

Delay-Tolerant Networking
Internet-Draft
Intended status: Standards Track
Expires: 20 July 2026

E.J. Birrane
E.A. Annis
B. Sipos
JHU/APL
16 January 2026

DTNMA Application Resource Identifier (ARI)
draft-ietf-dtn-ari-08

Abstract

This document defines the structure, format, and features of the naming scheme for the objects defined in the Delay-Tolerant Networking Management Architecture (DTNMA) Application Management Model (AMM), in support of challenged network management solutions described in the DTNMA document.

This document defines the DTNMA Application Resource Identifier (ARI), using a text-form based on the common Uniform Resource Identifier (URI) and a binary-form based on Concise Binary Object Representation (CBOR). These meet the needs for a concise, typed, parameterized, and hierarchically organized set of managed data elements.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <https://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 20 July 2026.

Copyright Notice

Copyright (c) 2026 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Revised BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Revised BSD License.

Table of Contents

1. Introduction	3
1.1. Scope	4
1.2. Use of ABNF	5
1.3. Use of CDDL	5
1.4. Example ARI Line Folding	6
1.5. Terminology	6
2. ARI Purpose	6
2.1. Resource Parameterization	7
2.2. Compressible Structure	7
2.2.1. Enumerated Path Segments	8
2.2.2. Relative Paths	8
2.2.3. Patterning	8
3. ARI Logical Structure	8
3.1. Names, Enumerations, Comparisons, and Canonicalizations	9
3.2. Literals	9
3.2.1. Object Reference Patterns	11
3.3. Object References	13
3.3.1. Organization ID	14
3.3.2. Model ID	14
3.3.3. Model Revision	15
3.3.4. Object Type	16
3.3.5. Object ID	16
3.3.6. Parameters	16
3.4. Namespace References	17
4. ARI Text Form	18
4.1. URIs and Percent Encoding	18
4.2. Literals	19
4.2.1. Typed Literal Values	20
4.2.2. Untyped Literal Values	28
4.2.3. Preferred Encodings	30
4.3. Object References	30
4.4. Namespace References	32
4.5. Relative References	32
5. ARI Binary Form	33
5.1. Intermediate CBOR	33
5.2. Literals	33

5.3. Object References	39
5.4. Namespace References	41
5.5. Relative References	42
6. Processing Activities	43
6.1. ID Segment Translation	43
6.2. Parameter Key Translation	44
6.3. Relative Reference Resolution	44
7. Transcoding Considerations	44
8. Interoperability Considerations	45
8.1. ARI Type Support	46
8.2. ARI Transport	46
9. Security Considerations	47
10. IANA Considerations	47
10.1. URI Schemes Registry	47
10.2. DTN Management Architecture	48
11. References	54
11.1. Normative References	54
11.2. Informative References	56
Appendix A. Example Equivalences	58
A.1. Primitive-Typed Literal	59
A.2. Timestamp Literal	60
A.3. Semantic-Typed Literal	60
A.4. Complex CBOR Literal	60
A.5. Non-parameterized Object Reference	61
A.6. Parameterized Object Reference	61
A.7. Recursive Structure with Percent Encodings	62
A.8. Full EXECSET and RPTSET with all ARI variations	63
Appendix B. Implementation Guidance	68
Implementation Status	69
Acknowledgments	70
Authors' Addresses	70

1. Introduction

The unique limitations of Delay-Tolerant Networking (DTN) transport capabilities [RFC4838] necessitate increased reliance on individual node behavior. These limitations are considered part of the expected operational environment of the system and, thus, contemporaneous end-to-end data exchange cannot be considered a requirement for successful communication.

The primary DTN transport mechanism, Bundle Protocol version 7 (BPv7) [RFC9171], standardizes a store-and-forward behavior required to communicate effectively between endpoints that may never co-exist in a single network partition. BPv7 might be deployed in static environments, but the design and operation of BPv7 cannot presume that to be the case.

Similarly, the management of any BPv7 protocol agent (BPA) (or any software reliant upon DTN for its communication) cannot presume to operate in a resourced, connected network. Just as DTN transport must be delay-tolerant, DTN network management must also be delay-tolerant.

The DTN Management Architecture (DTNMA) [RFC9675] outlines an architecture that achieves this result through the self-management of a DTN node as configured by one or more remote managers in an asynchronous and open-loop system. An important part of this architecture is the definition of a conceptual data schema for defining resources configured by remote managers and implemented by the local autonomy of a DTN node.

The DTNMA Application Management Model (AMM) [I-D.ietf-dtn-amm] defines a logical schema that can be used to represent data types and structures, autonomous controls, and other kinds of information expected to be required for the local management of a DTN node. The AMM further describes a physical data model, called the Application Data Model (ADM), that can be defined in the context of applications to create resources in accordance with the AMM schema. These named resources can be predefined in moderated publications or custom-defined as part of the Operational Data Model (ODM) of an agent.

Every AMM resource must be uniquely identifiable. To accomplish this, an expressive naming scheme is required. The Application Resource Identifier (ARI) provides this naming scheme. This document defines the ARI, based on the structure of a Uniform Resource Identifier (URI) of [RFC3986], meeting the needs for a concise, typed, parameterized, and hierarchically organized naming convention. Additionally, a binary form of ARI encoding based on Concise Binary Object Representation (CBOR) of [RFC8949] is defined for more compact interchange of the same information.

1.1. Scope

The ARI scheme is based on the structure of a URI [RFC3986] in accordance with the practices outlined in [RFC8820].

ARIs are designed to support the identification requirements of the AMM logical schema. As such, this specification will discuss these requirements to the extent necessary to explain the structure and use of the ARI syntax.

This specification does not constrain the syntax or structure of any existing URI (or part thereof). As such, the ARI scheme does not impede the ownership of any other URI scheme and is therefore clear of the concerns presented in [RFC7320].

This specification does not discuss the manner in which ARIs might be generated, populated, and used by applications. The operational utility and configuration of ARIs in a system are described in other documents associated with DTN management, to include the DTNMA and AMM specifications.

This specification does not describe the way in which path prefixes associated with an ARI are standardized, moderated, or otherwise populated. Path suffixes may be specified where they do not lead to collision or ambiguity.

This specification does not describe the mechanisms for generating either standardized or custom ARIs in the context of any given application, protocol, or network.

1.2. Use of ABNF

This document defines text structure using the Augmented Backus-Naur Form (ABNF) of [RFC5234]. The entire ABNF structure can be extracted from the XML version of this document using the XPath expression:

```
'//sourcecode[@type="abnf"]'
```

The following initial fragment defines the top-level rules of this document's ABNF.

```
start = ari
```

From the document [RFC3986] the definitions are taken for pchar, unreserved, pct-encoded. From the document [RFC3339] the definitions are taken for date-time, full-date, and duration. From the document [RFC5234] the definitions are taken for bit, hexdig, digit, and char-val. From the document [RFC8259] the definitions are taken for char and unescaped.

1.3. Use of CDDL

This document defines CBOR structure using the Concise Data Definition Language (CDDL) of [RFC8610]. The entire CDDL structure can be extracted from the XML version of this document using the XPath expression:

```
'//sourcecode[@type="cddl"]'
```

The following initial fragment defines the top-level rules of this document's CDDL, which includes the example CBOR content.

```
start = ari

; Limited sizes to fit the AMM data model
int32 = (-2147483648 .. 2147483647) .within int
uint32 = uint .le 4294967295
int64 = (-9223372036854775808 .. 9223372036854775807) .within int
uint64 = uint

; Restricted identifier text
id-text = tstr .regexp "![A-Za-z_][A-Za-z0-9_\\-\\.]*"
; Restricted identifier enumerations
id-int = int32
```

This document does not rely on any CDDL symbol names from other documents.

1.4. Example ARI Line Folding

The URI encoding of Section 4 does not allow blank space characters and some of the example ARIs are longer than RFC recommended line lengths, the examples uses the "single backslash" folding strategy of Section 7 of [RFC8792] for line wrapping. The header from that strategy is not used explicitly in this document, so any use of indentation in example ARIs will make use of this folding strategy.

1.5. Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

The terms "Application Data Model", "Application Resource Identifier", "Operational Data Model", "Externally Defined Data", "Variable", "Constant", "Control", "Literal", "Namespace", "Operator", "Report", "State-Based Rule", "Table", and "Time-Based Rule" are used without modification from the definitions provided in [I-D.ietf-dtn-amm].

2. ARI Purpose

AMM resources (contained within ADM and ODM namespaces) are referenced in the context of autonomous applications on an agent. The naming scheme of these resources must support certain features to inform DTNMA processing in accordance with the ADM logical schema.

This section defines the set of unique characteristics of the logical ARI scheme, the combination of which provides a unique utility for naming. While certain other naming schemes might incorporate certain elements, there are no such schemes that both support needed features and exclude prohibited features.

2.1. Resource Parameterization

The ADM schema allows for the parameterization of resources to both reduce the overall data volume communicated between DTN nodes and to remove the need for any round-trip data negotiation.

Parameterization reduces the communicated data volume when parameters are used as filter criteria. By associating a parameter with a data source, data characteristic, or other differentiating attribute, DTN nodes can locally process parameters to construct the minimal set of information to either process for local autonomy or report to remote managers in the network.

Parameterization eliminates the need for round-trip negotiation to identify where information is located or how it should be accessed. When parameters define the ability to perform an associative lookup of a value, the index or location of the data at a particular DTN node can be resolved locally as part of the local autonomy of the node and not communicated back to a remote manager.

2.2. Compressible Structure

The ability to encode information in very concise formats enables DTN communications in a variety of ways. Reduced message sizes increase the likelihood of message delivery, require fewer processing resources to secure, store, and forward, and require less resources to transmit.

The ARI syntax supports the following design elements to aid in the creation of more concise encodings: fixed-depth of hierarchy, enumerated forms of path segments, relative paths, and patterning. For example, the binary form of ARI (Section 5) supports the ability to identify objects in as few as 5 bytes and the ability to contain specific primitive values in as few as a single byte.

2.2.1. Enumerated Path Segments

Because the ARI structure includes paths segments with stable enumerated identifiers, each segment can be represented by either its text name or its integer enumeration. For human-readability in text form the text name is preferred, but for binary encoding and for comparisons the integer form is preferred. It is a translation done by the entity handling an ARI to switch between preferred representations (see Section 7); the data model of both forms of the ARI allows for either.

2.2.2. Relative Paths

Within an ARI, as defined in Section 3, literal values require no external context to interpret and so have no use for relative paths. Object reference values, however, always refer to an object within an ADM or ODM namespace and this namespace can be used as a base for resolving URI References as discussed in Section 4.5 and Section 5.5.

2.2.3. Patterning

Patterning in this context refers to the structuring of ARI information to allow defining value-matching logic as a function of wildcards and other expressions of general structure. Patterns allow for both better compression and fewer ARI representations by allowing a single ARI pattern to stand-in for a variety of actual ARI values.

This benefit is best achieved when the structure of the ARI is both expressive enough to include information that is useful to pattern match, and regular enough to understand how to create these patterns. A concrete definition for logic of and encodings for object reference patterns is defined in Section 3.2.1.

3. ARI Logical Structure

This section describes the information model of the ARI scheme to inform the discussion of the ARI syntax in Section 4 and Section 5. At the top-level, an ARI represents one of of the following AMM value classes defined in later subsections.

Literal values: These are values are those whose value and identifier are equivalent and are discussed in more detail in Section 3.2.

Object Reference values: These values refer to an individual object, possibly with parameters, and are discussed in Section 3.3.

Namespace Reference values: These values refer to an individual

namespace and are discussed in Section 3.4.

3.1. Names, Enumerations, Comparisons, and Canonicalizations

Within the ARI logical model, there are a number of domains in which items are identified by a combination of text name and integer enumeration: ADMs, ODMs, literal types, object types, and objects. In all cases, within a single domain the text name and integer enumeration SHALL NOT be considered comparable. It is an explicit activity by any entity processing ARIs to make the translation between text name and integer enumeration (see Section 7).

Text names SHALL be restricted to begin with an alphabetic character followed by any number of other characters, as defined in the id-text ABNF rule. This excludes a large class of characters, including non-printing characters. When represented in text form, the text name for ODMs is prefixed with a "!" character to disambiguate it from an ADM name (see Section 3.3).

For text names, comparison and uniqueness SHALL be based on case-insensitive logic. The canonical form of text names SHALL be the lower case representation.

Integer enumerations for ADMs and ODMs SHALL be restricted to a magnitude less than 2^{63} to allow them to fit within a signed 64-bit storage. The ADM registration in Table 6 reserves high-valued code points for private and experimental ADMs, while the entire domain of ODM code points (negative integers) is considered private use. Integer enumerations for literal types and object types SHALL be restricted to a magnitude less than 2^{31} to allow them to fit within a signed 32-bit storage. The registrations in Table 2 and Table 3 respectively Integer enumerations for objects (within an ADM or ODM) SHALL be restricted to a magnitude less than 2^{31} to allow them to fit within a signed 32-bit storage, although negative-value object enumerations are disallowed.

For integer enumerations, comparison and uniqueness SHALL be based on numeric values not on encoded forms. The canonical form of integer enumerations in text form SHALL be the shortest length decimal representation.

3.2. Literals

Literals represent a special class of ARI which are not associated with any particular ADM or ODM. A literal has no other name other than its value, but literals may be explicitly typed in order to force the receiver to handle it in a specific way.

Because literals will be based on the CBOR data model [RFC8949] and its extended diagnostic notation, a literal has an intrinsic representable data type as well as an AMM data type. The CBOR primitive types are named CDDL rules as defined in Section 3.3 of [RFC8610].

When converting from AMM literal types, the chosen CBOR type SHALL be determined by the mapping in Table 1. Additionally, when handling typed literal ARIs any combination of AMM literal type and CBOR primitive type not in Table 1 SHALL be considered invalid. This restriction is enforced by the CDDL defined in Section 5. Additionally, when handling a literal of AMM type CBOR the well-formed-ness of the CBOR contained SHOULD be verified before the literal is treated as valid.

AMM Literal Type	Used CBOR Type
NULL	null
BOOL	bool
BYTE	uint
INT	int
UINT	uint
VAST	int
UVAST	uint
REAL32	float
REAL64	float
TEXTSTR	tstr
BYTESTR	bstr
Non-primitive types	
TP	lit-time
TD	lit-time
LABEL	lit-label

CBOR	lit-cbor	
+-----+	+-----+	
ARITYPE	lit-label	
+-----+	+-----+	
OBJPAT	lit-objpat	
+=====+	+=====+	
Containers		
+=====+	+=====+	
AC	ari-collection	
+-----+	+-----+	
AM	ari-map	
+-----+	+-----+	
TBL	ari-tbl	
+-----+	+-----+	
EXECSET	exec-set	
+-----+	+-----+	
RPTSET	rpt-set	
+-----+	+-----+	

Table 1: AMM Literal Types to
CBOR Types

The text forms of time point and difference values defined in Section 4.2.1 allow both human-friendly representations (derived from [RFC3339] encodings) and direct numeric representation. The internal form of TP and TD values SHALL be as a decimal fraction supporting nanosecond precision. The internal form of TP and TD values SHALL be as a decimal fraction with a decimal mantissa large enough to hold a 64-bit signed integer (number of nanoseconds).

For the CBOR type, an implementation MAY perform CBOR decoding to validate that the byte string value is in fact well-formed CBOR but SHALL preserve the original byte string when transforming between the internal form and different encoded forms.

3.2.1. Object Reference Patterns

One of the considerations discussed in Section 2.2.3 is the ability to construct patterns to match and group objects by their various parts of their identity.

Although the logical structure of and encoded forms of Object Reference Patterns are similar to Object Reference values, they are different and they cannot be used interchangeably. The context used to interpret and match a pattern SHALL be explicit and separate from that used to interpret and dereference an Object Reference value.

While an Object Reference value can be used to dereference to a specific managed object and invoke behavior on that object, an Object Reference Pattern is used solely to perform matching logic against specific objects. The potential for an Object Reference Pattern to match specific objects SHALL NOT be interpreted as the existence of such hypothetical objects or any possible references to those objects.

As defined in the AMM, these patterns contain four separate parts (organization ID, model ID, object type, and object ID) and are used to match the four parts of an object path independently. Each part of the pattern contains one of the following to match the corresponding part of the object path:

- Single value: This will match only a single identifier value as either integer enumeration or text name.
- Range: This will match any identifier value contained in a disjoint set of integer intervals over the signed 32-bit domain.
- Wildcard: This will match any possible identifier in that part.

A diagram for logic and encoded fields of the range is shown in Figure 1. The text form of range encoding (Section 4.2.1) contains only the "min" and "max" values from the included intervals; it elides the least minimum and largest maximum if they are the limits of the integer domain. The binary form of range encoding (Section 5.2) contains only the "least" value and the width of each subsequent included and excluded interval; it elides the least value and last included width if the intervals cover the limits of the domain.

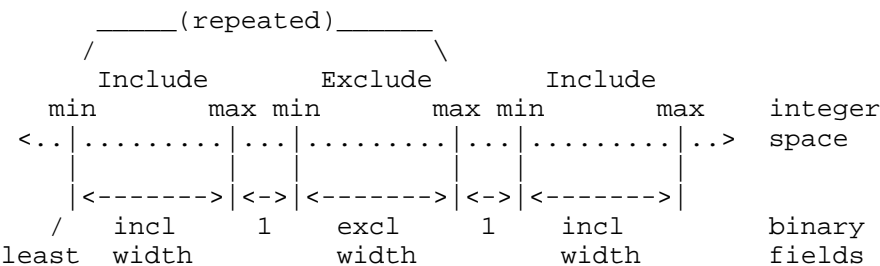


Figure 1: Integer Range Encoding

The most simple finite range is a single value, for example the value 10 is text encoded as "10" and binary encoded (shown using EDN) as [10,0] (which is a special case because this can also use the non-range specific value encoding). The most simple disjoint range

containing the values 2, 4, and 5 is text encoded as "2,4..5" and binary encoded as [2,0,0,1] because the first included interval is width zero, excluded interval is width zero, and last included interval is width 1. The largest possible range covering the entire 32-bit number space can be text encoded as "-2147483648..4294967295" and binary encoded as [-0x80000000,0xFFFFFFFF] or the more concise elided form of ".." and [null,null] respectively (and even this is a special case because this can also be represented by the wildcard value true, but that is a normalization decision not a coding one).

3.3. Object References

Object references are composed of two parts: object identifier and optional parameters. The object identifier can be dereferenced to a specific object in the ADM/ODM, while the parameters provide additional information for certain types of object and only when allowed by the parameter "signature" from the ADM/ODM.

The object identifier itself contains these components, described in the following subsections: organization ID, model ID with optional model revision, object type, and object ID. When encoded in text form (see Section 4), the identifier components correspond to the URI authority and path segments.

Each identifier component has two possible forms; one more human-friendly and one more compressible. Values can only be converted between forms based on a local registry of data models. There is nothing intrinsic in either form which relates to the other form (_i.e._, each name and enumeration value is arbitrary).

Text name form: This form corresponds with a human-readable identifier for the component. A text form component SHALL contain only URI path segment characters (pchar) representing valid UTF-8 text in accordance with [RFC3629]. This requirement applies to text components generally, specific component uses will restrict this valid domain further.

There is no intrinsic limit to the length of a text form component. Implementations are RECOMMENDED to not impose an arbitrary small limit to text form components.

Integer enumeration form: This form corresponds with a compressible value suitable for on-the-wire encoding between Manager and Agent. Sorting and matching integer components is also faster than text form. Every integer form component SHALL be small enough to be represented as a 32-bit signed integer.

An Agent implementation MAY be incapable of handling text form object identifier components, and this needs to be communicated to any associated Manager prior to encoding any ARIs for that Agent.

The organization ID, model ID, and model revision together are referred to as a "namespace". ADM resources exist within namespaces to eliminate the possibility of a conflicting resource name, aid in the application of patterns, and improve the compressibility of the ARI. Namespaces SHALL NOT be used as a security mechanism to manage access. An Agent or Manager SHALL NOT infer security information or access control based solely on namespace information in an ARI.

3.3.1. Organization ID

Organization IDs are used to segment the full domain of object identifiers into separately managed logical segments. Some organizations will be well known and registered with IANA, others will choose to make use of the private use reserved range for enumerations and names. This document allocates well known organization IDs in Section 10.2 for the IETF and IANA, as well as for example models.

Well Known organizations: A well known organization ID SHALL be registered with IANA. All text form well known organization IDs SHALL match the id-text rule of Section 4.1 and not begin with the bang character "!". All integer form well known organization IDs SHALL be non-negative.

Private Use organizations: A private use organization ID need not be registered with IANA. All text form private use organization IDs SHALL match the id-text rule and begin with the bang character "!". All integer form private use organization IDs SHALL be negative.

3.3.2. Model ID

Model IDs are used to segment a full organizational domain into individual object namespaces. There are two types of model IDs corresponding to the two types of namespaces described in the AMM Section 3.1.3 of [I-D.ietf-dtn-amm] as follows.

ADM namespace: A model ID for an ADM namespace is unique within an organization and, except for private or experimental use, SHOULD be registered with IANA (see Section 10.2) or an organization-specific authority. All text form ADM model IDs SHALL match the id-text rule of Section 4.1 and not begin with the bang character "!". All integer form ADM model IDs SHALL be non-negative.

ODM namespace: A model IDs for an ODM namespace does not have universal registration and SHALL be considered to be private use. It is expected that runtime ODM namespaces will be allocated and managed per-user and per-mission. All text form ODM model IDs SHALL match the id-text rule and begin with the bang character "!". All integer form ODM model IDs SHALL be negative.

3.3.3. Model Revision

A single ADM is allowed and expected to change over time, and references to objects within an ADM can include an identifier for a specific ADM revision. Each released revision of an ADM SHALL be identified by a specific revision date in the Gregorian calendar. An ARI referencing an object within an ADM MAY contain a specific model revision.

Because an ODM is expected to be dynamic and modified during Agent runtime, there is no meaning to a revision on an ODM. An ARI referencing an object within an ODM SHALL NOT contain a model revision.

The internal representation of revision dates is an implementation matter, but the encoded representations defined in this document include both text and integer form consistent with external standards (cited by the encodings of Section 4 and Section 5). Note that because ARI structure includes some recursion (*e.g.*, object reference parameters and container members are themselves ARIs), it is possible to have a model revision present at any depth of a complex ARI.

Because an Agent is able to implement only a single revision of any model, the model revision data SHALL NOT be present in an ARI being communicated to or from an Agent. If an Agent receives an ARI containing model revision information and that is inconsistent with the revision implemented by the Agent, it is an implementation matter for how to detect and handle the inconsistency.

The reason for including revision information in an ARI is to allow a Manager to keep that information intrinsic to the rest of the ARI in its own bookkeeping. A Manager SHALL recursively remove any model revision(s) before transmitting an ARI to an Agent, which satisfies the earlier paragraph. A Manager MAY add model revision(s) to an ARI received from an Agent to help its own data bookkeeping.

3.3.4. Object Type

Due to the flat structure of an ADM, as defined in Section 4 of [I-D.ietf-dtn-amm], all managed objects are of a specific and unchanging type from a set of available DTNMA object types. The preferred form for object types in text ARIs is the text name, while in binary form it is the integer enumeration (see Section 7). Both of these forms have values registered with IANA in Table 3.

The following subsection explains the form of those object identifiers.

3.3.5. Object ID

Within a single ADM or ODM namespace and a single object type, all managed objects have similar characteristics and all objects are identified by a single text name or integer enumeration. The preferred form for object names in text ARIs is the text name, while in binary form it is the integer enumeration.

Any ADM-defined object will have both name and enumeration, while a ODM-defined object can have either but not both. Conversion between the two forms requires access to the original ADM, and its specific revision, in which the object was defined. All text form object names SHALL match the id-text rule of Section 4.1..

3.3.6. Parameters

The ADM logical schema allows many object types to be parameterized when defined in the context of an application or a protocol.

If two instances of an ADM resource have the same namespace and same object type and object name but have different parameter values, then those instances are unique and the ARIs for those instances SHALL also be unique. Therefore, parameters are considered part of the ARI syntax.

The ADM logical schema defines specialized uses for the term "parameter" to disambiguate each purpose, as defined below.

Formal Parameters:

Formal parameters define the type, name, and order of the information that customizes an ARI. They represent the unchanging "signature" of the parameterized object. Because ARIs represent a use of an object and not its definition, formal parameters are not present in an ARI and instead are part of an object model.

Given Parameters:

Given parameters represent the data values used to distinguish different instances of a parameterized object. A given parameter is an AMM value and is represented by an ARI within the context of a parameter list (AC) or parameter map (AM). Because of necessary normalizing (of type and default value) based on formal parameters, multiple given parameters can correspond with the same meaning (see Actual Parameters below).

Additionally, there are two ways in which the value of a given parameter can be specified: parameter-by-value and parameter-by-name.

Parameter-By-Value: This method involves directly supplying the value as part of the actual parameter. It is the default method for supplying values.

Parameter-By-Name: This method involves specifying the name of an other parameter and using that other parameter's value as a substitute for the value of this parameter. This method is useful when a parameterized ARI is produced by an AMM object which itself is parameterized. The original ARI parameters contain literal ARIs with LABEL type, and when the value is produced based on input parameters the substitution is made. In this way, an actual parameter can be "flowed down" to produced values at runtime.

Actual Parameters:

Actual parameters represent a normalized set of values taken from a set of given parameters and normalized using a set of corresponding formal parameters. An actual parameter is an AMM value and is represented by an ARI.

Because normalizing can cause a given parameter to change type (in order to conform to a formal parameter type) or take on a default value (when not present in the given parameters), a single set of actual parameters can correspond with multiple options for given parameters.

3.4. Namespace References

A namespace reference is composed of the same initial parts as an object reference: the organization ID and model ID with optional model revision. These are used to refer to an entire ADM or ODM namespace. It is treated as a separate class of ARI in order to separate it from the additional required parts of an Object Reference (object type and object ID) and optional given parameters. The purpose of a namespace reference is to give an ARI value to the whole namespace separately from any of its contained objects.

4. ARI Text Form

This section defines how the data model explained in Section 3 is encoded as text conforming to the URI syntax of [RFC3986] with value encoding heavily influenced by the CBOR Extended Diagnostic Notation (EDN). One significant difference from the ARI encoding from EDN is that ARI identifiers and literal enumerated values are handled in a case-insensitive way, while EDN specifies enumerated values (such as "undefined" and "Infinity" as case sensitive).

When used within the context of a base ARI, the relative reference form of Section 4.5 can be used. In all other cases an ARI must be an absolute-path form and contain a scheme.

While this text description is normative, the ABNF schema in this section provides a more explicit and machine-parsable text schema. The scheme name of the ARI is "ari" and the scheme-specific part of the ARI follows one of the forms corresponding to the literal value (Section 4.2), object reference (Section 4.3), or namespace reference (Section 4.4).

```
ari = lit-ari / objref-ari / nsref-ari

; Scheme prefix used in specific contexts
ARIPREFIX = "ari:"
```

4.1. URIs and Percent Encoding

Due to the intrinsic structure of the URI, on which the text form of ARI is based, there are limitations on the syntax available to the scheme-specific-part [RFC7595]. One of these limitations is that each path segment can contain only characters in the pchar ABNF rule defined in [RFC3986]. For most parts of the ARI this restriction is upheld by the values themselves: ADM/ODM names, literal and object type names, and object names have a limited character set all within the rule val-seg. For literals and nested parameters though, the percent encoding of Section 2.4 of [RFC3986] is needed.

The following rule val-seg is an allowed superset of all path segment patterns, which includes ARI-specific control characters in val-delims. These control characters SHOULD NOT be percent encoded when encoding an ARI. ARI decoders SHALL handle percent encoded forms of all segment contents (_i.e._, each portion of an ARI needs to be percent decoded exactly once).

```

; A subset of URI "segment-nz" and "pchar" rules which matches all
; identifiers, primitive values, and typed-literal values.
val-seg = 1*( unreserved / pct-encoded / val-delims )
; A subset of URI "reserved" rule.
val-delims = "!" / "'" / "(" / ")" / "*" / "+" / "," / ";" / "="
           / ":" / "@"

; A text name must start with an alphabetic character or underscore
; with an optional bang prefix.
id-text = ["!"] (ALPHA / "_") *(ALPHA / DIGIT / "_" / "-" / ".")
; An integer enum must contain only digits with no zero padding
; The value must also be 32-bit signed range,
; but that is not enforced by this ABNF.
id-int = ["-"] nopad-number

nopad-number = "0" / non-zero-number
non-zero-number = %x31-39 *DIGIT

```

In the ARI text examples in this document the URIs have been percent-decoded for clarity, as might be done in an ARI display and editing tool. But the actual encoded form of the human-friendly ARI `ari:"text"` is `ari:%22text%22`. Outside of literals, the safe characters which are not be percent-encoded are the structural delimiters `/()``=;`, used for parameters and ARI collections.

```

| Even with the allowed character set of val-seg there are still
| some literal value control characters that do not fit within
| the URI pchar set and which need to be percent-encoded,
| specifically the double-quote character " for text strings.

```

One other aspect of convenience for human editing of text-form ARIs is linear white space. The current ABNF pattern, staying within the URI pattern, do not allow for whitespace to separate list items or otherwise. A human editing an ARI could find it convenient to include whitespace following commas between list items, or to separate large lists across lines. Any tool that allows this kind of convenience of editing SHALL collapse any white space within a single ARI before encoding its contents.

4.2. Literals

Based on the structure of Section 3.2, the text form of the literal ARI contains only a URI path with an optional AMM literal type. A literal has no concept of a namespace or context, so the path is always absolute. When the path has two segments, the first is the AMM literal type and the second is the encoded literal value. When the path has a single segment it is the encoded literal value. As a shortcut, an ARI with only a single path segment is necessarily an

untyped literal so the leading slash is elided.

The text form of literal ARI has two layers of coding: the URI path structure and the type-specific value coding. These are distinguished below in the ABNF by the `lit-ssp-struct` symbol being for the general path structure and type-specific coding defined in Section 4.2.1.

```
lit-ari = [ARIPREFIX] lit-ssp-struct

lit-ssp-struct = lit-container
                / lit-typeval-struct
                / lit-notype-struct

; More complex text for non-primitive values
lit-typeval-struct = "/" lit-type-id "/" val-seg
; Type ID is restricted to valid literal types, within the "val-seg"
; set, and never percent encoded
lit-type-id = id-text / id-int

; Containers use different structure than lit-typeval-struct
; because of rule recursion
lit-container = lit-ac / lit-am / lit-tbl / lit-execset / lit-rptset
lit-ac = "/" ("AC" / "17") "/" ari-collection
lit-am = "/" ("AM" / "18") "/" ari-map
lit-tbl = "/" ("TBL" / "19") "/" ari-tbl
lit-execset = "/" ("EXECSET" / "20") "/" exec-set
lit-rptset = "/" ("RPTSET" / "21") "/" rpt-set

; The untyped value is a subset of the "lit-notype" symbol
lit-notype-struct = val-seg
```

4.2.1. Typed Literal Values

The definition in this section is a specialization of the `lit-typeval-struct` structure for specific literal types.

An ARI encoder or decoder SHALL handle both text name and integer enumeration forms of the `lit-type` symbol. When present and able to be looked up, the literal type SHOULD be a text name.

The text form of typed values SHALL be one of the following, based on the associated `lit-type` value in the ARI:

NULL:

This type contains only the single value "null" as in the following ABNF rule.

null = "null"

The canonical form of the null value SHALL be lower case.
Decoding of values in the null type SHALL be case-insensitive.

BOOL:

This type contains the two fixed values "true" and "false" as in the following ABNF rule.

bool = "true" / "false"

The canonical form of boolean values SHALL be lower case.
Decoding of values in the boolean type SHALL be case-insensitive.

BYTE, INT, UINT, VAST, or UVAST:

The integer types, signed or unsigned, match the following integer ABNF rule. Each specific type will limit the domain of valid values within this more general encoding.

```
integer = optsign uinteger
optsign = ["+" / "-"]
; Just the numeric digits
uinteger = uint-dec / uint-bin / uint-hex
uint-dec = 1*DIGIT
uint-bin = "0b" 1*BIT
uint-hex = "0x" 1*HEXDIG
```

The canonical form of integer values SHALL NOT include a leading plus character. The canonical form of hexadecimal integer values SHALL use upper case letters. Decoding of values in the integer type SHALL be case-insensitive.

REAL32 or REAL64:

The floating-point types match the following float ABNF rule, which includes a hexadecimal form that avoids decimal conversion of the underlying value. All of these forms conform to the text representations defined in Section 5.12 of [IEEE.754-2019] for binary16, binary32, or binary64 values.

```
float = float-exp / float-dec / float-hex / float-inf / float-nan
; Decimal exp agrees with C99 '%e' format, point required
float-exp = optsign float-dec-mant "e" optsign 1*DIGIT
float-dec-mant = *DIGIT "." 1*DIGIT / 1*DIGIT "."
; Decimal fraction agrees with C99 '%f' format, point required
float-dec = optsign float-dec-mant
; Hexadecimal exp agrees with C99 '%a' format, point required
float-hex = optsign "0x" float-hex-mant "p" optsign 1*DIGIT
float-hex-mant = *HEXDIG "." 1*HEXDIG / 1*HEXDIG "."
; Non-finite enumerated values
float-inf = optsign "Infinity"
float-nan = "NaN"
```

The canonical form of float values SHALL NOT include a leading plus character. The canonical form of alphabetic characters within float values SHALL be the form in which the ABNF rules are defined above. Decoding of values in the float type SHALL be case-insensitive.

TEXTSTR:

The text string type matches the following tstr ABNF rule after percent-decoding, which is consistent with the CBOR EDN definition. As an alternative, if the text value matches the id-text character set it can be encoded directly without quoting.

```
tstr = tstr-quoted / id-text
; double-quoted and escaped text logic from Section 7 of RFC 8259
tstr-quoted = DQUOTE *char DQUOTE
DQUOTE = %x22
```

The canonical percent encoded form for a text string SHALL treat only the unreserved characters and "'" as safe. All other characters within the double-quoted text (including the double quotes) SHOULD be percent encoded.

BYTESTR:

The byte string type matches the following bstr ABNF rule after percent-decoding, which is consistent with the CBOR EDN definition restricted to only utf8, base16, and base64url encodings and disallowing the presence of linear whitespace for base16 and base64url.

```
bstr = bstr-utf8 / bstr-b16 / bstr-b64
; Escaped text logic from Section 7 of RFC 8259
bstr-utf8 = SQUOTE *char SQUOTE
; Encoding of Section 8 of RFC 4648
bstr-b16 = "h" SQUOTE *(2HEXDIG) SQUOTE
; Encoding of Section 5 of RFC 4648
bstr-b64 = "b64" SQUOTE *(ALPHA / DIGIT / "-" / "_") *EQ SQUOTE

SQUOTE = %x27
EQ = %x3D
```

The non-prefixed form of byte string SHALL only be used if the value contains only UTF-8 text in accordance with [RFC3629]. When using the "h" prefix the base16 encoding from Section 8 of [RFC4648] SHALL be used. The base16 encoding SHALL have a canonical form using upper case letters. When using the "b64" prefix only the base64url encoding from Section 5 of [RFC4648] SHALL be used as this avoids unnecessary percent encodings. The base64url encoding SHALL have a canonical form where padding characters are not used, which is compatible with EDN.

The canonical percent encoded form for a byte string SHALL treat only the unreserved characters and "'" as safe. All other characters within the single-quoted text SHOULD be percent encoded.

TP:

This type uses either the date-time ABNF rule of Appendix A of [RFC3339] always in the "Z" time-offset, or as a decimal representation of the relative time from the DTN Epoch.

```
lit-tp = date-time / float-dec / integer
```

All forms of TP values are limited to a decimal mantissa no larger than a 64-bit signed integer number of nanoseconds. This corresponds with an exact domain, in seconds offset from the DTN epoch, of -9223372036.854775808 to 9223372036.854775807 inclusive, or approximately between the years 1708 and 2292.

When the human-friendly text form of [RFC3339] is used, the date and time SHALL both be present along with the "Z" time-offset. The date component separator "-" and time component separator ":" are optional and MAY be omitted by an encoder. When separators are omitted, the text need not be percent encoded.

TD:

This type uses either the duration ABNF rule of Appendix A of [RFC3339] with a positive or negative sign prefix, or as a decimal representation of the relative time value.

lit-td = duration / float-dec / integer

All forms of TD values are limited to a decimal mantissa no larger than a 64-bit signed integer number of nanoseconds. This corresponds with an exact domain, in seconds, of -9223372036.854775808 to 9223372036.854775807 inclusive, or approximately 292 years.

When the human-friendly text form of [RFC3339] is used, the largest duration component allowed SHALL be the day ("D"). No year, month, or week components are allowed to be present. This text is unquoted and, due to the constraints on its value, need not be percent encoded.

Additionally, a zero-value duration is valid to be present in the human-friendly form. The zero-value duration SHOULD contain only a seconds ("S") component with no fractional part, *i.e.*, PT0S.

LABEL:

This type uses the following lit-label ABNF rule from this document. This text is unquoted and, due to the constraints on its value, need not be percent encoded.

lit-label = id-text / id-int

CBOR:

This type uses the following lit-cbor ABNF rule for its value. The value encoding is the raw byte string of the embedded CBOR item.

lit-cbor = bstr

ARITYTYPE:

This type uses the shared lit-label ABNF rule for its value to identify a built-in type. This means the value can either be the integer enumeration of the type or its text name. Processors will treat either form the same if they support the associated built-in type.

OBJPAT:

This type uses the following lit-objpat ABNF rule for its value to match sets of objects using each of their identifiers as described in Section 3.2.1. The range encoding uses included integer intervals identified by their minimum and maximum included values.

The first interval can elide the minimum value as an indicator that the interval minimum is the domain minimum value. The last interval can elide the maximum value as an indicator that the interval maximum is the domain maximum value.

```
lit-objpat = 4objpat-part
; Each independent path segment pattern
objpat-part = "(" objpat-item ")"
objpat-item = objpat-wildcard / objpat-single / objpat-range

objpat-wildcard = "*"
objpat-single = id-text / id-int

; Each interval in comma-separated list
objpat-range = objpat-intvl *( "," objpat-intvl )
; Either end can be elided to include the respective domain limit
objpat-intvl = id-int / ([id-int] ".." [id-int])
```

The canonical form for each range orders the included intervals in ascending order. The canonical form of interval uses the single-value variation when its minimum and maximum are equal. The canonical form of interval uses indefinite limits as a shortened encoding for the minimum and maximum value of the domain.

AC:

This type uses the following ari-collection ABNF rule for its value. Each item of the collection is an already-percent-encoded text-form ARI.

```
; A comma-separated list of any form of ARI with enclosure
ari-collection = "(" [ari-collection-list] ")"
ari-collection-list = ari *( "," ari)
```

AM:

This type uses the following ari-map ABNF rule for its value. Each key of the map is an already-percent-encoded text-form literal-value ARI. Each value of the map is an already-percent-encoded text-form ARI.

```
; A comma-separated list of pairs, each delimited by equal-sign
; with literal-only keys
ari-map = "(" [ari-map-list] ")"
ari-map-list = ari-map-pair *( "," ari-map-pair)
ari-map-pair = lit-notype "=" ari
```

Although an AM value logically has no specific ordering of its contained pairs, an encoded AM value necessarily has an ordering. The canonical encoded form of an AM value SHALL use the same key

ordering defined for the CBOR map deterministic ordering in Section 4.2.1 of [RFC8949]. An implementation MAY choose to preserve AM key ordering between encoded forms.

TBL:

This type uses the following ari-tbl ABNF rule for its value. The value is prefixed by the number of columns in the table, followed by an AC representing each separate row in the table. Each item of the table is an already-percent-encoded text-form ARI.

```
ari-tbl = "c=" nopad-number ";" *ari-collection
```

The length of each row in a TBL value SHALL be equal to the column count (c parameter) of the TBL.

	Note that although the TBL uses the same syntax as AC for
	encoding each row, the value itself is not necessarily
	internally represented by a sequence of AC values.

EXECSET:

This type uses the following exec-set ABNF rule for its value.

```
exec-set = "n=" exec-nonce ";" exec-targets
exec-nonce = null / uinteger / bstr
exec-targets = ari-collection
```

The value is prefixed by the optional nonce value for the associated execution(s), followed by an AC representing each item to be executed (either a CTRL reference, MAC-producing reference, or MAC value literal). When using the integer type, the nonce value SHALL be restricted to only non-negative integers. Each item of the targets is an already-percent-encoded text-form ARI. An EXECSET target list SHALL be non-empty to be a valid value.

	Note that although the EXECSET uses the same syntax as AC
	for encoding each row, the value itself is not necessarily
	internally represented by an AC.

RPTSET:

This type uses the following rpt-set ABNF rule for its value. The value consists of the optional nonce value for the associated execution(s), a reference absolute time for all contained reports, and the list of contained reports. Each contained report consists of a relative creation time for the report (relative to the RPTSET reference time), a reference to the source of the report, and an AC representing each item in the report. Each item of a report is an already-percent-encoded text-form ARI, as is the source reference. The nonce and timestamps are all literal values constrained to specific value types. An RPTSET report list SHALL be non-empty to be a valid value.

```
rpt-set = "n=" exec-nonce ";r=" lit-ari ";" [rpt-list] ")"
rpt-list = rpt-container *("," rpt-container)
rpt-container = "t=" lit-ari ";s=" ari ";" rpt-items
rpt-items = ari-collection
```

| Note that although the report container uses the same syntax
| as AC for encoding each report item list, the list itself is
| not necessarily internally represented by an AC.

The canonical encoded form of the reports list is ordered by increasing "t" field.

Some examples of typed literal values are below. The represented values for TP, TD, and CBOR types are the same just with different text representations. The three ARITYPE values refer to the same built-in type.

```
ari:/NULL/null
ari:/BOOL/true
ari:/BYTE/10
ari:/INT/10
ari:/UINT/10
ari:/VAST/10
ari:/UVAST/10
ari:/REAL32/1e10
ari:/LABEL/name
ari:/TP/20230102T030405Z
ari:/TP/2023-01-02T03:04:05Z
ari:/TP/725943845
ari:/TD/+PT1H
ari:/TD/3600
ari:/LABEL/name
ari:/CBOR/h'0a'
ari:/ARITYPE/int
ari:/ARITYPE/INT
ari:/ARITYPE/4
```

The following object reference patterns will match objects meeting all of the following: is in the organization "example" (65535), is in a model with a private use (negative) enumeration or model number 1, has any object type, and has an object enumeration between 10 and 100 inclusive. The second example is a longer form representing the same pattern but using explicit interval limits at the end of associated numeric domain.

```
ari:/OBJPAT/(65535)(..-1,1)(*)(10..100)
ari:/OBJPAT/(65535)(-2147483648..-1,1..1)\
  (-2147483648..2147483647)(10..100)
```

These are typed container values in text form:

```
ari:/AC/(1,2,3)
ari:/AM/(1=2,2=4,3=9)
ari:/TBL/c=3;(1,true,%22A%22)(2,false,%22B%22)
```

And these are a pair of EXECSET and RPTSET having identical nonce value (folded according to Section 1.4):

```
ari:/EXECSET/n=1234;(\
  //example/adm-a/CTRL/dothing,\
  //example/adm-a/CONST/amacro\
)
ari:/RPTSET/n=1234;r=/TP/20230102T030405Z;(\
  t=/TD/PT0S;s="//example/adm-a/CTRL/dothing;(null),\
  t=/TD/PT5S;s="//example/adm-a/CONST/amacro;(null)\
)
```

4.2.2. Untyped Literal Values

The definition in this section is a specialization of the lit-notype-struct structure for specific primitive types.

When untyped, the literal value SHALL be one of the primitive types named by the lit-notype ABNF rule below. The separate value encodings use rules from Section 4.2.1.

```
lit-notype = undefined / null / bool / float / integer / tstr / bstr
undefined = "undefined"
```

	The order of this symbol sequence is significant because the
	unquoted tstr must be matched after the enumerated types of
	undefined, null, and bool.

The undefined value is only valid as an untyped literal, it has no associated built-in type. The canonical form of the undefined value SHALL be lower case. Decoding of the undefined value SHALL be case-insensitive.

Integer values in the untyped literal SHALL be limited to the union of all integer types in the AMM (BYTE, INT, UINT, VAST, and UVAST). This has a total domain of $-(2^{63})$ to $2^{64}-1$ inclusive.

Floating point values in the untyped literal SHALL be limited to the union of all floating point types in the AMM (REAL32 and REAL64).

Some example of untyped literals are below.

```
ari:undefined
ari:null
ari:true
ari:FALSE
ari:1.1
ari:1.
ari:.1
ari:1.1e+06
ari:0x1.4p+3
ari:0x1.p+3
ari:0x.4p3
ari:Infinity
ari:-Infinity
ari:NaN
ari:10
ari:0xA
ari:0b1010
ari:-0x10
ari:0x7FFFFFFFFFFFFFFF
ari:-0x8000000000000000
ari:'bytes'
ari:h'6279746573'
ari:b64'Ynl0ZXN0'
ari:text
ari:%22text%22
ari:%22hi%5Cu1234%22
ari:%22hi%5CuD834%5CuDD1E%22
```

4.2.3. Preferred Encodings

Several of the literal types defined in Section 4.2.1 allow the same AMM value to be represented by multiple logically equivalent encodings. Because these encodings are not equivalent in represented text size or processing needs it is useful for an ARI processor to allow a user to determine preferred encodings when processing text-form ARIs.

Integer values: These options correspond to the BYTE, INT, UINT, VAST, and UVAST types and untyped int values. These values can be encoded as either base-2, base-10, or base-16. In uses which are more resource constrained and less human-facing a processor MAY encode integers only in base-16.

Floating point values: These options correspond to the REAL32 and REAL64 types and untyped float values. These values can be encoded as either decimal fraction, decimal exponential, or hexadecimal exponential. In uses which are more resource constrained and less human-facing a processor MAY encode floats only as hexadecimal.

Byte string values: These options correspond to the BYTESTR and CBOR types and untyped bstr values. These values can be encoded as either raw bytes, base16, or base64url. In uses which are more resource constrained and less human-facing a processor MAY encode byte strings only as base16.

Time values: These options correspond to the TP and TD types. These values can be encoded as either delimited text, non-delimited text, or decimal fraction. In uses which are more resource constrained and less human-facing a processor MAY encode time values only as decimal fraction.

4.3. Object References

Based on the structure of Section 3.3, the text form of the object reference ARI is a URI with an authority, corresponding to the organization ID, and three path segments, corresponding to the model ID, object type and object ID respectively. Those components (excluding parameters as defined below) are referred to as the object identifier.

An ARI encoder or decoder SHALL handle both text name and integer enumeration forms of the organization ID, model ID, object type, and object ID.

The final segment containing the object ID MAY contain parameters enclosed by parentheses "(" and ")". There is no semantic distinction between the absence of parameters and the empty parameter list. The contents of the parameters SHALL be interpreted as a literal AC or AM in accordance with Section 4.2.

The parameters as a whole SHALL be the percent encoded form of the constituent ARIs, excluding the structural delimiters /(),=.

Implementations are advised to be careful about the percent encoded vs. decoded cases of each of the nested ARIs within parameters to avoid duplicate encoding or decoding. It is recommended to dissect the parameters and ARI collections in their encoded form first, and then to dissect and percent decode each separately and recursively.

The object reference ARI has the following ABNF definition.

```
; The structure of an object reference shares the "nsref-ari" rule
objref-ari = nsref-ari objref-underns

; Path under the namespace, with optional parameters
objref-underns = obj-type "/" obj-id [paramlist]

; Parameters are either AC or AM
paramlist = ari-collection / ari-map

; Organization ID, within the "val-seg" set
org-id = id-text / id-int
; Model ID and optional revision, within the "val-seg" set
model-seg = model-id ["@" model-rev]
model-id = id-text / id-int
model-rev = full-date ; from RFC 3339
; Type is restricted to valid object types, within the "val-seg" set
obj-type = id-text / id-int
; Object ID, within the "val-seg" set
obj-id = id-text / id-int
```

Some examples of object reference values are below.

```
ari://example/adm-a@2024-06-25/EDD/someobj
ari://example/adm-a/EDD/someobj
ari://example/adm-a/CTRL/otherobj(true,3)
ari://example/adm-a/CTRL/otherobj(%22a%20param%22,/UINT/10)
ari://65535/1/-1/0
ari://example/!odm-b/VAR/counter
ari://65535/-20/-11/84
```

4.4. Namespace References

Based on the structure of Section 3.4, the text form of the namespace reference ARI is a URI with an authority part and one path segment, corresponding to the model ID with optional revision, followed by a path separator. The trailing path separator SHALL be present in the canonical form of a namespace reference.

The namespace reference ARI has the following ABNF definition.

```
; Absolute namespace with optional scheme
nsref-ari = [ARIPREFIX] "/" org-id "/" model-seg "/"
```

Some examples of namespace reference values are below.

```
ari://example/adm-a@2024-06-25/
ari://example/adm-a/
ari://65535/1/
ari://example/!odm-b/
ari://65535/-20/
```

4.5. Relative References

The text form of ARI can contain a relative reference, as defined in Section 4.1 of [RFC3986], only for the specific case of eliding the namespace of object reference (Section 4.3) and namespace reference (Section 4.4) values.

For a relative reference value, its namespace portion SHALL start with a relative path segment "." or "..", rather than an authority part, as in the ABNF rule augmentation below. In accordance with [RFC3986] a relative reference SHALL NOT contain a scheme part.

```
; Relative reference to other model in same org as context
nsref-ari =/ "../" model-seg "/"
; Relative reference to context namespace
nsref-ari =/ "./"
```

Examples of text name and integer enumerated identifiers in a relative reference are below.

```
./CTRL/do-thing
./CTRL/otherobj(%22a%20param%22,/UINT/10)
./-2/30
```

Relative reference resolution is discussed in Section 6.3.

5. ARI Binary Form

This section defines how the data model explained in Section 3 is encoded as a binary sequence conforming to the CBOR syntax of [RFC8949]. Within this section the term "item" is used to mean the CBOR-decoded data item which follows the logical model of CDDL [RFC8610].

The binary form of the URI is intended to be used for machine-to-machine interchange so it is missing some of the human-friendly shortcut features of the ARI text form from Section 4. It still follows the same logical data model so it has a one-for-one representation of all of the styles of text-form ARI.

While this text description is normative, the CDDL schema in this section provides a more explicit and machine-parsable binary schema.

```
ari = lit-ari / objref-ari / nsref-ari / relref-ari
```

5.1. Intermediate CBOR

The CBOR item form is used as an intermediate encoding between the ARI data and the ultimate binary encoding. When decoding a binary form ARI, the CBOR must be both "well-formed" according to [RFC8949] and "valid" according to the CDDL model of this specification. Implementations are encouraged, but not required, to use a streaming form of CBOR encoder/decoder to reduce memory consumption of an ARI handler. For simple implementations or diagnostic purposes, a two stage conversion between ARI--CBOR and CBOR--binary can be more easily understood and tested.

5.2. Literals

Based on the structure of Section 3.2, the binary form of the literal ARI contains a data item along with an optional AMM literal type identifier. In order to keep the encoding as short as possible, the untyped literal is encoded as the simple value itself. Because the typed literal and the object reference forms uses CBOR array framing, this framing is used to disambiguate from the pure-value encoding of the lit-notype CDDL symbol.

When present, the literal type SHALL be an integer enumeration. When typed, the decoded literal value SHALL be a valid CBOR item conforming to the AMM literal type definition of the \$lit-typeval CDDL socket. When untyped, the decoded literal value SHALL be one of the primitive types named by the lit-notype CDDL symbol.

The literal ARI, both typed and untyped, has the following CDDL definition.

```
lit-ari = lit-typeval / lit-notype

; undefined value is only allowed as non-typed literal
lit-notype = undefined / null / bool / (int64 / uint64)
            / float / tstr / bstr

lit-typeval = $lit-typeval .within lit-typeval-struct
lit-typeval-struct = [
    lit-type: lit-type-id,
    lit-value: any
]
lit-type-id = (int32 .ge 0)
```

The typed value encodings are organized by AMM type in the following CDDL plug definitions.

NULL:

This type contains only the single null value as in the following plug.

```
$lit-typeval /= [0, null]
```

BOOL:

This type contains the two fixed special values as in the following plug, which uses a base CDDL rule.

```
$lit-typeval /= [1, bool]
```

BYTE, INT, UINT, VAST, or UVAST:

The integer types, signed or unsigned, match the following plugs.

```
$lit-typeval /= [2, uint .size 1] ; 1-byte
$lit-typeval /= [4, int32] ; 4-byte
$lit-typeval /= [5, uint32] ; 4-byte
$lit-typeval /= [6, int64] ; 8-byte
$lit-typeval /= [7, uint64] ; 8-byte
```

REAL32 or REAL64:

The floating-point types match the following plugs, which use base CDDL rules.

```
$lit-typeval /= [8, float16 / float32]
$lit-typeval /= [9, float16 / float32 / float64]
```

TEXTSTR or BYTESTR:

The text string and byte string types match the following plugs, which use base CDDL rules. No quoting or escaping is necessary to distinguish values in these types.

```
$lit-typeval /= [10, tstr]
$lit-typeval /= [11, bstr]
```

TP or TD:

The time point and time difference types match the following plugs, which use a common rule for time-related values. The lit-time rule allows expressing a time offset as either integer seconds or a decimal fraction. The time-fraction rule uses the same structure as CBOR tag #4 "decimal fraction" but limited in domain for time values.

```
; Absolute timestamp as seconds from DTN Epoch
$lit-typeval /= [12, lit-time]
; Relative time interval as seconds
$lit-typeval /= [13, lit-time]

lit-time = int / time-fraction
time-fraction = [
  exp: (-9 .. 9) .within int,
  mantissa: int64,
]
```

The canonical encoded form for time offset uses the integer variation if it results in a smaller encoded size. The canonical encoded form for decimal fraction uses a minimum-size mantissa (i.e., its decimal form has no trailing zeros).

LABEL:

This type uses the following plug rule.

```
$lit-typeval /= [14, lit-label]
lit-label = id-int / id-text
```

CBOR:

This type uses the following plug rule, which explicitly constrains the byte string well-formed-ness.

```
$lit-typeval /= [15, lit-cbor]
lit-cbor = bstr .cbor any
```

ARITYTYPE:

This type uses the same rule as the LABEL but with a context-independent interpretation as matching one of the built-in literal types. This means the value can either be the integer enumeration of the type or its text name. Processors will treat either form the same if they support the associated built-in type.

```
$lit-typeval /= [16, lit-label]
```

OBJPAT:

This type uses the following plug for its value to match sets of objects using each of their identifiers as described in Section 3.2.1. The range encoding uses a compressed form of interval logic to encode the least included value and only the widths of each subsequent included and excluded interval.

```
$lit-typeval /= [24, lit-objpat]
lit-objpat = [
  org-id: objpat-item,
  model-id: objpat-item,
  type-id: objpat-item,
  obj-id: objpat-item
]
objpat-item = objpat-wildcard / objpat-single / objpat-range

objpat-wildcard = true
objpat-single = id-int / id-text

; Intervals are in-order because widths cannot be negative
objpat-range = [
  ; least included value (or domain minimum)
  least: id-int / null,
  * (
    ; width of included interval
    incl: id-width
    ; width of excluded interval
    excl: id-width
  ),
  ; Width of last included interval (or domain maximum)
  incl: id-width / null
]
id-width = uint32
```

AC:

This type uses the following plug for its value. Each item of the array is itself an ARI value (not byte string wrapped).

```
$lit-typeval /= [17, ari-collection]
ari-collection = [*ari]
```

AM:

This type uses the following plug for its value. Each pair of the map is itself composed of ARI values, where keys are restricted to untyped literals.

```
$lit-typeval /= [18, ari-map]
ari-map-key = lit-notype
ari-map = { *ari-map-key => ari }
```

TBL:

This type uses the following plug for its value. The first item of the array indicates the number of columns in the table, all of the subsequent items is a concatenation of all rows in the table (_i.e._ a row-major flattened form).

```
$lit-typeval /= [19, ari-tbl]
ari-tbl = [
  ncol: uint, ; Number of columns in the table
  *ari
]
```

EXECSET:

This type uses the following plug for its value, where named fields correspond with the logical structure defined in the AMM.

```
$lit-typeval /= lit-execset
lit-execset = [20, exec-set]
exec-set = [
  nonce,
  exec-targets,
]
nonce = null / uint / bstr
; Valid target types are defined by the AMM
exec-targets = (*ari)
```

RPTSET:

This type uses the following plug for its value, where named fields correspond with the logical structure defined in the AMM. The ref-time field is interpreted as a absolute time using the same encoding as TP. The rel-time field is interpreted as a difference relative to ref-time using the same encoding as TD.

```
$lit-typeval /= lit-rptset
lit-rptset = [21, rpt-set]
rpt-set = [
  nonce,
  ref-time: lit-time,
  * rpt-container
]
rpt-container = [
  rel-time: lit-time,
  source: objref-ari,
  rpt-items
]
rpt-items = (* ari)
```

The canonical encoded form of the reports list is ordered by increasing rel-time field.

Some example of the forms for a literal are below. In all cases, the use of CBOR EDN means that the symbol case and numeric base are artifacts of the EDN and not the CBOR items being used as examples.

These first are untyped primitive values in a sequence:

```
undefined,
null,
true,
1.1,
NaN,
10,
'bytes',
"text"
```

And these are typed values in a sequence:

```
[0, null],
[1, true],
[2, 10],
[4, 10],
[5, 10],
[6, 10],
[7, 10],
[8, 1e10],
[12, 725943845],
[13, [2, 36]],
[14, "name"],
[15, <<10>>],
[16, 4]
```

The following reference patterns are equivalent to the examples in Section 4.2, where the first example uses an indefinite endpoint and wildcard to compress the ranges.

```
[24, [
  65535,
  [null, 2147483647, 1, 1], / for (..-1,1) /
  true, / for (*) /
  [10, 90] / for (10..100) /
]],
[24, [
  65535,
  [-2147483648, 2147483647, 1, 1], / for (..-1,1) /
  [-2147483648, 4294967295], / for (-2147483648..2147483647) /
  [10, 90] / for (10..100) /
]]
```

These are typed container values in binary form equivalent to the examples in Section 4.2.

```
[17, [1, 2, 3]],
[18, {1: 2, 2: 4, 3: 9}],
[19, [3, 1, true, "A", 2, false, "B"]]
```

5.3. Object References

Based on the structure of Section 3.3, the binary form of the object reference ARI is a CBOR-encoded item. An ARI SHALL be encoded as a CBOR array with between four and six items. The first items correspond to the organization ID, model ID, optional model revision, object type, and object ID. Those items together are referred to as the object identifier. The optional last item of the array is the parameter list.

Each of the organization ID, model ID, object type, and object ID of an object identifier SHALL be represented as either text name id-text or integer enumeration id-int form or the null value to indicate its absence.

The model revision SHALL be represented as a CBOR-tagged text date in accordance with [RFC8943]. The presence or absence of the model revision can be detected by decoders due to its mandatory CBOR tag 1004; the following item (object type) is always untagged. ARI processors MAY decompose and integer-decode the model revision date in order to optimize storage or comparisons. For example, when matching ADM revisions during an ARI dereference activity per Section 6.3 of [I-D.ietf-dtn-amm]. Any model revision date handling SHALL NOT affect the text representation if re-encoding the ARI.

Because the date format does not allow variation of text encoding, this is easily satisfied and there is no need to preserve the original text value if decoding.

When present, the parameters SHALL be either the ari-collection or ari-map structure. In other words, just the value-portion of the AC or AM typed literal because no other disambiguation needs to be made for the parameter type.

The object reference ARI has the following CDDL definition.

```
; Type-agnostic object reference
objref-ari = $objref-ari .within ref-ari-struct

; General structure of any reference
ref-ari-struct = [
  org-id / null,
  (
    model-id,
    ? model-rev
  ) / null,
  obj-type-id / null,
  obj-id / null,
  ?params
]

; Identifier for a single object
obj-ident<obj-type> = (
  org-id,
  model-id,
  ? model-rev,
  obj-type,
  obj-id,
)

org-id = id-int / id-text
model-id = id-int / id-text
model-rev = #6.1004(tstr) ; from RFC 8943
obj-type-id = (id-int .lt 0) / id-text
obj-id = (id-int .ge 0) / id-text

params = ari-collection / ari-map

; Generic usable for restricting objref-ari by type
objref-type<obj-type> = [
  obj-ident<obj-type>,
  ?params
]
```



```
; IANA-assigned object types
IDENT = -1
$objref-ari /= objref-type<IDENT>
CONST = -2
$objref-ari /= objref-type<CONST>
CTRL = -3
$objref-ari /= objref-type<CTRL>
EDD = -4
$objref-ari /= objref-type<EDD>
OPER = -6
$objref-ari /= objref-type<OPER>
SBR = -8
$objref-ari /= objref-type<SBR>
TBR = -10
$objref-ari /= objref-type<TBR>
VAR = -11
$objref-ari /= objref-type<VAR>
TYPEDEF = -12
$objref-ari /= objref-type<TYPEDEF>
```

An example object reference without parameters is:

```
[65535, 1, -1, 0]
```

Another example object reference with parameters is:

```
[65535, 1, -2, 3, ["a param", [4, 10]]]
```

5.4. Namespace References

Based on the structure of Section 3.4, the binary form of the namespace reference ARI is a specialization of the object reference encoding as defined in the following CDDL.

```
nsref-ari = [
  org-id,
  model-id,
  ? model-rev,
  null,
  null
] .within ref-ari-struct
```

Some examples of namespace reference values are below, both with text form and integer form identifiers. For equivalence, model "adm-a" has enumeration 1 and "!odm-b" has enumeration -20.

```
[ "example", "adm-a", 1004("2024-06-25"), null, null],  
[ 65535, 1, 100(19899), null, null],  
[ "example", "adm-a", null, null],  
[ 65535, 1, null, null],  
[ "example", "!odm-b", null, null],  
[ 65535, -20, null, null]
```

5.5. Relative References

The binary form of ARI includes the ability to encode a relative reference only for the specific case of eliding the namespace of Object Reference values. In this case the organization ID, and possibly model ID, portions of an object identifier are replaced by the CBOR value null as defined below. If the model ID is absent, there SHALL NOT be a model revision present.

These restrictions are captured in the following CDDL.

```
relref-ari = [  
  null,  
  (  
    model-id,  
    ? model-rev,  
  ) / null,  
  obj-type-id,  
  obj-id,  
  ?params  
] .within ref-ari-struct
```

Examples of text name and integer enumerated identifiers in a relative reference are below.

These example corresponds to the URI references ./CTRL/do-thing and ./-2/30 respectively.

```
[null, null, "CTRL", "do-thing"],  
[null, null, -2, 30]
```

These example corresponds to the URI references ../!odm10/var/threshold and ../-10/-11/2 respectively.

```
[null, "!odm10", "var", "threshold"],  
[null, -10, -11, 2]
```

Relative reference resolution is discussed in Section 6.3.

6. Processing Activities

Separate from the forms of encoding the ARI information, there are some ARI-specific processing activities that can be used to transform one ARI value into another without changing the higher-level meaning of the value. Both of these activities require an external context to work in a meaningful way.

6.1. ID Segment Translation

Both the text and binary form of the ARI allow ID segments to contain either a text name or an integer enumeration. ARIs can be translated one-for-one between the two forms without loss.

When translating literal types into text form and code point lookup tables are available, the literal type SHOULD be converted to its text name. When translating literal types from text form and code point lookup tables are available, the literal type SHOULD be converted from its text name. The conversion between AMM literal type name and enumeration requires a lookup table based on the registrations in Table 2.

The LABEL literal type offers a choice of text name or integer enumeration forms. The conversion between these LABEL value forms requires a lookup table based on the context in which the value is being used. LABEL values used as value-producing object given parameters refer to names or ordinals of the corresponding formal parameters. LABEL values used within the table filtering control refer to names or ordinals of corresponding table template columns.
// TBD other contexts?

When translating object references into text form and code point lookup tables are available, any enumerated item SHOULD be converted to its text name. When translating object references from text form and code point lookup tables are available, any enumerated item SHOULD be converted from its text name. The conversion between AMM object-type text name and integer enumeration requires a lookup table based on the registrations in Table 3. The conversion between text name and integer enumeration for either organization ID, model ID, or object ID require lookup tables based on ADMs and ODMs known to the processing entity.

ID segment translation SHALL apply recursively to any nested values in literal containers or in object reference parameters.

6.2. Parameter Key Translation

Both the text and binary form of the ARI allow object reference parameters to be structured as either a list or a map. When parameters are present as a map the map keys can be either text name or an integer ordinal of a formal parameter. Parameter map can be translated one-for-one between the two forms without loss.

When translating object reference parameter maps into text form and code point lookup tables are available, any integer map key SHOULD be converted to its text name. When translating object reference parameter maps from text form and code point lookup tables are available, any integer map key SHOULD be converted from its text name. The conversion between text name and integer ordinal for parameter map keys require lookup tables based on formal parameters within ADMS and ODMs known to the processing entity.

Parameter key translation SHALL apply recursively to any nested values in literal containers or in object reference parameters.

6.3. Relative Reference Resolution

Resolving an ARI containing a relative reference (where an object reference namespace is elided) SHALL consist of replacing the namespace component with the namespace of the resolving context. Relative reference resolution SHALL apply recursively to any nested values in literal containers or in object reference parameters.

Because relative references can only exist in a specific named context (*_e.g._*, an ADM module), resolution occurs at the point of decoding and extracting ARI values from that context. The AMM [I-D.ietf-dtn-amm] gives more specifics of how and when resolution is handled by an ARI user.

7. Transcoding Considerations

When translating literal values into text form, it is necessary to canonicalize the CBOR extended diagnostic notation of the item. The following applies to generating text form from CBOR items:

- * Only the outermost value SHALL contain the "ari:" scheme prefix. This applies to cases of either literal container types or object reference parameter values.
- * The canonical presentation form of CBOR null and bool values SHALL be the names identified in Section 8 of [RFC8949].

- * The canonical presentation form of CBOR int and float values SHALL be the decimal encoding defined in Section 8 of [RFC8949].
- * The canonical presentation form of CBOR tstr values SHALL be the definite-length, non-concatenated encoding defined in Section 8 of [RFC8949].
- * The canonical presentation form of CBOR bstr values SHALL be the definite-length, base16 ("h" prefix), non-concatenated encoding defined in Section 8 of [RFC8949].
- * The canonical presentation form for literal type TP values SHALL be the human-friendly date-time encoding.
- * The canonical presentation form for literal type TD values SHALL be the human-friendly duration encoding.
- * When canonical presentation form for literal type CBOR values SHALL be the embedded CBOR encoding defined in Appendix G.3 of [RFC8610].

8. Interoperability Considerations

DTN challenged networks might interface with better resourced networks that are managed using non-DTN management protocols. When this occurs, the federated network architecture might need to define management gateways that translate between DTN and non-DTN management approaches.

| NOTE: It is also possible for DTN management be used end-to-end
| because this approach can also operate in less challenged
| networks. The opposite is not true; non-DTN management
| approaches should not be assumed to work in DTN challenged
| networks.

Where possible, ARIs should be translatable to other, non-DTN management naming schemes. This translation might not be one-to-one, as the features of the AMM and/or specific ADM may differ from features in other management architectures. Therefore, it is unlikely that a single naming scheme can be used for both DTN and non-DTN management.

8.1. ARI Type Support

As recommended in Section 3.2.5 of [I-D.ietf-dtn-amm], an implementation can make use of the private and experimental reserved code points for Literal Types (Table 2) and Reference Types (Table 3). In practice this requires all receivers of these ARIs to implement those private use built-in types so causes a high barrier to interoperability. Private use types are discouraged when the application need can be handled using a combination of well-known built-in types and semantic type definitions in some data model implemented on the Agent(s) and known to the Manager(s).

8.2. ARI Transport

It is expected that the principal form of exchange between Manager and Agent uses the purpose-built message form defined in [I-D.ietf-dtn-amp]. Beyond that principal interface there are likely to be more implementation-specific or ad-hoc mechanisms used to transport ARIs. Some of these could be between a Manager and its application(s) or between an Agent and its application(s). This section contains several interoperable recommendations for how to transport ARIs as encoded data between these entities.

URI List: This transport form uses text form ARIs separated as text lines following an existing file format. This form of encoding SHOULD be referred to by applications as the "uri" form.

When using this form, each ARI value SHALL be URI-encoded in accordance with Section 4, using the "ari:" scheme prefix to make it a proper non-reference URI, and appended by a CRLF line ending. Because of the restricted character set of encoded URIs, implementations MAY be more lax about decoding other line endings and still be interoperable. The encoded text MAY include non-URI comment lines as defined in Section 5 of [RFC2483].

When transporting this form using a mechanism that supports media type identification, the type "text/uri-list" SHOULD be used.

CBOR Sequence: This transport form uses binary form ARIs concatenated together into a single octet sequence. This form of encoding SHOULD be referred to by applications as the "cbor" form.

When using this form, each ARI value SHALL be CBOR-encoded in accordance with Section 5 and concatenated together into a CBOR sequence as defined in [RFC8742]. No other special considerations are needed for this form.

When transporting this form using a mechanism that supports media type identification, the type "application/cbor-seq" SHOULD be used. When transporting a single ARI value, the type "application/cbor" MAY be used instead to indicate the single encoded item.

CBOR Hexadecimal List: This transport form uses binary form ARIs which are then base16 encoded and separated as text lines. This form of encoding SHOULD be referred to by applications as the "cborhex" form.

When using this form, each ARI value SHALL be CBOR-encoded in accordance with Section 5, then base16 encoded in accordance with Section 8 of [RFC4648], and appended by a CRLF line ending. Each base16 encoded line MAY be prefixed by "0x" to disambiguate this form from the other two defined in this document.

If the transport mechanism supports media type identification, the type "text/plain" SHOULD be used.

The above are simply recommendations for better interoperability using existing, well-known media types. Implementations can use any additional media types at their own discretion.

| The URI List form and CBOR hexadecimal list form are equivalent
| to the how the examples in this document are presented. The
| CBOR hexadecimal list, when the "0x" prefix is not used, is
| also compatible with tools such as <https://cbor.me> and similar
| command-line converters.

9. Security Considerations

Because ADM and ODM namespaces are defined by any entity, no security or permission meaning can be inferred simply from the expression of namespace.

10. IANA Considerations

This section provides guidance to the Internet Assigned Numbers Authority (IANA) regarding registration of schema and namespaces related to ARIs, in accordance with BCP 26 [RFC8126].

10.1. URI Schemes Registry

This document defines a new URI scheme "ari" in Section 4. A new entry has been added to the "URI Schemes" registry [IANA-URI] with the following parameters.

Scheme name:
ari

Status:
Permanent

Applications/protocols that use this scheme name:
The scheme is used by DTNMA Managers and Agents to identify managed objects.

Contact:
IETF Chair <chair@ietf.org>

Change controller:
IETF <ietf@ietf.org>

Reference:
Section 4 of [This document].

10.2. DTN Management Architecture

This document defines several new registries within a new "DTN Management Architecture" registry group.

This document defines a new registry "DTNMA Literal Types" within the "DTN Management Architecture" registry group [IANA-DTNMA] containing initial entries from Table 2. Enumerations in this registry SHALL be non-negative integers representable as CBOR uint type with an argument shorter than 4-bytes. Names in this registry SHALL be unique among all entries in this and the "DTNMA Object Types" registry. The registration procedure for this registry is Specification Required.

Enumeration	Name	Description	Reference
0	NULL	The singleton null value.	[This document]
1	BOOL	A native boolean true or false value.	[This document]
2	BYTE	An 8-bit unsigned integer. The valid domain is 0 to 2 ⁸ -1 inclusive.	[This document]
4	INT	A 32-bit signed integer.	[This

		The valid domain is -2^{31} to $2^{31}-1$ inclusive	document]
5	UINT	A 32-bit unsigned integer. The valid domain is 0 to $2^{32}-1$ inclusive	[This document]
6	VAST	A 64-bit signed integer. The valid domain is -2^{63} to $2^{63}-1$ inclusive	[This document]
7	UFAST	A 64-bit unsigned integer. The valid domain is 0 to $2^{64}-1$ inclusive	[This document]
8	REAL32	A 32-bit [IEEE.754-2019] floating point number.	[This document]
9	REAL64	A 64-bit [IEEE.754-2019] floating point number.	[This document]
10	TEXTSTR	A text string composed of (unicode) characters in accordance with [RFC3629].	[This document]
11	BYTESTR	A byte string composed of 8-bit values.	[This document]
12	TP	An absolute time point (TP) in the DTN Epoch of Section 4.2.6 of [RFC9171].	[This document]
13	TD	A relative time difference (TD) with a sign.	[This document]
14	LABEL	A text label of a parent object parameter. This is only valid in a nested parameterized ARI.	[This document]
15	CBOR	A byte string containing an encoded CBOR item. The structure is opaque to the Agent but guaranteed well-	[This document]

		formed for the ADM using it.	
16	ARITYPE	An integer or text value representing one of the code points in this DTNMA Literal Types registry or the DTNMA Object Types registry.	[This document]
17	AC	An array containing an ordered list of ARIs.	[This document]
18	AM	A map containing keys of primitive ARIs and values of ARIs.	[This document]
19	TBL	A two-dimensional table containing cells of ARIs.	[This document]
20	EXECSET	A structure containing values to be executed by an Agent.	[This document]
21	RPTSET	A structure containing reports of values sampled from an Agent.	[This document]
22 to 254		_Unassigned_	
255	LITERAL	A reserved type name for the union of all possible literal types.	[This document]
256 to 64383		_Unassigned_	
64384 to 64511		Reserved for experimental use	[This document]
64512 to 65535		Reserved for private use	[This document]
65536 to 2147483647		Reserved for future use	[This document]

Table 2: DTNMA Literal Types

This document defines a new registry "DTNMA Object Types" within the "DTN Management Architecture" registry group [IANA-DTNMA] containing initial entries from Table 3. Enumerations in this registry SHALL be negative integers representable as CBOR nint type with an argument shorter than 4-bytes. Names in this registry SHALL be unique among all entries in this and the "DTNMA Literal Types" registry. The registration procedure for this registry is Specification Required.

Enumeration	Name	Description	Reference
__-1__	IDENT	Identity Object	[This document]
__-2__	CONST	Constant	[This document]
__-3__	CTRL	Control	[This document]
__-4__	EDD	Externally Defined Data	[This document]
__-6__	OPER	Operator	[This document]
__-8__	SBR	State-Based Rule	[This document]
__-10__	TBR	Time-Based Rule	[This document]
__-11__	VAR	Variable	[This document]
__-12__	TYPEDDEF	Named semantic type	[This document]
-13 to -254		_Unassigned_	
__-255__	NAMESPACE	A reserved type name for any namespace reference.	[This document]
__-256__	OBJECT	A reserved type name for the union of all possible object reference types.	[This document]
-257 to		_Unassigned_	

-64384			
+-----+	+-----+	+-----+	+-----+
-64385 to		Reserved for experimental	[This
-64512		use	document]
+-----+	+-----+	+-----+	+-----+
-64513 to		Reserved for private use	[This
-65536			document]
+-----+	+-----+	+-----+	+-----+
-65537 to		Reserved for future use	[This
-2147483648			document]
+-----+	+-----+	+-----+	+-----+

Table 3: DTNMA Object Types

IANA has annotated the "DTNMA Literal Types" and "DTNMA Object Types" registries with the following note:

```
| The enumeration and name of all entries managed by IANA are unique
| across both the "DTNMA Literal Types" and "DTNMA Object Types"
| registries.
```

This document defines a new registry "DTNMA Organizations" within the "DTN Management Architecture" registry group [IANA-DTNMA] containing initial entries from Table 5. Enumerations in this registry are 32-bit signed integers. Names in this registry are text containing only lower-case alphabetic and numeric characters, a subset of the id-text ABNF rule. The registration procedures for this registry are indicated in Table 4.

+=====+	+=====+
Enumeration Range	Registration Procedure
+=====+	+=====+
-2147483648 to 0	Reserved
+-----+	+-----+
1 to 65535	Expert Review
+-----+	+-----+
65536 to 2147483647	First Come First Served
+-----+	+-----+

Table 4: DTNMA Organizations Registration Procedures

The expert review range of this registry covers the values which have smaller encoding in binary form (all 3 octets or fewer). Experts are encouraged to be biased towards approving registrations unless they are abusive, frivolous, duplicative, or actively harmful (not merely of dubious value). Registrants are encouraged to use the first-come-first-served range if a large number of organizations are needed by one registrant.

Enumeration	Name	Contact	Notes
-2147483648 to -2147483777			Reserved for Experimental Use
-2147483776 to -1			Reserved for Private Use
0			Reserved
1	ietf	IETF Chair <chair@ietf.org>	All IETF standardized models will use this organization
2	iana	IETF Chair <chair@ietf.org>	All IANA registry models will use this organization
3 to 65534			_Unassigned_
65535	example	IETF Chair <chair@ietf.org>	Reserved for example ADMs
65536 to 2147483647			_Unassigned_

Table 5: DTNMA Organizations

This document defines a new registry "DTNMA IETF Data Models" within the "DTN Management Architecture" registry group [IANA-DTNMA] containing initial entries from Table 6. The purpose of this registry is to hold models defined by IETF published documents. All of the ADMs in this registry are under the "ietf" organization (enumeration 1) from Table 5. Enumerations in this registry are 32-bit signed integers. The registration procedure for this registry is standards action because these models are managed by the IETF.

Enumeration	Name	Reference	Notes
-2147483648 to -1			Reserved for ODMs
0 to 2147483647			_Unassigned_

Table 6: DTNMA IETF Data Models

This document defines a new registry "DTNMA IANA Data Models" within the "DTN Management Architecture" registry group [IANA-DTNMA] containing initial entries from Table 6. The purpose of this registry is to hold models containing object definitions which are purely ADM-defined, and involve no runtime variability, such as IDENT and TYPEDEF objects. All of the ADMs in this registry are under the "iana" organization (enumeration 2) from Table 5. Enumerations in this registry are 32-bit signed integers. The registration procedure for this registry is standards action because these models are managed by IANA.

Enumeration	Name	Reference	Notes
-2147483648 to -1			Reserved for ODMs
0 to 2147483647			_Unassigned_

Table 7: DTNMA IANA Data Models

11. References

11.1. Normative References

[IANA-DTNMA]

IANA, "Delay-Tolerant Networking Management Architecture (DTNMA) Parameters",
[<https://www.iana.org/assignments/TBA/>](https://www.iana.org/assignments/TBA/).

[IANA-URI] IANA, "Uniform Resource Identifier (URI) Schemes",

[<https://www.iana.org/assignments/uri-schemes/>](https://www.iana.org/assignments/uri-schemes/).

[IEEE.754-2019]

IEEE, "IEEE Standard for Floating-Point Arithmetic",
 IEEE IEEE 754-2019, DOI 10.1109/IEEESTD.2019.8766229, 18
 July 2019, <https://ieeexplore.ieee.org/document/8766229>.

- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<https://www.rfc-editor.org/info/rfc2119>>.
- [RFC2483] Mealling, M. and R. Daniel, "URI Resolution Services Necessary for URN Resolution", RFC 2483, DOI 10.17487/RFC2483, January 1999, <<https://www.rfc-editor.org/info/rfc2483>>.
- [RFC3339] Klyne, G. and C. Newman, "Date and Time on the Internet: Timestamps", RFC 3339, DOI 10.17487/RFC3339, July 2002, <<https://www.rfc-editor.org/info/rfc3339>>.
- [RFC3629] Yergeau, F., "UTF-8, a transformation format of ISO 10646", STD 63, RFC 3629, DOI 10.17487/RFC3629, November 2003, <<https://www.rfc-editor.org/info/rfc3629>>.
- [RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, DOI 10.17487/RFC3986, January 2005, <<https://www.rfc-editor.org/info/rfc3986>>.
- [RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, DOI 10.17487/RFC4648, October 2006, <<https://www.rfc-editor.org/info/rfc4648>>.
- [RFC5234] Crocker, D., Ed. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", STD 68, RFC 5234, DOI 10.17487/RFC5234, January 2008, <<https://www.rfc-editor.org/info/rfc5234>>.
- [RFC7595] Thaler, D., Ed., Hansen, T., and T. Hardie, "Guidelines and Registration Procedures for URI Schemes", BCP 35, RFC 7595, DOI 10.17487/RFC7595, June 2015, <<https://www.rfc-editor.org/info/rfc7595>>.
- [RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <<https://www.rfc-editor.org/info/rfc8174>>.
- [RFC8259] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", STD 90, RFC 8259, DOI 10.17487/RFC8259, December 2017, <<https://www.rfc-editor.org/info/rfc8259>>.

- [RFC8742] Bormann, C., "Concise Binary Object Representation (CBOR) Sequences", RFC 8742, DOI 10.17487/RFC8742, February 2020, <<https://www.rfc-editor.org/info/rfc8742>>.
- [RFC8943] Jones, M., Nadalin, A., and J. Richter, "Concise Binary Object Representation (CBOR) Tags for Date", RFC 8943, DOI 10.17487/RFC8943, November 2020, <<https://www.rfc-editor.org/info/rfc8943>>.
- [RFC8949] Bormann, C. and P. Hoffman, "Concise Binary Object Representation (CBOR)", STD 94, RFC 8949, DOI 10.17487/RFC8949, December 2020, <<https://www.rfc-editor.org/info/rfc8949>>.
- [RFC9171] Burleigh, S., Fall, K., and E. Birrane, III, "Bundle Protocol Version 7", RFC 9171, DOI 10.17487/RFC9171, January 2022, <<https://www.rfc-editor.org/info/rfc9171>>.
- [I-D.ietf-dtn-amm]
Birrane, E. J., Sipos, B., and J. Ethier, "DTNMA Application Management Model (AMM) and Data Models", Work in Progress, Internet-Draft, draft-ietf-dtn-amm-05, 3 July 2025, <<https://datatracker.ietf.org/doc/html/draft-ietf-dtn-amm-05>>.

11.2. Informative References

- [IEEE.1003.1-2024]
IEEE, "IEEE/Open Group Standard for Information Technology--Portable Operating System Interface (POSIX) Base Specifications, Issue 8", IEEE 1003-1-2024, DOI 10.1109/IEEESTD.2024.10555529, 12 June 2024, <<https://ieeexplore.ieee.org/document/10555529>>.
- [ISO.9899-1999]
ISO, "Programming Languages -- C", ISO/IEC 9899:1999, December 1999, <<https://www.iso.org/standard/29237.html>>.
- [RFC4838] Cerf, V., Burleigh, S., Hooke, A., Torgerson, L., Durst, R., Scott, K., Fall, K., and H. Weiss, "Delay-Tolerant Networking Architecture", RFC 4838, DOI 10.17487/RFC4838, April 2007, <<https://www.rfc-editor.org/info/rfc4838>>.
- [RFC7320] Nottingham, M., "URI Design and Ownership", RFC 7320, DOI 10.17487/RFC7320, July 2014, <<https://www.rfc-editor.org/info/rfc7320>>.

- [RFC7942] Sheffer, Y. and A. Farrel, "Improving Awareness of Running Code: The Implementation Status Section", BCP 205, RFC 7942, DOI 10.17487/RFC7942, July 2016, <<https://www.rfc-editor.org/info/rfc7942>>.
- [RFC8126] Cotton, M., Leiba, B., and T. Narten, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 8126, DOI 10.17487/RFC8126, June 2017, <<https://www.rfc-editor.org/info/rfc8126>>.
- [RFC8610] Birkholz, H., Vigano, C., and C. Bormann, "Concise Data Definition Language (CDDL): A Notational Convention to Express Concise Binary Object Representation (CBOR) and JSON Data Structures", RFC 8610, DOI 10.17487/RFC8610, June 2019, <<https://www.rfc-editor.org/info/rfc8610>>.
- [RFC8792] Watsen, K., Auerswald, E., Farrel, A., and Q. Wu, "Handling Long Lines in Content of Internet-Drafts and RFCs", RFC 8792, DOI 10.17487/RFC8792, June 2020, <<https://www.rfc-editor.org/info/rfc8792>>.
- [RFC8820] Nottingham, M., "URI Design and Ownership", BCP 190, RFC 8820, DOI 10.17487/RFC8820, June 2020, <<https://www.rfc-editor.org/info/rfc8820>>.
- [RFC9675] Birrane, III, E., Heiner, S., and E. Annis, "Delay-Tolerant Networking Management Architecture (DTNMA)", RFC 9675, DOI 10.17487/RFC9675, November 2024, <<https://www.rfc-editor.org/info/rfc9675>>.
- [I-D.ietf-dtn-amp]
Birrane, E. J. and B. Sipos, "DTNMA Asynchronous Management Protocol (AMP)", Work in Progress, Internet-Draft, draft-ietf-dtn-amp-03, 15 January 2026, <<https://datatracker.ietf.org/doc/html/draft-ietf-dtn-amp-03>>.
- [github-dtnma-ace]
JHU/APL, "The DTNMA AMM CODEC Engine (ACE)", <<https://github.com/JHUAPL-DTNMA/dtnma-ace>>.
- [github-dtnma-tools]
JHU/APL, "A reference implementation of the DTN Management Architecture (DTNMA) Agent and related Tools", <<https://github.com/JHUAPL-DTNMA/dtnma-ace>>.

Appendix A. Example Equivalences

This section contains examples of converting between text name and integer enumeration of ARI components and converting between text form and binary form of entire ARIs. The examples in this section rely on the ADM and ODM definitions in Table 8 under the well-known "example" organization from Table 5 and a private use organization "!private" with an enumeration -40.

Organization Name	Model Enumeration	Model Name
example	1	adm-a
example	2	adm-b
example	-10	!odm10
!private	30	adm-a

Table 8: Example ADMs

Given those namespaces, the example objects are listed in Table 9 where the Namespace column uses the ARI text form convention.

Organization Name	Model Name	Object Type	Enumeration	Name	Signature
example	adm-a	EDD	3	num-bytes	()
example	adm-a	CTRL	2	do-thing	(AC targets, UINT count)
example	adm-a	CONST	1	rptt-with-param	(ARI var, TEXTSTR text)
example	adm-a	TYPEDEF	1	distance	()
example	!odm10	VAR	2	threshold	()
!private	adm-a	VAR	1	my-counter	()

Table 9: Example Objects

The TYPEDEF distance is defined to be an augmented use of uint with scale of 1.0 and unit of meter.

Each of the following examples illustrate the comparison of ARI forms in different situations, covering the gamut of what can be expressed by an ARI.

A.1. Primitive-Typed Literal

This is the literal value 4 interpreted as a 32-bit unsigned integer. The ARI text (which is identical to its percent-encoded form) is:

```
ari:/UINT/4
```

which is translated to enumerated form:

```
ari:/5/4
```

and converted to CBOR item:

```
[5, 4]
```

and finally to the binary string of:

0x820504

A.2. Timestamp Literal

This is the timestamp "2000-01-01T00:16:40Z" which is DTN Epoch plus 1000 seconds. The ARI text (which is identical to its percent-encoded form) is:

ari:/TP/20000101T001640Z

which is translated to enumerated form:

ari:/12/1000

and converted to CBOR item:

[12, 1000]

and finally to the binary string of:

0x820c1a000f4240

A.3. Semantic-Typed Literal

This is the literal value 20 interpreted as a semantic type distance from adm-a. The ARI text (which is identical to its percent-encoded form) is:

ari://example/adm-a/TYPEDDEF/distance(20)

which is translated to enumerated form:

ari://65535/1/-12/1(20)

and converted to CBOR item:

[65535, 1, -12, 1, [20]]

and finally to the binary string of:

0x8519FFFF012B018114

A.4. Complex CBOR Literal

This is a literal value embedding a complex CBOR structure. The CBOR diagnostic expression being encoded is

<<{"test": [3, 4.5]}>>

The embedded item can be CBOR-encoded to a byte string and percent-encoded, along with a translated type enumeration of:

```
ari:/15/h'A164746573748203F94480'
```

and converted to CBOR item (note the byte string is no longer text-encoded):

```
[15, h'A164746573748203F94480']
```

and finally to the binary string of:

```
0x820F4BA164746573748203F94480
```

A.5. Non-parameterized Object Reference

This is a non-parameterized num-bytes object in the ADM namespace. The ARI text (which is identical to its percent-encoded form) is:

```
ari://example/adm-a/edd/num-bytes
```

which is translated to enumerated form:

```
ari://65535/1/-4/3
```

and converted to CBOR item:

```
[65535, 1, -4, 3]
```

and finally to the binary string of:

```
0x831A000100002303
```

A.6. Parameterized Object Reference

This is an parameterized do-thing object in the ADM namespace. Additionally, the parameters include two relative-path relative references to other objects in the same ADM, which are resolved after text-decoding (see Section 6.3). The ARI text (which is identical to its percent-encoded form) is:

```
ari://example/adm-a/ctrl/do-thing(/AC/(\
    ./edd/num-bytes,\
    ../!odm10/var/threshold,\
    //!private/adm-a/var/my-counter\
),3)
```

which is translated to enumerated and resolved form:

```
ari://65535/1/-3/2(/17/(\  
  //65535/1/-4/3,\  
  //65535/-10/-11/2,\  
  //-40/30/-11/1\  
,3)
```

and converted to CBOR item:

```
[65535, 1, -3, 2, [  
  [17, [  
    [65535, 1, -4, 3],  
    [65535, -10, -11, 2],  
    [-40, 30, -11, 1]  
  ]],  
  3  
]]
```

and finally to the binary string of:

```
0x8519FFFF012202828211838419FFFF0123038419FFFF292A02843827181E2A0103
```

A.7. Recursive Structure with Percent Encodings

This is a complex example having nested ARIs, some with percent-encoding needed. The human-friendly (but not valid URI) text for this case is:

```
ari://example/adm-a/rptt/rptt-with-param("text")
```

which is percent encoded to the real URI:

```
ari://example/adm-a/rptt/rptt-with-param(%22text%22)
```

which is translated to enumerated form:

```
ari://65535/1/-7/1(%22text%22)
```

and converted to CBOR item:

```
[65535, 1, -7, 1, ["text"]]
```

and finally to the binary string of:

```
0x8519FFFF012601816474657874
```

A.8. Full EXECSET and RPTSET with all ARI variations

This section contains fully valid EXECSET and RPTSET

```
ari:/EXECSET/n=1234;(\
//65535/1/CTRL/12(\
  undefined,\
  null,\
  true,\
  false,\
  10,\
  -10,\
  1e+06,\
  hi,\
  %22%24%21%22,\
  h'6869'\
),\
//65535/1/CTRL/123(\
  /NULL/null,\
  /BOOL/true,\
  /BOOL/false,\
  /BYTE/10,\
  /INT/-1000,\
  /UINT/1000,\
  /VAST/-1000000,\
  /UVAST/1000000,\
  /REAL32/1e+10,\
  /REAL64/1e+20,\
  /REAL64/NaN,\
  /REAL64/Infinity,\
  /REAL64/-Infinity,\
  /TEXTSTR/hi,\
  /TEXTSTR/%22%24%21%22,\
  /BYTESTR/'hi',\
  /BYTESTR/h'6869',\
  /BYTESTR/b64'aGk',\
  /TP/20250624T120000.3Z,\
  /TP/804081600.3,\
  /TD/PT1H0.05S,\
  /TD/3600.05,\
  /LABEL/name,\
  /LABEL/1234,\
  /CBOR/h'187B',\
  /ARITYPE/UINT,\
  /OBJPAT/(65535)(..-1,1)(*)(10..100),\
  /OBJPAT/(*)(*)(*)(*),\
  /AC/(2,4),\
  /AM/(1=2,3=4),\

```

```

    /TBL/c=2;(1,2)(3,4)\
  ),\
//65535/1/CTRL/1234(\
  0=//65535/2/IDENT/12,\
  1=//65535/2/TYPDEF/21\
)\
)

```

This has the following CBOR

```

[
  20,
  [
    1234,
    [65535, 1, -3, 12, [
      undefined,
      null,
      true,
      false,
      10,
      -10,
      1e6,
      "hi",
      "$.?!' \t@+:",
      'hi'
    ]],
    [65535, 1, -3, 123, [
      [0, null],
      [1, true],
      [1, false],
      [2, 10],
      [4, -1000],
      [5, 1000],
      [6, -1000000],
      [7, 1000000],
      [8, 1e10],
      [9, 1e20],
      [9, NaN],
      [9, Infinity],
      [9, -Infinity],
      [10, "hi"],
      [10, "$.?!' \t@+:"],
      [11, 'hi'],
      [11, 'hi'],
      [11, 'hi'],
      [12, [-1, 8040816003]],
      [12, [-1, 8040816003]],
      [13, [-2, 360005]],

```



```
[13, [-2, 360005]],  
[14, "name"],  
[14, 1234],  
[15, <<123>>],  
[16, 5],  
[24, [65535, [null,2147483648,1,1], true, [10,90]]],  
[24, [true, true, true, true]],  
[17, [2, 4]],  
[18, {1: 2, 3: 4}],  
[19, [2, 1,2, 3,4]]  
]],  
[65535, 1, -3, 1234, {  
  0: [65535, 2, -1, 12],  
  1: [65535, 2, -12, 21]  
}]  
]  
]
```

```
ari:/RPTSET/n=1234;r=/TP/20250624T120000Z;(\
  t=/TD/PT;s=//65535/1/CTRL/54(2,1);(\
    undefined,\
    null,\
    true,\
    false,\
    10,\
    -10,\
    1e+06,\
    hi,\
    %22%24%21%22,\
    h'6869'\
  ),\
  t=/TD/PT1.234S;s=//65535/1/CTRL/543(2=1);(\
    /NULL/null,\
    /BOOL/true,\
    /BOOL/false,\
    /BYTE/10,\
    /INT/-1000,\
    /UINT/1000,\
    /VAST/-1000000,\
    /UVAST/1000000,\
    /REAL32/1e+10,\
    /REAL64/1e+20,\
    /REAL64/NaN,\
    /REAL64/Infinity,\
    /REAL64/-Infinity,\
    /TEXTSTR/hi,\
    /TEXTSTR/%22%24%21%22,\
    /BYTESTR/'hi',\
    /BYTESTR/h'6869',\
    /BYTESTR/b64'aGk',\
    /TP/20250624T120000.3Z,\
    /TP/804081600.3,\
    /TD/PT1H0.05S,\
    /TD/3600.05,\
    /LABEL/name,\
    /LABEL/1234,\
    /CBOR/h'187B',\
    /ARITYPE/UINT,\
    /AC/(2,4),\
    /AM/(1=2,3=4),\
    /TBL/c=2;(1,2)(3,4)\
  )\
)
```

```
[
  21,
  [
    1234,
    804081600,
    [
      0,
      [65535, 1, -3, 54, [2, 1]],
      undefined,
      null,
      true,
      false,
      10,
      -10,
      1e6,
      "hi",
      "$.?!' \t@+:",
      'hi'
    ],
    [
      [-3, 1234],
      [65535, 1, -3, 543, {2: 1}],
      [0, null],
      [1, true],
      [1, false],
      [2, 10],
      [4, -1000],
      [5, 1000],
      [6, -1000000],
      [7, 1000000],
      [8, 1e10],
      [9, 1e20],
      [9, NaN],
      [9, Infinity],
      [9, -Infinity],
      [10, "hi"],
      [10, "$.?!' \t@+:"],
      [11, 'hi'],
      [11, 'hi'],
      [11, 'hi'],
      [12, [-1, 8040816003]],
      [12, [-1, 8040816003]],
      [13, [-2, 360005]],
      [13, [-2, 360005]],
      [14, "name"],
      [14, 1234],
      [15, <<123>>],
      [16, 5],
```

```
[17, [2, 4]],  
[18, {1: 2, 3: 4}],  
[19, [2, 1, 2, 3, 4]]  
]  
]  
]
```

Appendix B. Implementation Guidance

When implementing text-form ARI decoding as a token lexer and LR(1) parser, such as POSIX lex and yacc tools [IEEE.1003.1-2024], there are some techniques that can avoid pitfalls.

One is to use the generic val-seg rule from Section 4.1 as the pattern to match any URI path segment. This requires multi-phase processing in many cases, where the val-seg rule matches the segment which is then percent-decoded and primitive or context-specific rules are then used to decode the segment further. While the use of percent encoding is discouraged when unnecessary, and for many ARI values will be unnecessary, the generic URI syntax still allows for percent encoding of segments and a robust ARI decoder needs to handle that.

To avoid lexer ambiguities related to recursive ARI structures (which are present in containers and parameters), it is best to combine the entire object identifier of an Object Reference into a single token consisting of slash-separated val-seg segments. When the lexer uses capture length to perform tie-breaking between multiple rules that match an input text this will cause the object identifier rule to be preferred over Literal value rules (will which always contain fewer path segments).

When handling literal values, the text encodings defined in this document agree in many cases with text encodings provided by the C language starting with C99 [ISO.9899-1999] for the following uses.

Signed Integer: This agrees with the `printf()` format `%lld` and the parsing function `strtoll()`. Signed hexadecimal must be encoded with the sign separately.

Unsigned Integer: This agrees with the `printf()` format `%llu` and `%llx` and the parsing function `strtoull()`.

Floating Point: This agrees with the `printf()` formats `%e` `%f` `%g` and `%a` and the parsing function `strtod()`. This includes parsing of non-finite values but encoders might need to handle those values specially.

Decimal Fractions: These are used for TP and TD types. The sign, integer portion, and fractional portion can each be handled as separate components, with the latter two using unsigned integer codecs (see above).

Absolute Times: This is used only for the TP type. The integer portion is are compatible with the `strftime()` and `strptime()` formats of `%Y%m%dT%H%M%S`, with the output having the "Z" zone suffix appended and the input being stripped of optional "-" and ":" delimiters. The fractional portion can be handled the same as for decimal fractions (above).

Relative Times: This is used only for the TD type. There is no direct handling of text form time duration but each separate component can be handled as an unsigned decimal integer or decimal fraction (see above).

Implementation Status

This section is to be removed before publishing as an RFC.

[NOTE to the RFC Editor: please remove this section before publication, as well as the reference to [RFC7942], [github-dtnma-ace], and [github-dtnma-tools].]

This section records the status of known implementations of the protocol defined by this specification at the time of posting of this Internet-Draft, and is based on a proposal described in [RFC7942]. The description of implementations in this section is intended to assist the IETF in its decision processes in progressing drafts to RFCs. Please note that the listing of any individual implementation here does not imply endorsement by the IETF. Furthermore, no effort has been spent to verify the information presented here that was supplied by IETF contributors. This is not intended as, and must not be construed to be, a catalog of available implementations or their features. Readers are advised to note that other implementations can exist.

A full implementation in Python language of the ARI encoding and decoding requirements for both text URI and binary CBOR forms, and for translating between text name and integer enumeration (based on ADM contents) is present in the `apl-fy24` development branch of [github-dtnma-ace]. This repository includes unit test vectors for verifying ARI encoding and decoding.

A full implementation in C11 language of the ARI encoding and decoding requirements for both text URI and binary CBOR forms is present in the `apl-fy24` development branch of [github-dtnma-tools].

This repository includes unit test vectors for verifying ARI encoding and decoding. It also has built-item (executable) testing which interoperates the Python ACE library with REFDA and REFDM executables.

Acknowledgments

Thanks to Justin Ethier of the Johns Hopkins University Applied Physics Laboratory for review and implementation testing of ARI encoder/decoder (CODEC) software.

Authors' Addresses

Edward J. Birrane, III
The Johns Hopkins University Applied Physics Laboratory
11100 Johns Hopkins Rd.
Laurel, MD 20723
United States of America
Phone: +1 443 778 7423
Email: Edward.Birrane@jhuapl.edu

Emery Annis
The Johns Hopkins University Applied Physics Laboratory
Email: Emery.Annis@jhuapl.edu

Brian Sipos
The Johns Hopkins University Applied Physics Laboratory
Email: brian.sipos+ietf@gmail.com