>
        <MemberTypeDefinition>
            <Sequence>
                …
                <SequenceOf name="event-time-stamps" memberType="0-BACnetTimeStamp" … />
                …
                <SequenceOf name="event-priorities" memberType="Unsigned" … />
            </Sequence>
        </MemberTypeDefinition>
    </SequenceOf>
    <Boolean name="more-events"…/>
</Sequence>
```

An inherited 'memberTypeDefinition' metadata cannot be changed, and a subsequent 'memberType' metadata cannot override it. Since the inherited <MemberTypeDefinition> is anonymous, it is not possible to create a type that extends it. Therefore, once defined to a specific type, the member type of a collection cannot change.

Y.11.5 'memberRelationship'

This optional metadata, of type Enumerated, is used to indicate the default type of relationship the collection data item has with its children. It shall default to be of type "0-BACnetRelationship" and can set to be any type that extends "0-BACnetRelationship". This metadata sets the default relationship that this collection has with all of its children. This default can be overridden with the 'relationship' metadata on an individual child. See Clause Y.4.48.

Y.12 Primitive Data

Primitive data is represented by a single data item and its associated metadata. The base types available for modeling primitive data are: BitString, Boolean, Date, DatePattern, DateTime, DateTimePattern, Double, Enumerated, Integer, Null, ObjectIdentifier, ObjectIdentifierPattern, OctetString, Raw, Real, String, StringSet, Time, TimePattern, Unsigned, and WeekNDay. These are individually described more fully in the following clauses.

Y.12.1 Null

Null data is modeled with the Null base type. Other than the common metadata described in Clause Y.4, there are no other metadata for this base type.

Y.12.2 Boolean

Boolean data is modeled with the Boolean base type. In addition to the common metadata described in Clause Y.4, the Boolean base type also supports the value specifier described in Clause Y.7, and the named values described in Clause Y.5.

Y.12.3 Unsigned

Unsigned integer data is modeled with the Unsigned base type. In addition to the common metadata described in Clause Y.4, the Unsigned base type also supports the value specifier described in Clause Y.7, the range restrictions described in Clause Y.8, the named values described in Clause Y.5, and the units specifier described in Y.9.

Y.12.4 Integer

Signed integer data is modeled with the Integer base type. In addition to the common metadata described in Clause Y.4, the Integer base type also supports the value specifier described in Clause Y.7, the range restrictions described in Clause Y.8, the named values described in Clause Y.5, and the units specifier described in Y.9.

Y.12.5 Real

Single precision floating point data is modeled with the Real base type. In addition to the common metadata described in Clause Y.4, the Real base type also supports the value specifier described in Clause Y.7, the range restrictions described in Clause Y.8, the named values described in Clause Y.5, and the units specifier described in Y.9.

Y.12.6 Double

Double precision floating point data is modeled with the Double base type. In addition to the common metadata described in Clause Y.4, the Double base type also supports the value specifier described in Clause Y.7, the range restrictions described in Clause Y.8, the named values described in Clause Y.5, and the units specifier described in Y.9.

Y.12.7 OctetString

Octet String primitive data is modeled with the OctetString base type. In addition to the common metadata described in Clause Y.4, the OctetString base type also supports the value specifier described in Clause Y.7, the length restrictions described in clause Y.10, and the named values described in Clause Y.5.

Y.12.8 Raw

Unknown or unparsable ("raw") primitive data is modeled with the Raw base type. In addition to the common metadata described in Clause Y.4, the Raw base type also supports the value specifier described in Clause Y.7, the length restrictions described in clause Y.10, and the named values described in Clause Y.5. A Raw base type is functionally equivalent to an OctetString but has an implied semantic of "unknown type". See also the Unknown base type, Clause Y.13.4.

Y.12.9 String

Character string primitive data is modeled with the String base type. In addition to the common metadata described in Clause Y.4, the String base type also supports the value specifiers described in Clause Y.7, the length restrictions described in clause Y.10, and the named values described in Clause Y.5.

String data values are localizable. A localized String data value is modeled as a collection of individual strings, with one member for each locale that has been assigned a value. There can only be one member per locale. All the members together make up the logical value of the String data. Serialization and/or query contexts may affect whether a single member or all members are used to represent the value of the String data.

Y.12.10 StringSet

A set of related character strings is modeled with the StringSet base type. In addition to the common metadata described in Clause Y.4, the StringSet base type also supports the value specifiers described in Clause Y.7, the length restrictions described in Clause Y.10, and the named values described in Clause Y.5. The set of strings is not ordered and the order can change when strings are added or removed from the set.

A string shall not appear more than once in the set. Empty strings, strings containing a semicolon, and strings starting with "+" or "-" are not allowed.

For textual serialization contexts, the value of a StringSet is a single character string value containing a semicolon-separated concatenation of the individual strings in the set.

StringSet data is not localizable.

Y.12.11 BitString

BitString primitive data is modeled with the BitString base type. In addition to the common metadata described in Clause Y.4, the BitString base type also support the value specifiers described in Clause Y.7, the length restrictions described in clause Y.10, and the named values described in Clause Y.5.

For textual serialization contexts, the value of a BitString, expressed in the value, is a character string containing a semicolon-separated list of named or numeric bits that are "true". Identifiers for bits that are "false" are not present in the value string. The names for the named bit values are defined by child Bit data items of the optional 'namedBits' metadata. The individual bits in the value string are identified either by textual identifier, matching exactly the name of a Bit data item, or numerically, representing the numerical bit position within the bit string. For readability, the name form is preferred, when possible.

Bit names are not allowed to begin with the "+" or "-" character.

Y.12.12 Enumerated

Enumerated primitive data is modeled with the Enumerated base type. In addition to the common metadata described in Clause Y.4, the Enumerated base type also supports the value specifier described in Clause Y.7, the range restrictions described in Clause Y.8, and the named values described in Clause Y.5.

An extensible Enumerated data range may be defined with the range restrictions metadata. If neither 'minimum' nor 'maximum' are present, then the enumeration is not extensible and only the values specified by the named values are possible. If none of 'minimum', 'maximum', or 'namedValues' is specified, then the enumeration is unbounded.

For textual serialization contexts, the value of an Enumerated base type is a string. This string is either a decimal formatted number, in the same form as Unsigned, or a string that matches exactly the name of a child of 'namedValues' metadata. For readability, the name form is preferred to the numeric form, when possible.

For non-extensible enumerations, if the number format is used, it shall match the value of one of the children of the 'namedValues' metadata. For extensible enumerations, the numeric value is not restricted to match a child of 'namedValues', but its value may be restricted by the 'minimum' and 'maximum' metadata, if present.

Y.12.13 Date

Data that represents either a single specific date or a wholly "unspecified" date is modeled with the Date base type. In addition to the common metadata described in Clause Y.4, the Date base type also supports the value specifiers described in Clause Y.7, the range restrictions described in Clause Y.8, and the named values described in Clause Y.5.

Y.12.14 DatePattern

Date data that is allowed to contain individually "unspecified" fields is modeled with the DatePattern base type. In addition to the common metadata described in Clause Y.4, the DatePattern base type also supports the value specifier described in Clause Y.7 and the named values described in Clause Y.5.

For textual serialization contexts, the value of a DatePattern base type is a string. The format of the string value is "YYYY-MM-DD" or "YYYY-MM-DD W", where:

YYYY is either a four-digit year or a single asterisk ("*") character to indicate "unspecified",

MM is either a two-digit month or a single asterisk ("*") character to indicate "unspecified",

DD is either a two-digit day of the month or a single asterisk ("*") character to indicate "unspecified",

W is either the one-digit day of the week (1=Monday) or a single asterisk ("*") character to indicate "unspecified".

The numeric fields shall have leading zeros to achieve the number of digits specified. The YYYY, MM and DD fields are separated by a single dash ("-") character and the optional W field is separated from the DD field by a single space character. It the W field is not present, then neither is the space separator.

The W field is required to be present if any of the YYYY, MM, or DD fields is "unspecified". It is allowed to be absent only if the YYYY, MM, and DD fields specify a single date and the W field can thus be calculated unambiguously.

The allowed special values for the date fields are defined in Clause 21.

Y.12.15 DateTime

DateTime data that represents either a single specific date and time or a wholly "unspecified" date and time is modeled with the DateTime base type. In addition to the common metadata described in Clause Y.4, the DateTime base type also supports the value specifiers described in Clause Y.7, the range restrictions described in Clause Y.8, and the named values described in Clause Y.5.

Y.12.16 DateTimePattern

DateTime data that is allowed to contain individually "unspecified" fields is modeled with the DateTimePattern base type. In addition to the common metadata described in Clause Y.4, the DateTimePattern base type also supports the value specifier described in Clause Y.7 and the named values described in Clause Y.5.

For textual serialization contexts, the value of a DateTimePattern base type is a string. The format of the string value is "YYYY-MM-DD hh:mm:ss.nn " or "YYYY-MM-DD W hh:mm:ss.nn ", where:

YYYY is either a four-digit year or a single asterisk ("*") character to indicate "unspecified",

MM is either a two-digit month or a single asterisk ("*") character to indicate "unspecified",

DD is either a two-digit day of the month or a single asterisk ("*") character to indicate "unspecified",

W is either the one-digit day of the week (1=Monday) or a single asterisk ("*") character to indicate "unspecified",

hh is either a two-digit hour or a single asterisk ("*") character to indicate "unspecified",

mm is either a two-digit minute or a single asterisk ("*") character to indicate "unspecified",

ss is either a two-digit second or a single asterisk ("*") character to indicate "unspecified",

nn is either the two-digit hundredths or a single asterisk ("*") character to indicate "unspecified".

The numeric fields shall have leading zeros to achieve the number of digits specified. The YYYY, MM, and DD fields are separated by a single dash ("-") character, and the optional W field is separated from the DD field and from the hh field by a single space character. If the W field is not present, then neither is the preceding space separator. The hh, mm, and ss fields are separated from each other by a single colon (":") character, and the nn field is separated from the ss field by a single period (".") character.

The W field is required to be present if any of the YYYY, MM, or DD fields is "unspecified". It is allowed to be absent only if the YYYY, MM, and DD fields specify a single date and the W field can thus be calculated unambiguously.

The allowed special values for the date fields are defined in Clause 21.

Y.12.17 Time

Time data that represents either a single specific time or a wholly "unspecified" time is modeled with the Time base type. In addition to the common metadata described in Clause Y.4, the Time base type also supports the value specifiers described in Clause Y.7, the range restrictions described in Clause Y.8, and the named values described in Clause Y.5.

Y.12.18 TimePattern

Time data that that is allowed to contain individually "unspecified" fields is modeled with the TimePattern base type. In addition to the common metadata described in Clause Y.4, the Time base type also supports the value specifier described in Clause Y.7 and the named values described in Clause Y.5.

For textual serialization contexts, the value of a TimePattern base type is a string. The format of the string value is "hh:mm:ss.nn", where:

hh is either a two-digit hour or a single asterisk ("*") character to indicate "unspecified",

mm is either a two-digit minute or a single asterisk ("*") character to indicate "unspecified",

ss is either a two-digit second or a single asterisk ("*") character to indicate "unspecified",

nn is either the two-digit hundredths or a single asterisk ("*") character to indicate "unspecified".

The numeric fields shall have leading zeros to achieve the number of digits specified. The hh, mm, and ss fields are separated by a single colon (":") character and the nn field is separated from the ss field by a single period (".") character.

Y.12.19 Link

The Link base type contains a dereferenceable reference to another location. See Clause Y.16.1.

Y.13 Constructed Data

Constructed data is modeled by data items that contain one or more children that provide the value for the construct.

Y.13.1 Sequence

Data that is an ordered sequence of named members is modeled with the Sequence base type. In addition to the common metadata described in Clause Y.4, the Sequence base type also supports optional children representing the members of the sequence.

The allowed children of a Sequence are: Any, BitString, Boolean, Choice, Collection, Composition, Date, DatePattern, DateTime, DateTimePattern, Double, Enumerated, Integer, Null, ObjectIdentifier, ObjectIdentifierPattern, OctetString, Raw, Real, Sequence, SequenceOf, String, StringSet, Time, TimePattern, Unknown, Unsigned, and WeekNDay, Array, List, Collection, and Object.

The name of a child in a Sequence is significant. It is used to match this child with a corresponding child in a type definition. The name of a child in a Sequence shall be unique among its siblings of the Sequence.

Named children provided in an instance shall exist in the type definition and shall be of the same base type, with the exception that the Any base type in a definition can be replaced by any appropriate base type in an instance.

The order of definition of sequence members is significant. New children added as part of a new definition using the 'extends' metadata are added to the end of the existing children in the sequence. Serialization contexts are required to provide a mechanism to maintain this order. If not provided inherently (e.g., XML is inherently ordered), then the serialization context shall define the mechanism to accomplish this. For example, see Clause Z.3.1 for ordering in JSON.

The order of sequence members in an instance of a defined type is not significant because the members are matched to their position in the definition by their name and not by their position in the instance.

Y.13.2 Choice

Data that is a single choice from multiple possible choices is modeled with the Choice base type. In addition to the common metadata described in Clause Y.4, the Choice base type also supports an optional 'choices' metadata that defines the available choices and a single child holding the currently chosen data.

The single named child provides the value for the Choice and shall exist as a child of the 'choices' metadata and shall be of the same base type, with the exception that the Any base type in a definition can be replaced by any base type in an instance.

A single named child provided in a definition of a Choice can be used to provide a default value for the type. A child in an instance of that type replaces the default child from the definition since there can only be one chosen child at a time.

Y.13.2.1 'choices'

The list of possible choices for the Choice base type is provided by the optional 'choices' metadata, of type Collection of Any. All of the children of the 'choices' metadata shall have non-empty names with values unique among their siblings.

The allowed children of the 'choices' metadata are: Any, BitString, Boolean, Choice, Collection, Composition, Date, DatePattern, DateTime, DateTimePattern, Double, Enumerated, Integer, Null, ObjectIdentifier, ObjectIdentifierPattern, OctetString, Raw, Real, Sequence, SequenceOf, String, StringSet, Time, TimePattern, Unknown, Unsigned, WeekNDay, Array, List, and Object.

The order of the definition of choice members in 'choices' is not significant. The names of the children are significant and are used to match the child in an instance or overlay with a corresponding child in a type definition. The names shall be unique among siblings.

The 'choices' metadata can only appear in a definition context, since adding new choices constitutes a structural change to the data. When inheriting a 'choices' metadata from a definition, the newly specified children are logically added to the list of existing children of the inherited 'choices', but in no prescribed order.

Y.13.2.2 'allowedChoices'

This optional metadata, of type StringSet, indicates a restricted list of the available choices that are allowed to be present in an instance. The value of this metadata is a semicolon-separated concatenation of the names of the allowed children of the 'choices' metadata. This is typically used by a derived type to restrict the available choices that it inherited from its definition.

Absence of this metadata means that there are no restrictions on what children of 'choices' can be present in an instance.

Y.13.3 Array

Data that is an ordered collection of unnamed and indexable members is modeled with the Array base type. In addition to the common metadata described in Clause Y.4, the Array base type also supports the additional capabilities of Collections described in Clause Y.11.

Array members do not have names in the data model; however, when required by a textual serialization context, the represented name of an array member shall be equal to its index in the array, in decimal, starting with "1".

Children provided in a type definition of an Array can be used to provide a default value for the type. However, any children in an instance of that type completely replace the default children since instance values of collections are not merged with their definition.

Y.13.4 Unknown

An ordered collection of unnamed and indexable members is modeled with the Unknown base type. In addition to the common metadata described in Clause Y.4, the Unknown base type also supports the additional capabilities of Collections described in Clause Y.11.

The Unknown base type is used to model complex data when the structure of the data can be determined but the semantic of the structure cannot (Array, Sequence, etc). For example when decoding unknown BACnet property data, the structure can be determined via the open / close tags and individual application and context tagged elements, but whether the data is an Array, Sequence, or other complex type cannot be determined. In this case the property data is modelled using Unknown and primitive types.

Unknown members do not have names in the data model; however, when required by a textual serialization context, the represented name of an array member shall be equal to its index in the array, in decimal, starting with "1".

The Unknown base type is functionally equivalent to the Array base type but has an implied semantic of "unknown type". See also the Raw base type, Clause Y.12.8.

Y.13.5 List

Data that is an unordered collection of unnamed members is modeled with the List base type. In addition to the common metadata described in Clause Y.4, the List base type also supports the additional capabilities of Collections described in Clause Y.11.

List members do not have names in the data model; however, when required by a textual serialization context, the represented name of a list member shall be equal to its position in the list, in decimal, starting with "1".

Children provided in a type definition of a List can be used to provide a default value for the type. However, any children in an instance of that type completely replace the default children since instance values of collections are not merged with their definition.

Y.13.6 SequenceOf

Data that is an ordered collection of unnamed members is modeled with the SequenceOf base type. In addition to the common metadata described in Clause Y.4, the SequenceOf base type also supports the additional capabilities of Collections described in Clause Y.11.

SequenceOf members do not have names in the data model; however, when required by a textual serialization context, the represented name of a member shall be equal to its position, in decimal, starting with "1".

Children provided in a type definition of a SequenceOf can be used to provide a default value for the type. However, any children in an instance of that type completely replace the default children since instance values of collections are not merged with their definition.

Y.13.7 Collection

Data that is an unordered collection of non-predefined named members is modeled with the Collection base type. The Collection base type is a generic container that can be used for a variety of modeling purposes, such as building trees of data. The 'nodeType', 'nodeSubtype' and 'tags' metadata can be useful for giving these generic data containers meaning. In addition to the common metadata described in Clause Y.4, the Collection base type also supports the additional capabilities of Collections described in Clause Y.11.

Collection members always have names.

Children provided in a type definition of a Collection can be used to provide a default value for the type. However, any children in an instance of that type completely replace the default children since instance values of collections are not merged with their definition.

Y.13.8 Composition

A data structure that is an unordered collection of predefined named members can be modeled with the Composition base type. In addition to the common metadata described in Clause Y.4, the Composition base type also supports children representing the members of the structure.

The allowed children of a Composition are: Any, Array, BitString, Boolean, Choice, Collection, Composition, Date, DatePattern, DateTime, DateTimePattern, Double, Enumerated, Integer, List, Null, Object, ObjectIdentifier, ObjectIdentifierPattern, OctetString, Raw, Real, Sequence, SequenceOf, String, StringSet, Time, TimePattern, Unknown, Unsigned, and WeekNDay.

The name of the children is significant and is used to match a child in an instance with a corresponding child in a type definition. The names shall be unique among siblings. Named children provided in an instance shall exist in the type definition and shall be of the same base type, with the exception that the Any base type in a definition can be replaced by any appropriate base type in an instance.

The order of the definition of children in a Composition is not significant. New children added as part of a new definition using the 'extends' metadata are added to the inherited children in no prescribed order.

Y.13.9 Object

Structured data that has an identification of some kind can be modeled as the Object base type. This includes, but is not limited to, BACnet objects. Structurally, an Object base type is the same as a Composition base type and has all the same characteristics, with the exception that the Object members are often called "properties". Additionally, however, an Object is assumed to have an addressable identity of its own. For BACnet Objects, this addressable identity is provided by the "object-identifier" property. When used to represent BACnet objects, the "Property_List" property shall not be included.

Object definitions are generally publically defined in some way, either through a Standard Setting Organization's publications, or through a vendor's web site.

The CSML type name for standard BACnet objects shall be constructed from the Clause 21 identifier in the BACnetObjectType enumeration. The CSML type name shall be "0-" plus the Clause 21 identifier with dashes removed and the initial letter of each word capitalized, plus the word "Object". e.g., the Clause 21 identifier "trend-log-multiple" becomes "0-TrendLogMultipleObject" as a type name.

Y.13.10 'truncated'

This optional metadata, of type Boolean, indicates that a representation of a constructed data item has been truncated for some reason and that its children are not included in the textual serialization. This is to distinguish a representation with an elided set of children from a data item that has no children. This shall only be used when there are no children present in the textual serialization. Therefore, it can only be used to mean "there are children" and it cannot be used to mean "there are more children than these". This metadata applies only to constructed base types.

For example, <List truncated="true".../> has children that are not shown, <List ... /> is an empty list, and <List truncated="true" ...>...some children...</List> is invalid.

Y.13.11 'partial'

This optional metadata, of type Boolean, indicates that a representation of a constructed data item has only some of its children included in the textual serialization. This is implied to be "true" in a definition context when 'extends' is used, otherwise it defaults to "false". It is required to be present and "true" if an instance declares that it conforms to a type but only some of the required members are present. This can result from selective rendering, e.g., use of the "select" query parameter in Annex W, or partial specification, e.g., BACnet object creation in Annex W.

This metadata applies only to constructed base types.

It is needed when not all required children are present: e.g., {"$partial":true,"object-name":"new","object-type":0} requires "$partial" because required members, like "object-identifier", are not present in the serialization.

It is also needed for unambiguous specification of constructs that contain optional members: e.g.,

{"object-identifier":"loop,1", "device-identifier":"device,1234"}	means the optional "device-identifier" is present.
{"object-identifier":"loop,1"}	means the optional "device-identifier" is absent.
{"object-identifier":"loop,1", "$partial":true}	makes no statement on the presence of "device-identifier".

Y.13.12 'displayOrder'

This optional metadata, of type Unsigned, provides a hint for user interfaces to order the contents of the otherwise unordered base types Collection, Composition, Object, and List. The numerical values are not required to be sequential. They can start at zero, can have duplicates, and can skip by any amount. If supported, user interfaces can use this information to sort the display of the data members. Lower numbers sort "first", whatever that means for the display.

Y.14 Data of Undefined Type

Data whose type is not known or not restricted by a definition is modeled with the Any base type.

Y.14.1 Any

The Any primitive type is used in definitions and in .multi to describe data items for which the type is not known until instantiated in the model. In addition to the common metadata described in Clause Y.4, the Any base type also supports an 'allowedTypes' metadata that defines which actual types are allowed to replace the Any. When instantiated with data, the Any element is replaced with any primitive or constructed base type, subject to the 'allowedTypes' restrictions.

Y.14.2 'allowedTypes'

This optional metadata, of type StringSet, indicates a list of types that are allowed to be substituted for an Any base type. This metadata is only allowed on an Any base type. The value of this metadata is equal to a semicolon-separated concatenation of the strings suitable for use as a 'type' metadata value.

Absence of this metadata means that there are no restrictions on what types can be substituted for the Any.

Y.15 Logical Modeling

Data can be arranged in flexible ways using the generic Collection base type. Data can either contain other model components or refer to other data with the 'via', 'represents', or 'related' metadata. Using these methods, the data model can express hierarchical containment or arbitrary arrangements of data located elsewhere.

Y.16 Links

Expressing relationships between data is accomplished with links. RFC 5988 defines several standard relationship types, e.g., "self", "next", "via", etc. which are represented here as metadata named 'self', 'next', 'via', etc. In addition, for expression of other kinds of link relationships, this standard defines several new link relationship types, like 'viaMap'. All of these metadata are of base type Link.

For relationship types beyond those defined here, or for when there are multiple links for a given relationship type (e.g., multiple logical tree locations), a 'links' metadata is provided that is a Collection of Links.

Y.16.1 Link

The Link primitive base type contains a dereferenceable reference to another location. This is either a path to modeled data in the same server device or a complete URI to some kind of data in another location.

Y.16.1.1 'value'

The value, of type String, indicates the location of the referenced data. If the value does not contain a URI scheme then it is a path to data modeled on the same server. This can be a relative or absolute path.

The presence or absence of 'mediaType' is significant. See Clause Y.16.1.2

For example,
<Link name="foo" value="/some/path/to/data" />
<Link name="bar" value="http://someserver/path/to/data"/>
<Link name="baz" value="http://someserver/path/to/document.pdf" mediaType="application/pdf"/>

Y.16.1.2 'mediaType'

This optional metadata, of type String, indicates the content type for the location referenced by the value. It shall be equivalent to the HTTP Content-Type header that would be used to describe the contents.

Links that refer to a data model path shall not have the 'mediaType' metadata present, since the mediaType of data model resources is not fixed (it is determined by the client with query parameters).

Links that do not refer to data model path shall have the 'mediaType' metadata present, even it is "*/* to indicate "unknown media type".

Y.16.1.3 'rel'

This optional metadata, of type String, indicates the relationship type for the link, and shall be a "Relation Type" as defined by RFC 5988.

Y.16.2 Built-in Links

Some links that are commonly used are built-in as standard metadata.

Y.16.2.1 'self'

The 'self' metadata, of type Link, can be applied to any data. Its value contains the location of the data. The 'self' link is included always in callback posts, otherwise it is included only when the client asks for it (e.g., with metadata=cat-links or metadata=self).

Y.16.2.2 'edit'

The 'edit' metadata, of type Link, can be applied to any data. Its value contains the URI that can be used to modify the data. It is only included if the URI to modify is different from the URI to read.

Y.16.2.3 'next'

The 'next' metadata, of type Link, can be applied to constructed data that are only partially represented. Its value contains the URI that can be used to obtain the next portion of the partially represented constructed data item.

The value and format of the next URI is determined solely by the server device. It shall be considered opaque to the client and the client shall not attempt to parse it or interpret it in any way.

The server shall ensure that the use of the 'next' pointer functions consistently for the client. The combined results of a series of partial results using 'next' links shall be the same as if the entire result set had been retuned at once, with the exception that items that have been removed subsequent to the initial partial result shall not be included in future partial results. All items that were present at the time of the initial partial result shall be conveyed by some future partial result so that the reassembled list does not miss any items regardless of any internal reordering or shifting that may have occurred subsequent to the first partial result. The means of ensuring this, the duration of the validity of the context information for a 'next' link, is a local matter. If the context for a 'next' link has expired, the server shall return a WS_ERROR_EXPIRED_CONTEXT error rather than inconsistent results.

Y.16.2.4 'via'

The 'via' metadata, of type Link, can be applied to any data. Its value contains the URI that points to another data item that is the source of the data for this data item.

The ‘via’ and ‘sourceId’ metadata are used to indicate that the data item mirrors, or is calculated from, another data item. Where a data item is calculated from another data item, or where an aggregating server has collected data from other servers, either to provide a common place for convenience, or to provide archiving services for trends or alarms, these metadata items allow the source data item to be identified. A data item need not be in the same form as the source item.

The "via" metadata contains the dereferenceable URL of the source of the data and the "sourceId" contains the nondereferenceable "id" of the data.

Y.16.2.5 'related'

The 'related' metadata, of type Link, can be applied to any data. Its value contains the URI that points to a data item that is logically associated with this data item. This is often used by data items in trees that provide arbitrary arrangements of data and thus have their own parent-child relationships among themselves. This link allows them to refer to the data that the tree is arranging.

Y.16.2.6 'alternate'

The 'alternate' metadata, of type Link, can be applied to any data. Its value contains the URI that provides an alternate means of accessing the resource. e.g., if some of the data is not readable for a given authorization context, this can provide an alternate context to use.

Y.16.2.7 'subscription'

The 'subscription' metadata, of type Link, can be applied to the callback List wrapper. Its value contains the URI of the subscription that created the callback notification. This is provided so the subscriber can match the notification to a subscription and knows how to cancel it if desired.

Y.16.2.8 'viaMap'

The 'viaMap' metadata, of type Link, can be applied to logically modeled data. Its value contains the URI of the mapped data source that is associated with the logical data. This shall refer to a mapped data item, e.g., a BACnet object in the /.bacnet tree.

Y.16.2.9 'viaExternal'

The 'viaExternal' metadata, of type Link, can be applied to logically modeled data. Its value contains the URI of an external protocol that indicates the source of the data, e.g., "modbus://192.168.1.1/r/40000". This standard makes no specifications for formats of external URIs and makes no requirements of clients to support external protocols.

Y.16.2.10 'represents'

The 'represents' metadata, of type Link, can be applied to logically modeled data. Its value contains the URI of another data item that this data item is logically representing in a new context. This is typically used to build trees with alternate children.

Y.17 Change Indications

Data can indicate when it was created and/or modified using the 'published' and 'updated' metadata. Additionally the actor responsible for the creation or modification can be recorded in the 'author' metadata.

Y.17.1 'published'

This optional metadata, of type DateTime, can be applied to any data item. If present, it indicates the time when the data item was created in the server device.

Y.17.2 'updated'

This optional metadata, of type DateTime, can be applied to any data item. If present, it indicates the time when the data item's value was last modified.

Y.17.3 'author'

This optional metadata, of type String, can be applied to any data item. If present, it indicates an identifier for the actor responsible for making the last change to the data item's value. The format and meaning of the contents of this string is a local matter.

Y.18 Definitions, Types, Instances, and Inheritance

This data model has a type and instance system similar to many programming languages.

A data item's "base type" defines the set of metadata and children that it is allowed to have. Every data item has a type, whether explicitly or implicitly specified.

A "definition" is a named data item that can be referred to by another data item by using the 'type', 'extends', or 'overlays' metadata. To alert the consumer that a named type definition is being created, data items that are to be used as definitions are declared within a "definition context". The mechanism by which the "definition context" is indicated varies by the manifestation of the data model and is defined by other clauses. Because definitions cannot be redefined, and are not scoped by context or depth, their names are required to be globally unique. Overlays may be used to augment existing definitions without changing them structurally.

A "structural change" is defined as one that:

(a) adds a new member to a Sequence, Composition, or Object,
(b) adds a new choice to a Choice,
(c) changes or adds the member type of an Array, List, Collection, or SequenceOf,
(d) adds any new named values or named bits,

(e) changes or adds any of presence restrictions: 'optional', 'absent', 'requiredWith', 'requiredWithout'.

Changes other than this are considered nonstructural. Typically, non-structural changes are limited to assigning a value for the data, but in some cases, other nonstructural metadata may be changed as well.

The terms "type" and "definition" are mostly synonymous and are often used interchangeably in this annex, but the term "definition" always refers to a named data item (a "typedef" in some languages), whereas the term "type" can also refer to anonymous types created inline within another definition and to the base types like "String".

An "instance" is a data item that refers to a definition data item using the 'type' metadata. If no 'type' metadata is provided, the data item's definition is implicitly identified by the base type.

The data model defined in this annex is used for both definitions and instances. The two contexts are mostly identical: definitions can have values, which can be considered their "default values", and instances can change previously defined metadata like 'maximum'. However, there are some restrictions that are noted in this clause and elsewhere. For example, an instance cannot add new children to 'namedValues' or add new members to a Sequence. Those actions can only take place in a definition context.

Data items "inherit" from their definition by logically copying the metadata and children of the definition into themselves and then adding or overlaying the metadata and children that are specified for the data item itself.

Even though some metadata, like 'requiredWith', are interpreted as a concatenated list of strings, newly specified metadata values are not merged with inherited values. Metadata values are replaced in their entirety when a new value is specified. Collection metadata, like 'namedValues', 'namedBits, and 'choices', however, are merged with their definitions, with new children being logically added to the list of existing children.

Because of this logical copying behavior, the term "inherit" is used in this annex to mean not only the process of adopting the existing members of a Sequence, Composition, or Object when making an extension, as is the common use of the term in Object Oriented languages, but also the process of receiving all the metadata and children logically from a definition. In this data model, even base types that represent "primitive" data, like Unsigned are actually composed of multiple parts, like 'maximum' and 'value', each of which would be modeled in Object Oriented languages as individual properties or member variables of an "Unsigned" class or type and which may have default values that were defined when they were declared or that were overridden by subsequent definitions or constructors. Those properties or member variables are all logically part of any instance of that type and retain ("inherit" in this annex) their default values unless overridden by the instance. This data model is designed to support that expected behavior.

For example, given this definition for a Real named "555-Percent":

```
<Definitions>
    <Real name="555-Percent" value="50" minimum="0" maximum="100" units="percent" />
</Definitions>
```

Consider the following two other definitions.

```
<Definitions>
    <Real name="555-LimitedPercent1" type="555-Percent" minimum="10" maximum="90"/>
</Definitions>

<Definitions>
    <Real name="555-LimitedPercent2" value="50" minimum="10" maximum="90" units="percent"/>
</Definitions>
```

The types defined by "555-LimitedPercent1" and "555-LimitedPercent2" are logically equivalent because 555-LimitedPercent1 inherited the value and 'units' metadata from its definition, "555-Percent", and overrode the

'minimum' and 'maximum' metadata to new values, while the "555-LimitedPercent2" specifies all metadata itself without the use of a previous definition.

The above example also shows that a definition can specify any metadata that are allowed by its type, including 'value'. So consider some instances of "555-Percent":

<Real type="555-Percent" />

<Real type="555-Percent" value="50" />

<Real type="555-Percent" value="25" />

<Real type="555-Percent" value="25" maximum="50" />

The first two instances are identical because all of the metadata of the definition "555-Percent" are inherited by all instances, making the value="50" in the second instance redundant. However, the third instance changes the value and so the value is required. The fourth instance shows not only that values can be changed in instances, but that any non-structural metadata can be changed as well.

Therefore, those four instances of "555-Percent" are logically equivalent to these four instances, respectively.

<Real value="50" minimum="0" maximum="100" units="percent" />

<Real value="50" minimum="0" maximum="100" units="percent" />

<Real value="25" minimum="0" maximum="100" units="percent" />

<Real value="25" minimum="0" maximum="50" units="percent" />

The copying behavior of the definition is cascaded as needed until data items with inherent definitions are reached. If, while copying a set of children, one of the children itself has a 'type' or 'extends' metadata, before that child's own metadata and children are considered, the contents of its definition are logically copied into it.

For example, given these three definitions:

```
<Definitions>
    <Unsigned name="555-UnlimitedPercent" units="percent" />
    <Unsigned name="555-NormalPercent" type="555-UnlimitedPercent" maximum="100"/>
    <Unsigned name="555-LimitedPercent" type="555-NormalPercent" minimum="10" maximum="90"/>
</Definitions>
```

These two instances are logically equivalent:

```
<Unsigned type="555-LimitedPercent" value="75"/>
<Unsigned value="75" minimum="10" maximum="90" units="percent"/>
```

So far, these examples have been making new definitions by making nonstructural changes to existing definitions. In these cases, the 'type' metadata was used within the definition context, rather than 'extends'. This is because an action like specifying maximum="100" in the definition for "555-NormalPercent" above was not changing the data model. The 'maximum' metadata is always part of the data model for the Unsigned base type, so this is a nonstructural change.

When structural changes are needed, the 'extends' metadata is used instead. This alerts the consumer that changes to the data model are allowed and, typically, that new children of a Sequence, Composition, Object, 'choices', or 'namedValues' are being added.

For example, consider this definition for the type named "555-base".

```
<Definitions>
    <Sequence name="555-base">
        <Real name="foo"/>
    </Sequence>
</Definitions>
```

The following extension creates a new definition for a type named "555-derived", which is based on "555-base".

```
<Definitions>
    <Sequence name="555-derived" extends="555-base">
        <Real name="bar"/>
    </Sequence>
</Definitions>
```

An instance of "555-derived" thus contains the members defined in "555-base" as well as those added in "555-derived".

```
<Sequence type="555-derived">
    <Real name="foo" value="1.0"/>
    <Real name="bar" value="2.0"/>
</Sequence>
```

The 'extends' metadata is only used in the definition context, but is not limited to the outermost data item that is defining the new type. When used on an inner data item, a new anonymous type is created as an extension of the referenced type. Thus, anonymous types are either fully defined in-line without the use of the 'extends' metadata, or are an extension of an existing type by using the 'extends' metadata.

The following example shows all four methods of assigning a type for a new member. The "simple-member" uses a built-in type with no need for the 'type' or 'extends' metadata. The "typed-member" refers to the "555-derived" type using the 'type' metadata. The "full-anonymous-type-member" fully defines its anonymous type in-line, also without the use of the 'type' or 'extends' metadata. The "extension-anonymous-type-member" extends an existing type using the 'extends' metadata.

```
<Definitions>
    <Sequence name="555-example-1">
        <Real name="simple-member"/>
        <Sequence name="typed-member" type="555-derived"/>
        <Sequence name="full-anonymous-type-member">
            <Real name="foo" />
            <Real name="bar" />
        </Sequence>
        <Sequence name="extension-anonymous-type-member" extends="555-base">
            <Real name="bar" />
        </Sequence>
    </Sequence>
</Definitions>
```

The result is that the last three members defined above all have child members named "foo" and "bar".

An instance of that sequence, providing a value for each member, looks like this.

```
<Sequence type="555-example-1">
    <Real name="simple-member" value="55"/>
    <Sequence name="typed-member">
        <Real name="foo" value="1"/>
        <Real name="bar" value="2" />
    </Sequence>
    <Sequence name="full-anonymous-type-member">
        <Real name="foo" value="1"/>
        <Real name="bar" value="2" />
    </Sequence>
    <Sequence name="extension-anonymous-type-member">
        <Real name="foo" value="1"/>
        <Real name="bar" value="2" />
    </Sequence>
</Sequence>
```

The 'type' metadata on a data item is required only where it cannot be determined from the context in which the data appears. In the above example, the 'type' metadata for the three structured members of "555-example-1" was not given, because children are always matched by name with their corresponding child in the definition of their parent. This matching continues up the ancestor chain until the context cannot be determined, at which point a 'type', 'extends', or 'overlays' metadata shall be present to provide a context for all the descendants. The 'type' metadata is not disallowed in contexts where it can be inferred, but it is not required, and should be left off where brevity of textual serializations is desirable.

In this example, the 'type' metadata is required to define the type for the member "property-identifier".

```
<Definitions>
    <Sequence name="0-BACnetPropertyReference">
        <Enumerated name="property-identifier"     contextTag="0"  type="0-BACnetPropertyIdentifier/>
        <Unsigned     name="property-array-index" contextTag="1"  optional="true" />
    </Sequence>
</Definitions>
```

In the textual serialization below, the use of the 'type' metadata on the outer data item sets the context for interpretation of the inner items; therefore, the 'type' metadata is not needed on the "property-identifier" member because it is known from the definition of 0-BACnetPropertyReference.

```
<Sequence type="0-BACnetPropertyReference" >
    <Enumerated name="property-identifier" value="present-value" />
</Sequence>
```

The use of the 'type' metadata indicates that a data item is an instance of a previously defined type definition and that its metadata and children do not cause any structural changes. Conversely, the use of the 'extends' metadata indicates that structural changes are allowed and expected. Consequently, the 'type' metadata can be used in any context, but the 'extends' metadata can only be used in a definition context where a new type is being created.

A "structural change" is defined as one that adds a new member to a Sequence, Composition, or Object, adds a new choice to a Choice, adds any new named values, or changes the 'optional', 'absent', or 'contextTag' attributes. Changes other than this are considered nonstructural. Typically, non-structural changes are limited to assigning a value for the data, or defining an extended type for an existing item, but in some cases, other nonstructural metadata may be changed as well.

The 'type' of an item is allowed to change, but only to a type that is defined to be an extension of the original type. All types are considered to be an extension of Any, therefore an item defined as Any can be replaced with any other type. Likewise, an unspecified 'memberType', or a 'memberType' of "Any", can be replaced by any other type.

For an example of a nonstructural change, consider the definition:

```
<Definitions>
    <Sequence name="555-base">
        <Real name="foo"/>
    </Sequence>
</Definitions>
```

A new definition can be made from that without making structurally changes to "555-base" by using the 'type' metadata in a definition context. Below, the existing member is only given new metadata; in this case, new limits.

```
<Definitions>
    <Sequence name="555-limited-base" type="555-base">
        <Real name="foo" minimum="0.0" maximum="100.0" />
    </Sequence>
</Definitions>
```

If, however, a structure change is needed, the 'extends' metadata is used instead of the 'type' metadata. Below, the derived type adds a new member.

```
<Definitions>
    <Sequence name="555-limited-base" extends="555-base">
        <Real name="bar" />
    </Sequence>
</Definitions>
```

Inheriting 'namedValues' is only allowed in a definition context and involves overlaying existing children and adding new ones to the end.

For example, this enumeration definition creates an enumeration where red=0, green=1, and blue=6.

```
<Definitions>
    <Enumerated name="555-base-enum">
        <NamedValues>
            <Unsigned name="red" />
            <Unsigned name="green" />
            <Unsigned name="blue" value="6"/>
        </NamedValues>
    </Enumerated>
</Definitions>
```

An extension to that enumeration adds 'displayName' metadata to the existing red, green, and blue named values and adds new named values for purple and yellow. The order of existing named values is deliberately skewed in this example to illustrate that it does not matter since they have already been assigned values, but the order of the newly added purple and yellow is significant.

```
<Definitions>
    <Enumerated name="555-extended-enum" extends="555-base-enum">
        <NamedValues>
            <Unsigned name="green" displayName="Green"/>
            <Unsigned name="purple" displayName="Purple"/>
            <Unsigned name="blue" displayName="Blue"/>
            <Unsigned name="yellow" displayName="Yellow"/>
            <Unsigned name="red" displayName="Red"/>
        </NamedValues>
    </Enumerated>
</Definitions>
```

The above extension logically has an ordered list of named values.

```
<NamedValues>
   <Unsigned name="red" displayName="Red" />
   <Unsigned name="green" displayName="Green" />
   <Unsigned name="blue" displayName="Blue" value="6"/>
   <Unsigned name="purple" displayName="Purple" />
   <Unsigned name="yellow" displayName="Yellow" />
</NamedValues>
```

This ordering means that the automatically assigned values for purple and yellow will be 7 and 8, respectively. Since this was not obvious from the definition of "555-extended-enum", enumerations should explicitly assign values when those enumerations are mapped to external data or where the numerical values are otherwise significant outside of the data model.

Inheriting 'choices' is only allowed in a definition context and involves overlaying existing children and adding new ones to the list of choices (order is not significant).

For example, this choice definition creates two choices and defines the default value of the choice itself to be the "joe" choice.

```
<Definitions>
    <Choice name="555-base-choice">
        <Choices>
            <Unsigned name="fred" displayName="Fred"/>
            <Real name="joe" displayName="Joe"/>
        </Choices>
        <Real name="joe"/>
    </Choice>
</Definitions>
```

An extension to that choice changes a 'displayName' of the existing choices and adds a new "bob" choice. Additionally, it changes the default value of the choice itself to be "bob" rather than the default value for "555-base-choice", which was "joe". The order of existing choices is deliberately skewed in this example to illustrate that it does not matter, and the order of the resulting list of choices is not significant either.

```
<Definitions>
    <Choice name="555-extended-choice" extends="555-base-choice">
        <Choices>
            <Real name="joe" displayName="Joseph"/>
            <Double name="bob" displayName="Robert"/>
            <Unsigned name="fred" displayName="Frederick"/>
        </Choices>
        <Real name="bob"/>
    </Choice>
</Definitions>
```

Y.19 Data Revisions

It is typical that definitions of control systems data, including data defined by BACnet's Clauses 12 and 21, can evolve and change over time. New members, usually optional, are added to data constructs, new values are defined for enumerations and bit strings, and new properties are added to objects. Occasionally, some existing items are modified or removed. To model these changes, data modelers could choose to create new data types with new names. But doing so does not let consumers of these data definitions know that these "new" types are in fact slight modifications of previous definitions that they already know how to handle. In many cases, the "added" parts could have been easily handled or ignored.

Recognizing the dynamic nature of data definitions, CSML provides the ability to indicate revisions to existing data definitions. Clients therefore have the ability to work with simple extensible data types for the most common use cases, but also to be able to know about the details of the revision history for more advanced use cases.

Y.19.1 'addRev'

This optional metadata, of type String, indicates the data revision when a data item was added to a definition. This can only be applied in a definition context and is applicable to constructed members, choices, values of enumerations, and bits of bit strings.

Y.19.2 'remRev'

This optional metadata, of type String, indicates the data revision when a data item was removed from a definition. This can only be applied in a definition context and is applicable to constructed members, choices, values of enumerations, and bits of bit strings.

Y.19.3 'modRev'

This optional metadata, of type String, indicates the data revision when the definition of a data item was last modified. This can only be applied in a definition context and is applicable to members of constructed types and choices.

Y.19.4 'dataRev'

This optional metadata, of type String, indicates the data revision level of an instance of data. This information can be used to compare to the 'addRev', 'modRev', and 'remRev' information in the definition for this data type. This value applies to all enclosed data until overridden by another 'dataRev'. Therefore, for brevity, it is usually only applied at the outermost element where it applies.

This metadata, and the metadata 'addRev', 'remRev', and 'modRev', are of type String; however, they shall be restricted to being composed of 1 to 16 numeric digits. They shall be interpreted as a decimal number for comparison purposes, with leading zeros allowed. For example, "0005024" is less than "06789" and greater than "678".

Since 'dataRev' needs to be unambiguously interpreted for any given instance of data, it needs to have the same meaning in all type definitions in the inheritance chain for that instance. Therefore, if a definition is extended by another definition, that extension shall share a common meaning for the base definition's 'dataRev'. For example, BACnet data definitions define the meaning of 'dataRev' to be tied to the Protocol_Revision of the BACnet device that the data is associated with. Therefore, type definitions derived from BACnet data types shall inherit that same meaning for 'dataRev'.

Y.19.5 'revisions'

This optional metadata, of type Collection of Any, is used to hold extra information about past revisions of the data item. Each member of 'revisions' shall be the same base type as the data item and the name of the member shall indicate the data revision number that applies to that member.

Y.19.6 Indicating Definition Revisions

To accomplish this revisioning, three metadata are defined to annotate when data items were added, modified, or removed from a data definition. These metadata are 'addRev', 'modRev', and 'remRev'. Additionally, historical values for attributes of modified elements can be indicated under the 'revisions' metadata.

For BACnet data definitions, the data revision number is equal to the Protocol_Revision of the specification in which the data was defined or modified.

For example, consider the following definition:

```
<Definitions>
  <Object name="0-SomeObject" addRev="15" >
    ...
    <Real name="new-prop" propertyIdentifier="9991" maximum="100" writable="true" addRev="18" modRev="21" >
      <Revisions>
        <Real name="18" propertyIdentifier="9991" optional="true" requiredWith="old-prop" />
        <Real name="19" propertyIdentifier="9991" optional="true" requiredWith="old-prop" maximum="100" />
        <Real name="20" propertyIdentifier="9991" optional="true" maximum="100" />
      </Revisions>
    </Real>
    <Real name="old-prop" propertyIdentifier="9992" optional="true" remRev="20"  />
    ...
    <Real name="bad-idea" propertyIdentifier="9993" optional="true" addRev="18" modRev="19" remRev="20"  >
      <Revisions>
        <Real name="18" propertyIdentifier="9993" comment="full of promise" />
        <Real name="19" propertyIdentifier="9993" optional="true"  comment="no so sure now" />
      </Revisions>
    </Real>
    <Real name="better-idea" propertyIdentifier="9994" comment="replaces bad-idea" addRev="20" />
    ...
  </Object >
</Definitions>
```

The current object definition shows that "last known state" of all properties, even for those that have been removed. The optional metadata for each property contains the history of previous definitions. With this information, a simple client can simply determine the presence/absence of a property without needing to examine the 'revisions' metadata.

However, a more sophisticated client can determine the following full history of this object and its properties:

In data rev 15, this object type was first defined.
- At that time, the object contained a property named "old-prop", along with many other properties.

In data rev 18,
- The property named "new-prop" was added as optional but dependent on "old-prop".
- The property named "bad-idea" was added and was required.

In data rev 19,
- The property "new-prop" was restricted to a maximum of 100.
- The property "bad-idea" was made optional.

In data rev 20,
- The property "old-prop" was removed.
- Since "old-prop" is gone, "new-prop" is no longer dependent on it.
- The property "bad-idea" was removed.

- The property "better-idea" was added.

In data rev 21,

- The property "new-prop" is made writable.

Y.19.7 Indicating Instance Revisions

In addition to indicating what data revisions apply in the definition context, instances of data items can indicate what their current data revision is so that clients can consult the appropriate definition for that instance. This is indicated with the 'dataRev' metadata.

For data originating from a BACnet data source, this data revision number is equal to the Protocol_Revision of the device containing the data.

For example, first consider the following instance of the definition in the example above:

```
<Object name="B72_heat_valve" type="0-SomeObject" dataRev="18" >
   ...
   <Real name="old-prop"  value="0.0" />
   <Real name="new-prop" value="123.0" />
   <Real name="bad-idea"  value="666.0"  />
   ...
</Object >
```

In this instance, the 'dataRev' is 18. Therefore, "old-prop" is possibly present, and because it is actually present in this instance, so is "new-prop". But "new-prop" is not limited to 100. The required "bad-idea" property is also present.

Then consider the following instance:

```
<Object name="outside-damper" type="0-SomeObject" dataRev="23" >
   ...
   <Real name="new-prop" value="100.0" />
   <Real name="better-idea" value="12.34"  />
   ...
</Object >
```

Because the 'dataRev' is 23, "old-prop" is not possible, and "new-prop" is limited to 100. The "bad-idea" property has been replaced with the "better-idea" property.

Y.20 BACnet-Specific Base Types

Data items that are used to model data from BACnet sources have an additional set of available base types that correspond to BACnet specific datatypes.

Y.20.1 ObjectIdentifier

Object Identifier primitive data is modeled with the ObjectIdentifier base type. In addition to the common metadata described in Clause Y.4, the ObjectIdentifier base type also supports the value specifiers described in Clause Y.4, the range restrictions described in Clause Y.8, and the named values described in Clause Y.5.

For textual serialization contexts, the value of an ObjectIdentifier is a string. The format of this string is "T,N", where

T represents the type. The type identifier is defined by the context where the ObjectIdentifier appears. In BACnet contexts, it is either the object type formatted as a decimal number with no leading zeroes, or a standard type name exactly equal to the names specified in the definition for BACnetObjectTypes in Clause 21, or from the definition of "0-BACnetObjectType", or from the definition of the type indicated by the 'objectType' metadata, if available.

N represents the object instance number and is a decimal number with no leading zeroes.

Y.20.2 ObjectIdentifierPattern

Object Identifier primitive data that allows independent specification of type and instance is modeled with the ObjectIdentifierPattern base type. In addition to the common metadata described in Clause Y.4, the ObjectIdentifierPattern base type also supports the value specifiers described in Clause Y.4, the range restrictions described in Clause Y.8, and the named values described in Clause Y.5.

For textual serialization contexts, the value of an ObjectIdentifierPattern is a string. The format of this string is "T,N", where

T is either a type identifier or a single asterisk ("*") character to indicate "unspecified". The type identifier is defined by the context where the ObjectIdentifier appears. In BACnet contexts, it is either a decimal number with no leading zeroes, or a standard type name exactly equal to the names specified in the definition for BACnetObjectTypes in Clause 21, or from the definition of "0-BACnetObjectType", or from the definition of the type indicated by the 'objectType' metadata, if available.

N is either an object instance number or a single asterisk ("*") character to indicate "unspecified". The instance number is a decimal number with no leading zeros.

Y.20.3 WeekNDay

Month-Week-Day primitive data is encoded with the WeekNDay base type. In addition to the common metadata described in Clause Y.4, the WeekNDay base type also supports the value specifier described in Clause Y.4 and the named values described in Clause Y.5.

For textual serialization contexts, the value of a WeekNDay is a string. The format of the string value is "M,W,D", where:

M is either a decimal month identifier or an asterisk ("*") character to indicate "unspecified",

W is either a decimal week identifier or an asterisk ("*") character to indicate "unspecified",

D is either a decimal day-of-week identifier or an asterisk ("*") character to indicate "unspecified".

The numeric fields do not have leading zeros. The M, W, and D fields are separated by a comma (",") character. The range and meaning of the numeric values for M, W, and D is described in the BACnet WeekNDay production in Clause 21.

Y.21 BACnet-Specific Metadata

Data that is used to model data from BACnet sources have an additional set of available metadata that are specific to BACnet objects and services.

Y.21.1 'writableWhen'

This optional metadata, of type Enumerated, indicates the conditions under which the data value may be writable. The choices for the value and their meanings are defined in the following table. The choices for the value and their meanings are defined in the following table.

Table Y-6. Values for Metadata 'writableWhen'

Metadata Value	Meaning
"out-of-service"	When Out_Of_Service property is TRUE
"commandable"	When this property is commandable
"other"	Non-standard requirement. Descriptive text should be provided by 'writableWhenText' metadata.

If the 'writableWhenText' metadata is present then the implied value for this metadata is "other". Therefore, the 'writableWhenText' metadata may be present without requiring the presence of the 'writableWhen' metadata.

However, if a value for the 'writableWhen' metadata is specified and is not equal to "other", then the 'writableWhenText' metadata is not inherited from a definition and the 'writableWhenText' metadata shall not be specified in the same context.

When this metadata is equal to "other", the optional 'writableWhenText' metadata can be used to provide display text to describe the nonstandard condition.

A common case in Clause 12 objects is the requirement that Present_Value be writable when Out_Of_Service is true.

```
<Definitions>
    <Object name="0-AnalogInputObject">
        …
        <Real name="present-value" writableWhen="out-of-service" … />
        …
    </Object>
</Definitions>
```

Y.21.2 'writableWhenText'

This optional localizable metadata, of type String, is used to provide display text for the writability condition when the 'writableWhen' metadata has the value of "other". While the 'writableWhen' metadata is an enumeration of fixed strings as defined by this standard, the 'writableWhenText' metadata can contain arbitrary text that shall consist of plain printable characters with no formatting markup or line breaks.

For example, if the writability condition is not one of the standard conditions, then the 'writableWhen' metadata has the value of "other" and the members of 'writableWhenText' (encoded in XML as <WritableWhenText> elements) provide the display text (the default locale in this example is "en").

```
<Unsigned name="trend-memory-allocation" writableWhen="other" >
    <WritableWhenText>The Device object's Device Status property is "download required"</WritableWhenText>
</Unsigned>
```

If a 'writableWhenText' metadata is present in a context without the 'writableWhen' metadata, the 'writableWhen' metadata is implicitly assigned the value "other".

For example, the following shows that it is not necessary to include writableWhen="other" in a context with a 'writableWhenText' string (encoded in XML as a <WritableWhenText> element).

```
<Unsigned name="aux-input" >
    <WritableWhenText>The "Aux Disable" property is TRUE</WritableWhenText>
</Unsigned>
```

Y.21.3 'requiredWhen'

This optional metadata, of type Enumerated, indicates the conditions under which optional data shall be present. The choices for the value and their meanings are defined in Table Y-7.

Table Y-7. Values for Metadata 'requiredWhen'

Metadata Value	Meaning
"intrinsic-supported"	If the object supports intrinsic reporting
"cov-notify-supported"	If the object supports COV reporting
"cov-subscribe-supported"	If the device supports execution of either the SubscribeCOV or SubscribeCOVProperty service
"present-value-commandable"	If Present_Value is commandable
"segmentation-supported"	If Segmentation of any kind is supported
"virtual-terminal-supported"	If Virtual Terminal services are supported

"time-sync-execution"	If the device supports the execution of the TimeSynchronization service
"utc-time-sync-execution"	If the device supports the execution of the UTCTimeSynchronization service
"time-master"	If the device is a Time Master
"backup-restore-supported"	If the device supports the backup and restore procedures
"slave-proxy-supported"	If the device is capable of being a Slave-Proxy device
"slave-discovery-supported"	If the device is capable of being a Slave-Proxy device that implements automatic discovery of slaves
"other"	Non-standard requirement. Descriptive text should be provided by requiredWhenText metadata.

If the 'requiredWhenText' metadata is specified then the implied value for this metadata is "other". Therefore, the 'requiredWhenText' metadata may be present without requiring the presence of the 'requiredWhen' metadata. However, if a value for the 'requiredWhen' metadata is specified and is not equal to "other", then the 'requiredWhenText' metadata is not inherited and a 'requiredWhenText' metadata shall not be specified in the same context.

When this metadata is equal to "other", optional 'requiredWhenText' metadata can be used to provide display text to describe the nonstandard condition.

Many properties in Clause 12 objects have their presence dependent on a standard condition.

```
<Definitions>
    <Object name="0-DeviceObject">
        …
        <Unsigned name="max-segments-accepted" optional="true"
                  requiredWhen="segmentation-supported" … />
        …
    </Object>
</Definitions>
```

Y.21.4 'requiredWhenText'

This optional localizable metadata, of type String, is used to provide display text for the presence requirements when the 'requiredWhen' metadata has the value of "other". While the 'requiredWhen' metadata is an enumeration of fixed strings as defined by this standard, the 'requiredWhenText' metadata can contain arbitrary text that shall consist of plain printable characters with no formatting markup or line breaks.

For example, if the presence requirement is not one of the standard conditions, then the 'requiredWhen' metadata has the value of "other" and the members of the 'requiredWhenText' metadata (encoded in XML as <RequiredWhenText> elements) provide the display text (the default locale in this example is "en").

```
<Unsigned name="auxbaud" optional="true" requiredWhen="other" >
    <RequiredWhenText>The device is configured as a gateway</RequiredWhenText>
</Unsigned>
```

If a 'requiredWhenText' metadata is present in a context without the 'requiredWhen' metadata, then the 'requiredWhen' metadata is implicitly assigned the value "other".

For example, the following shows that it is not necessary to include requiredWhen="other" in a context with a 'requiredWhenText' member (encoded in XML as a <RequiredWhenText> element).

```
<Unsigned name="aux-limit" >
    <RequiredWhenText>The object is configured to support aux input<RequiredWhenText>
</Unsigned>
```

Y.21.5 'contextTag'

This optional metadata, of type Unsigned, indicates the context tag that should be used when encoding this data in ASN.1 according to the rules in Clause 20.

If this metadata is absent, then the data is "application tagged" when encoding in ASN.1. Once assigned a value by a definition, it cannot be changed by an instance or a derived type definition.

For example, because the deviceIdentifier field of the BACnetDeviceObjectReference construct is optional, the fields are context tagged.

```
<Definitions>
    <Sequence name="0-BACnetDeviceObjectReference">
        <ObjectIdentifier name="deviceIdentifier" contextTag="0" optional="true" />
        <ObjectIdentifier name="objectIdentifier" contextTag="1" />
    </Sequence>
</Definitions>
```

Y.21.6 'propertyIdentifier'

This optional metadata, of type Unsigned, indicates the property identifier that is to be used when accessing this data's value as a BACnet property.

If this metadata is absent, then the data is not intended to be accessed as a BACnet property. Once assigned a value by a definition, it cannot be changed by an instance or a derived type definition.

The following example declares that the Present_Value of an Analog Output object is accessible with property identifier 85.

```
<Definitions>
    <Object name="0-AnalogOutputObject">
        …
        <Real name="present-value" propertyIdentifier="85" … />
        …
    </Object >
</Definitions>
```

Y.21.7 'objectType'

This optional metadata, of type String, indicates the type name of the Enumerated type that shall be used for the object type identifier portion of the ObjectIdentifier and ObjectIdentifierPattern base types. This type shall be an extension of "0-BACnetObjectType".

[Change BACnetNodeType production in **Clause 21**, p. 687]

```
...
BACnetNodeType ::= ENUMERATED {
        unknown         (0),
        system          (1),
        network         (2),
        device          (3),
        organizational  (4),
        area            (5),
        equipment       (6),
        point           (7),
        collection      (8),
        property        (9),
```

functional (10),
other (11),
subsystem (12),
building (13),
floor (14),
section (15),
module (16),
tree (17),
member (18),
protocol (19),
room (20),
zone (21)
}

[Add new production to **Clause 21**, p. 709]

BACnetRelationship ::= ENUMERATED {
unknown (0),
default (1),
-- Note that all of the following are in even/odd pairs indicating forward and reverse relationships.
-- Proprietary extensions shall also follow the same even/odd pairing so that consumers can determine, for
-- any given relationship, what the opposite relationship is.
contains (2),
contained-by (3),
uses (4),
used-by (5),
commands (6),
commanded-by (7),
adjusts (8),
adjusted-by (9),
ingress (10),
egress (11),
supplies-air (12),
receives-air (13),
supplies-hot-air (14),
receives-hot-air (15),
supplies-cool-air (16),
receives-cool-air (17),
supplies-power (18),
receives-power (19),
supplies-gas (20),
receives-gas (21),
supplies-water (22),
receives-water (23),
supplies-hot-water (24),
receives-hot-water (25),
supplies-cool-water (26),
receives-cool-water (27),
supplies-steam (28),
receives-steam (29),
...
}
-- Enumerated values 0-1023 are reserved for definition by ASHRAE. Enumerated values 1024-65535
-- may be used by others subject to the procedures and constraints described in Clause 23.

[Change **Table 23-1**, p. 718]

Table 23-1. Extensible Enumerations

Enumeration Name	Reserved Range	Maximum Value
...	...	...
BACnetLightingTransition	0-63	255
BACnetRelationship	*0-1023*	*65535*

[Change **Clause 12.29.5**, p. 314]

12.29.5 Node_Type

This property, of type BACnetNodeType, provides a general classification of the object in the hierarchy of objects.

It is intended as a general suggestion to a client application about the contents of a Structured View object, and is not intended to convey an exact definition. Further refinement of classification is provided by the Node_Subtype property. The allowable values for this property are:

{UNKNOWN, SYSTEM, *SUBSYSTEM,* NETWORK, DEVICE, ORGANIZATIONAL, AREA, *BUILDING, FLOOR, ROOM, ZONE,* EQUIPMENT, *SECTION, MODULE,* POINT, COLLECTION, PROPERTY, FUNCTIONAL*, TREE, MEMBER, PROTOCOL,* OTHER}

Where the following are suggested interpretations:

UNKNOWN	Indicates that a value for Node_Type is not available or has not been configured at this time
SYSTEM	An entire mechanical system
SUBSYSTEM	*A portion of a mechanical system*
NETWORK	A communications network
DEVICE	Contains a set of elements which collectively represents a BACnet device, a logical device, or a physical device
ORGANIZATIONAL	Business concepts such as departments or people
AREA	Geographical concept such as a *region, area,* campus~~, building, floor~~, etc.
BUILDING	*Geographical concept of a building*
FLOOR	*Geographical concept of a floor of a building*
ROOM	*Geographical concept of a room of a building*
ZONE	*Abstract concept that may span or subdivide geographical boundaries for a particular purpose such as HVAC, lighting, security, fire, etc.*
EQUIPMENT	Single piece of equipment that may be a collection of "Points" *and "Sections"*
SECTION	*A section of a piece of equipment that may be a collection of "Points" or other "Sections"*

MODULE	*A modular part of a device (physical or logical)*
POINT	Contains a set of elements which collectively defines a single point of data, either a physical input or output of a control or monitoring device, or a software calculation or configuration setting
COLLECTION	A generic container used to group things together, such as a collection of references to all space temperatures in a building
PROPERTY	Defines a characteristic or parameter of the parent node
FUNCTIONAL	Single system component such as a control module or a logical component such as a function block
TREE	*A logical hierarchical arrangement*
MEMBER	*Part of a collection*
PROTOCOL	*A grouping of data reachable by a single protocol*
OTHER	Everything that does not fit into one of these broad categories

135-2012*am*-3. Rework Annex Q to be an XML syntax for the abstract model.

Rationale

The data model defined by the existing Annex Q is currently tied closely to the definition of the XML syntax. Section 2 of this addendum separated out the data model, and this part edits Annex Q itself to refer to that common model.

It is intended that the net result of extracting the data model and reworking the syntax definition is that no substantive changes are made to the resulting Structure and meaning of existing XML, with the following exceptions:
- The namespace is changed from "http://www.bacnet.org/CSML/1.0" to "http://bacnet.org/csml/1.2"
- The <Documentation> element is changed from a fixed content type of "XHTML" to variable type based on a 'mediaType' metadata, and its content is changed from "mixed" to xs:string to simplify XML parsing by removing a base requirement for parsing XML other than CSML.
- Non-file contexts (e.g. web services) can now start with any data element, like <String>, rather than always using a <CSML> wrapper.
- The 'defaultLocale' can be applied to any data element when it is the top level element.
- Any other changes to structure or meaning are unintentional and will be corrected.

Additions:
- Comments can now be localized.
- New metadata added: nodeType, nodeSubtype, fault, overridden, outOfService, inAlarm, isMultiline, links, tags.

[Change Annex Q, p. 962]

ANNEX Q - XML DATA FORMATS (NORMATIVE)

(This annex is part of this standard and is required for its use.)

This annex defines formats for XML data exchanged between various BAS systems. This data may have a variety of purposes and may be conveyed through files or by other means.

Q.1 Introduction

The Extensible Markup Language (XML) is a format for structured text that can be used to represent a variety of data in a machine-readable form. The XML syntax used in this standard conforms to the "Extensible Markup Language (XML) 1.0 (Fifth Edition)", and the XML datatypes used in this standard, indicated by the prefix "xs:", refer to the datatypes defined by "XML Schema Part 2: Datatypes Second Edition."

This XML syntax is based on the abstract data model in Annex Y. The abstract model is composed of "data" and "metadata". While most of the abstract data model metadata are mapped directly to XML attributes, several data model metadata are actually complex data types that cannot be represented as primitive XML attributes. In these cases, the mapping to XML elements is defined by this annex.

For representing BACnet data, the ~~This~~ syntax allows data structure definitions, such as those in Clause 21, and instances of those definitions to be represented in XML. The syntax is optimized for efficient representation of BACnet data and is sufficiently flexible not to be limited to modeling BACnet data exclusively.

~~Additionally, the~~ *The* syntax *also* allows for human language descriptions, range restrictions, and usage information to be added to the basic data structure definitions.

Q.1.1 Design

This XML syntax is designed to provide a ~~common~~ syntax ~~and data model~~ that can be used to represent both standard and proprietary data types along with any accompanying descriptive information. It is a general data

definition and instance language rather than a syntax specific for the data defined in this standard. To allow the flexibility to present proprietary data along with standard data with a consistent syntax and data model, the syntax uses a datatype-centric form, such as <String name="present-value" value="75"> and <Boolean name="proprietary-value" value="true"> rather than a syntax specific to ~~standard BACnet~~ data names, such as <PresentValue>75</PresentValue>. This allows any kind of data, standard or proprietary, to be represented in the future without changing the XML schema or the code for the low level parsing of the XML into a native form for higher level processing to consume. It also allows the names of the data to be values that are not legal as XML element tag names or attribute names.

For the purposes of document brevity and human readability, this syntax represents data in XML attribute form rather than element body text form wherever possible. However, this representation choice does not limit extensibility of the data model because, like the *data model metadata* ~~attributes in the BACnet Web services data model~~ described in ~~Annex N~~ *Annex Y*, most attributes defined in this annex can be extended to have attributes of their own using an extension syntax defined in Clause Q.7. This allows XML brevity for the common cases without limiting extensibility when needed.

[Note: see new Clause Q.1.1.1]

~~With the exception of the <Documentation> element, which contains rich text XHTML-formatted documentation, this standard does not use mixed content in any element body. So simple consumers that ignore formatted documentation only need to process attribute text and simple element body text.~~

Validation of this syntax may be accomplished with XML data validators such as XML Schema. These validators can be used to validate the type and range of data in elements and attributes of the syntax itself. This syntax is also intended to represent a higher-level data model, such as BACnet objects and properties. Higher level data model validation, such as whether a property is allowed in a given object or whether its value is within its declared minimum and maximum limits, is beyond the capabilities of XML syntax validators like XML Schema and shall therefore be performed as needed by the application consuming this XML.

[Note: the following restriction is now defined in Annex Y]

~~To simplify processing and avoid the definition of potentially complex scoping rules, all datatype definitions are given globally unique names. The management of the names to ensure global uniqueness is a local matter to the organization producing the XML and should at least consist of a prefix for the name that is unique to an organization and then have the organization manage everything that follows that prefix. For brevity, if an organization has a BACnet Vendor ID, the prefix can consist of that ID as a decimal number followed by a dash character ('-'). In all cases, however, a reversed domain name, like "com.companyname.controlsdivision.", can be used as a prefix to ensure global uniqueness.~~

Q.1.1.1 ***XML Requirements and Restrictions***

This standard uses standard XML syntax but anticipates that some implementations may be resource constrained or use parsing libraries with limited capabilities. Therefore, the requirements for parsing XML are designed to be a compatible subset of the full XML specification. The following requirements are made of consumers and producers.

Consumers are required to:

(a) *parse and check the single default namespace specifier "xmlns" specified in Clause Q.2.1.*
(b) *parse and ignore namespace specifiers it doesn't understand.*
(c) *parse and ignore elements or attributes with namespace prefixes for namespaces it doesn't understand.*
(d) *support XML entities "quot", "amp", "apos", "lt", and "gt" (e.g., ">").*
(e) *parse CDATA sections (e.g., "<![CDATA[...something...]]>"). The CDATA wrapper is not part of the value of the String (e.g., value above is "...something...").*
(f) *support numeric character references in the form "&#nnnn;" and "&#xhhhh;", where nnnn is the code point in decimal and hhhh is the code point in hexadecimal. The 'x' shall be lowercase. The nnnn or hhhh can be any number of digits and can include leading zeros. The hhhh can mix uppercase and lowercase.*

(g) *parse all standard elements and attributes defined by this standard.*
(h) *parse body text for elements that specifically call for it (e.g., <Documentation>, <ErrorText>, etc.)*
(i) *check valid contents of standard elements and attributes that the implementation supports. Based on the implementation's capabilities, contents of unsupported items can be ignored.*

Correspondingly, consumers are not required to:
(a) *process namespace specifiers other than the default "xmlns".*
(b) *process elements and attributes with namespace prefixes.*
(c) *parse entity definitions.*
(d) *parse entities other than "quot", "amp", "apos", "lt", and "gt".*
(e) *parse mixed content.*

Therefore, producers shall not:
(a) *generate entity definitions.*
(b) *generate entities other than "quot", "amp", "apos", "lt", and "gt".*
(c) *generate mixed content.*
(d) *generate a namespace prefix on standard elements and attributes.*

Q.1.2 Syntax Examples

Some examples using the Clause 21 datatypes will provide an introduction to the form and capabilities of the syntax. The full details of the XML elements and attributes are defined in Clauses Q.3 and ~~Q.4, and a~~ *Q.4. The* description of the data model and the system for defining and extending data types and expressing instances of those types is described in ~~Clause Q.5~~ *Annex Y.*

...

BACnetPropertyReference ::= SEQUENCE {

~~propertyIdentifier~~ *property-identifier*	[0] BACnetPropertyIdentifier,
~~propertyArrayIndex~~ *property-array-index*	[1]Unsigned OPTIONAL --used only with array datatype -- if omitted with an array the entire array is referenced

}

In XML, this sequence also specifies names, context tags, optionality, and can even capture comments:

```
<Definitions>
  <Sequence name="0-BACnetPropertyReference">
    <Enumerated name="~~propertyIdentifier~~ property-identifier" contextTag="0"
                type="0-BACnetPropertyIdentifier" />
    <Unsigned name="~~propertyArrayIndex~~ property-array-index" contextTag="1" optional="true"
                comment="Used only with array datatype. If omitted, the entire array is referenced."/>
  </Sequence>
</Definitions>
```

An XML representation of a value of that sequence may provide values for each member that is present, using a representation appropriate to that member's type, and omit optional members that are not present.

```
  <Sequence type="0-BACnetPropertyReference" >
    <Enumerated name="~~propertyIdentifier~~ property-identifier" value="present-value" />
  </Sequence>
...
```

Q.2 *XML* Document Structure

The XML elements and attributes defined in this annex may be used for a variety of purposes and are always enclosed in a <CSML> element *when stored in files.*

When used in other contexts, such as web services, any of the elements, other than <Definitions>, <TagDefinitions>, and <Includes> that are defined as allowed children of <CSML> can be used as the top level element. In these cases, the XML namespace specifier and optional 'defaultLocale' attribute defined for the CSML element shall be placed on the top level element.

For example, when used in a file context, the elements are wrapped in a <CSML> element:

```
<?xml version='1.0' encoding='utf-8'?>
<CSML xmlns="http://bacnet.org/csml/1.2" defaultLocale="en-GB" >
   <Definitions>... </Definitions>
   <List name="a-list" ...>...</List>
</CSML>
```

For other contexts, any data element is allowed to be the top level element:

```
<?xml version='1.0' encoding='utf-8'?>
<List name="a-list" xmlns="http://bacnet.org/csml/1.2" defaultLocale="en-GB" ...>...</List>
```

Q.2.1 <CSML>

When used in a file context, the ~~The~~ XML syntax defined by this annex is enclosed in the element <CSML> ("Control Systems Modeling Language") that has ~~a single optional attribute, 'defaultLocale', and~~ an xml namespace of ~~"http://www.bacnet.org/CSML/1.0"~~ *"http://bacnet.org/csml/1.2"*.

The valid child elements of <CSML> are any number and combination of the <Definitions>*, <TagDefinitions>, and Includes> elements,* ~~element~~ and the data elements <Any>, <Array>, <BitString>, <Boolean>, <Choice>, <*Collection*>, <*Composition*>, <Date>, <DatePattern>, <DateTime>, <DateTimePattern>, <Double>, <Enumerated>, <Integer>, <List>, <Null>, <Object>, <ObjectIdentifier>, <ObjectIdentifierPattern>, <OctetString>, <*Raw*>, <Real>, <Sequence>, <SequenceOf>, <String>, <*StringSet*>, <Time>, <TimePattern>, <*Unknown*> <Unsigned>, and <WeekNDay>, all described elsewhere in this annex.

The <CSML> element is structurally equivalent to a <Collection>. Therefore the attributes "published", "author", "description", "dataRev", etc., can be used to provide helpful information to consumers.

...

Q.2.1.1 'defaultLocale'

This optional attribute of ~~<CSML>~~*the top level element*, of type xs:language, specifies the locale (RFC 3066, language plus optional country tag) that becomes the "default locale" for the enclosed elements. All human language data in the enclosed elements is for the default locale unless otherwise indicated.

If this attribute is not present, then the default locale for the ~~document~~ *enclosed data* remains unspecified. In this case, the interpretation of any human language content is a local matter, and the use of the 'locale' attribute to specify alternate locales is not permitted elsewhere in the document.

Q.2.1.2 <Definitions>

This optional child element ~~of <CSML>~~ provides the "definition context" as used in this annex *and referenced by the data model in Annex Y*. There may be multiple <Definitions> elements under a <CSML> element and these definitions may appear in any position and in any order~~, with the only restriction that data types shall be defined before they are used~~.

...

The valid child elements of <Definitions> are any number and combination of the data elements <Any>, <Array>, <BitString>, <Boolean>, <Choice>, <*Composition*>, <*Collection*>, <Date>, <DatePattern>, <DateTime>, <DateTimePattern>, <Double>, <Enumerated>, <Integer>, <List>, <Null>, <Object>, <ObjectIdentifier>,

<ObjectIdentifierPattern>, <OctetString>, <*Raw*>, <Real>, <Sequence>, <SequenceOf>, <String>, <*StringSet*>, <Time>, <TimePattern>, <Unknown>, <Unsigned>, and <WeekNDay>, all described elsewhere in this annex.
...

[Insert new **Clauses Q.2.1.3 and Q.2.1.4**, p. 966]

Q.2.1.3 <TagDefinitions>

This optional child element of <CSML> provides the definitions of one or more "tags" that provide semantic meaning to other data. See Clause Y.1.4. There can be multiple <TagDefinitions> elements under a <CSML> element and these definitions can appear in any position.

The valid child elements of <TagDefinitions> are any number and combination of the data elements <BitString>, <Boolean>, <Date>, <DatePattern>, <DateTime>, <DateTimePattern>, <Double>, <Enumerated>, <Integer>, <Null>, <ObjectIdentifier>, <ObjectIdentifierPattern>, <OctetString>, <Real>, <String>, <Time>, <TimePattern>, <Unsigned>, and <WeekNDay>, all described elsewhere in this annex.

The <Null> element shall be used to define a tag that simply indicates semantic meaning by its presence and has no value. See Clause Y.1.4.

For example, an Example Organization publishes descriptive definitions for a semantic tag named "foo", and a value tag named "baz" (with organizational prefixes for uniqueness).

```
<CSML ...>
  <TagDefinitions>
    <Null name="org.example.tags.foo" displayName="Foo" description="... " >
      <DisplayName locale="tlh">Füu</DisplayName>
      <Description locale="tlh">...</Description>
    </Null>
    <String name="org.example.tags.baz" displayName="Baz"  description="... " >
      <DisplayName locale="tlh">Bæž</DisplayName>
      <Description locale="tlh">...</Description>
    </String>
  <TagDefinitions>
</CSML>
```

Q.2.1.4 <Includes>

This optional child element of <CSML> provides one or more "include directives" instructing the consumer to include all of the information contained in other CSML file(s). The valid child elements of <Includes> are <Link> elements. There can be multiple <Includes> elements under a <CSML> element and these may appear in any position

Q.2.1.4.1 <Link>

This optional child element of <Includes> provides a single "include directive" instructing the consumer to include all of the information contained in another CSML file.

Q.2.1.4.1.1 'value'

This required attribute of the <Link> element, of type xs:anyURI, identifies the location of another CSML file.

If a URI scheme is given, it is restricted to being "http", "https", or "bacnet".

If a URI scheme is not given, then the string shall be processed as an absolute or relative path according to the following rules:

If the referring file is contained in a zip file, then the value shall be evaluated as a path within that zip container, relative to the referring file's base path. For example, for a zip file containing files /base.xml, /other.xml, /foo/bar.xml, and /foo/baz.xml, the following are valid references:

```
in /base.xml:
   <Includes>
      <Link value="/other.xml" />          ( OR "other.xml" )
      <Link value="/foo/bar.xml".../>      ( OR "foo/bar.xml" )
   </Includes>

in /foo/bar.xml:
   <Includes>
      <Link value="/base.xml"/>            (OR "../base.xml" )
      <Link value="/foo/baz.xml".../>      ( OR "../foo/baz.xml" OR "baz.xml" )
   </Includes>
```

Otherwise, the absolute or relative path is processed with respect to the base URI of the referring file according to RFC 3986.

There is no relative path format defined for the "bacnet" URI scheme; however, the use of the reserved path segment ".this", meaning "this device", can be used to create references to other BACnet files in the same device. See Clause Q.Y.

[Replace **Clause Q.3** entirely, p. 966]

Q.3 Expressing Data

The data model in Annex Y defines data in terms of "data" and "metadata", which are made out of base types. The base types Any, Array, BitString, Boolean, Choice, Collection, Composition, Date, DatePattern, DateTime, DateTimePatern, Double, Enumerated, Integer, List, Logical, Null, Object, ObjectIdentifier, ObjectIdentifierPattern, OctetString, Raw, Real, Sequence, SequenceOf, String, StringSet, Time, TimePattern, Unknown, Unsigned, and WeekNDay are expressed in XML using the correspondingly named data elements: <Any>, <Array>, <BitString>, etc. The set of allowed data model metadata and children is defined by the data model. Data model children are expressed as direct child elements in the XML, as peers to some of the XML child elements as described by Clause Q.4.

The 'name' attribute of the children of <Array>, <List>, and <SequenceOf> base types is optional, but may be required by some contexts, such as the web services defined in Annex W.

[Replace **Clause Q.4** entirely, p. 1000]

Q.4 Expressing Metadata

Data model metadata items are not limited to primitives as XML attributes are. Therefore, these clauses define the mapping from data model metadata to XML attributes and/or child elements.

Q.4.1 Primitive Metadata

Primitive metadata is expressed directly as XML attributes with the same name as the metadata. The XML attribute type corresponds to the base type of the data model metadata. Most of these are fixed. For example, the base type of the data model 'displayName' metadata is String, therefore its XML 'displayName' attribute type is xs:string. However, some metadata like 'maximum' vary their base type based on the containing base type, therefore their XML type can be xs:float, xs:double, xs:nonNegativeInteger, etc. based on the containing element. The rules for this variability are defined by the data model.

Q.4.2 Localizable Metadata

In the data model, a localizable text metadata value is modeled as a collection of strings, with one member for each locale that has been assigned a value. The members of these value collections are represented in XML as repeatable child elements according to the following table. If the locale of the member is different from the default locale, then the locale is represented as a 'locale' XML attribute of the child element. The body text of these elements is of type xs:string.

Metadata Name	XML Child Element	Used For...	Child 'locale' attribute
'displayName'	<DisplayName>	non-default locales	Required
'displayNameForWriting'	<DisplayNameForWriting>	non-default locales	Required
'description'	<Description>	non-default locales	Required
'documentation'	<Documentation>	all locales	Optional
'comment'	<Comment>	non-default locales	Required
'writableWhenText'	<WritableWhenText>	all locales	Optional
'requiredWhenText'	<RequiredWhenText>	all locales	Optional
'unitsText'	<UnitsText>	all locales	Optional
'errorText'	<ErrorText>	all locales	Optional

For example, these data items have localized values for some of their metadata.

```
<String name="n1234" displayName="Something" description="Some Thing" comment="it's great" value="Hi" >
   <DisplayName locale="de-DE">Etwas</DisplayName>
   <Description locale="de-DE">Eine Sache</Description>
   <Comment locale="de-DE">Es ist toll</Comment>
  <Documentation>Documentation is always in element form</Documentation>
  <Documentation locale="de-DE">Die Dokumentation ist immer in elementarer Form</Documentation>
</String>
<Real name="abc" units="degrees-Celsius" writableWhen="other" requiredWhen="other" >
   <UnitsText locale="en">ºC</UnitsText>
   <UnitsText locale="de">Grad Celsius</UnitsText>
   <WritableWhenText>The unit is held upside down</WritableWhenText>
   <WritableWhenText locale="de-DE"<Das Gerät wird kopfstehed gehalten</WritableWhenText>
   <RequiredWhenText>In hostile territory</RequiredWhenText>
   <RequiredWhenText locale="de-DE>In feindlichem Gebiet</RequiredWhenText>
</Real>
```

Q.4.3 Container Metadata

Some data model metadata are containers for other data. Some of these represent their members under a single container XML element and some use repeated elements, according to the following tables.

Metadata Name	XML Child Element Container	XML Container Members
'namedValues'	<NamedValues>	multiple, any type
'namedBits'	<NamedBits>	multiple, <Bit>
'choices'	<Choices>	multiple, any type
'memberTypeDefinition'	<MemberTypeDefinition>	single, any type
'links'	<Links>	multiple, <Link>
'priorityArray'	<PriorityArray>	<Choice>
'relinquishDefault'	<RelinquishDefault>	single, any type
'failures'	<Failures>	<Link>
'valueTags'	<ValueTags>	multiple, any type
'revisions'	<Revisions>	multiple, any type

[Replace **Clause Q.5** entirely, p. 1000]

Q.5 Expressing Values

The primitive data elements <BitString>, <Boolean>, <Date>, <DatePattern>, <DateTime>, <DateTimePattern>, <Double>, <Enumerated>, <Integer>, <Link>, <ObjectIdentifier>, <ObjectIdentifierPattern>, <Real>, <String>, <StringSet>, <Time>, <TimePattern>, <Unsigned>, and <WeekNDay> all can specify their values in attribute form as described in the following table.

XML Data Element	XML 'value' Attribute Type
<Boolean>	xs:boolean
<Unsigned>	xs:nonNegativeInteger
<Integer>	xs:integer
<Real>	xs:float
<Double>	xs:double
<OctetString>, <Raw>	xs:hexBinary
<Date>	xs:date
<DateTime>	xs:dateTime
<Time>	xs:time
<Link>	xs:anyURI
<String>, <StringSet>, <BitString>, <Enumerated>, <DatePattern>, <DateTimePattern>, <TimePattern>, <ObjectIdentifier>, <ObjectIdentifierPattern>, <WeekNDay>	xs:string

In addition to the XML attribute form of value, the data elements <OctetString>, <Raw>, <String>, and <BitString> can also specify their values in element form using a <Value> child element.

For <OctetString> and <Raw> elements, the value can be represented as either an XML 'value' attribute or as an XML <Value> child element. The XML 'value' attribute, of type xs:hexBinary, is in hexadecimal format, which is easier to process both for humans and machines, but is not as succinct as xs:base64Binary for large amounts of data. If a short amount of data is to be conveyed, the attribute form should be used. However, if a large amount of data is to be conveyed, the <Value> child element, of type xs:base64Binary, should be used. The threshold to select between the two methods is a local matter. Since the 'value' attribute and the <Value> child element are two ways to specify the same value, they are mutually exclusive in the same XML context, and when one is present, it overrides the other that may have been inherited.

For the <String> element, the XML 'value' attribute represents the data value in the default locale. Values in other locales, or values containing character data unsuitable for XML attributes, are represented using the optional child element <Value>. Since the 'value' attribute and a <Value> child element without a 'locale' attribute are two ways to specify the same value, they are mutually exclusive in the same XML context, and when one is present, it overrides the other that may have been inherited.

For the <BitString> element, the XML 'value' attribute represents the concatenation of all bits that are set (equal to true). If a short amount of data is to be conveyed, the attribute form should be used. However, if a large amount of data is to be conveyed, the optional <Value> child element should be used. This <Value> element is a container for individual <Bit> elements for the bits that are true. The threshold to select between the two methods is a local matter. Since the 'value' attribute and the <Value> child elements are two ways to specify the same value, they are mutually exclusive in the same XML context, and when one is present, it overrides the other that may have been inherited.

If a value is uninitialized, then a properly formed value shall be present of which the content is a local matter, and the 'error' metadata shall be set to WS_ERR_UNINITIALIZED_VALUE. Malformed or empty values shall not be used to indicate uninitialized values. Note that "uninitialized" and "unspecified" are not the same thing. The term "uninitialized value" means that the server or client is not in possession of any kind of value for a data item. The

term "unspecified value" applies only to Date, Time, and DateTime base types and means that the value is indeed "initialized", but it is known to represent an "unspecified" time or date.

The xs:dateTime format shall include the time zone designator and no more than 9 digits of fractional seconds.

Q.5.1 Localizable Values

In the data model, a localizable text value is modeled as a collection of strings, with one member for each locale that has been assigned a value. The members of these value collections are represented in XML as repeatable <Value> child elements. If the locale of the member is different from the default locale, then the locale is represented as a 'locale' XML attribute of the child element. The body text of these elements is of type xs:string.

For example, this data item has localized values.

```
<String name="n1234" value="Good Day">
  <Value locale="de-DE">Guten Tag</Value>
  <Value locale="fr">Bonjour</Value>
</String>
```

[Change **Clause Q.7**, p. 1007]

Q.7 Extensibility

Both the XML syntax and the data it represents can be extended.

Q.7.1 XML ~~extensions~~ *Extensions*

Documents conforming to this standard can be extended through the use of XML attributes and elements from other XML namespaces. XML attributes from other namespaces are allowed on any standard element, and elements from other namespaces are allowed under any standard element that already has child elements defined for it in this standard. ~~With the exception of the <Documentation> element, this~~ *This* standard does not use mixed content, so any element ~~other than <Documentation>~~ that uses body text may not be extended with elements from other namespaces.

Because these cannot be represented in the data model, it is recommended that their usage be limited. There is no requirement that consumers of this XML understand or process any of these extensions. Consumers are allowed to consume extensions that are known to the consumer and to ignore the rest.

Q.7.2 Data Model Extensions

~~Extensions to the data represented by this standard XML syntax is accomplished with the <Extensions> element defined in Clause Q.3.2.6. Normally, these extensions represent data that is beyond what is accessible through standard BACnet binary services but which may be of interest to the consumer of the XML, or they may represent extended data that is accessible through BACnet Web services or by other means.~~

~~Standard elements and attributes can both be extended with proprietary attributes.~~ *Proprietary metadata, as well as extended metadata information (such as metadata for metadata) that cannot be represented as XML attributes or child elements according to the rules in the preceding clauses, are represented under an <Extensions> child element. Extended data is not restricted in type or depth.*

Each extended metadata item is represented as a child element of the <Extensions> element using an XML element corresponding to the base type of the metadata item. The 'name' attribute of the XML child corresponds to the name of the metadata item. The names of proprietary *metadata* ~~attributes~~ shall begin with ~~a period character (".")~~ *prefix* to prevent conflict with standard attribute names. ~~While not required, it is recommended that proprietary attributes also use a vendor-specific prefix, following the required period character, to prevent conflicts among proprietary attributes.~~ *This prefix shall be one of:*

1) *A reversed registered DNS name, followed by a period character. e.g., "com.example.", or*
2) *A BACnet vendor identifier in decimal, followed by a dash character. e.g., "555-", or*

3) *A period character ".". While not required, it is recommended that proprietary attributes beginning with a period character also use a vendor-specific prefix, following the required period character, to prevent conflicts among proprietary attributes. Note that option 3 is retained for historical compatibility; new implementations are required to use option 1 or 2.*

~~Standard~~ *For example, standard metadata*, like ~~displayName~~ *'displayName'*, can be extended with standard ~~attributes~~ *metadata* that are appropriate to their ~~datatypes~~ *base type, like 'maximumLength'*.

While this clause provides the syntax and method for extending the standard ~~attributes~~ *metadata*, it makes no requirement that consumers of this XML understand or process any of these extensions. *Consumers are allowed to consume extensions that are known to the consumer and to ignore the rest.* ~~When extending standard attributes, the names used for the extensions use the naming convention of the corresponding attributes in the Annex N data model and are shown in the following table. The table also indicates an "effective type" which defines the standard element type that shall be used as the child of <Extensions> when extending the standard attribute.~~

[Remove **Table Q-7** from this Clause, p. 1008]

The following example shows the standard ~~attribute~~ *metadata*, 'maximumLength', being extended with the standard ~~attributes~~ *metadata* 'writable' and 'writeEffective', and the standard element <String> being extended with a proprietary attribute ~~".999-WritePrivilegeLevel"~~ *"555-UIGroup"*.

```
<Definitions>
    <Object name="555-ExampleObject">
        <String name="write-me" writable="true" maximumLength="50" >
            <Extensions>
                <Unsigned name="~~Maximum~~maximumLength" writable="true" writeEffective="on-device-restart" />
                <Integer name="~~.999-WritePrivilegeLevel~~" "555-UIGroup" value="6" />
            </Extensions>
        </Real>
    </Object >
</Definitions>
```

[Insert new **Clause Q.8**, p. 1009]

Q.8 BACnet URI Scheme

In cases where a URI is needed to refer to data that is accessible using BACnet services, the following format shall be used, where angle brackets indicate variable text and square brackets indicate optionality:

bacnet://<device>/<object>[/<property>[/<index>]]

The <device> segment is the device instance number in decimal. A <device> identifier of ".this" means 'this device' so that it can be used in static files that do not need to be changed when the device identifier changes.

The <object> identifier is in the form "<type>,<instance>" where <type> is either a decimal number or exactly equal to the Clause 21 identifier text of BACnetObjectType, and <instance> is a decimal number.

The <property> identifier is either a decimal number or exactly equal to the Clause 21 identifier text of BACnetPropertyIdentifier. If it is omitted, it defaults to "present-value" except for BACnet File objects, where absence of <property> refers to the entire content of the file accessed with Stream Access.

The <index> is the decimal number for the index of an array property.

135-2012*am*-4. Add a JSON syntax for the abstract model.

Rationale

The RESTful web services allow the JavaScript Object Notation (JSON) to be used as an alternate syntax to XML. This section of the addendum creates a new annex, as a parallel to Annex Q, to provide the syntax rules for expressing the common data model in JSON. While likely used only for web services, it nonetheless has full parallels to the XML <CSML> and <Definitions> capabilities, for use in files.

[Add new **Annex Z**, p. 1026]

ANNEX Z - JSON DATA FORMATS (NORMATIVE)

(This annex is part of this standard and is required for its use.)

This annex defines formats for JSON data exchanged between various BAS systems. This data may have a variety of purposes and may be conveyed through files or by other means.

Z.1 Introduction

The Javascript Object Notation (JSON) is a format for structured text that can be used to represent a variety of data in a machine-readable form. The JSON syntax used in this standard conforms to RFC 4627.

This JSON syntax is based on the abstract data model in Annex Y. The abstract model is composed of "data" and "metadata". Data items are expressed as JSON Objects and metadata are expressed as JSON object members of a data item.

For representing BACnet data, the syntax allows data structure definitions, such as those in Clause 21, and instances of those definitions to be represented in JSON. The syntax is optimized for efficient representation of BACnet data and is sufficiently flexible not to be limited to modeling BACnet data exclusively.

The syntax also allows for human language descriptions, range restrictions, and usage information to be added to the basic data structure definitions.

Z.1.1 Design

This JSON syntax is designed to provide a syntax that can be used to represent both standard and proprietary data types along with any accompanying descriptive information. It is a general data definition and instance language rather than a syntax specific for the data defined in this standard. Because JSON is often used with interpreted languages like JavaScript, where member variables can be created on the fly, the full utility of a JSON syntax can best be realized by using JSON object member names, as opposed the anonymous JSON array members. Therefore, unlike the "datatype-centric" approach of the XML defined in Annex Q, the JSON syntax takes a "name-centric" approach., such as "present-value": { "$base":"String", "$value":75 } and "proprietary-value":{ "$base":"Boolean", "$value":true }. By including the "$base" type information, a "data-centric" view is still possible, allowing processors to work with unknown member names, knowing only its datatype.

For the purposes of brevity and human readability, this syntax represents metadata data as simply as possible rather than always as fully formed base types, in the same manner as Annex Q chooses to represent metadata as XML attributes as much as possible. However, this representation choice does not limit extensibility of the data model because, like the data model metadata described in Annex Y, the metadata representations defined in this annex can be extended to have metadata of their own using an extension syntax defined in Clause Z.6.2.. This allows JSON brevity for the common cases without limiting extensibility when needed.

Z.1.2 Syntax Examples

Some examples using the Clause 21 datatypes will provide an introduction to the form and capabilities of the syntax. The full details of the JSON objects and members is defined is subsequence clauses. The description of the data model and the system for defining and extending data types and expressing instances of those types is described in Annex Y. In this syntax, names of a data items are used as the JSON names; names of metadata items are prefixed with "$" and names for things that are not part of the Annex Y common data model are prefixed with "$$".

Enumerations in Clause 21 are defined as a mapping between an unsigned value and a textual identifier.

BACnetFileAccessMethod ::= ENUMERATED {
    record-access (0),
    stream-access (1)
    }

The definition of that same enumeration in JSON creates the mapping with a series of named unsigned values.

```
{
  "$$definitions": {
    "$base":"Collection",
    "BACnetFileAccessMethod":{
      "$base": "Enumerated",
      "$namedValues": {
        "record-access":{ "$base":"Unsigned", "value":0 },
        "stream-access":{ "$base":"Unsigned", "value":1 }
      }
    }
  }
}
```

A JSON representation of a value of that enumeration uses the textual identifier rather than the number when the type is known.

```
    "myvalue": {  "$base":"Enumerated", "$value":"record-access" }
```

Some enumerations in BACnet are extensible, however, and in those cases, and in cases where the type is not known, a number is used in place of the textual identifier. This is discussed in more detail in Clause Y.12.12.

Bit Strings in Clause 21 are similarly defined as a mapping between a bit position and a textual identifier.

BACnetEventTransitionBits ::= BIT STRING {
    to-offnormal (0),
    to-fault (1),
    to-normal (2)
    }

The definition of that bit string in JSON also defines the mapping with a series of named unsigned values, where the value specifies the bit position.

```
{
  "$$definitions": {
    "$base":"Collection",
    "BACnetEventTransitionBits":{
      "$base": "BitString",
      "$length": 3,
      "$namedBits": {
        "to-offnormal":{ "$base":"Bit", "$bit":0 },
        "to-fault": { "$base":"Bit", "$bit":1 },
        "to-normal": { "$base":"Bit", "$bit":2 }
      }
    }
  }
}
```

A JSON representation of a value of that bit string is a list of the textual identifiers for the bits that are set.

"myvalue":{ "$base":"BitString", "$value":"to-offnormal;to-normal" }

Similar to the extensible enumeration case, if a textual representation of a bit is not known, then a number indicating its bit-position is used instead.

Constructed data definitions in Clause 21 define a set of named members and can also specify context tags, optionality, and comments about the data members.

BACnetPropertyReference ::= SEQUENCE {
    property-identifier   [0] BACnetPropertyIdentifier,
    property-array-index   [1] Unsigned OPTIONAL   --used only with array datatype
                                                        -- if omitted with an array the entire array is referenced
    }

In JSON, this sequence also specifies names, context tags, optionality, and can even capture comments:

```
{
  "$$definitions": {
    "$base":"Collection",
    "BACnetPropertyReference": {
      "$base":"Sequence",
      "$$order":"property-identifier;property-array-index",
      "property-identifier": {
        "$base":"Enumerated",
        "$contextTag":0,
        "$type":"BACnetPropertyIdentifier"
      },
      "property-array-index": {
        "$base":"Unsigned",
        "$contextTag":1,
        "$optional":true,
        "$comment":"Used only with array datatype. If omitted, the entire array is referenced."
      }
    }
  }
}
```

A JSON representation of a value of that sequence may provide values for each member that is present, using a representation appropriate to that member's type, and omit optional members that are not present.

```
"myvalue":{
   "$base":"Sequence",
   "property-identifier": { "$base":Enumerated",  "$value":"present-value" }
}
```

Choices in Clause 21 are defined as a choice of named members with specified types.

```
BACnetClientCOV ::= CHOICE {
          real-increment              REAL,
          default-increment           NULL
          }
```

In JSON, this choice is also defined as a choice of named members with specified types.

```
{
   "$$definitions": {
      "$base":"Collection",
      "BACnetClientCOV": {
         "$base":"Choice",
         "$choices": {
            "real-increment":{ "$base":"Real" },
            "default-increment"{ "$base:"Null" }
        }
      }
   }
}
```

A JSON representation of a value of that choice indicates which member is chosen and gives a value for it.

```
"myvalue": { "$base":"Choice", "real-increment":  { "$value":75.0 } }
```

Variable length collections of identical members are defined in Clause 21 using the SEQUENCE OF construct.

```
BACnetDailySchedule ::= SEQUENCE {
          day-schedule       [0] SEQUENCE OF BACnetTimeValue
          }
```

In JSON, this collection is represented by the SequenceOf element which takes a 'memberType' metadata.

```
{
   "$$definitions": {
     "$base":"Collection",
      "BACnetDailySchedule": {
         "$base":"Sequence",
         "day-schedule": {
            "$base":"SequenceOf",
            "$contextTag":0,
            "$memberType":"BACnetTimeValue"
         }
      }
   }
}
```

A JSON representation of a value of that SEQUENCE OF provides a collection of unnamed (numbered) members of the appropriate type.

```
"myvalue":{
   "$base":"Sequence",
   "day-schedule": {
      "$base":"SequenceOf",
      "1":  {
          "$base":"Sequence",
          "time":{ "$base":"Time", "$value":"08:00:00.00" },
          "value":{ "$base":"Unsigned", "$value":1 }
      },
      "2": {
         "$base":"Sequence",
         "time": { "$base":"Time", "$value":"17:00:00.00" },
         "value"{ "$base":"Unsigned", "$value":0 }
      }
   }
}
```

Z.2 JSON Document Structure

In JSON, the top level object is anonymous. If required by context, e.g., in the body of a POST where a new name is being proposed, an explicit "$name" can be included to provide a name for the top level data item. This top level object can have an optional member named "$$defaultLocale" (see Clause Z.2.1).

When used in a file context, the top level object shall be an Annex Y data item of base type Collection. This top level object is referred to as the "CSML object" (Control Systems Modeling Language). This object can have an optional "$$definitions" member (Clause Z.2.2), an optional "$$tagDefinitions" member (Clause Z.2.3), an optional "$$includes" member (Clause Z.2.4), and any number and combination of members allowed by the Collection base type. Since this top level object in a file is a Collection, the metadata "published", "author", "description", "dataRev", etc., can be used to provide helpful information to consumers.

For example:

```
{
   "$base":"Collection",
   "$$defaultLocale":"en-GB",
   "$$includes": {
      "$base":"List",
      "1":{ "$base":"Link", $value":"http://example.com/csml/defs/common.json" }
   }
   "$$definitions": {
      "$base":"Collection",
      "some-type-name":{ "$base":"Sequence",  ... ... a definition ... },
      "some-other-type-name":{ "$base":"Choice",  ... another  definition ... }
   },
   "some-data":{ "$base":"Real",  ... },
   "some-more-data": { "$base":"Sequence", ...}
}
```

When used in other contexts, such as web services, any of the data items that are allowed members of a CSML object can be used as the top level object.

Z.2.1 "$$defaultLocale"

This optional string member of the top level object specifies the locale (RFC 3066, language plus optional country tag) that becomes the "default locale" for the enclosed data. All human language data in the enclosed data is for the default locale unless otherwise indicated.

If this member is not present, then the default locale for the enclosed data is unspecified. In this case, the interpretation of any human language content is a local matter, and the use of the 'locale' metadata to specify alternate locales is not permitted elsewhere in the document.

Z.2.2 "$$definitions"

This optional member, of type "Collection", provides the "definition context" as referenced by the data model in Annex Y. Unlike Annex Q, which can have multiple <Definitions> XML elements, thereby creating multiple definition contexts, JSON documents are limited to a single definition context at a single level. The allowed members of the "$$definitions" Collection are any number and combination of members with base types of Any, Array, BitString, Boolean, Choice, Collection, Composition, Date, DatePattern, DateTime, DateTimePattern, Double, Enumerated, Integer, List, Collection, Null, Object, ObjectIdentifier, ObjectIdentifierPattern, OctetString, Raw, Real, Sequence, SequenceOf, String, StringSet, Time, TimePattern, Unknown, Unsigned, or WeekNDay.

The 'base' metadata shall be specified for all data in a definition context unless it can be derived from the presence of 'type', 'extends', or 'overlays' metadata.

For example, this file contains a mixture of definitions and data with a default locale:

```
{
  "$$defaultLocale":"en-GB",
  "$$definitions": {
      "some-type-name":{"$base":"Enumeration" ... a definition ... },
      "some-other-type-name":{ "$base":"Choice"... another  definition ... }
  },
  "some-data":{ "$base":"Real",  ... },
  "some-more-data": { "$base":"Sequence", ...}
}
```

Z.2.3 "$$tagDefinitions"

This optional member of a CSML object, of type "Collection", provides the tag definitions, as referenced by the data model in Annex Y. Unlike Annex Q, which can have multiple <TagDefinitions> XML elements, thereby creating multiple tag definition sections, JSON documents are limited to a single tag definition section. The allowed members of the "$$tagDefinitions" Collection are any number and combination of members with base types of BitString, Boolean, Date, DatePattern, DateTime, DateTimePattern, Double, Enumerated, Integer, Null, ObjectIdentifier, ObjectIdentifierPattern, OctetString, Real, String, StringSet, Time, TimePattern, Unsigned, or WeekNDay.

The Null base type shall be used to define a semantic tag that simply indicates its meaning by its presence and has no value.

For example, an Example Organization publishes descriptive definitions for a semantic tag named "foo", and a value tag named "baz" (with organizational prefixes for uniqueness), and provides metadata for human understanding.

```
{
  "$$defaultLocale":"en-GB",
  "$$tagDefinitions": {
     "$base":"Collection"
     "org.example.tags.foo":{
        "$base":"Null",
        "$displayName":"Foo",
        "$displayName$$tlh":"Füu",
        "$description":"... "
     },
     "org.example.tags.baz":{
        "$base":"Choice",
        "$displayName":"Baz",
        "$displayName$$tlh":"Bæž",
        "$description":"... "
     }
  }
}
```

Z.2.4 "$$includes"

This optional member of a CSML object, of base type "List of Link", provides a list of directives to include other CSML documents. The value restrictions for the Links follow the same rules defined in Clause Q.2.1.4 for XML includes. Consumers of JSON documents are only required to include other JSON documents.

The value of the Link identifies the location of another CSML file with the following restrictions:

If a URI scheme is given, it is restricted to being "http", "https", or "bacnet".

If a URI scheme is not given, then the string shall be processed as an absolute or relative path according to the following rules:

If the referring file is contained in a zip file, then the Link's value shall be evaluated as a path within that zip container, relative to the referring file's base path. For example, for a zip file containing files /base.json, /other.json, /foo/bar.json, and /foo/baz.json, the following are valid references:

in /base.json:

```
  "$$includes": {
     "$base":"List"
      "1":{"$base":"Link", "$value":"/other.json" ...}    ( OR "other.json" )
      "2":{ "$base":"Link" "$value":"/foo/bar.json" ...} ( OR "foo/bar.json" )
  }
```

in /foo/bar.json:

```
  "$$includes": {
     "$base":"List"
      "1":{"$base":"Link", "$value":"/other.json" ...}      ( OR "../other.json" )
      "2":{"$base":"Link", "$value":"/foo/baz.json" ...}  ( OR "../foo/baz.json" OR "baz.json" )
  }
```

Otherwise, the absolute or relative path is processed with respect to base URI of the referring file according to RFC 3986.

There is no relative path format defined for the "bacnet" scheme; however, the use of the reserved path segment ".this", meaning "this device", can be used to create references to other BACnet files in the same device. See Clause Q.8.

For example, this file includes two other files:

```
{
   "$$includes": {
      "$base":"List"
       "1":{ "$base":"Link"  "$value":"http://example.com/csml/defs/common.json" mediaType="application/json"},
       "2":{ "$base":"Link" "$value":"a/relative/path/to/blah.json" mediaType="application/json"}
   },
   "$$definitions": {  ... },
   ...
}
```

Z.3 Expressing Data

The data model in Annex Y defines data in terms of "data" and "metadata". Data items are expressed in JSON using a JSON object with the "$base" member to identify the base type of the data, e.g., {"$base":"Array"...}, {"$base":"BitString"...}, etc. The set of allowed metadata and children for the various base types is defined by the data model.

With the exception of the outermost anonymous JSON object, all other data and metadata have JSON member names. These names are equal to the data or metadata item's name. In the case of base types Array, List, and SequenceOf, the position of the member, in decimal, is used for the member name, starting with "1".

Z.3.1 Order

The Array, Sequence, and SequenceOf base types are defined to be ordered collections, however, the members of a JSON object are unordered. The names for Array and SequenceOf members are numeric, so the order is implied, but the names for Sequence members have no such restrictions. Therefore, producers of JSON serializations for the Sequence base type shall add an order indicator to each member of the Sequence when required. This indicator is required for definitions and for instances that do not have a derivable definition. For instances that have an explicit 'type' or are contained in a data structure for which a definition is known, the order indicator would be redundant information and can be omitted.

Z.3.2 $$order

This optional string member of any JSON object specifies the order of children in that object. The value is a semicolon-separated list of children names.

Z.4 Expressing Metadata

Metadata items are not limited to primitives. Therefore, these clauses define the mapping from data model metadata to JSON object member types.

All standard metadata names are prefixed with a dollar sign character ("$") to distinguish them from data children. All nonstandard metadata shall be represented under the $extensions member. See Clause Z.6.2.

Z.4.1 Primitive Metadata

Primitive data model metadata are expressed directly as object members with the same name as the data model metadata item. The object member type corresponds to the base type of the data model metadata. Most of these are fixed. For example, the base type of the data model 'displayName' metadata is String, therefore its JSON "displayName" object member type is a JSON string. However, some metadata like 'maximum' vary their base type based on the containing base type, therefore their JSON type can be floating point, integer, etc. based on the containing element. The rules for this variability are defined by the data model.

Z.4.2 Localizable Metadata

In the data model, a localizable text metadata value is modeled as a collection of strings, with one member for each locale that has been assigned a value. The members of these value collections are represented in JSON as individual strings with the same name as the data model metadata. If the locale of the member is different from the default locale, then the JSON name is postfixed with a double dollar character sequence ("$$") and the corresponding locale. This includes:

'displayName', 'displayNameForWriting', 'description', 'documentation', 'comment', 'writableWhenText', 'requiredWhenText', 'unitsText', 'errorText'

For example, these data items have localized values for some of their data and metadata; including the String's value itself.

```
{
   "$base":"String",
   "$displayName":"Something",
   "$displayName$$de-DE":"Etwas",
   "$description":"Some Thing",
   "$description$$de-DE":"Eine Sache",
   "$comment":"it's great",
   "$comment$$de-DE":"Es ist toll",
   "$documentation":"This is some helpful documentation",
   "$documentation$$de-DE":"Dies ist eine hilfreiche Dokumentation",
   "$value":"Good Day",
   "$value$$de-DE":"Guten Tag"
}

{
   "$base":"Real",
   "$error":0,
   "$errorText": "The device is not feeling well today",
   "$errorText$$de-DE": "Das Gerät fühlt sich heute nicht gut",
   "$units":"degrees-celsius",
   "$unitsText":"ºC",
   "$unitsText$$de-DE":"Grad Celsius",
   "$writableWhen":"other",
   "$writableWhenText":"The unit is held upside down",
   "$writableWhenText$$de-DE":"Das Gerät ist kopfstehend gehalten",
   "$requiredWhen":"other",
   "$requiredWhenText":"In hostile territory",
   "$requiredWhenText$$de-DE":"In feindlichem Gebiet"
}
```

Z.4.3 Container Metadata

Some data model metadata are containers for other data. All of these simply represent their members as appropriate base types under a JSON object named the same as the data model metadata. This applies to:

'namedValues', 'namedBits', 'choices', 'memberTypeDefinition', 'links'

Z.5 Expressing Values

The primitive base types BitString, Boolean, Date, DatePattern, DateTime, DateTimePattern, Double, Enumerated, Integer, Link, ObjectIdentifier, ObjectIdentifierPattern, Real, String, Time, TimePattern, Unsigned, and WeekNDay all specify their values as a JSON object member named "$value" with a type described in the following table (See Annex Q for reference for the "xs:" textual formats).

Table Z-1. JSON Type and Format for Base Types

Base Type	JSON "$value" Type	Textual Format
Boolean	boolean	xs:boolean
Double	number	xs:double
Real	number	xs:float
Unsigned	number	xs:nonNegativeInteger
Integer	number	xs:integer
OctetString, Raw	string	xs:hexBinary
Date	string	xs:date
DateTime	string	xs:dateTime
Time	string	xs:time
String, Link	string	xs:string
BitString, StringSet, Enumerated, DatePattern, DateTimePattern, TimePattern, ObjectIdentifier, ObjectIdentifierPattern, WeekNDay	string	defined by data model

If a value is uninitialized, then a properly formed value shall be present, the contents of which is a local matter, and the 'error' metadata shall be set to WS_ERR_UNINITIALIZED_VALUE. Malformed or empty values shall not be used to indicate uninitialized values. Note that "uninitialized" and "unspecified" are not the same thing. The term "uninitialized value" means that the server or client is not in possession of any kind of value for a primitive data item. The term "unspecified value" applies only to Date, Time, and DateTime base types and means that the value is indeed "initialized", but it is known to represent an "unspecified" time or date.

The xs:dateTime format shall include the time zone designator and no more than 9 digits of fractional seconds.

String values containing multiple lines shall use the escape sequence "\n" as the line separator.

Z.5.1 Localizable Value

In the data model, a localizable text value is modeled as a collection of strings, with one member for each locale that has been assigned a value. The members of these value collections are represented in JSON as individual strings, with the default locale in the member "$value". If the locale of the member is different from the default locale, then the JSON name "$value" is postfixed with a double dollar character sequence ("$$") and the corresponding locale.

For example, these data items have localized values:

```
{
   "$base":"String",
   "$value":"Hello",
   "$value$$de-DE":"Guten Tag"
}
```

```
{
   "$base":"Real",
   "$error":0,
   "$errorText": "The device is not feeling well today",
   "$errorText$$de-DE": "Das Gerät fühlt sich heute nicht gut",
   "$units":"degrees-Celsius",
   "$unitsText":"ºC",
   "$unitsText$$de-DE":"Grad Celsius",
   "$writableWhen":"other",
   "$writableWhenText":"The unit is held upside down",
   "$writableWhenText$$de-DE":"Das Gerät ist kopfstehend gehalten",
   "$requiredWhen":"other",
   "$requiredWhenText":"In hostile territory",
   "$requiredWhenText$$de-DE":"In feindlichem Gebiet"
}
```


Z.6 Extensibility

Both the JSON syntax and the data it represents can be extended.

Z.6.1 JSON Extensions

Documents conforming to this standard can be extended through the use of JSON object members names other than those defined by this standard. These extensions shall be prefixed with a double dollar sequence ("$$"). Proprietary extensions shall also use a vendor-specific prefix, following the required "$$", to prevent conflicts among proprietary extensions. This vendor-specific prefix shall be either:

1) A reversed registered DNS name, followed by a period character. e.g., "com.example.", or
2) A BACnet vendor identifier in decimal, followed by a dash character. e.g., "555-"

For example, "setpoint": { "$base":"Real", "$value":75.0, "$$555-extraJSONThing":"xzy" }

Because these extensions are not considered data or metadata and cannot be represented in the data model, it is recommended that their usage be limited. There is no requirement that consumers of this JSON understand or process any of these extensions. Consumers are allowed to consume extensions that are known to the consumer and to ignore the rest.

Z.6.2 Data Model Extensions

Standard primitive metadata items, like 'displayName', normally represent their value as a JSON primitive, like "$displayName":"abc". However, they can be also extended with metadata that are appropriate to their base type, like 'maximumLength'. To represent such metadata-of-metadata, the JSON object form shall be used, like "$displayName":{"$value":"abc","$maximumLength":50}.

Proprietary metadata shall be represented under a "$extensions" object member. Such data is not restricted in type or depth. Each extended metadata item shall be represented as a member of the "$extensions" object member using a JSON object corresponding to the base type of the metadata in the data model. The names of proprietary metadata shall begin with a prefix to prevent conflict with standard attribute names. This prefix shall be one of:

1) A reversed registered DNS name, followed by a period character. e.g., "com.example."
2) A BACnet vendor identifier in decimal, followed by a dash character. e.g., "555-"
3) A dot (period) character ".". While not required, it is recommended that proprietary attributes beginning with a period character also use a vendor-specific prefix, following the required period character, to prevent conflicts among proprietary attributes. Option 3 is retained for historical compatibility; new implementations are required to use option 1 or 2.

While this clause provides the syntax and method for extending the standard metadata, it makes no requirement that consumers of this JSON understand or process any of these extensions. Consumers are allowed to consume extensions that are known to the consumer and to ignore the rest.

The following example shows the standard metadata, 'maximumLength', being extended with the standard metadata 'writable' and 'writeEffective', and a standard String being extended with a proprietary extension "555-UIGroup".

```
{ "$$definitions": {
     "555-ExampleObject": {
       "$base":"Object",
       "write-me":{
          "$base":"String",
          "$writable":true,
          "$maximumLength":{"$value":50,"$writable":true,"$writeEffective":"on-device-restart"},
          "$extensions":{
             "555-UIGroup": { "$base":"Integer", "$value":6 }
          }
       }
    }
  }
}
```

135-2012*am*-5. Deprecate Annex N SOAP services and add a migration guide.

Rationale

The data model in Annex Q (XML) and the 135-2012 Annex N (SOAP services) are similar but not exactly the same. Section 2 of this addendum extracted and abstracted the data model from Annex Q in to a common model described in new Annex Y. This section deprecates the SOAP services and provides a guide for migrating clients to the new REST based services defined in new Annex W, and the new data model.

[Change **Annex N**, p. 915]

ANNEX N - *FORMER* BACnet/WS WEB SERVICES INTERFACE (~~NORMATIVE~~ *INFORMATIVE*)

~~(This annex is part of this standard and is required for its use.)~~ *(This annex is not part of this standard but is included for informative purposes only.)*

This annex is included for historical purposes. The SOAP-based services defined in this annex are deprecated. New designs are recommended to use the REST-based services defined in Annex W.

The following is the text of the original Annex N included for reference:

This annex defines a data model and Web service interface for integrating facility data from disparate data sources with a variety of business management applications. The data model and access services are generic and can be used to model and access data from any source, whether the server owns the data locally or is acting as a gateway to other standard or proprietary protocols.
...

[Add new **Annex V**, p. 1026]

ANNEX V Migration from SOAP Services (INFORMATIVE)

(This annex is not part of this standard but is included for informative purposes only.)

Annex N formerly defined a set of SOAP-based services for accessing simple abstract data and the histories of that data. This clause now provides migration information for moving those services to the REST services defined in Annex W.

In general, the "Path" parameter of the SOAP services is now represented in the REST URI path and the items in the SOAP "Options" parameter are represented as HTTP query parameters. The names of the SOAP options are mostly the same as the names of the REST query parameters; however, all query parameters require a value.

The data model used for the SOAP services arranged data into "nodes" and "attributes". These concepts are now called "data" and "metadata" to prevent confusion with XML "attributes".

V.1 Services

Many of the SOAP services have direct equivalents in REST while some require restructuring the questions or answers to achieve the same result.

V.1.1 getValue Service

The getValue service returned localizable plain text values. This service is replaced by an HTTP GET with the same path. However, only character string data is now localizable. All other data types are returned in canonical form. For example:

```
GET /path/to/primitive/data?alt=plain
...
```

```
200 OK
...
75.5
```

V.1.2 getValues Service

This service returned a list of localizable plain text values of attributes from multiple paths. There is no direct replacement for this service. However, multiple data items can now be read with an HTTP POST to the "/.multi" resource. The results are only available in XML or JSON. For example:

```
POST /.multi
...
<Composition>
  <List name="values">
    <Any name="1" via="/path/one"/>
    <Any name="2" via="/path/two"/>
  </List>
<Composition>
```

```
200 OK
...
<Composition>
  <List name="values">
    <Real name="1" via="/path/one" value="75.5"/>
    <Unsigned name="2" via="/path/two" value="100"/>
  </List>
<Composition>
```

V.1.3 getRelativeValues Service

This service returned a list of localizable plain text values of attributes from multiple paths where the paths were relative to a given starting path. There is no direct replacement for this service. However, multiple data items can now be read with an HTTP POST to the "/.multi" resource, but the paths used for the 'via' metadata are all absolute so the client will have to duplicate the base path for each. The results are only available in XML or JSON. See example for getValues service.

V.1.4 getArray Service

This service returned "array attributes" in plain text on separate lines. Since the new REST services support the exchange of structured data, there are no direct equivalent to "array attributes". Collections of data are returned in Array, List, SequenceOf, and Collections base types in either XML or JSON.

The closest equivalent to an "array attribute" for character strings is the StringSet base type that is used for the 'children', 'tags', and other metadata that is a collection of strings represented as a single primitive data type. The value is only available as a semicolon concatenated aggregation of the component strings. It is not possible to return the component strings on separate lines so the client will need to use the semicolon delimiter to separate the components.

```
GET /path/to/data/$children?alt=plain
...
```

```
200OK
...
child1;child2;child3
```

V.1.5 getArrayRange Service

This service returned a range of items from "array attributes" in plain text on separate lines. Since the new REST services support the exchange of structured data, there are no direct equivalent to "array attributes". Collections of data are returned in Array, List, SequenceOf, and Collection base types in either XML or JSON.

The parameters of this service are replaced by using the 'skip' and 'max-results' for the "Index" and "Count" SOAP parameters. For example:

GET /path/to/list/data?skip=10&max-results=5

V.1.6 getArraySize Service

This service returned the number of items in an "array attribute". There is no direct equivalent to "array attribute", however, the size of any collection can be obtained by querying the 'count' metadata on the collection data. For example:

GET /path/to/array/$count

V.1.7 setValue Service

The setValue service accepted localizable plain text values. This service is simply replaced by an HTTP PUT with the same path. For example:

```
PUT /path/to/data?alt=plain
...
100.0
```

V.1.8 setValues Service

This service accepted a list of localizable plain text values of attributes from multiple paths, each value on a separate text line. There is no direct replacement for this service. However, multiple data items can now be written with an HTTP POST to the "/.multi" resource. For example:

```
POST /.multi
...
<Composition>
   <List name="values">
      <Real name="1" via="/path/one" value="75.5"/>
      <Unsigned name="2" via="/path/two" value="100"/>
   </List>
<Composition>
```

V.1.9 getHistoryPeriodic Service

This service retrieved a list of localizable plain text values for periodic trend values, each value on a separate line. This is replaced by using the 'historyPeriodic' function. The SOAP "start", "interval", "count", and "resampleMethod" are replaced by the function parameters "start", "period", "count", and "method", respectively. For example:

GET /path/to/data/historyPeriodic(start=2011-12-03T00:00:00Z,period=3600,count=24,method=average)

V.1.10 getDefaultLocale Service

This service returned the default locale, or empty string if localization is not supported. It is replaced by reading /.info/.default-locale. For example:

GET /.info/.default-locale?alt=plain

V.1.11 getSupportedLocales Service

This service returned the (possibly empty) list of supported locales, each on a separate line. It is replaced by reading /.info/.supported-locales, with the exception that the locales are returned in a semicolon-separated concatenation rather than on separate lines. For example:

GET /.info/.supported-locales?alt=plain

V.2 Service Options

The SOAP services took "service options" that are not represented by HTTP query parameters. The following table represents the mapping between the two.

V.2.1 readback

There is no REST equivalent to the SOAP "readback" option.

V.2.2 errorString, errorPrefix

The REST query parameters "error-string" and "error-prefix" serve the equivalent function as these SOAP service options.

V.2.3 locale, writeSingleLocale

The REST query parameter "locale" serves a very similar function to the SOAP options "locale" and "writeSingleLocale". See Clause W.17

V.2.4 canonical, precision

The REST services support multiple locales for string values but do not support localizing representation of numbers and dates. Therefore, the "canonical" and "precision" SOAP query parameters have no equivalent and clients will have to convert from the always-canonical number and date formats to a localized format for display, if desired.

V.2.5 noEmptyArrays

The "noEmptyArrays" SOAP option was created to work around a defect in a common VisualBasic library that could not handle empty arrays in the response. This is no longer needed and has no equivalent in the REST services.

[Change **Clause H.6**, p. 1026]

H.6 Using BACnet with the *Former* BACnet/WS Web Services Interface *Defined by* ~~(~~Annex N~~)~~

Note that the SOAP-based Web Services interface defined by Annex N is deprecated for new designs. This clause is thus provided only for historical purposes.

This clause provides examples of the correspondence between BACnet/WS node attributes to specific properties of BACnet ...

135-2012*am*-6. Change Clause 21 identifiers to use a consistent format.

Rationale

The REST services defined in this addendum make use of structured data. When representing data that is defined in Clause 21, the names of the data are defined to be the Clause 21 ASN.1 identifiers. However, the identifiers in Clause 21 are currently in two different styles. Since there has been no standard usage of these identifiers "on the wire", the consistency has never mattered. However, now that standard names are required to be exchanged between implementations, the formats must be exactly equal. Before this becomes fixed forever, this is a good opportunity to make them use a consistent style. The historic format was to use camelCase for service structures and dash-separated for data structures but there are exceptions in both cases. Since the data structures have been studied and quoted much more than the service structures, the dash-separated format is chosen to be the consistent style.

[Change **Clause 21**, p. 639, and other clauses, as identified]

[For the entire Clause 21, change all sequence/choice member identifiers and value/bit identifiers that are currently in camelCase format to be in dash-separated format, e.g., "quitOnFailure" becomes "quit-on-failure", with the following exceptions:

- the following identifiers are preserved: "weekNDay", "pH"
- identifiers followed by numbers are preserved together: "wiegand26", "currency1", etc.
- all-caps fragments like "APDU", "ACK", "ID", "COV", "VT", and "PDU" shall be converted to lower case and preceded and/or followed with a dash where appropriate (e.g. "invokeID" becomes "invoke-id").
- in BACnetTimeStamp, "dateTime" becomes "datetime"
- primitive type names shall not be hyphenated or abbreviated: "bit-string" becomes "bitstring", "octet" becomes "octetstring", etc.
- Since DatePattern, TimePattern, and DateTimePattern are base types in the abstract data model in Annex Y, "date-pattern", "time-pattern", etc., become "datepattern" and "timepattern" in the style of other primitive type names like "bitstring".
- for BACnetObjectIdentifier types, "object-identifier" shall be the used as a name of a field or property, e.g. as a peer to "'device-identifier" (where "objectIdentifier" is currently used), and "objectidentifier " shall be used as a type name identifier, e.g. as a peer to "datetime", "bitstring", etc., (where "objectid" is currently used)

The member identifiers and value/bit identifiers are affected by this change but the names of the data structures themselves are not. For example, in the constructs below, "logData" becomes "log-data" but "BACnetLogMultipleRecord" remains unchanged.

BACnetLogMultipleRecord ::= SEQUENCE {
    timestamp [0] BACnetDateTime,
    ~~logData~~*log-data* [1] BACnetLogData
    }

BACnetLimitEnable ::= BIT STRING {
    ~~lowLimitEnable~~*low-limit-enable* (0),
    ~~highLimitEnable~~*high-limit-enable* (1)
    }

Additionally, the following identifiers shall be changed to be made consistent with other similar names.

- in BACnetChannelValue, BACnetLogData, BACnetLogRecord, BACnetPriorityValue, BACnetPropertyStates, and BACnetNotificationParameters, "signed" becomes "integer" and "signed-value" becomes "integer-value", however, "signed-out-of-range" remains unchanged.
- in BACnetLogData, BACnetLogRecord, BACnetEventParameter, BACnetFaultParameter, BACnetNotificationParameters, "enum" becomes "enumerated" and "enum-value" becomes "enumerated-value", and references to these in Clause 12 are changed also.

Other clauses shall be changed to be made consistent with these changes to Clause 21. e.g.,

- in Clause 12.1.11, "floatScale" becomes "float-scale", "integerScale" becomes "integer-scale", etc.
- in Clause 12.1.13, "moduloDivide" becomes "modulo-divide"
- in Clause 20.1.2, "max-APDU-length-accepted" becomes "max-apdu-length-accepted", "invokeID" becomes "invoke-id", etc.

]

[Add a new entry to **History of Revisions**, p. 1027]

(This History of Revisions is not part of this standard. It is merely informative and does not contain requirements necessary for conformance to the standard.)

HISTORY OF REVISIONS

…	…	**…**
1	19	**Addendum am to ANSI/ASHRAE 135-2012** Approved by ASHRAE on April 29, 2016, and by the American National Standards Institute on April 29, 2016. 1. Extend BACnet/WS with RESTful services for complex data types and subscriptions. 2. Extract data model from Annex Q into separate common model. 3. Rework Annex Q to be an XML syntax for the common model. 4. Add a JSON syntax for the common model. 5. Deprecate Annex N SOAP services and add a migration guide. 6. Change Clause 21 identifiers to use a consistent format.

POLICY STATEMENT DEFINING ASHRAE'S CONCERN FOR THE ENVIRONMENTAL IMPACT OF ITS ACTIVITIES

ASHRAE is concerned with the impact of its members' activities on both the indoor and outdoor environment. ASHRAE's members will strive to minimize any possible deleterious effect on the indoor and outdoor environment of the systems and components in their responsibility while maximizing the beneficial effects these systems provide, consistent with accepted Standards and the practical state of the art.

ASHRAE's short-range goal is to ensure that the systems and components within its scope do not impact the indoor and outdoor environment to a greater extent than specified by the Standards and Guidelines as established by itself and other responsible bodies.

As an ongoing goal, ASHRAE will, through its Standards Committee and extensive Technical Committee structure, continue to generate up-to-date Standards and Guidelines where appropriate and adopt, recommend, and promote those new and revised Standards developed by other responsible organizations.

Through its *Handbook*, appropriate chapters will contain up-to-date Standards and design considerations as the material is systematically revised.

ASHRAE will take the lead with respect to dissemination of environmental information of its primary interest and will seek out and disseminate information from other responsible organizations that is pertinent, as guides to updating Standards and Guidelines.

The effects of the design and selection of equipment and systems will be considered within the scope of the system's intended use and expected misuse. The disposal of hazardous materials, if any, will also be considered.

ASHRAE's primary concern for environmental impact will be at the site where equipment within ASHRAE's scope operates. However, energy source selection and the possible environmental impact due to the energy source and energy transportation will be considered where possible. Recommendations concerning energy source selection should be made by its members.

ASHRAE · 1791 Tullie Circle NE · Atlanta, GA 30329 · www.ashrae.org

About ASHRAE

ASHRAE, founded in 1894, is a global society advancing human well-being through sustainable technology for the built environment. The Society and its members focus on building systems, energy efficiency, indoor air quality, refrigeration, and sustainability. Through research, Standards writing, publishing, certification and continuing education, ASHRAE shapes tomorrow's built environment today.

For more information or to become a member of ASHRAE, visit www.ashrae.org.

To stay current with this and other ASHRAE Standards and Guidelines, visit www.ashrae.org/standards.

Visit the ASHRAE Bookstore

ASHRAE offers its Standards and Guidelines in print, as immediately downloadable PDFs, on CD-ROM, and via ASHRAE Digital Collections, which provides online access with automatic updates as well as historical versions of publications. Selected Standards and Guidelines are also offered in redline versions that indicate the changes made between the active Standard or Guideline and its previous version. For more information, visit the Standards and Guidelines section of the ASHRAE Bookstore at www.ashrae.org/bookstore.

IMPORTANT NOTICES ABOUT THIS STANDARD

To ensure that you have all of the approved addenda, errata, and interpretations for this Standard, visit www.ashrae.org/standards to download them free of charge.

Addenda, errata, and interpretations for ASHRAE Standards and Guidelines are no longer distributed with copies of the Standards and Guidelines. ASHRAE provides these addenda, errata, and interpretations only in electronic form to promote more sustainable use of resources.

4/16