Specification v1.0-beta

Revision 2021-11-13

Overview

UAVCAN is an open lightweight protocol designed for reliable intravehicular communication in aerospace and robotic applications over robust transports. It is created to address the challenge of deterministic on-board data exchange between systems and components of advanced intelligent vehicles.

The name UAVCAN stands for *Uncomplicated Application-level Vehicular Communication And Networking*.

Features:

- Democratic network – no bus master, no single point of failure.
- Publish/subscribe and request/response (RPC[1]) communication semantics.
- Efficient exchange of large data structures with automatic decomposition and reassembly.
- Lightweight, deterministic, easy to implement, and easy to validate.
- Suitable for deeply embedded, resource constrained, hard real-time systems.
- Supports dual and triply modular redundant transports.
- Supports high-precision network-wide time synchronization.
- Provides rich data type and interface abstractions – an interface description language is a core part of the technology which allows deeply embedded sub-systems to interface with higher-level systems directly and in a maintainable manner while enabling simulation and functional testing.
- The specification and high quality reference implementations in popular programming languages are free, open source, and available for commercial use under the permissive MIT license.

License

Disclaimer of warranty

Limitation of liability

[1] Remote procedure call.

Table of contents

List of tables

List of figures

1 Introduction

This is a non-normative chapter covering the basic concepts that govern development and maintenance of the specification.

1.1 Overview

UAVCAN is a lightweight protocol designed to provide a highly reliable communication method supporting publish-subscribe and remote procedure call semantics for aerospace and robotic applications via robust vehicle bus networks. It is created to address the challenge of deterministic on-board data exchange between systems and components of next-generation intelligent vehicles: manned and unmanned aircraft, spacecraft, robots, and cars.

UAVCAN can be approximated as a highly deterministic decentralized object request broker with a specialized interface description language and a highly efficient data serialization format suitable for use in real-time safety-critical systems with optional modular redundancy.

The name UAVCAN stands for *Uncomplicated Application-level Vehicular Computing And Networking*.

UAVCAN is a standard open to everyone, and it will always remain this way. No authorization or approval of any kind is necessary for its implementation, distribution, or use.

The development and maintenance of the UAVCAN specification is governed through the public discussion forum, software repositories, and other resources available via the official website at uavcan.org.

Engineers seeking to leverage UAVCAN should also consult with *The UAVCAN Guide* – a separate textbook available via the official website.

1.2 Document conventions

Non-normative text, examples, recommendations, and elaborations that do not directly participate in the definition of the protocol are contained in footnotes[2] or highlighted sections as shown below.

> Non-normative sections such as examples are enclosed in shaded boxes like this.

Code listings are formatted as shown below. All such code is distributed under the same license as this specification, unless specifically stated otherwise.

```
// This is a source code listing.
fn main() {
    println!("Hello World!");
}
```

A byte is a group of eight (8) bits.

Textual patterns are specified using the standard POSIX Extended Regular Expression (ERE) syntax; the character set is ASCII and patterns are case sensitive, unless explicitly specified otherwise.

Type parameterization expressions use subscript notation, where the parameter is specified in the subscript enclosed in angle brackets: $\texttt{type}_{\texttt{<parameter>}}$.

Numbers are represented in base-10 by default. If a different base is used, it is specified after the number in the subscript[3].

DSDL definition examples provided in the document are illustrative and may be incomplete or invalid. This is to ensure that the examples are not cluttered by irrelevant details. For example, `@extent` or `@sealed` directives may be omitted if not relevant.

1.3 Design principles

Democratic network — There will be no master node. All nodes in the network will have the same communication rights; there should be no single point of failure.

Facilitation of functional safety — A system designer relying on UAVCAN will have the necessary guarantees and tools at their disposal to analyze the system and ensure its correct behavior.

[2] This is a footnote.

[3] E.g., $\mathrm{BADC0FFEE}_{16} = 50159747054$, $10101_2 = 21$.

High-level communication abstractions — The protocol will support publish/subscribe and remote procedure call communication semantics with statically defined and statically verified data types (schema). The data types used for communication will be defined in a clear, platform-agnostic way that can be easily understood by machines, including humans.

Facilitation of cross-vendor interoperability — UAVCAN will be a common foundation that different vendors can build upon to maximize interoperability of their equipment. UAVCAN will provide a generic set of standard application-agnostic communication data types.

Well-defined generic high-level functions — UAVCAN will define standard services and messages for common high-level functions, such as network discovery, node configuration, node software update, node status monitoring, network-wide time synchronization, plug-and-play node support, etc.

Atomic data abstractions — Nodes shall be provided with a simple way of exchanging large data structures that exceed the capacity of a single transport frame[4]. UAVCAN should perform automatic data decomposition and reassembly at the protocol level, hiding the related complexity from the application.

High throughput, low latency, determinism — UAVCAN will add a very low overhead to the underlying transport protocol, which will ensure high throughput and low latency, rendering the protocol well-suited for hard real-time applications.

Support for redundant interfaces and redundant nodes — UAVCAN shall be suitable for use in applications that require modular redundancy.

Simple logic, low computational requirements — UAVCAN targets a wide variety of embedded systems, from high-performance on-board computers to extremely resource-constrained microcontrollers. It will be inexpensive to support in terms of computing power and engineering hours, and advanced features can be implemented incrementally as needed.

Rich data type and interface abstractions — An interface description language will be a core part of the technology which will allow deeply embedded sub-systems to interface with higher-level systems directly and in a maintainable manner while enabling simulation and functional testing.

Support for various transport protocols — UAVCAN will be usable with different transports. The standard shall be capable of accommodating other transport protocols in the future.

API-agnostic standard — Unlike some other networking standards, UAVCAN will not attempt to describe the application program interface (API). Any details that do not affect the behavior of an implementation observable by other participants of the network will be outside of the scope of this specification.

Open specification and reference implementations — The UAVCAN specification will always be open and free to use for everyone; the reference implementations will be distributed under the terms of the permissive MIT License or released into the public domain.

1.4 Capabilities

The maximum number of nodes per logical network is dependent on the transport protocol in use, but it is guaranteed to be not less than 128.

UAVCAN supports an unlimited number of composite data types, which can be defined by the specification (such definitions are called *standard data types*) or by others for private use or for public release (in which case they are said to be *application-specific* or *vendor-specific*; these terms are equivalent). There can be up to 256 major versions of a data type, and up to 256 minor versions per major version.

UAVCAN supports 8192 message subject identifiers for publish/subscribe exchanges and 512 service identifiers for remote procedure call exchanges. A small subset of these identifiers is reserved for the core standard and for publicly released vendor-specific types (chapter 5).

Depending on the transport protocol, UAVCAN supports at least eight distinct communication priority levels (section 4.1.1.3).

The list of transport protocols supported by UAVCAN is provided in chapter 4. Non-redundant, doubly-redundant and triply-redundant transports are supported. Additional transport layers may be added in future revisions of the protocol.

[4] A *transport frame* is an atomic transmission unit defined by the underlying transport protocol. For example, a CAN frame.

Application-level capabilities of the protocol (such as time synchronization, file transfer, node software update, diagnostics, schemaless named registers, diagnostics, plug-and-play node insertion, etc.) are listed in section 5.3.

The core specification does not define nor explicitly limit any physical layers for a given transport, however; properties required by UAVCAN may imply or impose constraints and/or minimum performance requirements on physical networks. Because of this, the core standard does not control compatibility below a supported transport layer between compliant nodes on a physical network (i.e. there are no, anticipated, compatibility concerns between compliant nodes connected to a virtual network where hardware constraints are not enforced nor emulated). Additional standards specifying physical-layer requirements, including connectors, may be required to utilize this standard in a vehicle system.

The capabilities of the protocol will never be reduced within a major version of the specification but may be expanded.

1.5 Management policy

The UAVCAN maintainers are tasked with maintaining and advancing this specification and the set of public regulated data types[5] based on their research and the input from adopters. The maintainers will be committed to ensuring long-term stability and backward compatibility of existing and new deployments. The maintainers will publish relevant announcements and solicit inputs from adopters via the discussion forum whenever a decision that may potentially affect existing deployments is being made.

The set of standard data types is a subset of public regulated data types and is an integral part of the specification; however, there is only a very small subset of required standard data types needed to implement the protocol. A larger set of optional data types are defined to create a standardized data exchange environment supporting the interoperability of COTS[6] equipment manufactured by different vendors. Adopters are invited to take part in the advancement and maintenance of the public regulated data types under the management and coordination of the UAVCAN maintainers.

1.6 Referenced sources

The UAVCAN specification contains references to the following sources:

- CiA 103 — Intrinsically safe capable physical layer.
- CiA 801 — Application note — Automatic bit rate detection.
- IEEE 754 — Standard for binary floating-point arithmetic.
- IEEE Std 1003.1 — IEEE Standard for Information Technology – Portable Operating System Interface (POSIX) Base Specifications.
- IETF RFC2119 — Key words for use in RFCs to Indicate Requirement Levels.
- ISO 11898-1 — Controller area network (CAN) — Part 1: Data link layer and physical signaling.
- ISO 11898-2 — Controller area network (CAN) — Part 2: High-speed medium access unit.
- ISO/IEC 10646 — Universal Coded Character Set (UCS).
- ISO/IEC 14882 — Programming Language C++.
- semver.org — Semantic versioning specification.
- “A Passive Solution to the Sensor Synchronization Problem”, Edwin Olson.
- “Implementing a Distributed High-Resolution Real-Time Clock using the CAN-Bus”, M. Gergeleit and H. Streich.
- “In Search of an Understandable Consensus Algorithm (Extended Version)”, Diego Ongaro and John Ousterhout.

1.7 Revision history

1.7.1 v1.0 – work in progress

- The maximum data type name length has been increased from 50 to 255 characters.
- The default extent function has been removed (section 3.4.5.5). The extent now has to be specified explicitly always unless the data type is sealed.

1.7.2 v1.0-beta – Sep 2020

Compared to v1.0-alpha, the differences are as follows (the motivation is provided on the forum):

- The physical layer specification has been removed. It is now up to the domain-specific UAVCAN-based

[5] The related technical aspects are covered in chapters 2 and 3.

[6] Commercial off-the-shelf equipment.

standards to define the physical layer.
- The subject-ID range reduced from $[0, 32767]$ down to $[0, 8191]$. This change may be reverted in a future edition of the standard, if found practical.
- Added support for delimited serialization; introduced related concepts of *extent* and *sealing* (section 3.4.5.5). This change enables one to easily evolve networked services in a backward-compatible way.
- Enabled the automatic runtime adjustment of the transfer-ID timeout on a per-subject basis as a function of the transfer reception rate (section 4.1.4).

1.7.3 v1.0-alpha – Jan 2020

This is the initial version of the document. The discussions that shaped the initial version are available on the public UAVCAN discussion forum.

2 Basic concepts

2.1 Main principles

2.1.1 Communication

2.1.1.1 Architecture

A UAVCAN network is a decentralized peer network, where each peer (node) has a unique numeric identifier[7] — *node-ID* — ranging from 0 up to a transport-specific upper boundary which is guaranteed to be not less than 127. Nodes of a UAVCAN network can communicate using the following communication methods:

Message publication — The primary method of data exchange with one-to-many publish/subscribe semantics.

Service invocation — The communication method for one-to-one request/response interactions[8].

For each type of communication, a predefined set of data types is used, where each data type has a unique name. Additionally, every data type definition has a pair of major and minor version numbers, which enable data type definitions to evolve in arbitrary ways while ensuring a well-defined migration path if backward-incompatible changes are introduced. Some data types are standard and defined by the protocol specification (of which only a small subset are required); others may be specific to a particular application or vendor.

2.1.1.2 Subjects and services

Message exchanges between nodes are grouped into *subjects* by the semantic meaning of the message. Message exchanges belonging to the same subject pertain to the same function or process within the system.

Request/response exchanges between nodes are grouped into *services* by the semantic meaning of the request and response, like messages are grouped into subjects. Requests and their corresponding responses that belong to the same service pertain to the same function or process within the system.

Each message subject is identified by a unique natural number – a *subject-ID*; likewise, each service is identified by a unique *service-ID*. An umbrella term *port-ID* is used to refer either to a subject-ID or to a service-ID (port identifiers have no direct manifestation in the construction of the protocol, but they are convenient for discussion). The sets of subject-ID and service-ID are orthogonal.

Port identifiers are assigned to various functions, processes, or data streams within the network at the system definition time. Generally, a port identifier can be selected arbitrarily by a system integrator by changing relevant configuration parameters of connected nodes, in which case such port identifiers are called *non-fixed port identifiers*. It is also possible to permanently associate any data type definition with a particular port identifier at a data type definition time, in which case such port identifiers are called *fixed port identifiers*; their usage is governed by rules and regulations described in later sections.

A port-ID used in a given UAVCAN network shall not be shared between functions, processes, or data streams that have different semantic meaning.

A data type of a given major version can be used simultaneously with an arbitrary number of non-fixed different port identifiers, but not more than one fixed port identifier.

2.1.2 Data types

2.1.2.1 Data type definitions

Message and service types are defined using the *data structure description language* (DSDL) (chapter 3). A DSDL definition specifies the name, major version, minor version, attributes, and an optional fixed port-ID of the data type among other less important properties. Service types define two inner data types: one for request, and the other for response.

2.1.2.2 Regulation

Data type definitions can be created by the UAVCAN specification maintainers or by its users, such as equipment vendors or application designers. Irrespective of the origin, data types can be included into the set of

[7] Here and elsewhere in this specification, *ID* and *identifier* are used interchangeably unless specifically indicated otherwise.
[8] Like remote procedure call (RPC).

data type definitions maintained and distributed by the UAVCAN specification maintainers; definitions belonging to this set are termed *regulated data type definitions*. The specification maintainers undertake to keep regulated definitions well-maintained and may occasionally amend them and release new versions, if such actions are believed to benefit the protocol. User-created (i.e., vendor-specific or application-specific) data type definitions that are not included into the aforementioned set are called *unregulated data type definitions*.

Unregulated definitions that are made available for reuse by others are called *unregulated public data type definitions*; those that are kept closed-source for private use by their authors are called *(unregulated) private data type definitions*[9].

Data type definitions authored by the specification maintainers for the purpose of supporting and advancing this specification are called *standard data type definitions*. All standard data type definitions are regulated.

Fixed port identifiers can be used only with regulated data type definitions or with private definitions. Fixed port identifiers shall not be used with public unregulated data types, since that is likely to cause unresolvable port identifier collisions[10]. This restriction shall be followed at all times by all compliant implementations and systems[11].

	Regulated	**Unregulated**
Public	Standard and contributed (e.g., vendor-specific) definitions. Fixed port identifiers are allowed; they are called *regulated port-ID*.	Definitions distributed separately from the UAVCAN specification. Fixed port identifiers are *not allowed*.
Private	Nonexistent category.	Definitions that are not available to anyone except their authors. Fixed port identifiers are permitted (although not recommended); they are called *unregulated fixed port-ID*.

Table 2.1: Data type taxonomy

DSDL processing tools shall prohibit unregulated fixed port identifiers by default, unless they are explicitly configured otherwise.

Each of the two sets of port identifiers (which are subject identifiers and service identifiers) are segregated into three categories:

- Application-specific port identifiers. These can be assigned by changing relevant configuration parameters of the connected nodes (in which case they are called *non-fixed*), or at the data type definition time (in which case they are called *fixed unregulated*, and they generally should be avoided due to the risks of collisions as explained earlier).
- Regulated non-standard fixed port identifiers. These are assigned by the specification maintainers for non-standard contributed vendor-specific public data types.
- Standard fixed port identifiers. These are assigned by the specification maintainers for standard regulated public data types.

Data type authors that want to release regulated data type definitions or contribute to the standard data type set should contact the UAVCAN maintainers for coordination. The maintainers will choose unoccupied fixed port identifiers for use with the new definitions, if necessary. Since the set of regulated definitions is maintained in a highly centralized manner, it can be statically ensured that no identifier collisions will take place within it; also, since the identifier ranges used with regulated definitions are segregated, regulated port-IDs will not conflict with any other compliant UAVCAN node or system[12].

2.1.2.3 *Serialization*

A DSDL description can be used to automatically generate the serialization and deserialization code for every defined data type in a particular programming language. Alternatively, a DSDL description can be used to

[9] The word "unregulated" is redundant because private data types cannot be regulated, by definition. Likewise, all regulated definitions are public, so the word "public" can be omitted.

[10] Any system that relies on data type definitions with fixed port identifiers provided by an external party (i.e., data types and the system in question are designed by different parties) runs the risk of encountering port identifier conflicts that cannot be resolved without resorting to help from said external party since the designers of the system do not have control over their fixed port identifiers. Because of this, the specification strongly discourages the use of fixed unregulated private port identifiers. If a data type definition is ever disclosed to any other party (i.e., a party that did not author it) or to the public at large it is important that the data type *not* include a fixed port-identifier.

[11] In general, private unregulated fixed port identifiers are collision-prone by their nature, so they should be avoided unless there are very strong reasons for their usage and the authors fully understand the risks.

[12] The motivation for the prohibition of fixed port identifiers in unregulated public data types is derived directly from the above: since there is no central repository of unregulated definitions, collisions would be likely.

construct appropriate serialization code manually by a human. DSDL ensures that the memory footprint and computational complexity per data type are constant and easily predictable.

Serialized message and service objects[13] are exchanged by means of the transport layer (chapter 4), which implements automatic decomposition of long transfers into several transport frames[14] and reassembly from these transport frames back into a single atomic data block, allowing nodes to exchange serialized objects of arbitrary size (DSDL guarantees, however, that the minimum and maximum size of the serialized representation of any object of any data type is always known statically).

2.1.3 High-level functions

On top of the standard data types, UAVCAN defines a set of standard high-level functions including: node health monitoring, node discovery, time synchronization, firmware update, plug-and-play node support, and more (section 5.3).

<table>
<tr><td colspan="3">Applications</td></tr>
<tr><td>Required functions</td><td>Standard functions</td><td>Custom functions</td></tr>
<tr><td>Required data types</td><td>Standard data types</td><td>Custom data types</td></tr>
<tr><td colspan="3">Serialization</td></tr>
<tr><td colspan="3">Transport</td></tr>
</table>

Figure 2.1: UAVCAN architectural diagram

2.2 Message publication

Message publication refers to the transmission of a serialized message object over the network to other nodes. This is the primary data exchange mechanism used in UAVCAN; it is functionally similar to raw data exchange with minimal overhead, additional communication integrity guarantees, and automatic decomposition and reassembly of long payloads across multiple transport frames. Typical use cases may include transfer of the following kinds of data (either cyclically or on an ad-hoc basis): sensor measurements, actuator commands, equipment status information, and more.

Information contained in a published message is summarized in table 2.2.

Property	Description
Payload	The serialized message object.
Subject-ID	Numerical identifier that indicates how the payload should be interpreted.
Source node-ID	The node-ID of the transmitting node (excepting anonymous messages).
Transfer-ID	An integer value that is used for message sequence monitoring, multi-frame transfer reassembly, deduplication, automatic management of redundant transports, and other purposes (section 4.1.1.7).

Table 2.2: Published message properties

2.2.1 Anonymous message publication

Nodes that don't have a unique node-ID can publish only *anonymous messages*. An anonymous message is different from a regular message in that it doesn't contain a source node-ID.

UAVCAN nodes will not have an identifier initially until they are assigned one, either statically (which is generally the preferred option for applications where a high degree of determinism and high safety assurances are required) or automatically (i.e., plug-and-play). Anonymous messages are used to facilitate the plug-and-play function (section 5.3.12).

2.3 Service invocation

Service invocation is a two-step data exchange operation between exactly two nodes: a client and a server. The steps are[15]:

1. The client sends a service request to the server.
2. The server takes appropriate actions and sends a response to the client.

Typical use cases for this type of communication include: node configuration parameter update, firmware

[13] An *object* means a value that is an instance of a well-defined type.
[14] A *transport frame* means a block of data that can be atomically exchanged over the transport layer network, e.g., a CAN frame.
[15] The request/response semantic is facilitated by means of hardware (if available) or software acceptance filtering and higher-layer logic. No additional support or non-standard transport layer features are required.

update, an ad-hoc action request, file transfer, and other functions of similar nature.

Information contained in service requests and responses is summarized in table 2.3. Both the request and the response contain same values for all listed fields except payload, where the content is application-defined.

Property	Description
Payload	The serialized request/response object.
Service-ID	Numerical identifier that indicates how the service should be handled.
Client node-ID	Source node-ID during request transfer, destination node-ID during response transfer.
Server node-ID	Destination node-ID during request transfer, source node-ID during response transfer.
Transfer-ID	An integer value that is used for request/response matching, multi-frame transfer re-assembly, deduplication, automatic management of redundant transports, and other purposes (section 4.1.1.7).

Table 2.3: Service request/response properties

3 Data structure description language

The data structure description language, or *DSDL*, is a simple domain-specific language designed for defining composite data types. The defined data types are used for exchanging data between UAVCAN nodes via one of the standard UAVCAN transport protocols[16].

3.1 Architecture

3.1.1 General principles

In accordance with the UAVCAN architecture, DSDL allows users to define data types of two kinds: message types and service types. Message types are used to exchange data over publish-subscribe one-to-many message links identified by subject-ID, and service types are used to perform request-response one-to-one exchanges (like RPC) identified by service-ID. A service type is composed of exactly two inner data types: one of them is the request type (its instances are transferred from client to server), and the other is the response type (its instances are transferred from the server back to the client).

Following the deterministic nature of UAVCAN, the size of a serialized representation of any message or service object is bounded within statically known limits. Variable-size entities always have a fixed size limit defined by the data type designer.

DSDL definitions are strongly statically typed.

DSDL provides a well-defined means of data type versioning, which enables data type maintainers to introduce changes to released data types while ensuring backward compatibility with fielded systems.

DSDL is designed to support extensive static analysis. Important properties of data type definitions such as backward binary compatibility and data field layouts can be checked and validated by automatic software tools before the systems utilizing them are fielded.

DSDL definitions can be used to automatically generate serialization (and deserialization) source code for any data type in a target programming language. A tool that is capable of generating serialization code based on a DSDL definition is called a *DSDL compiler*. More generically, a software tool designed for working with DSDL definitions is called a *DSDL processing tool*.

3.1.2 Data types and namespaces

Every data type is located inside a *namespace*. Namespaces may be included into higher-level namespaces, forming a tree hierarchy.

A namespace that is at the root of the tree hierarchy (i.e., not nested within another one) is called a *root namespace*. A namespace that is located inside another namespace is called a *nested namespace*.

A data type is uniquely identified by its namespaces and its *short name*. The short name of a data type is the name of the type itself excluding the containing namespaces.

A *full name* of a data type consists of its short name and all of its namespace names. The short name and the namespace names included in a full name are called *name components*. Name components are ordered: the root namespace is always the first component of the name, followed by the nested namespaces, if there are any, in the order of their nesting; the short name is always the last component of the full name. The full name is formed by joining its name components via the ASCII dot character "`.`" (ASCII code 46).

A *full namespace* name is the full name without the short name and its component separator.

A *sub-root namespace* is a nested namespace that is located immediately under its root namespace. Data types that reside directly under their root namespace do not have a sub-root namespace.

The name structure is illustrated in figure 3.1.

[16] The standard transport protocols are documented in chapter 4. UAVCAN doesn't prohibit users from defining their own application-specific transports as well, although users doing that are likely to encounter compatibility issues and possibly a suboptimal performance of the protocol.

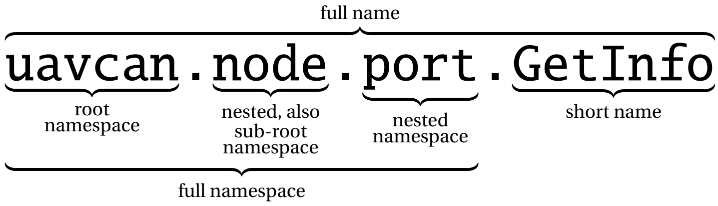


Figure 3.1: Data type name structure

A set of full namespace names and a set of full data type names shall not intersect[17].

Data type names and namespace names are case-sensitive. However, names that differ only in letter case are not permitted[18]. In other words, a pair of names which differ only in letter case is considered to constitute a name collision.

A name component consists of alphanumeric ASCII characters (which are: `A-Z`, `a-z`, and `0-9`) and underscore ("`_`", ASCII code 95). An empty string is not a valid name component. The first character of a name component shall not be a digit. A name component shall not match any of the reserved word patterns, which are listed in table 3.2.5.

The length of a full data type name shall not exceed 255 characters[19].

Every data type definition is assigned a major and minor version number pair. In order to uniquely identify a data type definition, its version numbers shall be specified. In the following text, the term *version* without a majority qualifier refers to a pair of major and minor version numbers.

Valid data type version numbers range from 0 to 255, inclusively. A data type version where both major and minor components are zero is not allowed.

3.1.3 File hierarchy

DSDL data type definitions are contained in UTF-8 encoded text files with a file name extension `.uavcan`.

One file defines exactly one version of a data type, meaning that each combination of major and minor version numbers shall be unique per data type name. There may be an arbitrary number of versions of the same data type defined alongside each other, provided that each version is defined at most once. Version number sequences can be non-contiguous, meaning that it is allowed to skip version numbers or remove existing definitions that are neither oldest nor newest.

A data type definition may have an optional fixed port-ID[20] value specified.

The name of a data type definition file is constructed from the following entities joined via the ASCII dot character "`.`" (ASCII code 46), in the specified order:

- Fixed port-ID in decimal notation, unless a fixed port-ID is not provided for this definition.
- Short name of the data type (mandatory, always non-empty).
- Major version number in decimal notation (mandatory).
- Minor version number in decimal notation (mandatory).
- File name extension "`uavcan`" (mandatory).

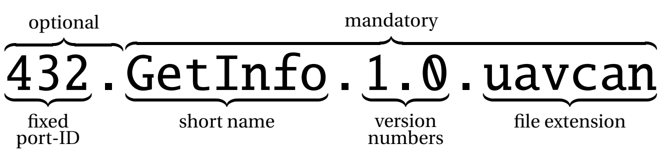


Figure 3.2: Data type definition file name structure

DSDL namespaces are represented as directories, where one directory defines exactly one namespace, possibly nested. The name of the directory defines the name of its data type name component. A directory defining a namespace will always define said namespace in its entirety, meaning that the contents of a namespace cannot be spread across different directories sharing the same name. One directory cannot define more than one

[17] For example, a namespace "`vendor.example`" and a data type "`vendor.example.1.0`" are mutually exclusive. Note the data type name shown in this example violates the naming conventions which are reviewed in a separate section.
[18] Because that may cause problems with case-insensitive file systems.
[19] This includes the name component separators, but not the version.
[20] Chapter 2.

level of nesting[21].

Directory tree	Entry description
`vendor_x/`	Root namespace `vendor_x`.
`foo/`	Nested namespace (also sub-root) `vendor_x.foo`.
`100.Run.1.0.uavcan`	Data type definition v1.0 with fixed service-ID 100.
`100.Status.1.0.uavcan`	Data type definition v1.0 with fixed subject-ID 100.
`ID.1.0.uavcan`	Data type definition v1.0 without fixed port-ID.
`ID.1.1.uavcan`	Data type definition v1.1 without fixed port-ID.
`bar_42/`	Nested namespace `vendor_x.foo.bar_42`.
`101.List.1.0.uavcan`	Data type definition v1.0 with fixed service-ID 101.
`102.List.2.0.uavcan`	Data type definition v2.0 with fixed service-ID 102.
`ID.1.0.uavcan`	Data type definition v1.0 without fixed port-ID.

Figure 3.3: DSDL directory structure example

3.1.4 Elements of data type definition

A data type definition file contains an exhaustive description of a particular version of the said data type in the *data structure description language* (DSDL).

A data type definition contains an ordered, possibly empty collection of *field attributes* and/or unordered, possibly empty collection of *constant attributes*.

A data type may describe either a *structure object* or a *tagged union object*. The value of a structure object is a function of the values of all of its field attributes. A tagged union object is formed from at least two field attributes, but it is capable of holding exactly one field attribute value at any given time. The value of a tagged union object is a function of which field attribute value it is holding at the moment and the value of said field attribute.

A field attribute represents a named dynamically assigned value of a statically defined type that can be exchanged over the network as a member of its containing object. A padding field attribute is a special kind of field attribute which is used for data alignment purposes; such field attributes are not named.

A constant attribute represents a named statically defined value of a statically defined type. Constants are never exchanged over the network, since they are assumed to be known to all involved nodes by virtue of them sharing compatible definitions of the data type.

Constant values are defined via *DSDL expressions*, which are evaluated at the time of DSDL definition processing. There is a special category of types called *expression types*, instances of which are used only during expression evaluation and cannot be exchanged over the network.

Data type definitions can also contain various auxiliary elements reviewed later, such as deprecation markers (notifying its users that the data type is no longer recommended for new designs) or assertions (special statements introduced by data type designers which are statically validated by DSDL processing tools).

Service type definitions are a special case: they cannot be instantiated or serialized, they do not contain attributes, and they are composed of exactly two inner data type definitions[22]. These inner types are the service request type and the service response type, separated by the *service response marker*. They are otherwise ordinary data types except that they are unutterable[23] and they derive some of their properties[24] from their *parent service type*.

3.1.5 Serialization

Every object that can be exchanged between UAVCAN nodes has a well-defined *serialized representation*. The value and meaning of an object can be unambiguously recovered from its serialized representation, provided that the type of the object is known. Such recovery process is called *deserialization*.

A serialized representation is a sequence of binary digits (bits); the number of bits in a serialized representation is called its *bit length*. A *bit length set* of a data type refers to the set of bit length values of all possible serialized representations of objects that are instances of the data type.

[21] For example, "`foo.bar`" is not a valid directory name. The valid representation would be "`bar`" nested in "`foo`".
[22] A service type can be thought of as a specialized namespace that contains two types and has some of the properties of a type, such as name and version.
[23] Cannot be referred to. Another commonly used term is "Voldemort type".
[24] Like version numbers or deprecation status.

A data type whose bit length set contains more than one element is said to be *variable length*. The opposite case is referred to as *fixed length*.

The data type of a serialized message or service object exchanged over the network is recovered from its subject-ID or service-ID, respectively, which is attached to the serialized object, along with other metadata, in a manner dictated by the applicable transport layer specification (chapter 4). For more information on port identifiers and data type mapping refer to section 2.1.1.2.

The bit length set is not defined on service types (only on their request and response types) because they cannot be instantiated.

3.2 Grammar

This section contains the formal definition of the DSDL grammar. Its notation is introduced beforehand. The meaning of each element of the grammar and their semantics will be explained in the following sections.

3.2.1 Notation

The following definition relies on the PEG[25] notation described in table 3.2.1[26]. The text of the formal definition contains comments which begin with an octothorp and last until the end of the line.

Pattern	Description
`"text"`	Denotes a terminal string of ASCII characters. The string is case-sensitive.
(space)	Concatenation. E.g., `korovan paukan excavator` matches a sequence where the specified tokens appear in the defined order.
`abc / ijk / xyz`	Alternatives. The leftmost matching alternative is accepted.
`abc?`	Optional greedy match.
`abc*`	Zero or more expressions, greedy match.
`abc+`	One or more expressions, greedy match.
`~r"regex"`	An IEEE POSIX Extended Regular Expression pattern defined between the double quotes. The expression operates on the ASCII character set and is always case-sensitive. ASCII escape sequences "`\r`", "`\n`", and "`\t`" are used to denote ASCII carriage return (code 13), line feed (code 10), and tabulation (code 9) characters, respectively.
`~r'regex'`	As above, with single quotes instead of double quotes.
`(abc xyz)`	Parentheses are used for grouping.

Table 3.1: Notation used in the formal grammar definition

3.2.2 Definition

At the top level, a DSDL definition file is an ordered collection of statements; the order is determined by the relative placement of statements inside the DSDL source file: statements located closer the beginning of the file precede those that are located closer to the end of the file.

From the top level down to the expression rule, the grammar is a valid regular grammar, meaning that it can be parsed using standard regular expressions.

The grammar definition provided here assumes lexerless parsing; that is, it applies directly to the unprocessed source text of the definition.

All characters used in the definition belong to the ASCII character set.

[25] Parsing expression grammar.

[26] Inspired by Parsimonious – an MIT-licensed software product authored by Erik Rose; its sources are available at https://github.com/erikrose/parsimonious.

```
definition  = line (end_of_line line)*  # An empty file is a valid definition. Trailing end-of-line is optional.
line        = statement? _? comment?    # An empty line is a valid line.
comment     = ~r"#[^\r\n]*"
end_of_line = ~r"\r?\n"                 # Unix/Windows
_           = ~r"[ \t]+"                # Whitespace

identifier = ~r"[a-zA-Z_][a-zA-Z0-9_]*"

# ==================================================== Statements ====================================================

statement = statement_directive
          / statement_service_response_marker
          / statement_attribute

statement_attribute = statement_constant
                    / statement_field
                    / statement_padding_field

statement_constant      = type _ identifier _? "=" _? expression
statement_field         = type _ identifier
statement_padding_field = type_void ""  # The trailing empty symbol is to prevent the node from being optimized away.

statement_service_response_marker = ~r"---+"  # Separates request/response, specifies that the definition is a service.

statement_directive = statement_directive_with_expression
                    / statement_directive_without_expression
statement_directive_with_expression    = "@" identifier _ expression  # The expression type shall match the directive.
statement_directive_without_expression = "@" identifier

# ==================================================== Data types ====================================================

type = type_array
     / type_scalar

type_array = type_array_variable_inclusive
           / type_array_variable_exclusive
           / type_array_fixed

type_array_variable_inclusive = type_scalar _? "[" _? "<=" _? expression _? "]"  # Expression shall yield integer.
type_array_variable_exclusive = type_scalar _? "[" _? "<"  _? expression _? "]"
type_array_fixed              = type_scalar _? "[" _?         expression _? "]"

type_scalar = type_versioned
            / type_primitive
            / type_void

type_versioned         = identifier ("." identifier)* "." type_version_specifier
type_version_specifier = literal_integer_decimal "." literal_integer_decimal

type_primitive = type_primitive_truncated
               / type_primitive_saturated

type_primitive_truncated =  "truncated" _   type_primitive_name
type_primitive_saturated = ("saturated" _)? type_primitive_name     # Defaults to this.

type_primitive_name = type_primitive_name_boolean
                    / type_primitive_name_unsigned_integer
                    / type_primitive_name_signed_integer
                    / type_primitive_name_floating_point

type_primitive_name_boolean          = "bool"
type_primitive_name_unsigned_integer = "uint"  type_bit_length_suffix
type_primitive_name_signed_integer   = "int"   type_bit_length_suffix
type_primitive_name_floating_point   = "float" type_bit_length_suffix

type_void = "void" type_bit_length_suffix

type_bit_length_suffix = ~r"[1-9]\d*"

# ==================================================== Expressions ====================================================

expression = ex_logical    # Aliased for clarity.

expression_list = (expression (_? "," _? expression)*)?     # May be empty.

expression_parenthesized = "(" _? expression _? ")"         # Used for managing precedence.

expression_atom = expression_parenthesized                  # Ordering matters.
                / type
                / literal
                / identifier

# Operators. The precedence relations are expressed in the rules; the order here is from lower to higher.
# Operators that share common prefix (e.g. < and <=) are arranged so that the longest form is specified first.
ex_logical        = ex_logical_not    (_? op2_log _? ex_logical_not)*
ex_logical_not    = op1_form_log_not / ex_comparison
ex_comparison     = ex_bitwise        (_? op2_cmp _? ex_bitwise)*
ex_bitwise        = ex_additive       (_? op2_bit _? ex_additive)*
ex_additive       = ex_multiplicative (_? op2_add _? ex_multiplicative)*
ex_multiplicative = ex_inversion      (_? op2_mul _? ex_inversion)*
ex_inversion      = op1_form_inv_pos / op1_form_inv_neg / ex_exponential
```

```
ex_exponential    = ex_attribute      (_? op2_exp _? ex_inversion)?     # Right recursion
ex_attribute      = expression_atom   (_? op2_attrib _? identifier)*

# Unary operator forms are moved into separate rules for ease of parsing.
op1_form_log_not = "!" _? ex_logical_not     # Right recursion
op1_form_inv_pos = "+" _? ex_exponential
op1_form_inv_neg = "-" _? ex_exponential

# Logical operators; defined for booleans.
op2_log = op2_log_or / op2_log_and
op2_log_or  = "||"
op2_log_and = "&&"

# Comparison operators.
op2_cmp = op2_cmp_equ / op2_cmp_geq / op2_cmp_leq / op2_cmp_neq / op2_cmp_lss / op2_cmp_grt  # Ordering is important.
op2_cmp_equ = "=="
op2_cmp_neq = "!="
op2_cmp_leq = "<="
op2_cmp_geq = ">="
op2_cmp_lss = "<"
op2_cmp_grt = ">"

# Bitwise integer manipulation operators.
op2_bit = op2_bit_or / op2_bit_xor / op2_bit_and
op2_bit_or  = "|"
op2_bit_xor = "^"
op2_bit_and = "&"

# Additive operators.
op2_add = op2_add_add / op2_add_sub
op2_add_add = "+"
op2_add_sub = "-"

# Multiplicative operators.
op2_mul = op2_mul_mul / op2_mul_div / op2_mul_mod  # Ordering is important.
op2_mul_mul = "*"
op2_mul_div = "/"
op2_mul_mod = "%"

# Exponential operators.
op2_exp = op2_exp_pow
op2_exp_pow = "**"

# The most tightly bound binary operator - attribute reference.
op2_attrib = "."

# ===================================================== Literals =====================================================

literal = literal_set           # Ordering is important to avoid ambiguities.
        / literal_real
        / literal_integer
        / literal_string
        / literal_boolean

# Set.
literal_set = "{" _? expression_list _? "}"

# Integer.
literal_integer = literal_integer_binary
                / literal_integer_octal
                / literal_integer_hexadecimal
                / literal_integer_decimal
literal_integer_binary      = ~r"0[bB](_?(0|1))+"
literal_integer_octal       = ~r"0[oO](_?[0-7])+"
literal_integer_hexadecimal = ~r"0[xX](_?[0-9a-fA-F])+"
literal_integer_decimal     = ~r"(0(_?0)*)+|([1-9](_?[0-9])*)"

# Real. Exponent notation is defined first to avoid ambiguities.
literal_real = literal_real_exponent_notation
             / literal_real_point_notation
literal_real_exponent_notation = (literal_real_point_notation / literal_real_digits) literal_real_exponent
literal_real_point_notation    = (literal_real_digits? literal_real_fraction) / (literal_real_digits ".")
literal_real_fraction          = "." literal_real_digits
literal_real_exponent          = ~r"[eE][+-]?" literal_real_digits
literal_real_digits            = ~r"[0-9](_?[0-9])*"

# String.
literal_string = literal_string_single_quoted
               / literal_string_double_quoted
literal_string_single_quoted = ~r"'[^'\\]*(\\[^\r\n][^'\\]*)*'"
literal_string_double_quoted = ~r'"[^"\\]*(\\[^\r\n][^"\\]*)*"'

# Boolean.
literal_boolean = literal_boolean_true
                / literal_boolean_false
literal_boolean_true  = "true"
literal_boolean_false = "false"
```

3.2.3 Expressions

Symbols representing operators belong to the ASCII (basic Latin) character set.

Operators of the same precedence level are evaluated from left to right.

The attribute reference operator is a special case: it is defined for an instance of any type on its left side and an attribute identifier on its right side. The concept of "attribute identifier" is not otherwise manifested in the type system. The attribute reference operator is not explicitly documented for any data type; instead, the documentation specifies the set of available attributes for instances of said type, if there are any.

Symbol	Precedence	Description
`+`	3	Unary plus
`-` (hyphen-minus)	3	Unary minus
`!`	8	Logical not

Table 3.2: Unary operators

Symbol	Precedence	Description
`.` (full stop)	1	Attribute reference (parent object on the left side, attribute identifier on the right side)
`**`	2	Exponentiation (base on the left side, power on the right side)
`*`	4	Multiplication
`/`	4	Division
`%`	4	Modulo
`+`	5	Addition
`-` (hyphen-minus)	5	Subtraction
`|` (vertical line)	6	Bitwise or
`^` (circumflex accent)	6	Bitwise xor
`&`	6	Bitwise and
`==` (dual equals sign)	7	Equality
`!=`	7	Inequality
`<=`	7	Less or equal
`>=`	7	Greater or equal
`<`	7	Less
`>`	7	Greater
`||` (dual vertical line)	9	Logical or
`&&`	9	Logical and

Table 3.3: Binary operators

3.2.4 Literals

Upon its evaluation, a literal yields an object of a particular type depending on the syntax of the literal, as specified in this section.

3.2.4.1 Boolean literals

A boolean literal is denoted by the keyword "`true`" or "`false`" represented by an instance of primitive type "`bool`" (section 3.4.3) with an appropriate value.

3.2.4.2 Numeric literals

Integer and real literals are represented as instances of type "`rational`" (section 3.3.1).

The digit separator character "`_`" (underscore) does not affect the interpretation of numeric literals.

The significand of a real literal is formed by the integer part, the optional decimal point, and the optional fraction part; either the integer part or the fraction part (not both) can be omitted. The exponent is optionally specified after the letter "`e`" or "`E`"; it indicates the power of 10 by which the significand is to be scaled. Either the decimal point or the letter "`e`"/"`E`" with the following exponent (not both) can be omitted from a real literal.

> An integer literal `0x123` is represented internally as $\frac{291}{1}$.
>
> A real literal `.3141592653589793e+1` is represented internally as $\frac{3141592653589793}{1000000000000000}$.

3.2.4.3 *String literals*

String literals are represented as instances of type "`string`" (section 3.3.2).

A string literal is allowed to contain an arbitrary sequence of Unicode characters, excepting escape sequences defined in table 3.2.4.3 which shall follow one of the specified therein forms. An escape sequence begins with the ASCII backslash character "\".

Sequence	Interpretation
`\\`	Backslash, ASCII code 92. Same as the escape character.
`\r`	Carriage return, ASCII code 13.
`\n`	Line feed, ASCII code 10.
`\t`	Horizontal tabulation, ASCII code 9.
`\'`	Apostrophe (single quote), ASCII code 39. Regardless of the type of quotes around the literal.
`\"`	Quotation mark (double quote), ASCII code 34. Regardless of the type of quotes around the literal.
`\u????`	Unicode symbol with the code point specified by a four-digit hexadecimal number. The placeholder "?" represents a hexadecimal character `[0-9a-fA-F]`.
`\U????????`	Like above, the code point is specified by an eight-digit hexadecimal number.

Table 3.4: String literal escape sequences

```
@assert "oh,\u0020hi\U0000000aMark" == 'oh, hi\nMark'
```

3.2.4.4 *Set literals*

Set literals are represented as instances of type "`set`" (section 3.3.3) parameterized by the type of the contained elements which is determined automatically.

A set literal declaration shall specify at least one element, which is used to determine the element type of the set.

The elements of a set literal are defined as DSDL expressions which are evaluated before a set is constructed from the corresponding literal.

```
@assert {"cells", 'interlinked'} == {"inter" + "linked", 'cells'}
```

3.2.5 Reserved identifiers

DSDL identifiers and data type name components that match any of the case-insensitive patterns specified in table 3.2.5 cannot be used to name new entities. The semantics of such identifiers is predefined by the DSDL specification, and as such, they cannot be used for other purposes. Some of the reserved identifiers do not have any functions associated with them in this version of the DSDL specification, but this may change in the future.

POSIX ERE ASCII pattern	Example	Special meaning
truncated		Cast mode specifier
saturated		Cast mode specifier
true		Boolean literal
false		Boolean literal
bool		Primitive type category
u?int\d*	uint8	Primitive type category
float\d*	float	Primitive type category
u?q\d+_\d+	q16_8	Primitive type category (future)
void\d*	void	Void type category
optional		Reserved for future use
aligned		Reserved for future use
const		Reserved for future use
struct		Reserved for future use
super		Reserved for future use
template		Reserved for future use
enum		Reserved for future use
self		Reserved for future use
and		Reserved for future use
or		Reserved for future use
not		Reserved for future use
auto		Reserved for future use
type		Reserved for future use
con		Compatibility with Microsoft Windows
prn		Compatibility with Microsoft Windows
aux		Compatibility with Microsoft Windows
nul		Compatibility with Microsoft Windows
com\d	com1	Compatibility with Microsoft Windows
lpt\d	lpt9	Compatibility with Microsoft Windows
_.*_	_offset_	Special-purpose intrinsic entities

Table 3.5: Reserved identifier patterns (POSIX ERE notation, ASCII character set, case-insensitive)

3.3 Expression types

Expression types are a special category of data types whose instances can only exist and be operated upon at the time of DSDL definition processing. As such, expression types cannot be used to define attributes, and their instances cannot be exchanged between nodes.

Expression types are used to represent values of constant expressions which are evaluated when a DSDL definition is processed. Results of such expressions can be used to define various constant properties, such as array length boundaries or values of constant attributes.

Expression types are specified in this section. Each expression type has a formal DSDL name for completeness; even if such types can't be used to define attributes, a well-defined formal name allows DSDL processing tools to emit well-formed and understandable diagnostic messages.

3.3.1 Rational number

At the time of DSDL definition processing, integer and real numbers are represented internally as rational numbers where the range of numerator and denominator is unlimited[27]. DSDL processing tools are not permitted to introduce any implicit rational number transformations that may result in a loss of information.

The DSDL name of the rational number type is "`rational`".

Rational numbers are assumed to be stored in a normalized form, where the denominator is positive and the greatest common divisor of the numerator and the denominator is one.

A rational number can be used in a context where an integer value is expected only if its denominator equals one.

[27]Technically, the range may only be limited by the memory resources available to the DSDL processing tool.

Implicit conversions between boolean-valued entities and rational numbers are not allowed.

Op	Type	Constraints	Description
`+`	`(rational)` → `rational`		No effect.
`-`	`(rational)` → `rational`		Negation.
`**`	`(rational,rational)` → `rational`	Power denominator equals one	Exact exponentiation.
`**`	`(rational,rational)` → `rational`	Power denominator greater than one	Exponentiation with implementation-defined accuracy.
`*`	`(rational,rational)` → `rational`		Exact multiplication.
`/`	`(rational,rational)` → `rational`	Non-zero divisor	Exact division.
`%`	`(rational,rational)` → `rational`	Non-zero divisor	Exact modulo.
`+`	`(rational,rational)` → `rational`		Exact addition.
`-`	`(rational,rational)` → `rational`		Exact subtraction.
`|`	`(rational,rational)` → `rational`	Denominators equal one	Bitwise or.
`^`	`(rational,rational)` → `rational`	Denominators equal one	Bitwise xor.
`&`	`(rational,rational)` → `rational`	Denominators equal one	Bitwise and.
`!=`	`(rational,rational)` → `bool`		Exact inequality.
`==`	`(rational,rational)` → `bool`		Exact equality.
`<=`	`(rational,rational)` → `bool`		Less or equal.
`>=`	`(rational,rational)` → `bool`		Greater or equal.
`<`	`(rational,rational)` → `bool`		Strictly less.
`>`	`(rational,rational)` → `bool`		Strictly greater.

Table 3.6: Operators defined on instances of rational numbers

3.3.2 Unicode string

This type contains a sequence of Unicode characters. It is used to represent string literals internally.

The DSDL name of the Unicode string type is "`string`".

A Unicode string containing one symbol whose code point is within $[0,127]$ (i.e., an ASCII character) is implicitly convertible into a `uint8`-typed constant attribute value, where the value of the constant is to be equal the code point of the symbol.

Op	Type	Description
`+`	`(string,string)` → `string`	Concatenation.
`!=`	`(string,string)` → `bool`	Inequality of Unicode NFC normalized forms. NFC stands for *Normalization Form Canonical Composition* – one of standard Unicode normalization forms where characters are recomposed by canonical equivalence.
`==`	`(string,string)` → `bool`	Equality of Unicode NFC normalized forms.

Table 3.7: Operators defined on instances of Unicode strings

The set of operations and conversions defined for Unicode strings is to be extended in future versions of the specification.

3.3.3 Set

A set type represents an unordered collection of unique objects. All objects shall be of the same type. Uniqueness of elements is determined by application of the equality operator "==".

The DSDL name of the set type is "`set`".

A set can be constructed from a set literal, in which case such set shall contain at least one element.

The attributes and operators defined on set instances are listed in the tables 3.3.3 and 3.3.3, where E represents the set element type.

Name	Type	Constraints	Description
`min`	E	Operator "<" is defined $(E, E) \to$ `bool`	Smallest element in the set determined by sequential application of the operator "<".
`max`	E	Operator "<" is defined $(E, E) \to$ `bool`	Greatest element in the set determined by sequential application of the operator "<".
`count`	`rational`		Cardinality.

Table 3.8: Attributes defined on instances of sets

Op	Type	Constraints	Description
`==`	$(\texttt{set}_{<E>}, \texttt{set}_{<E>}) \to \texttt{bool}$		Left equals right.
`!=`	$(\texttt{set}_{<E>}, \texttt{set}_{<E>}) \to \texttt{bool}$		Left does not equal right.
`<=`	$(\texttt{set}_{<E>}, \texttt{set}_{<E>}) \to \texttt{bool}$		Left is a subset of right.
`>=`	$(\texttt{set}_{<E>}, \texttt{set}_{<E>}) \to \texttt{bool}$		Left is a superset of right.
`<`	$(\texttt{set}_{<E>}, \texttt{set}_{<E>}) \to \texttt{bool}$		Left is a proper subset of right.
`>`	$(\texttt{set}_{<E>}, \texttt{set}_{<E>}) \to \texttt{bool}$		Left is a proper superset of right.
`|`	$(\texttt{set}_{<E>}, \texttt{set}_{<E>}) \to \texttt{set}_{<E>}$		Union.
`^`	$(\texttt{set}_{<E>}, \texttt{set}_{<E>}) \to \texttt{set}_{<E>}$		Disjunctive union.
`&`	$(\texttt{set}_{<E>}, \texttt{set}_{<E>}) \to \texttt{set}_{<E>}$		Intersection.
`**`	$(\texttt{set}_{<E>}, E) \to \texttt{set}_{<R>}$	E is not a set	Elementwise $(E, E) \to R$.
`**`	$(E, \texttt{set}_{<E>}) \to \texttt{set}_{<R>}$	E is not a set	Elementwise $(E, E) \to R$.
`*`	$(\texttt{set}_{<E>}, E) \to \texttt{set}_{<R>}$	E is not a set	Elementwise $(E, E) \to R$.
`*`	$(E, \texttt{set}_{<E>}) \to \texttt{set}_{<R>}$	E is not a set	Elementwise $(E, E) \to R$.
`/`	$(\texttt{set}_{<E>}, E) \to \texttt{set}_{<R>}$	E is not a set	Elementwise $(E, E) \to R$.
`/`	$(E, \texttt{set}_{<E>}) \to \texttt{set}_{<R>}$	E is not a set	Elementwise $(E, E) \to R$.
`%`	$(\texttt{set}_{<E>}, E) \to \texttt{set}_{<R>}$	E is not a set	Elementwise $(E, E) \to R$.
`%`	$(E, \texttt{set}_{<E>}) \to \texttt{set}_{<R>}$	E is not a set	Elementwise $(E, E) \to R$.
`+`	$(\texttt{set}_{<E>}, E) \to \texttt{set}_{<R>}$	E is not a set	Elementwise $(E, E) \to R$.
`+`	$(E, \texttt{set}_{<E>}) \to \texttt{set}_{<R>}$	E is not a set	Elementwise $(E, E) \to R$.
`-`	$(\texttt{set}_{<E>}, E) \to \texttt{set}_{<R>}$	E is not a set	Elementwise $(E, E) \to R$.
`-`	$(E, \texttt{set}_{<E>}) \to \texttt{set}_{<R>}$	E is not a set	Elementwise $(E, E) \to R$.

Table 3.9: Operators defined on instances of sets

3.3.4 Serializable metatype

Serializable types (which are reviewed in section 3.4) are instances of the serializable metatype. This metatype is convenient for expression of various relations and attributes defined on serializable types.

The DSDL name of the serializable metatype is "`metaserializable`".

Available attributes are defined on a per-instance basis.

3.4 Serializable types

3.4.1 General principles

Values of the serializable type category can be exchanged between nodes over the UAVCAN network. This is opposed to the expression types (section 3.3), instances of which can only exist while DSDL definitions are being evaluated. The data serialization rules are defined in section 3.7.

3.4.1.1 Alignment and padding

For any serializable type, its *alignment* A is defined as some positive integer number of bits such that the offset of a serialized representation of an instance of this type relative to the origin of the containing serialized representation (if any) is an integer multiple of A.

Given an instance of type whose alignment is A, it is guaranteed that its serialized representation is always an integer multiple of A bits long.

When constructing a serialized representation, the alignment and length requirements are satisfied by means of *padding*, which refers to the extension of a bit sequence with zero bits until the resulting alignment or length requirements are satisfied. During deserialization, the padding bits are ignored (skipped) irrespective of their value (also see related section 3.7.1.3).

> For example, given a variable-length entity whose length varies between 1 and 3 bits, followed by a field whose type has the alignment requirement of 8, one may end up with 5, 6, or 7 bits of padding inserted before the second field at runtime.
>
> The exact amount of such padding cannot always be determined statically because it depends on the size of the preceding entity; however, it is guaranteed that it is always strictly less than the alignment requirement of the following field or, if this is the last field in a group, the alignment requirement of its container.

3.4.2 Void types

Void types are used for padding purposes. The alignment of void types is 1 bit (i.e., no alignment).

Void-typed field attributes are set to zero when an object is serialized and ignored when it is deserialized. Void types can be used to reserve space in data type definitions for possible use in later versions of the data type.

The DSDL name pattern for void types is as follows: "`void[1-9]\d*`", where the trailing integer represents its width, in bits, ranging from 1 to 64, inclusive.

Void types can be referred to directly by their name from any namespace.

3.4.3 Primitive types

Primitive types are assumed to be known to DSDL processing tools a priori, and as such, they need not be defined by the user. Primitive types can be referred to directly by their name from any namespace.

The alignment of primitive types is 1 bit (i.e., no alignment).

3.4.3.1 Hierarchy

The hierarchy of primitive types is documented below.

- **Boolean types.** A boolean-typed value represents a variable of the Boolean algebra. A Boolean-typed value can have two values: true and false. The corresponding DSDL data type name is "`bool`".
- **Algebraic types.** Those are types for which conventional algebraic operators are defined.
  - **Integer types** are used to represent signed and unsigned integer values. See table 3.4.3.1.
    - **Signed integer types.** These are used to represent values which can be negative. The corresponding DSDL data type name pattern is "`int[1-9]\d*`", where the trailing integer represents the length of the serialized representation of the value, in bits, ranging from 2 to 64, inclusive.
    - **Unsigned integer types.** These are used to represent non-negative values. The corresponding DSDL data type name pattern is "`uint[1-9]\d*`", where the trailing integer represents the length of the serialized representation of the value, in bits, ranging from 1 to 64, inclusive.
  - **Floating point types** are used to approximately represent real values. The underlying serialized representation follows the IEEE 754 standard. The corresponding DSDL data type name pattern is "`float(16|32|64)`", where the trailing integer represents the type of the IEEE 754 representation. See table 3.4.3.1.

Category	DSDL names	Range, X is bit length
Signed integers	`int2`, `int3`, `int4` … `int62`, `int63`, `int64`	$\left[-\frac{2^X}{2}, \frac{2^X}{2}-1\right]$
Unsigned integers	`uint1`, `uint2`, `uint3` … `uint62`, `uint63`, `uint64`	$\left[0, 2^X-1\right]$

Table 3.10: Properties of integer types

DSDL name	Representation	Approximate epsilon	Approximate range
`float16`	IEEE 754 binary16	0.001	±65504
`float32`	IEEE 754 binary32	10^{-7}	$\pm10^{39}$
`float64`	IEEE 754 binary64	2×10^{-16}	$\pm10^{308}$

Table 3.11: Properties of floating point types

3.4.3.2 Cast mode

The concept of *cast mode* is defined for all primitive types. The cast mode defines the behavior when a primitive-typed entity is assigned a value that exceeds its range. Such assignment requires some of the information to be discarded; due to the loss of information involved, it is called a *lossy assignment*.

The following cast modes are defined:

Truncated mode — denoted with the keyword "`truncated`" placed before the primitive type name.

Saturated mode — denoted with the optional keyword "`saturated`" placed before the primitive type name. If neither cast mode is specified, saturated mode is assumed by default. This essentially makes the "`saturated`" keyword redundant; it is provided only for consistency.

When a primitive-typed entity is assigned a value that exceeds its range, the resulting value is chosen according to the lossy assignment rules specified in table 3.4.3.2. Cases that are marked as illegal are not permitted in DSDL definitions.

Type category	Truncated mode	Saturated mode (default)
Boolean	Illegal: boolean type with truncated cast mode is not allowed.	Falsity if the value is zero or false, truth otherwise.
Signed integer	Illegal: signed integer types with truncated cast mode are not allowed.	Nearest reachable value.
Unsigned integer	Most significant bits are discarded.	Nearest reachable value.
Floating point	Infinity with the same sign, unless the original value is not-a-number, in which case it will be preserved.	If the original value is finite, the nearest finite value will be used. Otherwise, in the case of infinity or not-a-number, the original value will be preserved.

Table 3.12: Lossy assignment rules per cast mode

Rules of conversion between values of different type categories do not affect compatibility at the protocol level, and as such, they are to be implementation-defined.

3.4.3.3 *Expressions*

At the time of DSDL definition processing, values of primitive types are represented as instances of the `rational` type (section 3.3.1), with the exception of the type `bool`, instances of which are usable in DSDL expressions as-is.

Op	Type	Description
`!`	(`bool`) → `bool`	Logical not.
`||`	(`bool`,`bool`) → `bool`	Logical or.
`&&`	(`bool`,`bool`) → `bool`	Logical and.
`==`	(`bool`,`bool`) → `bool`	Equality.
`!=`	(`bool`,`bool`) → `bool`	Inequality.

Table 3.13: Operators defined on instances of type boolean

3.4.3.4 *Reference list*

An exhaustive list of all void and primitive types ordered by bit length is provided below for reference. Note that the cast mode specifier is omitted intentionally.

1. `void1` `uint1` `bool`
2. `void2` `int2` `uint2`
3. `void3` `int3` `uint3`
4. `void4` `int4` `uint4`
5. `void5` `int5` `uint5`
6. `void6` `int6` `uint6`
7. `void7` `int7` `uint7`
8. `void8` `int8` `uint8`
9. `void9` `int9` `uint9`
10. `void10` `int10` `uint10`
11. `void11` `int11` `uint11`
12. `void12` `int12` `uint12`
13. `void13` `int13` `uint13`
14. `void14` `int14` `uint14`
15. `void15` `int15` `uint15`
16. `void16` `int16` `uint16` `float16`
17. `void17` `int17` `uint17`
18. `void18` `int18` `uint18`
19. `void19` `int19` `uint19`
20. `void20` `int20` `uint20`
21. `void21` `int21` `uint21`

22.	void22	int22	uint22	
23.	void23	int23	uint23	
24.	void24	int24	uint24	
25.	void25	int25	uint25	
26.	void26	int26	uint26	
27.	void27	int27	uint27	
28.	void28	int28	uint28	
29.	void29	int29	uint29	
30.	void30	int30	uint30	
31.	void31	int31	uint31	
32.	void32	int32	uint32	float32
33.	void33	int33	uint33	
34.	void34	int34	uint34	
35.	void35	int35	uint35	
36.	void36	int36	uint36	
37.	void37	int37	uint37	
38.	void38	int38	uint38	
39.	void39	int39	uint39	
40.	void40	int40	uint40	
41.	void41	int41	uint41	
42.	void42	int42	uint42	
43.	void43	int43	uint43	
44.	void44	int44	uint44	
45.	void45	int45	uint45	
46.	void46	int46	uint46	
47.	void47	int47	uint47	
48.	void48	int48	uint48	
49.	void49	int49	uint49	
50.	void50	int50	uint50	
51.	void51	int51	uint51	
52.	void52	int52	uint52	
53.	void53	int53	uint53	
54.	void54	int54	uint54	
55.	void55	int55	uint55	
56.	void56	int56	uint56	
57.	void57	int57	uint57	
58.	void58	int58	uint58	
59.	void59	int59	uint59	
60.	void60	int60	uint60	
61.	void61	int61	uint61	
62.	void62	int62	uint62	
63.	void63	int63	uint63	
64.	void64	int64	uint64	float64

3.4.4 Array types

An array type represents an ordered collection of values. All values in the collection share the same type, which is referred to as *array element type*. The array element type can be any type except:

- void type;
- array type[28].

The number of elements in the array can be specified as a constant expression at the data type definition time, in which case the array is said to be a *fixed-length array*. Alternatively, the number of elements can vary between zero and some static limit specified at the data type definition time, in which case the array is said to be a *variable-length array*. Variable-length arrays with unbounded maximum number of elements are not allowed.

Arrays are defined by adding a pair of square brackets after the array element type specification, where the brackets contain the *array capacity expression*. The array capacity expression shall yield a positive integer of type "`rational`" upon its evaluation; any other value or type renders the current DSDL definition invalid.

[28] Meaning that nested arrays are not allowed; however, the array element type can be a composite type which in turn may contain arrays. In other words, indirect nesting of arrays is permitted.

The array capacity expression can be prefixed with the following character sequences in order to define a variable-length array:

- “<” (ASCII code 60) — indicates that the integer value yielded by the array capacity expression specifies the non-inclusive upper boundary of the number of elements. In this case, the integer value yielded by the array capacity expression shall be greater than one.
- “<=” (ASCII code 60 followed by 61) — same as above, but the upper boundary is inclusive.

If neither of the above prefixes are provided, the resultant definition is that of a fixed-length array.

The alignment of an array equals the alignment of its element type[29].

3.4.5 Composite types

3.4.5.1 Kinds

There are two kinds of composite type definitions: message types and service types. All versions of a data type shall be of the same kind[30].

A service type defines two inner data types: one for service request object, and one for service response object, in that order. The two types are separated by the service response marker (“`---`”) on a separate line.

The presence of the service response marker indicates that the data type definition at hand is of the service kind. At most one service response marker shall appear in a given definition.

3.4.5.2 Dependencies

In order to refer to a composite type from another composite type definition (e.g., for nesting or for referring to an external constant), one has to specify the full data type name of the referred data type followed by its major and minor version numbers separated by the namespace separator character, as demonstrated on figure 3.4.

To facilitate look-up of external dependencies, implementations are expected to obtain from external sources[31] the list of directories that are the roots of namespaces containing the referred dependencies.

$$\underbrace{\texttt{uavcan.node.Heartbeat}}_{\text{full data type name}}\texttt{.}\underbrace{\texttt{1.0}}_{\text{version numbers}}$$

Figure 3.4: Reference to an external composite data type definition

If the referred data type and the referring data type share the same full namespace name, it is allowed to omit the namespace from the referred data type specification leaving only the short data type name, as demonstrated on figure 3.5. In this case, the referred data type will be looked for in the namespace of the referrer. Partial omission of namespace components is not permitted.

$$\underbrace{\texttt{Heartbeat}}_{\text{short data type name}}\texttt{.}\underbrace{\texttt{1.0}}_{\text{version numbers}}$$

Figure 3.5: Reference to an external composite data type definition located in the same namespace

Circular dependencies are not permitted. A circular dependency renders all of the definitions involved in the dependency loop invalid.

If any of the referred definitions are marked as deprecated, the referring definition shall be marked deprecated as well[32]. If a non-deprecated definition refers to a deprecated definition, the referring definition is malformed[33].

When a data type is referred to from within an expression context, it constitutes a literal of type “`metaserializable`” (section 3.3.4). If the referred data type is of the message kind, its attributes are accessible in the referring expression through application of the attribute reference operator “`.`”. The available attributes and their semantics are documented in the section 3.5.2.

[29] E.g., the alignment of `uint5[<=3]` or `int64[3]` is 1 bit (that is, no alignment).
[30] For example, if a data type version 0.1 is of a message kind, all later versions of it shall be messages, too.
[31] For example, from user-provided configuration options.
[32] Deprecation is indicated with the help of a special directive, as explained in section 3.6.
[33] This tainting behavior is designed to prevent unexpected breakage of type hierarchies when one of the deprecated dependencies reaches its end of life.

```
uint64 MY_CONSTANT = vendor.MessageType.1.0.OTHER_CONSTANT
uint64 MY_CONSTANT = MessageType.1.0.OTHER_CONSTANT
# The above is valid if the referring definition and the referred definition
# are located inside the root namespace "vendor".
@print MessageType.1.0
```

3.4.5.3 *Tagged unions*

Any data type definition can be supplied with a special directive (section 3.6) indicating that it forms a tagged union.

A tagged union type shall contain at least two field attributes. A tagged union shall not contain padding field attributes.

The value of a tagged union object is a function of the field attribute which value it is currently holding and the value of the field attribute itself.

To avoid ambiguity, a data type that is not a tagged union is referred to as a *structure*.

3.4.5.4 *Alignment and cumulative bit length set*

The alignment of composite types is one byte (8 bits)[34].

Per the definitions given in 3.4.1.1, a serialized representation of a composite type is padded to 8 bits by inserting padding bits after its last element until the resulting length is a multiple of 8 bits.

Given a set of field attributes, their *cumulative bit length set* is computed by evaluating every permutation of their respective bit length sets plus the required padding.

- For tagged unions, this amounts to the union of the bit length sets of each field plus the bit length set of the implicit union tag.
- Otherwise, the cumulative bit length set is the Cartesian product of the bit length sets of each field plus the required inter-field padding.

Related specifics are given in section 3.7 on data serialization.

3.4.5.5 *Extent and sealing*

As detailed in section 3.8, data types may evolve over time to accommodate new design requirements, new features, to rectify issues, etc. In order to allow gradual migration of deployed systems to revised data types, it is desirable to ensure that they can be modified in a way that does not render new definitions incompatible with their earlier versions. In this context there are two related concepts:

Extent — the minimum amount of memory, in bits, that shall be allocated to store the serialized representation of the type. The extent of any type is always greater than or equal the maximal value of its bit length set. It is always a multiple of 8.

Sealing — a type that is *sealed* is non-evolvable and its extent equals the maximal value of its bit length set[35]. A type that is not sealed is also referred to as *delimited*.

The extent is the growth limit for the maximal bit length of the type as it evolves. The extent should be at least as large as the longest serialized representation of any compatible version of the type, which ensures that an agent leveraging a particular version can correctly process any other compatible version due to the avoidance of unintended truncation of serialized representations.

Serialized representations of evolvable definitions may carry additional metadata which introduces a certain overhead. This may be undesirable in some scenarios, especially in cases where serialized representations of the definition are expected to be highly compact, thereby making the overhead comparatively more significant. In such cases, the designer may opt out of the extensibility by declaring the definition sealed. Serialized representations of sealed definitions do not incur the aforementioned overhead.

The related mechanics are described in section 3.7.5.3.

[34] Regardless of the content. It follows that empty composites can be inserted arbitrarily to force byte alignment of the next field(s) at runtime.
[35] I.e., the smallest possible extent.

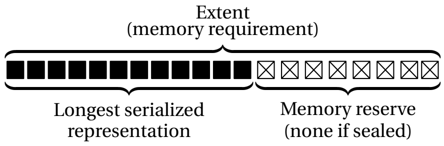


Figure 3.6: Serialized representation and extent

> Because of UAVCAN's commitment to determinism, memory buffer allocation can become an issue. When using a flat composite type (where each field is of a primitive type) with the implicit truncation rule, it is clear that the last defined fields are to be truncated out shall the allocated buffer be too small to accommodate the serialized representation in its entirety. If there is a composite-typed field, this behavior can no longer be relied on. The technical details explaining this are given in section 3.7.5.3.
>
> Conventional protocols manage this simply by delaying the memory requirement identification until runtime, which is unacceptable to UAVCAN. The solution for UAVCAN is to allow the data type author to require implementations to reserve more memory than required by the data type definition unless it is `@sealed` (or unless the implementation does use dynamic memory allocation).

The extent shall be set explicitly using the directive described in section 3.6.2, unless the definition is declared sealed using the directive described in section 3.6.3. The directives are mutually exclusive.

It is allowed for a sealed composite type to nest non-sealed (delimited) composite types, and vice versa.

3.4.5.6 *Bit length set*

The bit length set of a sealed composite type equals the cumulative bit length set of its fields plus the final padding (see section 3.4.5.4).

> The bit length set of the following is $\{8, 24, 40, 56\}$:
>
> ```
> uint16[<=3] foo
> @sealed
> ```
>
> The bit length set of the following is $\{16, 32, 48, 64\}$:
>
> ```
> uint16[<=3] foo
> int2 bar
> @sealed
> ```
>
> The bit length set of the following is $\{8, 16\}$:
>
> ```
> bool[<=3] foo
> @sealed
> ```

The bit length set of a non-sealed (delimited) composite type is dependent only on its extent X, and is defined as follows:

$$\left\{ B_{\mathrm{DH}} + 8b \mid b \in \mathbb{Z},\ 0 \le b \le \frac{X}{8} \right\}$$

where B_{DH} is the bit length of the *delimiter header* as defined in section 3.7.5.3.

This is intentionally not dependent on the fields of the composite because the definition of delimited composites should be possible to change without violating the backward compatibility.

If the bit length set was dependent on the field composition, then a composite A that nests another composite B could have made a fragile assumption about the offset of the fields that follow B that could be broken when B is modified. Example:

```
# A.1.0
B.1.0 x
float32 assume_aligned  # B.1.0 contains a single uint64, assume this field is 32-bit aligned?
@sealed
```

```
# B.1.0
uint64 x
@extent 8 * 8
```

Imagine then that `B.1.0` is replaced with the following:

```
# B.1.1
uint64 x
bool[<=64] y
@extent 8 * 8
```

Once this modification is introduced, the fragile assumption about the alignment of `A.1.0.assume_aligned` would be violated. To avoid this issue, the bit length set definition of delimited types intentionally discards the information about its field composition, forcing dependent types to avoid any non-trivial assumptions.

When serializing an object, the amount of memory needed for storing its serialized representation may be taken as the maximal value of its bit length set minus the size of the delimiter header, since this bound is tighter than the extent yet guaranteed to be sufficient. This optimization is not applicable to deserialization since the actual type of the object may not be known.

3.4.5.7 *Type polymorphism and equivalency*

Type polymorphism is supported in DSDL through structural subtyping. The following definition relies on the concept of *field attribute*, which is introduced in section 3.5.

Polymorphic relations are not defined on service types.

Let B and D be DSDL types that define b and d field attributes, respectively. Let each field attribute be assigned a sequential index according to its position in the DSDL definition (see section 3.2 on statement ordering).

1. Structure subtyping rule — D is a *structural subtype* of B if all conditions are satisfied:
   - neither B nor D define a tagged union[36];
   - neither B nor D are sealed[37];
   - the extent of B is not less than the extent of D[38];
   - B is not D;
   - $b \le d$;
   - for each field attribute of B at index i there is an equal[39] field attribute in D at index i.
2. Tagged union subtyping rule — D is a structural subtype of B if all conditions are satisfied:
   - both B and D define tagged unions;
   - neither B nor D are sealed;
   - the extent of B is not less than the extent of D;
   - B is not D;
   - $b \le d$;
   - $2^{\lceil \log_2 \max(8, \lceil \log_2 b \rceil) \rceil} = 2^{\lceil \log_2 \max(8, \lceil \log_2 d \rceil) \rceil}$[40];
   - for $i \in [0, b)$, the type of the field attribute of D at index i is the same or is a subtype of the type of the field

[36] This is because tagged unions are serialized differently.

[37] Sealed types are serialized in-place as if their definition was directly copied into the outer (containing) type (if any). This circumstance effectively renders them non-modifiable because that may break the bit layout of the outer types (if any). More on this in section 3.7.5.3.

[38] This is to uphold the Liskov substitution principle. A deserializer expecting an instance of B in a serialized representation should be invariant to the replacement $B \leftarrow D$. If the extent of D was larger, then its serialized representation could spill beyond the allocated container, possibly resulting in the truncation of the following data, which in turn could result in incorrect deserialization. See 3.7.

[39] Field attribute equality is defined in section 3.5.

[40] I.e., the length of the implicit union tag field should be the same.

attribute of B at index i.
- for $i \in [0, b)$, the name of the field attribute of D at index i is the same as the name of the field attribute of B at index i.

3. Empty type subtyping rule — D is a structural subtype of B if all conditions are satisfied:
   - $b = 0$[41];
   - neither B nor D are sealed;
   - the extent of B is not less than the extent of D;
   - B is not D.
4. Header subtyping rule — D is a structural subtype of B if all conditions are satisfied:
   - neither B nor D define a tagged union;
   - both B and D are sealed;
   - the first field attribute of D is of type B.

If D is a structural subtype of B, then B is a *structural supertype* of D.

D and B are *structurally equivalent* if D is a structural subtype and a structural supertype of B.

A *type hierarchy* is an ordered set of data types such that for each pair of its members one type is a subtype of another, and for any member its supertypes are located on the left.

[41] If B contains no field attributes, the applicability of the Liskov substitution principle is invariant to whether its subtypes are tagged union types or not.

Subtyping example for structure (non-union) types. First type:

```
float64 a       # Index 0
int16[<=9] b    # Index 1
@extent 32 * 8
```

The second type is a structural subtype of the first type:

```
float64 a       # Index 0
int16[<=9] b    # Index 1
uint8 foo       # Index 2
@extent 32 * 8
```

Subtyping example for union types. First type:

```
@union                            # The implicit union tag field is 8 bits wide
uavcan.primitive.Empty.1.0 foo
float16 bar
uint8 zoo
@extent 128 * 8
```

The second type is a structural subtype of the first type:

```
@union                            # The implicit union tag field is 8 bits wide
uavcan.diagnostic.Record.1.0 foo  # Subtype
float16 bar                       # Same
uint8 zoo                         # Same
int64[<=64] baz                   # New field
@extent 128 * 8
```

A structure type that defines zero fields is a structural supertype of any other structure type, regardless of either or both being a union, provided that its extent is sufficient. A structure type may have an arbitrary number of supertypes as long as the field equality constraint is satisfied.

Header subtyping example. The first type is named `A.1.0`:

```
float64 a
int16[<=9] b
@sealed
```

The second type is a structural subtype of the first type:

```
A.1.0 base
uint8 foo
@sealed
```

The following example in C demonstrates the concept of polymorphic compatibility detached from DSDL.

```c
struct base
{
    int a;
    float b;
};

struct derived_first
{
    int a;
    float b;
    double c;
};

struct derived_second
{
    int a;
    float b;
    short d;
};

float compute(struct base* value)
{
    return (float)value->a + value->b;
}

int main()
{
    struct derived_first foo  = { .a = 123, .b = -456.789F, .c = 123.456 };
    struct derived_second bar = { .a = -123, .b = 456.789F, .d = -123 };
    // Both derived_first and derived_second are structural subtypes of base. The program returns zero.
    return compute(&foo) + compute(&bar);
}
```

3.5 Attributes

An *attribute* is a named (excepting padding fields) entity associated with a particular object or type.

3.5.1 Composite type attributes

A composite type attribute that has a value assigned at the data type definition time is called a *constant attribute*; a composite type attribute that does not have a value assigned at the definition time is called a *field attribute*.

The name of a composite type attribute shall be unique within the data type definition that contains it, and it shall not match any of the reserved name patterns specified in the table 3.2.5. This requirement does not apply to padding fields.

3.5.1.1 *Field attributes*

A field attribute represents a named dynamically assigned value of a statically defined type that can be exchanged over the network as a member of its containing object. The data type of a field attribute shall be of the serializable type category (section 3.4), excepting the void type category, which is not allowed.

Exception applies to the special kind of field attributes — *padding fields*. The type of a padding field attribute shall be of the void category. A padding field attribute may not have a name.

A pair of field attributes is considered equal if, and only if, both field attributes are of the same type, and both share the same name or both are padding field attributes.

Example:

```
uint8[<=10] regular_field   # A field named "regular field"
void16                      # A padding field; no name is permitted
```

3.5.1.2 *Constant attributes*

A constant attribute represents a named statically assigned value of a statically defined type. Values of constant attributes are never exchanged over the network, since they are assumed to be known to all involved nodes by virtue of them sharing the same definition of the data type.

The data type of a constant attribute shall be of the primitive type category (section 3.4).

The value of the constant attribute is determined at the DSDL definition processing time by evaluating its *initialization expression*. The expression shall yield a compatible type upon its evaluation in order to initialize the value of its constant attribute. The set of compatible types depends on the type of the initialized constant attribute, as specified in table 3.5.1.2.

Expression type / Constant type category	`bool`	`rational`	`string`
Boolean	Allowed.	Not allowed.	Not allowed.
Integer	Not allowed.	Allowed if the denominator equals one and the numerator value is within the range of the constant type.	Allowed if the target type is `uint8` and the source string contains one symbol whose code point falls into the range $[0, 127]$.
Floating point	Not allowed.	Allowed if the source value does not exceed the finite range of the constant type. The final value is computed as the quotient of the numerator and the denominator with implementation-defined accuracy.	Not allowed.

Table 3.14: Permitted constant attribute value initialization patterns

Due to the value of a constant attribute being defined at the data type definition time, the cast mode of primitive-typed constants has no observable effect.

A real literal `1234.5678` is represented internally as $\frac{6172839}{5000}$, which can be used to initialize a `float16` value, resulting in 1235.0.

The specification states that the value of a floating-point constant should be computed with an implementation-defined accuracy. UAVCAN avoids strict accuracy requirements in order to ensure compatibility with implementations that rely on non-standard floating point formats. Such laxity in the specification is considered acceptable since the uncertainty is always confined to a single division expression per constant; all preceding computations, if any, are always performed by the DSDL compiler using exact rational arithmetic.

3.5.2 Local attributes

Local attributes are available at the DSDL definition processing time.

As defined in section 3.2, a DSDL definition is an ordered collection of statements; a statement may contain DSDL expressions. An expression contained in a statement number E may refer to a composite type attribute introduced in a statement number A by its name, where $A < E$ and both statements belong to the same data type definition[42]. The representation of the referred attribute in the context of the referring DSDL expression is specified in table 3.5.2.

Attribute category	Value type	Value
Constant attribute	Type of the constant attribute	Value of the constant attribute
Field attribute	Illegal	Illegal

Table 3.15: Local attribute representation

```
uint8 FOO = 123
uint16 BAR = FOO ** 2
@assert BAR == 15129
---  # The request type ends here; its attributes are no longer accessible.
#uint16 BAZ = BAR  # Would fail - BAR is not accessible here.
float64 FOO = 3.14
@assert FOO == 3.14
```

[42] Per 3.1.4, in case of services, this applies only to their request and response types.

3.5.3 Intrinsic attributes

Intrinsic attributes are available in any expression. Their values are constructed by the DSDL processing tool depending on the context, as specified in this section.

3.5.3.1 Offset attribute

The offset attribute is referred to by its identifier "`_offset_`". Its value is of type `set<rational>`.

In the following text, the term *referring statement* denotes a statement containing an expression which refers to the offset attribute. The term *bit length set* is defined in section 3.1.5.

The value of the attribute is a function of the field attribute declarations preceding the referring statement and the category of the containing definition.

If the current definition belongs to the tagged union category, the referring statement shall be located after the last field attribute definition. A field attribute definition following the referring statement renders the current definition invalid. For tagged unions, the value of the offset attribute is defined as the cumulative bit length set[43] of the union's fields, where each element of the set is incremented by the bit length of the implicit union tag field (section 3.7.5).

If the current data definition does not belong to the tagged union category, the referring statement may be located anywhere within the current definition. The value of the offset attribute is defined as the cumulative bit length set[44] of the fields defined in statements preceding the referring statement (see section 3.2 on statement ordering).

```
@union
uint8 a
#@print _offset_  # Would fail: it's a tagged union, _offset_ is undefined until after the last field
uint16 b
@assert _offset_ == {8 + 8,  8 + 16}
---
@assert _offset_ == {0}
float16 a
@assert _offset_ == {16}
void4
@assert _offset_ == {20}
int4 b
@assert _offset_ == {24}
uint8[<4] c
@assert _offset_ == 8 + {24,  32,  40,  48}
@assert _offset_ % 8 == {0}
# One of the main usages for _offset_ is statically proving that the following field is byte-aligned
# for all possible valid serialized representations of the preceding fields. It is done by computing
# a remainder as shown above. If the field is aligned, the remainder set will equal {0}. If the
# remainder set contains other elements, the field may be misaligned under some circumstances.
# If the remainder set does not contain zero, the field is never aligned.
uint8 well_aligned   # Proven to be byte-aligned.
```

3.6 Directives

Per the DSDL grammar specification (section 3.2), a directive may or may not have an associated expression. In this section, it is assumed that a directive does not expect an expression unless explicitly stated otherwise.

If the expectation of an associated directive expression or lack thereof is violated, the containing DSDL definition is malformed.

The effect of a directive is confined to the data type definition that contains it. That means that for service types, unless specifically stated otherwise, a directive placed in the request (response) type affects only the request (response) type.

3.6.1 Tagged union marker

The identifier of the tagged union marker directive is "`union`". Presence of this directive in a data type definition indicates that the data type definition containing this directive belongs to the tagged union category (section 3.4.5.3).

Usage of this directive is subject to the following constraints:

- The directive shall not be used more than once per data type definition.

[43] Section 3.4.5.4.
[44] Section 3.4.5.4.

- The directive shall be placed before the first composite type attribute definition in the current definition.

```
uint8[<64] name # Request is not a union
---
@union          # Response is a union
uint64 natural
#@union         # Would fail - @union is not allowed after the first attribute definition
float64 real
```

3.6.2 Extent specifier

The identifier of the extent specification directive is "`extent`". This directive declares the current data type definition to be delimited (non-sealed) and specifies its extent (section 3.4.5.5). The extent value is obtained by evaluating the provided expression. The expression shall be present and it shall yield a non-negative integer value of type "`rational`" (section 3.4.3) upon its evaluation.

Usage of this directive is subject to the following constraints (otherwise, the definition is malformed):

- The directive shall not be used more than once per data type definition.
- The directive shall be placed after the last attribute definition in the current data type[45].
- The value shall satisfy the constraints given in section 3.4.5.5.
- The data type shall not be sealed.

```
uint64 foo
@extent 256 * 8  # Make the extent 256 bytes large
#@sealed         # Would fail -- mutually exclusive directives
```

```
uint64[<=64] bar
@extent _offset_.max * 2
#float32 baz  # Would fail (protects against incorrectly computing
              # the extent when it is a function of _offset_)
```

3.6.3 Sealing marker

The identifier of the sealing marker directive is "`sealed`". This directive marks the current data type sealed (section 3.4.5.5).

Usage of this directive is subject to the following constraints (otherwise, the definition is malformed):

- The directive shall not be used more than once per data type definition.
- The extent directive shall not be used in this data type definition.

```
uint64 foo
@sealed       # The request type is sealed.
#@extent 128  # Would fail -- cannot specify extent for sealed type
---
float64 bar   # The response type is not sealed.
@extent 4000 * 8
```

3.6.4 Deprecation marker

The identifier of the deprecation marker directive is "`deprecated`". Presence of this directive in a data type definition indicates that the current version of the data type definition is nearing the end of its life cycle and may be removed soon. The data type versioning principles are explained in section 3.8.

Code generation tools should use this directive to reflect the impending removal of the current data type version in the generated code.

Usage of this directive is subject to the following constraints:

- The directive shall not be used more than once per data type definition.
- The directive shall be placed before the first composite type attribute definition in the definition.
- In case of service types, this directive may only be placed in the request type, and it affects the response type as well.

[45] This constraint is to help avoid issues where the extent is defined as a function of the offset past the last field of the type, and a new field is mistakenly added after the extent directive.

```
@deprecated        # Applies to the entire definition
uint8 FOO = 123
#@deprecated       # Would fail - shall be placed before the first attribute definition
---
#@deprecated       # Would fail - shall be placed in the request type
```

A C++ class generated from the above definition could be annotated with the `[[deprecated]]` attribute.

A Rust structure generated from the above definition could be annotated with the `#[deprecated]` attribute.

A Python class generated from the above definition could raise `DeprecationWarning` upon usage.

3.6.5 Assertion check

The identifier of the assertion check directive is "`assert`". The assertion check directive expects an expression which shall yield a value of type "`bool`" (section 3.4.3) upon its evaluation.

If the expression yields truth, the assertion check directive has no effect.

If the expression yields falsity, a value of type other than "`bool`", or fails to evaluate, the containing DSDL definition is malformed.

```
float64 real
@assert _offset_ == {32}    # Would fail: {64} != {32}
```

3.6.6 Print

The identifier of the print directive is "`print`". The print directive may or may not be provided with an associated expression.

If the expression is not provided, the behavior is implementation-defined.

If the expression is provided, it is evaluated and its result is displayed by the DSDL processing tool in a human-readable implementation-defined form. Implementations should strive to produce textual representations that form valid DSDL expressions themselves, so that they would produce the same value if evaluated by a DSDL processing tool.

If the expression is provided but cannot be evaluated, the containing DSDL definition is malformed.

```
float64 real
@print _offset_ / 6                # Possible output: {32/3}
@print uavcan.node.Heartbeat.1.0   # Possible output: uavcan.node.Heartbeat.1.0
@print bool[<4]                    # Possible output: saturated bool[<=3]
@print float64                     # Possible output: saturated float64
@print {123 == 123, false}         # Possible output: {true, false}
@print 'we all float64 down here\n' # Possible output: 'we all float64 down here\n'
```

3.7 Data serialization

3.7.1 General principles

3.7.1.1 Design goals

The main design principle behind the serialized representations described in this section is the maximization of compatibility with native representations used by currently existing and likely future computer microarchitectures. The goal is to ensure that the serialized representations defined by DSDL match internal data representations of modern computers, so that, ideally, a typical system will not have to perform any data conversion whatsoever while exchanging data over a UAVCAN network.

The implicit truncation and implicit zero extension rules introduced in this section are designed to facilitate structural subtyping and to enable extensibility of data types while retaining backward compatibility. This is a conscious trade-off between runtime type checking and long-term stability guarantees. This model assumes that data type compatibility is determined statically and is not, normally, enforced at runtime.

3.7.1.2 Bit and byte ordering

The smallest atomic data entity is a bit. Eight bits form one byte; within the byte, the bits are ordered so that the least significant bit is considered first (0-th index), and the most significant bit is considered last (7-th index).

Numeric values consisting of multiple bytes are arranged so that the least significant byte is encoded first; such format is also known as little-endian.

bit index (M 7 6 5 4 3 2 1 0 L) … bit index (M 7 6 5 4 3 2 1 0 L)

$$\underbrace{01010101}_{\text{least significant byte}}\ldots\underbrace{01010101}_{\text{most significant byte}}$$

Figure 3.7: Bit and byte ordering

3.7.1.3 *Implicit truncation of excessive data*

When a serialized representation is deserialized, implementations shall ignore any excessive (unused) data or padding bits remaining upon deserialization[46]. The total size of the serialized representation is reported either by the underlying transport layer, or, in the case of nested objects, by the *delimiter header* (section 3.7.5.3).

As a consequence of the above requirement the transport layer can introduce additional zero padding bits at the end of a serialized representation to satisfy data size granularity constraints. Non-zero padding bits are not allowed[47].

> Because of implicit truncation a serialized representation constructed from an instance of type B can be deserialized into an instance of type A as long as B is a structural subtype of A.
>
> Let x be an instance of data type B, which is defined as follows:
>
> ```
> float32 parameter
> float32 variance
> ```
>
> Let A be a structural supertype of B, being defined as follows:
>
> ```
> float32 parameter
> ```
>
> Then the serialized representation of x can be deserialized into an instance of A. The topic of data type compatibility is explored in detail in section 3.8.

3.7.1.4 *Implicit zero extension of missing data*

For the purposes of deserialization routines, the serialized representation of any instance of a data type shall *implicitly* end with an infinite sequence of bits with a value of zero (0).[48].

Despite this rule, implementations are not allowed to intentionally truncate trailing zeros upon construction of a serialized representation of an object[49].

The total size of the serialized representation is reported either by the underlying transport layer, or, in the case of nested objects, by the *delimiter header* (section 3.7.5.3).

[46] The presence of unused data should not be considered an error.

[47] Because padding bits may be misinterpreted as part of the serialized representation.

[48] This can be implemented by checking for out-of-bounds access during deserialization and returning zeros if an out-of-bounds access is detected. This is where the name "implicit zero extension rule" is derived from.

[49] Intentional truncation is prohibited because a future revision of the specification may remove the implicit zero extension rule. If intentional truncation were allowed, removal of this rule would break backward compatibility.

The implicit zero extension rule enables extension of data types by introducing additional fields without breaking backward compatibility with existing deployments. The topic of data type compatibility is explored in detail in section 3.8.

The following example assumes that the reader is familiar with the variable-length array serialization rules, explained in section 3.7.4.2.

Let the data type A be defined as follows:

```
uint8 scalar
```

Let x be an instance of A, where the value of `scalar` is 4. Let the data type B be defined as follows:

```
uint8[<256] array
```

Then the serialized representation of x can be deserialized into an instance of B where the field `array` contains a sequence of four zeros: $0, 0, 0, 0$.

3.7.1.5 *Error handling*

In this section and further, an object that nests other objects is referred to as an *outer object* in relation to the nested object.

Correct UAVCAN types shall have no serialization error states.

A deserialization process may encounter a serialized representation that does not belong to the set of serialized representations of the data type at hand. In such case, the invalid serialized representation shall be discarded and the implementation shall explicitly report its inability to complete the deserialization process for the given input. Correct UAVCAN types shall have no other deserialization error states.

Failure to deserialize a nested object renders the outer object invalid[50].

3.7.2 Void types

The serialized representation of a void-typed field attribute is constructed as a sequence of zero bits. The length of the sequence equals the numeric suffix of the type name.

When a void-typed field attribute is deserialized, the values of respective bits are ignored; in other words, any bit sequence of correct length is a valid serialized representation of a void-typed field attribute. This behavior facilitates usage of void fields as placeholders for non-void fields introduced in newer versions of the data type (section 3.8).

The following data type will be serialized as a sequence of three zero bits 000_2:

```
void3
```

The following bit sequences are valid serialized representations of the type: 000_2, 001_2, 010_2, 011_2, 100_2, 101_2, 110_2, 111_2.

Shall the padding field be replaced with a non-void-typed field in a future version of the data type, nodes utilizing the newer definition may be able to retain compatibility with nodes using older types, since the specification guarantees that padding fields are always initialized with zeros:

```
# Version 1.1
float64 a
void64
```

```
# Version 1.2
float64 a
float32 b  # Messages v1.1 will be interpreted such that b = 0.0
void32
```

[50] Therefore, failure in a single deeply nested object propagates upward, rendering the entire structure invalid. The motivation for such behavior is that it is likely that if an inner object cannot be deserialized, then the outer object is likely to be also invalid.

3.7.3 Primitive types

3.7.3.1 General principles

Implementations where native data formats are incompatible with those adopted by UAVCAN shall perform conversions between the native formats and the corresponding UAVCAN formats during serialization and deserialization. Implementations shall avoid or minimize information loss and/or distortion caused by such conversions.

Serialized representations of instances of the primitive type category that are longer than one byte (8 bits) are constructed as follows. First, only the least significant bytes that contain the used bits of the value are preserved; the rest are discarded following the lossy assignment policy selected by the specified cast mode. Then the bytes are arranged in the least-significant-byte-first order[51]. If the bit width of the value is not an integer multiple of eight (8) then the next value in the type will begin starting with the next bit in the current byte. If there are no further values then the remaining bits shall be zero (0).

> The value 1110 1101 1010_2 (3802 in base-10) of type `uint12` is encoded as follows. The bit sequence is shown in the base-2 system, where bytes (octets) are comma-separated:
>
> byte 0 byte 1
>
> 7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0
>
> $$\underbrace{11011010}_{\text{Least significant 8 bits of } 3802_{10}},\underbrace{????}_{\text{Next object or zero padding bits}}\underbrace{1110}_{\text{Most significant 4 bits of } 3802_{10}}$$

3.7.3.2 Boolean types

The serialized representation of a value of type `bool` is a single bit. If the value represents falsity, the value of the bit is zero (0); otherwise, the value of the bit is one (1).

3.7.3.3 Unsigned integer types

The serialized representation of an unsigned integer value of length n bits (which is reflected in the numerical suffix of the data type name) is constructed as if the number were to be written in base-2 numerical system with leading zeros preserved so that the total number of binary digits would equal n.

> The serialized representation of integer 42 of type `uint7` is 0101010_2.

3.7.3.4 Signed integer types

The serialized representation of a non-negative value of a signed integer type is constructed as described in section 3.7.3.3.

The serialized representation of a negative value of a signed integer type is computed by applying the following transformation:

$$2^n + x$$

where n is the bit length of the serialized representation (which is reflected in the numerical suffix of the data type name) and x is the value whose serialized representation is being constructed. The result of the transformation is a positive number, whose serialized representation is then constructed as described in section 3.7.3.3.

The representation described here is widely known as *two's complement*.

> The serialized representation of integer -42 of type `int7` is 1010110_2.

3.7.3.5 Floating point types

The serialized representation of floating point types follows the IEEE 754 series of standards as follows:

- `float16` — IEEE 754 binary16;
- `float32` — IEEE 754 binary32;
- `float64` — IEEE 754 binary64.

Implementations that model real numbers using any method other than IEEE 754 shall be able to model positive infinity, negative infinity, signaling NaN[52], and quiet NaN.

[51] Also known as "little endian".

[52] Per the IEEE 754 standard, NaN stands for "not-a-number" – a set of special bit patterns that represent lack of a meaningful value.

3.7.4 Array types

3.7.4.1 *Fixed-length array types*

Serialized representations of a fixed-length array of n elements of type T and a sequence of n field attributes of type T are equivalent.

Serialized representations of the following two data type definitions are equivalent:

```
AnyType[3] array
```

```
AnyType item_0
AnyType item_1
AnyType item_2
```

3.7.4.2 *Variable-length array types*

A serialized representation of a variable-length array consists of two segments: the implicit length field immediately followed by the array elements.

The implicit length field is of an unsigned integer type. The serialized representation of the implicit length field is injected in the beginning of the serialized representation of its array. The bit length of the unsigned integer value is first determined as follows:

$$b = \lceil \log_2 (c + 1) \rceil$$

where c is the capacity (i.e., the maximum number of elements) of the variable-length array and b is the minimum number of bits needed to encode c as an unsigned integer. An additional transformation of b ensures byte alignment of this implicit field when serialized[53]:

$$2^{\lceil \log_2 (\max(8,b)) \rceil}$$

The number of elements n contained in the variable-length array is encoded in the serialized representation of the implicit length field as described in section 3.7.3.3. By definition, $n \le c$; therefore, bit sequences where the implicit length field contains values greater than c do not belong to the set of serialized representations of the array.

The rest of the serialized representation is constructed as if the variable-length array was a fixed-length array of n elements[54].

[53] Future updates to the specification may allow this second step to be modified but the default action will always be to byte-align the implicit length field.

[54] Observe that the implicit array length field, per its definition, is guaranteed to never break the alignment of the following array elements. There may be no padding between the implicit array length field and its elements.

Data type authors must take into account that variable-length arrays with a capacity of ≤ 255 elements will consume an additional 8 bits of the serialized representation (where a capacity of ≤ 65535 will consume 16 bits and so on). For example:

```
uint8 first
uint8[<=6] second             # The implicit length field is 8 bits wide
@assert _offset_.max / 8 <= 7  # This would fail.
```

In the above example the author attempted to fit the message into a single Classic CAN frame but did not account for the implicit length field. The correct version would be:

```
uint8 first
uint8[<=5] second             # The implicit length field is 8 bits wide
@assert _offset_.max / 8 <= 7  # This would pass.
```

If the array contained three elements, the resulting set of its serialized representations would be equivalent to that of the following definition:

```
uint8 first
uint8 implicit_length_field  # Set to 3, because the array contains three elements
uint8 item_0
uint8 item_1
uint8 item_2
```

3.7.5 Composite types

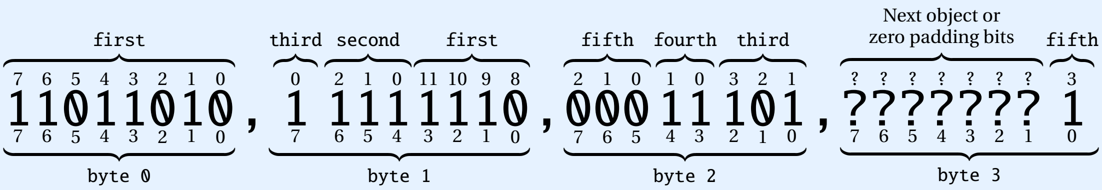
3.7.5.1 Sealed structure

A serialized representation of an object of a sealed composite type that is not a tagged union is a sequence of serialized representations of its field attribute values joined into a bit sequence, separated by padding if such is necessary to satisfy the alignment requirements. The ordering of the serialized representations of the field attribute values follows the order of field attribute declaration.

Consider the following definition, where the fields are assigned runtime values shown in the comments:

```
#                          decimal          bit sequence   comment
truncated uint12 first   # +48858     1011_1110_1101_1010   overflow, MSB truncated
saturated  int3  second  #     -1                     111   two's complement
saturated  int4  third   #     -5                    1011   two's complement
saturated  int2  fourth  #     -1                      11   two's complement
truncated uint4  fifth   #   +136               1000_1000   overflow, MSB truncated
@sealed
```

It can be seen that the bit layout is rather complicated because the field boundaries do not align with byte boundaries, which makes it a good case study. The resulting serialized byte sequence is shown below in the base-2 system:



Note that some of the complexity of the above illustration stems from the modern convention of representing numbers with the most significant components on the left moving to the least significant component of the number of the right. If you were to reverse this convention the bit sequences for each type in the composite would seem to be continuous as they crossed byte boundaries. Using this reversed representation, however, is not recommended because the convention is deeply ingrained in most readers, tools, and technologies.

3.7.5.2 Sealed tagged union

Similar to variable-length arrays, a serialized representation of a sealed tagged union consists of two segments: the implicit *union tag* value followed by the selected field attribute value.

The implicit union tag is an unsigned integer value whose serialized representation is implicitly injected in the beginning of the serialized representation of its tagged union. The bit length of the implicit union tag is determined as follows:

$$b = \lceil \log_2 n \rceil$$

where n is the number of field attributes in the union, $n \geq 2$ and b is the minimum number of bits needed to encode n as an unsigned integer. An additional transformation of b ensures byte alignment of this implicit field when serialized[55]:

$$2^{\lceil \log_2 (\max(8, b)) \rceil}$$

Each of the tagged union field attributes is assigned an index according to the order of their definition; the order follows that of the DSDL statements (see section 3.2 on statement ordering). The first defined field attribute is assigned the index 0 (zero), the index of each following field attribute is incremented by one.

The index of the field attribute whose value is currently held by the tagged union is encoded in the serialized representation of the implicit union tag as described in section 3.7.3.3. By definition, $i < n$, where i is the index of the current field attribute; therefore, bit sequences where the implicit union tag field contains values that are greater than or equal n do not belong to the set of serialized representations of the tagged union.

The serialized representation of the implicit union tag is immediately followed by the serialized representation of the currently selected field attribute value[56].

Consider the following example:

```
@sealed
@union           # In this case, the implicit union tag is one byte wide
uint16 FOO = 42  # A regular constant attribute
uint16 a         # Field index 0
uint8 b          # Field index 1
uint32 BAR = 42  # Another regular constant attribute
float64 c        # Field index 2
```

In order to serialize the field b, the implicit union tag shall be assigned the value 1. The following type will have an identical layout:

```
@sealed
uint8 implicit_union_tag  # Set to 1
uint8 b                   # The actual value
```

Suppose that the value of b is 7. The resulting serialized representation is shown below in the base-2 system:

$$\overset{\text{byte 0}}{\underbrace{00000001}_{\text{union tag}}},\overset{\text{byte 1}}{\underbrace{00000111}_{\text{field b}}}$$

[55] Future updates to the specification may allow this second step to be modified but the default action will always be to byte-align the implicit length field.

[56] Observe that the implicit union tag field, per its definition, is guaranteed to never break the alignment of the following field. There may be no padding between the implicit union tag field and the selected field.

Let the following data type be defined under the short name `Empty` and version 1.0:

```
# Empty. The only valid serialized representation is an empty bit sequence.
@sealed
```

Consider the following union:

```
@sealed
@union
Empty.1.0 none
AnyType.1.0 some
```

The set of serialized representations of the union given above is equivalent to that of the following variable-length array:

```
@sealed
AnyType.1.0[<=1] maybe_some
```

3.7.5.3 *Delimited types*

Objects of delimited (non-sealed) composite types that are nested inside other objects[57] are serialized into opaque containers that consist of two parts: the fixed-length *delimiter header*, immediately followed by the serialized representation of the object as if it was of a sealed type.

Objects of delimited composite types that are *not* nested inside other objects (i.e., top-level objects) are serialized as if they were of a sealed type (without the delimiter header). The delimiter header, therefore, logically belongs to the container object rather than the contained one.

Top-level objects do not require the delimiter header because the change in their length does not necessarily affect the backward compatibility thanks to the implicit truncation rule (section 3.7.1.3) and the implicit zero extension rule (section 3.7.1.4).

The delimiter header is an implicit field of type `uint32` that encodes the length of the serialized representation it precedes in bytes[58]. During deserialization, if the length of the serialized representation reported by its delimiter header does not match the expectation of the deserializer, the implicit truncation (section 3.7.1.3) and the implicit zero extension (section 3.7.1.4) rules apply.

The length encoded in a delimiter header cannot exceed the number of bytes remaining between the delimiter header and the end of the serialized representation of the outer object. Otherwise, the serialized representation of the outer object is invalid and is to be discarded (section 3.7.1.5).

It is allowed for a sealed composite type to nest non-sealed composite types, and vice versa. No special rules apply in such cases.

[57] Of any type, not necessarily composite; e.g., arrays.

[58] Remember that by virtue of the padding requirement (section 3.4.5.4), the length of the serialized representation of a composite type is always an integer number of bytes.

The resulting serialized representation of a delimited composite is identical to `uint8[<2**32]` (sans the higher alignment requirement). The implicit array length field is like the delimiter header, and the array content is the serialized representation of the composite as if it was sealed.

The following illustrates why this is necessary for robust extensibility. Suppose that some composite C contains two fields whose types are A and B. The fields of A are a_0, a_1; likewise, B contains b_0, b_1.

Suppose that C' is modified such that A' contains an extra field a_2. If A (and A') were sealed, this would result in the breakage of compatibility between C and C' as illustrated in figure 3.8 because the positions of the fields of B (which is sealed) would be shifted by the size of a_2.

The use of opaque containers allows the implicit truncation and the implicit zero extension rules to apply at any level of nesting, enabling agents expecting C to truncate a_2 away, and enabling agents expecting C' to zero-extend a_2 if it is not present, as shown in figure 3.9, where H_A is the delimiter header of A. Observe that it is irrelevant whether C (same as C') is sealed or not.

C	a_0	a_1	b_0	b_1	
	✓	✓	×	×	×
C'	a_0	a_1	a_2	b_0	b_1

Figure 3.8: Non-extensibility of sealed types

C	H_A	a_0	a_1	...	b_0	b_1
	✓	✓	✓	✓	✓	✓
C'	H_A	a_0	a_1	a_2	b_0	b_1

Figure 3.9: Extensibility of delimited types with the help of the delimiter header

This example also illustrates why the extent is necessary. Per the rules set forth in 3.4.5.5, it is required that the extent (i.e., the buffer memory requirement) of A shall be large enough to accommodate serialized representations of A', and, therefore, the extent of C is large enough to accommodate serialized representations of C'. If that were not the case, then an implementation expecting C would be unable to correctly process C' because the implicit truncation rule would have cut off b_1, which is unexpected.

The design decision to make the delimiter header of a fixed width may not be obvious so it's worth explaining. There are two alternatives: making it variable-length and making the length a function of the extent (section 3.4.5.5). The first option does not align with the rest of the specification because DSDL does not make use of variable-length integers (unlike some other formats, like Google Protobuf, for example), and because a variable-length length (sic!) prefix would have somewhat complicated the bit length set computation. The second option would make nested hierarchies (composites that nest other composites) possibly highly fragile because the change of the extent of a deeply nested type may inadvertently move the delimiter header of an outer type into a different length category, which would be disastrous for compatibility and hard to spot. There is an in-depth discussion of this issue (and other related matters) on the forum.

The fixed-length delimiter header may be considered large, but delimited types tend to also be complex, which makes the overhead comparatively insignificant, whereas sealed types that tend to be compact and overhead-sensitive do not contain the delimiter header.

In order to efficiently serialize an object of a delimited type, the implementation may need to perform a second pass to reach the delimiter header after the object is serialized, because before that, the value of the delimiter header cannot be known unless the object is of a fixed-size (i.e., the cardinality of the bit length set is one).

Consider:

```
uint8[<=4] x
```

Let $\mathtt{x} = [4, 2]$, then the nested serialized representation would be constructed as:

1. Memorize the current memory address M_{origin}.
2. Skip 32 bits.
3. Encode the length: 2 elements.
4. Encode $x_0 = 4$.
5. Encode $x_1 = 2$.
6. Memorize the current memory address M_{current}.
7. Go back to M_{origin}.
8. Encode a 32-bit wide value of $(M_{\text{current}} - M_{\text{origin}})$.
9. Go back to M_{current}.

However, if the object is known to be of a constant size, the above can be simplified, because there may be only one possible value of the delimiter header. Automatic code generation tools should take advantage of this knowledge.

3.8 Compatibility and versioning

3.8.1 Rationale

Data type definitions may evolve over time as they are refined to better address the needs of their applications. UAVCAN defines a set of rules that allow data type designers to modify and advance their data type definitions while ensuring backward compatibility and functional safety.

3.8.2 Semantic compatibility

A data type A is *semantically compatible* with a data type B if relevant behavioral properties of the application are invariant under the substitution of A with B. The property of semantic compatibility is commutative.

The following two definitions are semantically compatible and can be used interchangeably:

```
uint16 FLAG_A = 1
uint16 FLAG_B = 256
uint16 flags
@extent 16
```

```
uint8 FLAG_A = 1
uint8 FLAG_B = 1
uint8 flags_a
uint8 flags_b
@extent 16
```

It should be noted here that due to different set of field and constant attributes, the source code auto-generated from the provided definitions may be not drop-in replaceable, requiring changes in the application; however, source-code-level application compatibility is orthogonal to data type compatibility.

The following supertype may or may not be semantically compatible with the above depending on the semantics of the removed field:

```
uint8 FLAG_A = 1
uint8 flags_a
@extent 16
```

Let node A publish messages of the following type:

```
float32 foo
float64 bar
@extent 128
```

Let node B subscribe to the same subject using the following data type definition:

```
float32 foo
float64 bar
int16   baz  # Extra field; implicit zero extension rule applies.
@extent 128
```

Let node C subscribe to the same subject using the following data type definition:

```
float32 foo
# The field 'bar' is missing; implicit truncation rule applies.
@extent 128
```

Provided that the semantics of the added and omitted fields allow it, the nodes will be able to interoperate successfully despite using different data type definitions.

3.8.3 Versioning

3.8.3.1 General assumptions

The concept of versioning applies only to composite data types. As such, unless specifically stated otherwise, every reference to "data type" in this section implies a composite data type.

A data type is uniquely identified by its full name, assuming that every root namespace is uniquely named. There is one or more versions of every data type.

A data type definition is uniquely identified by its full name and the version number pair. In other words, there may be multiple definitions of a data type differentiated by their version numbers.

3.8.3.2 Versioning principles

Every data type definition has a pair of version numbers — a major version number and a minor version number, following the principles of semantic versioning.

For the purposes of the following definitions, a *release* of a data type definition stands for the disclosure of the data type definition to its intended users or to the general public, or for the commencement of usage of the data type definition in a production system.

In order to ensure a deterministic application behavior and ensure a robust migration path as data type definitions evolve, all data type definitions that share the same full name and the same major version number shall be semantically compatible with each other.

The versioning rules do not extend to scenarios where the name of a data type is changed, because that would essentially construe the release of a new data type, which relieves its designer from all compatibility requirements. When a new data type is first released, the version numbers of its first definition shall be assigned "1.0" (major 1, minor 0).

In order to ensure predictability and functional safety of applications that leverage UAVCAN, it is recommended that once a data type definition is released, its DSDL source text, name, version numbers, fixed port-ID, extent, sealing, and other properties cannot undergo any modifications whatsoever, with the following exceptions:

- Whitespace changes of the DSDL source text are allowed, excepting string literals and other semantically sensitive contexts.
- Comment changes of the DSDL source text are allowed as long as such changes do not affect semantic compatibility of the definition.
- A deprecation marker directive (section 3.6) can be added or removed[59].

Addition or removal of the fixed port identifier is not permitted after a data type definition of a particular version is released.

[59] Removal is useful when a decision to deprecate a data type definition is withdrawn.

Therefore, substantial changes can be introduced only by releasing new definitions (i.e., new versions) of the same data type. If it is desired and possible to keep the same major version number for a new definition of the data type, the minor version number of the new definition shall be one greater than the newest existing minor version number before the new definition is introduced. Otherwise, the major version number shall be incremented by one and the minor version shall be set to zero.

An exception to the above rules applies when the major version number is zero. Data type definitions bearing the major version number of zero are not subjected to any compatibility requirements. Released data type definitions with the major version number of zero are permitted to change in arbitrary ways without any regard for compatibility. It is recommended, however, to follow the principles of immutability, releasing every subsequent definition with the minor version number one greater than the newest existing definition.

For any data type, there shall be at most one definition per version. In other words, there shall be exactly one or zero definitions per combination of data type name and version number pair.

All data types under the same name shall be also of the same kind. In other words, if the first released definition of a data type is of the message kind, all other versions shall also be of the message kind.

All data types under the same name and major version number should share the same extent and the same sealing status. It is therefore advised to:

- Avoid marking types sealed, especially complex types, because it is likely to render their evolution impossible.
- When the first version is released, its extent should be sufficiently large to permit addition of new fields in the future. Since the value of extent does not affect the network traffic, it is safe to pick a large value without compromising the temporal properties of the system.

3.8.3.3 *Fixed port identifier assignment constraints*

The following constraints apply to fixed port-ID assignments:

$$
\begin{aligned}
\exists P(x_{a.b}) &\rightarrow \exists P(x_{a.c}) && \mid b < c;\ x \in (M \cup S) \\
\exists P(x_{a.b}) &\rightarrow P(x_{a.b}) = P(x_{a.c}) && \mid b < c;\ x \in (M \cup S) \\
\exists P(x_{a.b}) \wedge \exists P(x_{c.d}) &\rightarrow P(x_{a.b}) \neq P(x_{c.d}) && \mid a \neq c;\ x \in (M \cup S) \\
\exists P(x_{a.b}) \wedge \exists P(y_{c.d}) &\rightarrow P(x_{a.b}) \neq P(y_{c.d}) && \mid x \neq y;\ x \in T;\ y \in T;\ T = \{M, S\}
\end{aligned}
$$

where $t_{a.b}$ denotes a data type t version $a.b$ (a major, b minor); $P(t)$ denotes the fixed port-ID (whose existence is optional) of data type t; M is the set of message types, and S is the set of service types.

3.8.3.4 *Data type version selection*

DSDL compilers should compile every available data type version separately, allowing the application to choose from all available major and minor version combinations.

When emitting a transfer, the major version of the data type is chosen at the discretion of the application. The minor version should be the newest available one under the chosen major version.

When receiving a transfer, the node deduces which major version of the data type to use from its port identifier (either fixed or non-fixed). The minor version should be the newest available one under the deduced major version[60].

It follows from the above two rules that when a node is responding to a service request, the major data type version used for the response transfer shall be the same that is used for the request transfer. The minor versions may differ, which is acceptable due to the major version compatibility requirements.

A simple usage example is provided in this intermission.

Suppose a vendor named "Sirius Cybernetics Corporation" is contracted to design a cryopod management data bus for a colonial spaceship "Golgafrincham B-Ark". Having consulted with applicable specifications and standards, an engineer came up with the following definition of a cryopod status message type (named `sirius_cyber_corp.b_ark.cryopod.Status`):

```
# sirius_cyber_corp.b_ark.cryopod.Status.0.1

float16 internal_temperature    # [kelvin]
float16 coolant_temperature     # [kelvin]
```

[60] Such liberal minor version selection policy poses no compatibility risks since all definitions under the same major version are compatible with each other.

```
uint8 FLAG_COOLING_SYSTEM_A_ACTIVE = 1
uint8 FLAG_COOLING_SYSTEM_B_ACTIVE = 2
# Status flags in the lower bits.
uint8 FLAG_PSU_MALFUNCTION = 32
uint8 FLAG_OVERHEATING     = 64
uint8 FLAG_CRYOBOX_BREACH  = 128
# Error flags in the higher bits.
uint8 flags  # Storage for the above defined flags (this is not the recommended practice).

@extent 1024 * 8  # Pick a large extent to allow evolution. Does not affect network traffic.
```

The definition is then deployed to the first prototype for initial laboratory testing. Since the definition is experimental, the major version number is set to zero in order to signify the tentative nature of the definition. Suppose that upon completion of the first trials it is identified that the units should track their power consumption in real time for each of the three redundant power supplies independently.

It is easy to see that the amended definition shown below is not semantically compatible with the original definition; however, it shares the same major version number of zero, because the backward compatibility rules do not apply to zero-versioned data types to allow for low-overhead experimentation before the system is deployed and fielded.

```
# sirius_cyber_corp.b_ark.cryopod.Status.0.2

truncated float16 internal_temperature    # [kelvin]
truncated float16 coolant_temperature     # [kelvin]

saturated float32 power_consumption_0     # [watt] Power consumption by the redundant PSU 0
saturated float32 power_consumption_1     # [watt] likewise for PSU 1
saturated float32 power_consumption_2     # [watt] likewise for PSU 2
# breaking compatibility with Status.0.1 is okay because the major version is 0

uint8 FLAG_COOLING_SYSTEM_A_ACTIVE = 1
uint8 FLAG_COOLING_SYSTEM_B_ACTIVE = 2
# Status flags in the lower bits.
uint8 FLAG_PSU_MALFUNCTION = 32
uint8 FLAG_OVERHEATING     = 64
uint8 FLAG_CRYOBOX_BREACH  = 128
# Error flags in the higher bits.
uint8 flags  # Storage for the above defined flags (this is not the recommended practice).

@extent 512 * 8  # Extent can be changed freely because v0.x does not guarantee compatibility.
```

The last definition is deemed sufficient and is deployed to the production system under the version number of 1.0: `sirius_cyber_corp.b_ark.cryopod.Status.1.0`.

Having collected empirical data from the fielded systems, the Sirius Cybernetics Corporation has identified a shortcoming in the v1.0 definition, which is corrected in an updated definition. Since the updated definition, which is shown below, is semantically compatible[a] with v1.0, the major version number is kept the same and the minor version number is incremented by one:

```
# sirius_cyber_corp.b_ark.cryopod.Status.1.1

saturated float16 internal_temperature    # [kelvin]
saturated float16 coolant_temperature     # [kelvin]

float32[3] power_consumption    # [watt] Power consumption by the PSU

bool flag_cooling_system_a_active
bool flag_cooling_system_b_active
# Status flags (this is the recommended practice).

void3   # Reserved for other flags

bool flag_psu_malfunction
bool flag_overheating
bool flag_cryobox_breach
# Error flags (this is the recommended practice).

@extent 512 * 8  # Extent is to be kept unchanged now to avoid breaking compatibility.
```

Since the definitions v1.0 and v1.1 are semantically compatible, UAVCAN nodes using either of them can successfully interoperate on the same bus.

Suppose further that at some point a newer version of the cryopod module, equipped with better temperature sensors, is released. The definition is updated accordingly to use `float32` for the temperature fields instead of `float16`. Seeing as that change breaks the compatibility, the major version number has to be incremented by one, and the minor version number has to be reset back to zero:

```
# sirius_cyber_corp.b_ark.cryopod.Status.2.0

float32 internal_temperature    # [kelvin]
float32 coolant_temperature     # [kelvin]

float32[3] power_consumption    # [watt] Power consumption by the PSU

bool flag_cooling_system_a_active
bool flag_cooling_system_b_active
void3
bool flag_psu_malfunction
bool flag_overheating
bool flag_cryobox_breach

@extent 768 * 8  # Since the major version number is different, extent can be changed.
```

Imagine that later it was determined that the module should report additional status information relating to the coolant pump. Thanks to the implicit truncation (section 3.7.1.3), implicit zero extension (section 3.7.1.4), and the delimited serialization (section 3.7.5.3), the new fields can be introduced in a semantically-compatible way without releasing a new major version of the data type:

```
# sirius_cyber_corp.b_ark.cryopod.Status.2.1

float32 internal_temperature    # [kelvin]
float32 coolant_temperature     # [kelvin]

float32[3] power_consumption    # [watt] Power consumption by the PSU

bool flag_cooling_system_a_active
bool flag_cooling_system_b_active
void3
bool flag_psu_malfunction
bool flag_overheating
bool flag_cryobox_breach

float32 rotor_angular_velocity  # [radian/second] (usage of RPM would be non-compliant)
float32 volumetric_flow_rate    # [meter^3/second]
# Coolant pump fields (extension over v2.0; implicit truncation/extension rules apply)
# If zero, assume that the values are unavailable.

@extent 768 * 8
```

It is also possible to add an optional field at the end wrapped into a variable-length array of up to one element, or a tagged union where the first field is empty and the second field is the wrapped value. In this way, the implicit truncation/extension rules would automatically make such optional field appear/disappear depending on whether it is supported by the receiving node.

Nodes using v1.0, v1.1, v2.0, and v2.1 definitions can coexist on the same network, and they can interoperate successfully as long as they all support at least v1.x or v2.x. The correct version can be determined at runtime from the port identifier assignment as described in section 2.1.1.2.

In general, nodes that need to maximize their compatibility are likely to employ all existing major versions of each used data type. If there are more than one minor versions available, the highest minor version within the major version should be used in order to take advantage of the latest changes in the data type definition. It is also expected that in certain scenarios some nodes may resort to publishing the same message type using different major versions concurrently to circumvent compatibility issues (in the example reviewed here that would be v1.1 and v2.1).

The examples shown above rely on the primitive scalar types for reasons of simplicity. Real applications should use the type-safe physical unit definitions available in the SI namespace instead. This is covered in section 5.3.6.1.

[a] The topic of data serialization is explored in detail in section 3.7.

3.9 Conventions and recommendations

This section is dedicated to conventions and recommendations intended to help data type designers maintain a consistent style across the ecosystem and avoid some common pitfalls. All of the conventions and recommendations provided in this section are optional (not mandatory to follow).

3.9.1 Naming recommendations

The DSDL naming recommendations follow those that are widely accepted in the general software development industry.

- Namespaces and field attributes should be named in the `snake_case`.
- Constant attributes should be named in the `SCREAMING_SNAKE_CASE`.
- Data types (excluding their namespaces) should be named in the `PascalCase`.
- Names of message types should form a declarative phrase or a noun. For example, `BatteryStatus` or `OutgoingPacket`.
- Names of service types should form an imperative phrase or a verb. For example, `GetInfo` or `HandleIncomingPacket`.
- Short names, unnecessary abbreviations, and uncommon acronyms should be avoided.

3.9.2 Comments

Every data type definition file should begin with a header comment that provides an exhaustive description of the data type, its purpose, semantics, usage patterns, any related data exchange patterns, assumptions, constraints, and all other information that may be necessary or generally useful for the usage of the data type definition.

Every attribute of the data type definition, and especially every field attribute of it, should have an associated comment explaining the purpose of the attribute, its semantics, usage patterns, assumptions, constraints, and any other pertinent information. Exception applies to attributes supplied with sufficiently descriptive and unambiguous names.

A comment should be placed after the entity it is intended to describe; either on the same line (in which case it should be separated from the preceding text with at least two spaces) or on the next line (without blank lines in between). This recommendation does not apply to the file header comment.

3.9.3 Optional value representation

Data structures may include optional field attributes that are not always populated.

The recommended approach for representing optional field attributes is to use variable-length arrays with the capacity of one element.

Alternatively, such one-element variable-length arrays can be replaced with two-field unions, where the first field is empty and the second field contains the desired optional value. The described layout is semantically compatible with the one-element array described above, provided that the field attributes are not swapped.

Floating-point-typed field attributes may be assigned the value of not-a-number (NaN) per IEEE 754 to indicate that the value is not specified; however, this pattern is discouraged because the value would still have to be transferred over the bus even if not populated, and special case values undermine type safety.

Array-based optional field:

```
MyType[<=1] optional_field
```

Union-based optional field:

```
@sealed                     # Sic!
@union                      # The implicit tag is one byte long.
uavcan.primitive.Empty none # Represents lack of value, unpopulated field.
MyType some                 # The field of interest; field ordering is important.
```

The defined above union can be used as follows (suppose it is named `MaybeMyType`):

```
MaybeMyType optional_field
```

The shown approaches are semantically compatible.

The implicit truncation and the implicit zero extension rules allow one to freely add such optional fields at the end of a definition while retaining semantic compatibility. The implicit truncation rule will render them invisible to nodes that utilize older data type definitions which do not contain them, whereas nodes that utilize newer definitions will be able to correctly process objects serialized using older definitions because the implicit zero extension rule guarantees that the optional fields will appear unpopulated.

For example, let the following be the old message definition:

```
float64 foo
float32 bar
```

The new message definition with the new field is as follows:

```
float64 foo
float32 bar
MyType[<=1] my_new_field
```

Suppose that one node is publishing a message using the old definition, and another node is receiving it using the new definition. The implicit zero extension rule guarantees that the optional field array will appear empty to the receiving node because the implicit length field will be read as zero. Same is true if the message was nested inside another one, thanks to the delimiter header.

3.9.4 Bit flag representation

The recommended approach to defining a set of bit flags is to dedicate a `bool`-typed field attribute for each. Representations based on an integer sum of powers of two[61] are discouraged due to their obscurity and failure to express the intent clearly.

Recommended approach:

```
void5
bool flag_foo
bool flag_bar
bool flag_baz
```

Not recommended:

```
uint8 flags            # Not recommended
uint8 FLAG_BAZ = 1
uint8 FLAG_BAR = 2
uint8 FLAG_FOO = 4
```

[61] Which are popular in programming.

4 Transport layer

This chapter defines the transport layer of UAVCAN. First, the core abstract concepts are introduced. Afterwards, they are concretized for each supported underlying transport protocol (e.g., CAN bus); such concretizations are referred to as *concrete transports*.

When referring to a concrete transport, the notation "UAVCAN/X" is used, where *X* is the name of the underlying transport protocol. For example, "UAVCAN/CAN" refers to CAN bus.

As the specification is extended to add support for new concrete transports, some of the generic aspects may be pushed to the concrete sections if they are found to map poorly onto the newly supported protocols. Such changes are guaranteed to preserve full backward compatibility of the existing concrete transports.

4.1 Abstract concepts

The function of the transport layer is to facilitate exchange of serialized representations of DSDL objects[62] between UAVCAN nodes over the *transport network*.

4.1.1 Transport model

This section introduces an abstract implementation-agnostic model of the UAVCAN transport layer. The core relations are depicted in figure 4.1. Some of the concepts introduced at this level may not be manifested in the design of concrete transports; despite that, they are convenient for an abstract discussion.

<table>
<tr><th colspan="3">Taxonomy</th><th>Message transfers</th><th colspan="2">Service transfers</th><th>Description</th></tr>
<tr><td colspan="3">Transfer payload</td><td colspan="3">Serialized object</td><td>The serialized instance of a specific DSDL data type.</td></tr>
<tr><td rowspan="7">Transfer metadata</td><td colspan="2" rowspan="2"></td><td colspan="3">Transfer priority</td><td>Defines the urgency (time sensitivity) of the transferred object.</td></tr>
<tr><td colspan="3">Transfer-ID</td><td>An integer that uniquely identifies a transfer within its session.</td></tr>
<tr><td rowspan="5">Session specifier</td><td rowspan="2">Route specifier</td><td colspan="3">Source node-ID</td><td>Source node-ID is not specified for anonymous transfers.</td></tr>
<tr><td></td><td colspan="2">Destination node-ID</td><td>Destination node-ID is not specified for broadcast transfers.</td></tr>
<tr><td rowspan="3">Data specifier</td><td rowspan="2">Subject-ID</td><td colspan="2">Service-ID</td><td>Port-ID specifies how the serialized object should be processed.</td></tr>
<tr><td>Request</td><td>Response</td><td>Request/response specifier applies to services only.</td></tr>
<tr><td colspan="3">Transfer kind</td><td>Message (subject) or service transfer.</td></tr>
</table>

Figure 4.1: UAVCAN transport layer model

4.1.1.1 Transfer

A *transfer* is a singular act of data transmission from one UAVCAN node to zero or more other UAVCAN nodes over the transport network. A transfer carries zero or more bytes of *transfer payload* together with the associated *transfer metadata*, which encodes the semantic and temporal properties of the carried payload. The elements comprising the metadata are reviewed below.

Transfers are distinguished between *message transfers* and *service transfers* depending on the kind of the carried DSDL object. Service transfers are further differentiated between *service request transfers*, which are sent from the invoking node – *client node* – to the node that provides the service – *server node*, and *service response transfers*, which are sent from the server node to the client node upon handling the request.

A transfer is manifested on the transport network as one or more *transport frames*. A transport frame is an atomic entity carrying the entire transfer payload or a fraction thereof with the associated transfer metadata – possibly extended with additional elements specific to the concrete transport – over the transport network. The exact definition of a transport frame and the mapping of the abstract transport model onto it are specific to concrete transports[63].

4.1.1.2 Transfer payload

The transfer payload contains the serialized representation of the carried DSDL object[64].

Concrete transports may extend the payload with zero-valued *padding bytes* at the end to meet the transport-specific data granularity constraints. Usage of non-zero-valued padding bytes is prohibited for all implementations[65].

Concrete transports may extend the payload with a *transfer CRC* – an additional metadata field used for validating its integrity. The details of its implementation are dictated by the concrete transport specification.

The deterministic nature of UAVCAN in general and DSDL in particular allows implementations to statically determine the maximum amount of memory that is required to contain the serialized representation of a DSDL object of a particular type. Consequently, an implementation that is interested in receiving data objects of a particular type can statically determine the maximum length of the transfer payload.

Implementations should handle incoming transfers containing a larger amount of payload data than expected. In the event of such extra payload being received, a compliant implementation should discard the excessive (unexpected) data at the end of the received payload[66]. The transfer CRC, if applicable, shall be validated regardless of the presence of the extra payload in the transfer. See figure 4.2.

A *transport-layer maximum transmission unit* (MTU) is the maximum amount of data with the associated metadata that can be transmitted per transport frame for a particular concrete transport. All nodes connected

[62] DSDL and data serialization are reviewed in chapter 3.
[63] For example, UAVCAN/CAN (introduced later) defines a particular CAN frame format. Frames that follow the format are UAVCAN transport frames of UAVCAN/CAN.
[64] Chapter 3.
[65] Non-zero padding bytes are disallowed because they would interfere with the implicit zero extension rule (section 3.7).
[66] Such occurrence is not indicative of a problem so it should not be reported as such.

to a given transport network should share the same transport-layer MTU setting[67].

In order to facilitate the implicit zero extension rule introduced in section 3.7, implementations shall not discard a transfer even if it is determined that it contains less payload data than a predicted minimum.

A transfer whose payload exceeds the capacity of the transport frame is manifested on the transport network as a set of multiple transport frames; such transfers are referred to as *multi-frame transfers*. Implementations shall minimize the number of transport frames constituting a multi-frame transfer by ensuring that their payload capacity is not underutilized. Implementations should minimize the delay between transmission of transport frames that belong to the same transfer. Transport frames of a multi-frame transfer shall be transmitted following the order of the transfer payload fragments they contain.

A transfer whose payload does not exceed the capacity of the transport frame shall be manifested on the transport network as a single transport frame[68]; such transfers are referred to as *single-frame transfers*.

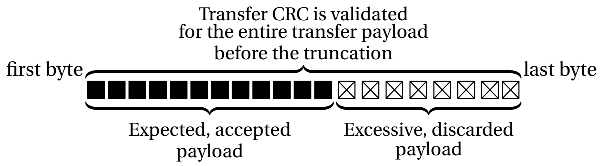


Figure 4.2: Transfer payload truncation

The requirement to discard the excessive payload data at the end of the transfer is motivated by the necessity to allow extensibility of data type definitions, as described in chapter 3. Additionally, excessive payload data may contain zero padding bytes if required by the concrete transport.

Let node A publish an object of the following type over the subject x:

```
float32 parameter
float32 variance
```

Let node B subscribe to the subject x expecting an object of the following type:

```
float32 parameter
```

The payload truncation requirement guarantees that the two nodes will be able to interoperate despite relying on incompatible data type definitions. Under this example, the duty of ensuring the semantic compatibility lies on the system integrator.

The requirement that all involved nodes use the same transport-layer MTU is crucial here. Suppose that the MTU expected by the node B is four bytes and the MTU of the node A is eight bytes. Under this setup, messages emitted by A would be contained in single-frame transfers that are too large for B to process, resulting in the nodes being unable to communicate. An attempt to optimize the memory utilization of B by relying on the fact that the maximum length of a serialized representation of the message is four bytes would be a mistake, because this assumption ignores the existence of subtyping and introduces leaky abstractions throughout the protocol stack.

The implicit zero extension rule makes deserialization routines sensitive to the trailing unused data. For example, suppose that a publisher emits an object of type:

```
uint16 foo
```

Suppose that the concrete transport at hand requires padding to 4 bytes, which is done with 55_{16} (intentionally non-compliant for the sake of this example). Suppose that the published value is 1234_{16}, so the resulting serialized representation is $[34_{16}, 12_{16}, 55_{16}, 55_{16}]$. Suppose that the receiving side relies on the implicit zero extension rule with the following definition:

```
uint16 foo
uint16 bar
```

[67] Failure to follow this rule may render nodes unable to communicate if a transmitting node emits larger transport frames than the receiving node is able to accept.

[68] In other words, multi-frame transfers are prohibited for payloads that can be transferred using a single-frame transfer.

The expectation is that `foo` will be deserialized as 1234_{16}, and `bar` will be zero-extended as 0000_{16}. If arbitrary padding values were allowed, the value of `bar` would become undefined; in this particular example it would be 5555_{16}.

Therefore, the implicit zero-extension rule requires that padding is done with zero bytes only.

4.1.1.3 *Transfer priority*

Transfers are prioritized by means of the *transfer priority* parameter, which allows at least 8 (eight) distinct priority levels. Concrete transports may support more than eight priority levels.

Transmission of transport frames shall be ordered so that frames of higher priority are transmitted first. It follows that higher-priority transfers may preempt transmission of lower-priority transfers.

Transmission of transport frames that share the same priority level should follow the order of their appearance in the transmission queue.

Priority of message transfers and service request transfers can be chosen freely according to the requirements of the application. Priority of a service response transfer should match the priority of the corresponding service request transfer.

Transfer prioritization is paramount for distributed real-time applications.

The priority level mnemonics and their usage recommendations are specified in the following list. The mapping between the mnemonics and actual numeric identifiers is transport-dependent.

Exceptional – The bus designer can ignore these messages when calculating bus load since they should only be sent when a total system failure has occurred. For example, a self-destruct message on a rocket would use this priority. Another analogy is an NMI on a microcontroller.

Immediate – Immediate is a “high priority message” but with additional latency constraints. Since exceptional messages are not considered when designing a bus, the latency of immediate messages can be determined by considering only immediate messages.

Fast – Fast and immediate are both “high priority messages” but with additional latency constraints. Since exceptional messages are not considered when designing a bus, the latency of fast messages can be determined by considering only immediate and fast messages.

High – High priority messages are more important than nominal messages but have looser latency requirements than fast messages. This priority is used so that, in the presence of rogue nominal messages, important commands can be received. For example, one might envision a failure mode where a temperature sensor starts to load a vehicle bus with nominal messages. The vehicle remains operational (for a time) because the controller is exchanging fast and immediate messages with sensors and actuators. A system safety monitor is able to detect the distressed bus and command the vehicle to a safe state by sending high priority messages to the controller.

Nominal – This is what all messages should use by default. Specifically the heartbeat messages should use this priority.

Low – Low priority messages are expected to be sent on a bus under all conditions but cannot prevent the delivery of nominal messages. They are allowed to be delayed but latency should be constrained by the bus designer.

Slow – Slow messages are low priority messages that have no time sensitivity at all. The bus designer need only ensure that, for all possible system states, these messages will eventually be sent.

Optional – These messages might never be sent (theoretically) for some possible system states. The system shall tolerate never exchanging optional messages in every possible state. The bus designer can ignore these messages when calculating bus load. This should be the priority used for diagnostic or debug messages that are not required on an operational system.

4.1.1.4 *Route specifier*

The *route specifier* defines the node-ID of the origin and the node-ID of the destination of a transfer.

A *broadcast transfer* is a transfer that does not have a specific destination; the decision of whether to process a broadcast transfer is delegated to receiving nodes[69]. A *unicast transfer* is a transfer that is addressed to a specific single node[70] whose node-ID is not the same as that of the origin; which node should process a unicast transfer is decided by the sending node.

A node that does not have a node-ID is referred to as *anonymous node*. Such nodes are unable to emit transfers other than *anonymous transfers*. An anonymous transfer is a transfer that does not have a specific source. Anonymous transfers have the following limitations[71]:

- An anonymous transfer can be only a message transfer.
- An anonymous transfer can be only a single-frame transfer.
- Concrete transports may introduce arbitrary additional restrictions on anonymous transfers or omit their support completely.

A message transfer can be only a broadcast transfer; unicast message transfers are not defined[72]. A service transfer can be only a unicast transfer; broadcast service transfers are prohibited.

Transfer kind	**Unicast**	**Broadcast**
Message transfer	Not defined	Valid
Service transfer	Valid	Prohibited

4.1.1.5 *Data specifier*

The *data specifier* encodes the semantic properties of the DSDL object carried by a transfer and its kind.

The data specifier of a message transfer is the subject-ID of the contained DSDL message object.

The data specifier of a service transfer is a combination of the service-ID of the contained DSDL service object and an additional binary parameter that segregates service requests from service responses.

4.1.1.6 *Session specifier*

The *session specifier* is a combination of the data specifier and the route specifier. Its function is to uniquely identify a category of transfers by the semantics of exchanged data and the agents participating in its exchange while abstracting over individual transfers and their concrete data[73].

The term *session* used here denotes the node's local representation of a logical communication channel that it is a member of. Following the stateless and low-context nature of UAVCAN, this concept excludes any notion of explicit state sharing between nodes.

> One of the key design principles is that UAVCAN is a stateless low-context protocol where collaborating agents do not make strong assumptions about the state of each other. Statelessness and context invariance are important because they facilitate behavioral simplicity and robustness; these properties are desirable for deterministic real-time distributed applications which UAVCAN is designed for.
>
> Design and verification of a system that relies on multiple agents sharing the same model of a distributed process necessitates careful analysis of special cases such as unintended state divergence, latency and transient states, sudden loss of state (e.g., due to disconnection or a software reset), etc. Lack of adequate consideration may render the resulting solution fragile and prone to unspecified behaviors.
>
> Some of the practical consequences of the low-context design include the ability of a node to immediately commence operation on the network without any prior initialization steps. Likewise, addition and removal of a subscriber to a given subject is transparent to the publisher.
>
> The above considerations only hold for the communication protocol itself. Applications whose functionality is built on top of the protocol may engage in state sharing if such is found to be beneficial[a].
>
> [a] Related discussion in https://forum.uavcan.org/t/idempotent-interfaces-and-deterministic-data-loss-mitigation/643.

> Some implementations of the UAVCAN communication stack may contain states indexed by the session specifier. For example, in order to emit a transfer, the stack may need to query the appropriate transfer-ID

[69] This does not imply that applications are required to be involved with every broadcast transfer. The opt-in logic is facilitated by the low-level routing and/or filtering features implemented by the network stack and/or the underlying hardware.

[70] Whose existence and availability is optional.

[71] Anonymous transfers are intended primarily for the facilitation of the optional plug-and-play feature (section 5.3) which enables fully automatic configuration of UAVCAN nodes upon their connection to the network. Some transports may provide native support for auto-configuration, rendering anonymous transfers unnecessary.

[72] Unicast message transfers may be defined in a future revision of this Specification.

[73] Due to the fact that anonymous transfers lack information about their origin, all anonymous transfers that share the same data specifier and destination are grouped under the same session specifier.

counter (section 4.1.1.7) by the session specifier of the transfer. Likewise, in order to process a received frame, the stack may need to locate the appropriate states keyed by the session specifier.

Given the intended application domains of UAVCAN, the temporal characteristics of such look-up activities should be well-characterized and predictable. Due to the fact that all underlying primitive parameters that form the session specifier (such as node-ID, port-ID, etc.) have statically defined bounds, it is trivial to construct a look-up procedure satisfying any computational complexity envelope, from constant-complexity $O(1)$ at the expense of heightened memory utilization, up to low-memory-footprint $O(n)$ if temporal predictability is less relevant.

For example, given a subject-ID, the maximum number of distinct sessions that can be observed by the local node will never exceed the number of nodes in the network minus one[a]. If the number of nodes in the network cannot be reliably known in advance (which is the case in most applications), it can be considered to equal the maximum number of nodes permitted by the concrete transport[b]. The total number of distinct sessions that can be observed by a node is a product of the number of distinct data specifiers utilized by the node and the number of other nodes in the network.

It is recognized that highly rigid safety-critical applications may benefit from avoiding any dynamic look-up by sacrificing generality, by employing automatic code generation, or through other methods, in the interest of greater determinism and robustness. In such cases, the above considerations may be irrelevant.

[a] A node cannot receive transfers from itself, hence minus one.
[b] E.g., 128 nodes for the CAN bus transport.

4.1.1.7 *Transfer-ID*

The *transfer-ID* is an unsigned integer value that is provided for every transfer. Barring the case of transfer-ID overflow reviewed below, each transfer under a given session specifier has a unique transfer-ID value. This parameter is crucial for many aspects of UAVCAN communication[74]; specifically:

Message sequence monitoring – transfer-ID allows receiving nodes to detect discontinuities in incoming message streams from remote nodes.

Service response matching – when a server responds to a request, it uses the same transfer-ID for the response transfer as in the request transfer, allowing the client to emit concurrent requests to the same server while being able to match each response with the corresponding local request state.

Transfer deduplication – the transfer-ID allows receiving nodes to detect and eliminate duplicated transfers. Transfer duplication may occur either spuriously as an artifact of a concrete transport[75] or deliberately as a method of deterministic data loss mitigation for unreliable links (section 4.1.3.3).

Multi-frame transfer reassembly – a transfer that is split over multiple transport frames is reassembled back upon reception with the help of transfer-ID: all transport frames that comprise a transfer share the same transfer-ID value.

Automatic management of redundant interfaces – in redundant transport networks, transfer-ID enables automatic switchover to a back-up interface shall the primary interface fail. The switchover logic can be completely transparent to the application, joining several independent redundant transport networks into a highly reliable single virtual communication channel.

For service response transfers the transfer-ID value shall be directly copied from the corresponding service request transfer[76].

A node that is interested in emitting message transfers or service request transfers under a particular session specifier, whether periodically or on an ad-hoc basis, shall allocate a transfer-ID counter state associated with said session specifier exclusively. The transfer-ID value of every emitted transfer is determined by sampling the corresponding counter keyed by the session specifier of the transfer; afterwards, the counter is incremented by one.

[74] One might be tempted to use the transfer-ID value for temporal synchronization of parallel message streams originating from the same node, where messages bearing the same transfer-ID value are supposed to correspond to the same moment in time. Such use is strongly discouraged because it is incompatible with transports that rely on overflowing transfer-ID values and because it introduces a leaky abstraction into the system. If temporal synchronization is necessary, explicit time stamping should be used instead.

[75] For example, in CAN bus, a frame that appears valid to the receiver may under certain (rare) conditions appear invalid to the transmitter, triggering the latter to retransmit the frame, in which case it will be duplicated on the side of the receiver. Sequence counting mechanisms such as transfer-ID allow implementations to circumvent this problem.

[76] This behavior facilitates request-response matching on the client node.

The initial value of a transfer-ID counter shall be zero. Once a new transfer-ID counter is created, it shall be kept at least as long as the node remains connected to the transport network; destruction of transfer-ID counter states is prohibited[77].

When the transfer-ID counter reaches the maximum value defined for the concrete transport, the next increment resets its value to zero. Transports where such events are expected to take place during operation are said to have *cyclic transfer-ID*; the number of unique transfer-ID values is referred to as *transfer-ID modulo*. Transports where the maximum value of the transfer-ID is high enough to be unreachable under all conceivable practical circumstances are said to have *monotonic transfer-ID*.

Transfer-ID difference for a pair of transfer-ID values a and b is defined for monotonic transfer-ID as their arithmetic difference $a - b$. For a cyclic transfer-ID, the difference is defined as the number of increment operations that need to be applied to b so that $a = b'$.

A C++ implementation of the cyclic transfer-ID difference operator is provided here.

```cpp
#include <cstdint>
/**
 * UAVCAN cyclic transfer-ID difference computation algorithm implemented in C++.
 * License: CC0, no copyright reserved.
 * @param a         Left-hand operand (minuend).
 * @param b         Right-hand operand (subtrahend).
 * @param modulo    The number of distinct transfer-ID values, or the maximum value plus one.
 * @returns         The number of increment operations separating b from a.
 */
[[nodiscard]]
constexpr std::uint8_t computeCyclicTransferIDDifference(const std::uint8_t a,
                                                         const std::uint8_t b,
                                                         const std::uint8_t modulo)
{
    std::int16_t d = static_cast<std::int16_t>(a) - static_cast<std::int16_t>(b);
    if (d < 0)
    {
        d += static_cast<std::int16_t>(modulo);
    }
    return static_cast<std::uint8_t>(d);
}
```

4.1.2 Redundant transports

UAVCAN supports transport redundancy for the benefit of a certain class of safety-critical applications. A redundant transport interconnects nodes belonging to the same network (all or their subset) via more than one transport network. A set of such transport networks that together form a redundant transport is referred to as a *redundant transport group*.

Each member of a redundant transport group shall be capable of independent operation such that the level of service of the resulting redundant transport remains constant as long as at least one member of the redundant group remains functional[78].

Networks containing nodes with different reliability requirements may benefit from nonuniform redundant transport configurations, where non-critical nodes are interconnected using a lower number of transports than critical nodes.

Designers should recognize that nonuniform redundancy may complicate the analysis of the network.

4.1.3 Transfer transmission

4.1.3.1 Transmission timeout

The transport frames of a time-sensitive transfer whose payload has lost relevance due to its transmission being delayed should be removed from the transmission queue[79]. The time interval between the point where the transfer is constructed and the point where it is considered to have lost relevance is referred to as *transmission timeout*.

The transmission timeout should be documented for each outgoing transfer port.

[77] The number of unique session specifiers is bounded and can be determined statically per application, so this requirement does not introduce non-deterministic features into the application even if it leverages aperiodic/ad-hoc transfers.

[78] Redundant transports are designed for increased fault tolerance, not for load sharing.

[79] Trailing transport frames of partially transmitted multi-frame transfers should be removed as well. The objective of this recommendation is to ensure that obsolete data is not transmitted as it may have adverse effects on the system.

4.1.3.2 *Pending service requests*

In the case of cyclic transfer-ID transports (section 4.1.1.7), implementations should ensure that upon a transfer-ID overflow a service client session does not reuse the same transfer-ID value for more than one pending request simultaneously.

4.1.3.3 *Deterministic data loss mitigation*

Performance of transport networks where the probability of a successful transfer delivery does not meet design requirements can be adjusted by repeating relevant outgoing transfers under the same transfer-ID value[80]. This tactic is referred to as *deterministic data loss mitigation*[81].

4.1.3.4 *Transmission over redundant transports*

Nodes equipped with redundant transports shall submit every outgoing transfer to the transmission queues of all available redundant transports simultaneously[82]. It is recognized that perfectly simultaneous transmission may not be possible due to different utilization rates of the redundant transports, different phasing of their traffic, and/or application constraints, in which case implementations should strive to minimize the temporal skew as long as that does not increase the latency.

An exception to the above rule applies if the payload of the transfer is a function of the identity of the transport instance that carries the transfer[83].

4.1.4 Transfer reception

4.1.4.1 *Definitions*

Transfer reassembly is the real-time process of reconstruction of the transfer payload and its metadata from a sequence of relevant transport frames.

Transfer-ID timeout is a time interval whose semantics are explained below. Implementations may define this value statically according to the application requirements. Implementations may automatically adjust this value per session at runtime as a function of the observed transfer reception interval. Transfer-ID timeout values greater than 2 (two) seconds are not recommended. Implementations should document the value of transfer-ID timeout or the rules of its computation.

Transport frame reception timestamp specifies the moment of time when the frame is received by a node. *Transfer reception timestamp* is the reception timestamp of the earliest received frame of the transfer.

An *ordered transfer sequence* is a sequence of transfers whose temporal order is covariant with their transfer-ID values.

4.1.4.2 *Behaviors*

For a given session specifier, every unique transfer (differentiated from other transfers in the same session by its transfer-ID) shall be received at most once[84].

For a given session specifier, a successfully reassembled transfer that is temporally separated from any other successfully reassembled transfer under the same session specifier by more than the transfer-ID timeout is considered unique regardless of its transfer-ID value.

If the optimal transfer-ID timeout value for a given session cannot be known in advance, it can be computed at runtime on a per-session basis[85]. The parameters of such computation are to be chosen according to the requirements of the application, but they should always be documented.

[80] Removal of intentionally duplicated transfers on the receiving side is natively guaranteed by this transport layer specification; no special activities are needed there to accommodate this feature.

[81] Discussed in https://forum.uavcan.org/t/idempotent-interfaces-and-deterministic-data-loss-mitigation/643.

[82] The objective of this requirement is to guarantee that a redundant transport remains fully functional as long as at least one transport in the redundant group is functional.

[83] An example of such a special case is the time synchronization algorithm documented in section 5.3.

[84] In other words, intentional and unintentional duplicates shall be removed. Intentional duplications are introduced by the deterministic data loss mitigation measure or redundant transports. Unintentional duplications may be introduced by various artifacts of the transport network.

[85] E.g., as a multiple of the average transfer reception interval.

Low transfer-ID timeout values increase the risk of undetected transfer duplication when such transfers are significantly delayed due to network congestion, which is possible with very low-priority transfers when the network load is high.

High transfer-ID timeout values increase the risk of an undetected transfer loss when a remote node suffers a loss of state (e.g., due to a software reset).

The ability to auto-detect the optimal transfer-ID timeout value per session at runtime ensures that the application can find the optimal balance even if the temporal properties of the network are not known in advance. As a practical example, an implementation could compute the exponential moving average of the transfer reception interval x for a given session and define the transfer-ID timeout as $2x$.

It is important to note that the automatic adjustment of the transfer-ID timeout should only be done on a per-session basis rather than for the entire port, because there may be multiple remote nodes emitting transfers on the same port at different rates. For example, if one node emits transfers at a rate r transfers per second, and another node emits transfers on the same port at a much higher rate $100r$, the resulting auto-detected transfer-ID timeout might be too low, creating the risk of accepting duplicates.

Implementations are recommended, but not required, to support reassembly of multi-frame transfers where the temporal ordering of the transport frames is distorted.

For a certain category of transport implementations, reassembly of multi-frame transfers from an unordered transport frame sequence increases the probability of successful delivery if the probability of a transport frame loss is non-zero and transport frames are intentionally duplicated.

Such intentional duplication occurs in redundant transports and if deterministic data loss mitigation is used. The reason is that the loss of a single transport frame is observed by the receiving node as its relocation from its original position in the sequence to the position of its duplicate.

Reassembled transfers shall form an ordered transfer sequence.

For a cyclic transfer-ID redundant transport whose redundant group contains n transports, if up to $n-1$ transports in the redundant group lose the ability to exchange transport frames between nodes, the transfer reassembly process shall be able to restore nominal functionality in an amount of time that does not exceed the transfer-ID timeout.

Cyclic transfer-ID transport implementations are recommended to insert a delay before performing an automatic fail-over. As indicated in the normative description, the delay may be arbitrary as long as it does not exceed the transfer-ID timeout value.

The fail-over delay allows implementations to uphold the transfer uniqueness requirement when the phasing of traffic on different transports within the redundant group differs by more than the transfer-ID overflow period.

For a monotonic transfer-ID redundant transport whose redundant group contains n transports, if up to $n-1$ transports in the redundant group lose the ability to exchange transport frames between nodes, the performance of the transfer reassembly process shall not be affected.

Monotonic transfer-ID transport implementations are recommended to always accept the first transfer to arrive regardless of which transport within the redundant group it was delivered over.

This behavior ensures that the total latency of a redundant transport equals the latency of the best-performing transport within the redundant group (i.e., the total latency equals the latency of the fastest transport). Since a monotonic transfer-ID does not overflow, there is no risk of failing to uphold the uniqueness guarantee unlike with the case of cyclic transfer-ID.

If anonymous transfers are supported by the concrete transport, reassembly of anonymous transfers shall be implemented by unconditional acceptance of their transport frames. Requirements pertaining to ordering and uniqueness do not apply.

Regardless of the concrete transport in use and its capabilities, UAVCAN provides the following guarantees (excluding anonymous transfers):

- Removal of duplicates. If a transfer is delivered, it is guaranteed that it is delivered once, even if intentionally duplicated by the origin.
- Correct ordering. Received transfers are ordered according to their transfer-ID values.
- Deterministic automatic fail-over in the event of a failure of a transport (or several) in a redundant group.

For anonymous transfers, ordering and uniqueness are impossible to enforce because anonymous transfers that originate from different nodes may share the same session specifier.

Reassembly of transfers from redundant interfaces may be implemented either on the per-transport-frame level or on the per-transfer level. The former amounts to receiving individual transport frames from redundant interfaces which are then used for reassembly; it can be seen that this method requires that all transports in the redundant group use identical application-level MTU (i.e., same number of transfer payload bytes per frame). The latter can be implemented by treating each transport in the redundant group separately, so that each runs an independent transfer reassembly process, whose outputs are then deduplicated on the per-transfer level; this method may be more computationally complex but it provides greater flexibility. A detailed discussion is omitted because it is outside of the scope of this specification.

4.2 UAVCAN/CAN

This section specifies a concrete transport based on ISO 11898 CAN bus. Throughout this section, “CAN” implies both Classic CAN 2.0 and CAN FD, unless specifically noted otherwise. CAN FD should be considered the primary transport protocol.

Parameter	Value	References
Maximum node-ID value	127 (7 bits wide).	2
Transfer-ID mode	Cyclic, modulo 32.	4.1.1.7
Number of transfer priority levels	8 (no additional levels).	4.1.1.3
Largest single-frame transfer payload	Classic CAN – 7 bytes, CAN FD – up to 63 bytes.	4.1.1.2
Anonymous transfers	Supported with non-deterministic collision resolution policy.	4.1.1.4

Table 4.1: UAVCAN/CAN transport capabilities

4.2.1 CAN ID field

UAVCAN/CAN transport frames are CAN 2.0B frames. The 29-bit CAN ID encodes the session specifier[86] of the transfer it belongs to along with its priority. The CAN data field of every frame contains the transfer payload (or, in the case of multi-frame transfers, a fraction thereof), the transfer-ID, and other metadata.

UAVCAN/CAN can share the same bus with other high-level CAN bus protocols provided that they do not make use of CAN 2.0B frames[87]. However, future revisions of UAVCAN/CAN may utilize CAN 2.0A as well, so backward compatibility with other high-level CAN bus protocols is not guaranteed.

UAVCAN/CAN can share the same bus with UAVCAN/CAN v0 – the earlier experimental revision of the protocol (not recommended for new designs). The protocol version can be determined at runtime on a per-frame basis as described in section 4.2.2.2.

UAVCAN/CAN utilizes two different CAN ID bit layouts for message transfers and service transfers. The bit layouts are summarized on figure 4.3. Tables 4.2.1 and 4.2.1 summarize the purpose of each field and their permitted values for message transfers and service transfers, respectively.

Message	Service, not message				Anonymous				Subject-ID													R	Source node-ID						
	Priority					R	R	R																					
Values	[0, 7]			0	𝔹	0	1	1	[0, 8191]													0	[0, 127]						
CAN ID bit	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CAN ID byte	3					2								1								0							

Service	Service, not message				Request, not response											Destination node-ID							Source node-ID						
	Priority					R	Service-ID																						
Values	[0, 7]			1	𝔹	0	[0, 511]									[0, 127]							[0, 127]						
CAN ID bit	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CAN ID byte	3					2								1								0							

Figure 4.3: CAN ID bit layout

Field	Width	Valid values	Description
Transfer priority	3	[0, 7] (any)	Section 4.1.1.3.
Service, not message	1	0	Always zero for message transfers.
Anonymous	1	{0, 1} (any)	Zero for regular message transfers, one for anonymous transfers.
Reserved bit 23	1	0	Discard frame if this field has a different value.
Reserved bit 22	1	1, any	Transmit 1; ignore (do not check) when receiving.
Reserved bit 21	1	1, any	Transmit 1; ignore (do not check) when receiving.
Subject-ID	13	[0, 8191] (any)	Subject-ID of the current message transfer.
Reserved bit 7	1	0	Discard frame if this field has a different value.
Source node-ID	7	[0, 127] (any)	Node-ID of the origin. For anonymous transfers, this field contains a pseudo-ID instead, as described in section 4.2.1.2.

Table 4.2: CAN ID bit fields for message transfers

[86] Section 4.1.1.6.

[87] For example, CANOpen or CANaerospace.

Field	Width	Valid values	Description
Transfer priority	3	$[0,7]$ (any)	Section 4.1.1.3.
Service, not message	1	1	Always one for service transfers.
Request, not response	1	$\{0,1\}$ (any)	One for service request, zero for service response.
Reserved bit 23	1	0	Discard frame if this field has a different value.
Service-ID	9	$[0,511]$ (any)	Service-ID of the encoded service object (request or response).
Destination node-ID	7	$[0,127]$ (any)	Node-ID of the destination: server if request, client if response.
Source node-ID	7	$[0,127]$ (any)	Node-ID of the origin: client if request, server if response.

Table 4.3: CAN ID bit fields for service transfers

4.2.1.1 *Transfer priority*

Valid values for transfer priority range from 0 to 7, inclusively, where 0 corresponds to the highest priority, and 7 corresponds to the lowest priority (according to the CAN bus arbitration policy).

In multi-frame transfers, the value of the priority field shall be identical for all frames of the transfer.

When multiple transfers of different types with the same priority contest for bus access, the following precedence is ensured (from higher priority to lower priority):

1. Message transfers (the primary method of data exchange in UAVCAN networks).
2. Anonymous (message) transfers.
3. Service response transfers (preempt requests).
4. Service request transfers (responses take precedence over requests to make service calls more atomic and reduce the number of pending states in the system).

Mnemonics for transfer priority levels are provided in section 4.1.1.3, and their mapping to the UAVCAN/CAN priority field is as follows:

Priority field value	Mnemonic name
0	Exceptional
1	Immediate
2	Fast
3	High
4	Nominal
5	Low
6	Slow
7	Optional

Since the value of transfer priority is required to be the same for all frames in a transfer, it follows that the value of the CAN ID is guaranteed to be the same for all CAN frames of the transfer. Given a constant transfer priority value, all CAN frames under a given session specifier will be equal.

4.2.1.2 *Source node-ID field in anonymous transfers*

The source node-ID field of anonymous transfers shall be initialized with a pseudorandom *pseudo-ID* value. The source of the pseudorandom data used for the pseudo-ID shall aim to produce different values for different CAN frame data field values.

A node transmitting an anonymous transfer shall abort its transmission and discard it upon detection of a bus error. Some method of media access control should be used at the application level for further conflict resolution.

CAN bus does not allow different nodes to transmit CAN frames with different data under the same CAN ID value. Owing to the fact that the CAN ID includes the node-ID of the transmitting node, this restriction does not affect non-anonymous transfers. However, anonymous transfers would violate this restriction because their source node-ID is not defined, hence the additional measures described in this section.

A possible way of initializing the source node pseudo-ID value is to compute the arithmetic sum of all bytes of the transfer payload, taking the least significant bits of the result as the pseudo-ID (usage of stronger hashes is encouraged). Implementations that adopt this approach will be using the same pseudo-ID value for identical transfer payloads, which is acceptable since this will not trigger an error on the bus.

Because the set of possible pseudo-ID values is small, a collision where multiple nodes emit CAN frames with different data but the same CAN ID is likely to happen despite the randomization measures described here. Therefore, if anonymous transfers are used, implementations shall account for possible errors on the CAN bus triggered by CAN ID collisions.

Automatic retransmission should be disabled for anonymous transfers (like in TTCAN). This measure allows the protocol to prevent temporary disruptions that may occur if the automatic retransmission on bus error is not suppressed.

Additional bus access control logic is needed at the application level because the possibility of identifier collisions in anonymous frames undermines the access control logic implemented in CAN bus controller hardware.

The described principles make anonymous transfers highly non-deterministic and inefficient. This is considered acceptable because the scope of anonymous transfers is limited to a very narrow set of use cases which tolerate their downsides. The UAVCAN specification employs anonymous transfers only for the plug-and-play feature defined in section 5.3. Deterministic applications are advised to avoid reliance on anonymous transfers completely.

None of the above considerations affect nodes that do not transmit anonymous transfers.

4.2.2 CAN data field

4.2.2.1 Layout

UAVCAN/CAN utilizes a fixed layout of the CAN data field: the last byte of the CAN data field contains the metadata, it is referred to as the *tail byte*. The preceding bytes of the data field contain the transfer payload, which may be extended with padding bytes and transfer CRC.

A CAN frame whose data field contains less than one byte is not a valid UAVCAN/CAN frame.

The bit layout of the tail byte is shown in table 4.4.

Table 4.4: Tail byte structure

Bit	Field	Single-frame transfers	Multi-frame transfers
7	**Start of transfer**	Always 1	First frame: 1, otherwise 0.
6	**End of transfer**	Always 1	Last frame: 1, otherwise 0.
5	**Toggle bit**	Always 1	First frame: 1, then alternates; section 4.2.2.2.
4			
3		Modulo 32 (range [0, 31])	
2	**Transfer-ID**	section 4.1.1.7	
1			
0		(least significant bit)	

4.2.2.2 Toggle bit

Transport frames that form a multi-frame transfer are equipped with a *toggle bit* which alternates its state every frame within the transfer for frame deduplication purposes[88].

The toggle bit can be used to facilitate operation of heterogeneous deployments where the experimental UAVCAN/CAN v0 shares the same CAN bus with the current version of the standard.

Whenever a new transfer is initiated, the original state of the toggle bit reflects the protocol version. Implementations that need to support simultaneous operation of two versions of the protocol can record the state of the toggle bit when the "start of transfer" bit is set, and keep this information indexed by the value of the CAN ID field (all frames of a transfer are guaranteed to share the same CAN ID). The resulting mapping from CAN ID to the protocol version can be used to route incoming frames to the implementation of the appropriate version of the protocol.

[88] A frame that appears valid to the receiving node may under certain conditions appear invalid to the transmitter, triggering the latter to retransmit the frame, in which case it will be duplicated on the side of the receiver.

Start of transfer	Toggle bit	Protocol version
1	0	UAVCAN v0 (experimental version).
1	1	UAVCAN v1 (this version).
0	x	Keep the state of the toggle bit from the first frame of the transfer to detect protocol version in multi-frame transfers.

Table 4.5: Protocol version detection based on the toggle bit

4.2.2.3 *Transfer payload decomposition*

The transport-layer MTU of Classic CAN-based implementations shall be 8 bytes (the maximum). The transport-layer MTU of CAN FD-based implementations should be 64 bytes (the maximum).

CAN FD does not guarantee byte-level granularity of the CAN data field length. If the desired length of the CAN data field cannot be represented due to the granularity constraints, zero padding bytes are used.

In single-frame transfers, padding bytes are inserted between the end of the payload and the tail byte.

In multi-frame transfers, the transfer payload is appended with trailing zero padding bytes followed by the transfer CRC (section 4.2.2.4). All transport frames of a multi-frame transfer except the last one shall fully utilize the available data field capacity; hence, padding is unnecessary there. The number of padding bytes is computed so that the length granularity constraints for the last frame of the transfer are satisfied.

Usage of padding bytes implies that when a serialized message is being deserialized by a receiving node, the byte sequence used for deserialization may be longer than the actual byte sequence generated by the emitting node during serialization. This behavior is compatible with the DSDL specification.

The weak MTU requirement for CAN FD is designed to avoid compatibility issues.

4.2.2.4 *Transfer CRC*

Payload of multi-frame transfers is extended with a transfer CRC for validating the correctness of their re-assembly. Transfer CRC is not used with single-frame transfers.

The transfer CRC is computed over the entire payload of the multi-frame transfer plus the trailing padding bytes, if any. The resulting CRC value is appended to the transfer payload after the padding bytes (if any) in the *big-endian byte order* (most significant byte first)[89].

The CRC function is the standard CRC-16-CCITT: initial value FFFF_{16}, polynomial 1021_{16}, not reversed, no output XOR, big endian. The value for an input sequence $(49, 50, \ldots, 56, 57)$ is 29B1_{16}. The following code

[89] This is the native byte order for this CRC function.

snippet provides a basic implementation of the transfer CRC algorithm in C++ (LUT-based alternatives exist).

```cpp
#include <cstdint>
#include <cstddef>

/// UAVCAN/CAN transfer CRC function implementation. License: CC0, no copyright reserved.
class CANTransferCRC
{
    std::uint16_t value_ = 0xFFFFU;

public:
    void add(const std::uint8_t byte)
    {
        value_ ^= static_cast<std::uint16_t>(byte) << 8U;
        for (std::uint8_t bit = 8; bit > 0; --bit)
        {
            if ((value_ & 0x8000U) != 0)
            {
                value_ = (value_ << 1U) ^ 0x1021U;
            }
            else
            {
                value_ = value_ << 1U;
            }
        }
    }

    void add(const std::uint8_t* bytes, std::size_t length)
    {
        while (length-- > 0)
        {
            add(*bytes++);
        }
    }

    [[nodiscard]] std::uint16_t get() const { return value_; }
};
```

4.2.3 Examples

Heartbeat from node-ID 42, nominal priority level, uptime starting from 0 and then incrementing by one every transfer, health status is 0, operating mode is 1, vendor-specific status code 161 ($A1_{16}$):

CAN ID (hex)	CAN data (hex)
107D552A	00 00 00 00 00 01 A1 E0
107D552A	01 00 00 00 00 01 A1 E1
107D552A	02 00 00 00 00 01 A1 E2
107D552A	03 00 00 00 00 01 A1 E3

`uavcan.primitive.String.1.0` under subject-ID 4919 (1337_{16}) published by an anonymous node, the string is “`Hello world!`” (ASCII); one byte of zero padding can be seen between the payload and the tail byte:

CAN ID (hex)	CAN data (hex)
11133775	0C 00 48 65 6C 6C 6F 20 77 6F 72 6C 64 21 00 E0
11133775	0C 00 48 65 6C 6C 6F 20 77 6F 72 6C 64 21 00 E1
11133775	0C 00 48 65 6C 6C 6F 20 77 6F 72 6C 64 21 00 E2
11133775	0C 00 48 65 6C 6C 6F 20 77 6F 72 6C 64 21 00 E3

Node info request from node 123 to node 42 via Classic CAN, then response; notice how the transfer CRC is scattered across two frames:

CAN ID (hex)	CAN data (hex)	ASCII	Comment
136B957B	E1	.	The request contains no payload.
126BBDAA	01 00 00 00 01 00 00 A1	....	Start of response, toggle bit is set.
126BBDAA	00 00 00 00 00 00 00 01	....	Toggle bit is cleared.
126BBDAA	00 00 00 00 00 00 00 21	.......!	Toggle bit is set.
126BBDAA	00 00 00 00 00 00 00 01	....	Etc.
126BBDAA	00 00 24 6F 72 67 2E 21	..$org.!	Array (string) length prefix.
126BBDAA	75 61 76 63 61 6E 2E 01	uavcan..	
126BBDAA	70 79 75 61 76 63 61 21	pyuavca!	
126BBDAA	6E 2E 64 65 6D 6F 2E 01	n.demo..	
126BBDAA	62 61 73 69 63 5F 75 21	basic_u!	
126BBDAA	73 61 67 65 00 00 9A 01	sage....	Transfer CRC, MSB.
126BBDAA	E7 61	.a	Transfer CRC, LSB.

`uavcan.primitive.array.Natural8.1.0` under subject-ID 4919 (1337_{16}) published by node 59, the array contains an arithmetic sequence $(0, 1, 2, \ldots, 89, 90, 91)$; the transport MTU is 64 bytes:

CAN ID (hex)	CAN data (hex)	Comment
1013373B	5C 00 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F 20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F 30 31 32 33 34 35 36 37 38 39 3A 3B 3C A0	First frame: 1. payload (array length prefix is 92); 2. tail byte.
1013373B	3D 3E 3F 40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F 50 51 52 53 54 55 56 57 58 59 5A 5B 00 00 00 00 00 00 00 00 00 00 00 00 00 00 **BC 19** 40	Last frame: 1. payload; 2. padding (underlined); 3. transfer CRC (bold); 4. tail byte.

4.2.4 Software design considerations

4.2.4.1 *Ordered transmission*

The CAN controller driver software shall guarantee that CAN frames with identical CAN ID values will be transmitted in their order of appearance in the transmission queue[90].

4.2.4.2 *Transmission timestamping*

Certain application-level functions of UAVCAN may require the driver to timestamp outgoing transport frames, e.g., the time synchronization function. A sensible approach to transmission timestamping is built around the concept of *loop-back frames*, which is described here.

If the application needs to timestamp an outgoing frame, it sets a special flag – the *loop-back flag* – on the frame before sending it to the driver. The driver would then automatically re-enqueue this frame back into the reception queue once it is transmitted (keeping the loop-back flag set so that the application is able to distinguish the loop-back frame from regular received traffic). The timestamp of the loop-backed frame would be of the moment when it was delivered to the bus.

The advantage of the loop-back based approach is that it relies on the same interface between the application and the driver that is used for regular communications. No complex and dangerous callbacks or write-backs from interrupt handlers are involved.

4.2.4.3 *Inner priority inversion*

Implementations should take necessary precautions against the problem of inner priority inversion.

Suppose the application needs to emit a frame with the CAN ID X. The frame is submitted to the CAN controller's registers and the transmission is started. Suppose that afterwards it turned out that there is a new frame with the CAN ID $(X-1)$ that needs to be sent, too, but the previous frame X is in the way, and it is blocking the transmission of the new frame. This may turn into a problem if the lower-priority frame is losing arbitration on the bus due to the traffic on the bus having higher priority than the current frame, but

[90] This is because multi-frame transfers use identical CAN ID for all frames of the transfer, and UAVCAN requires that all frames of a multi-frame transfer shall be transmitted in the correct order.

lower priority than the next frame that is waiting in the queue.

A naive solution to this is to continuously check whether the priority of the frame that is currently being transmitted by the CAN controller is lower than the priority of the next frame in the queue, and if it is, abort transmission of the current frame, move it back to the transmission queue, and begin transmission of the new one instead. This approach, however, has a hidden race condition: the old frame may be aborted at the moment when it has already been received by remote nodes, which means that the next time it is re-transmitted, the remote nodes will see it duplicated. Additionally, this approach increases the complexity of the driver and can possibly affect its throughput and latency.

Most CAN controllers offer a robust solution to the problem: they have multiple transmission mailboxes (usually at least 3), and the controller always chooses for transmission the mailbox which contains the highest priority frame. This provides the application with a possibility to avoid the inner priority inversion problem: whenever a new transmission is initiated, the application should check whether the priority of the next frame is higher than any of the other frames that are already awaiting transmission. If there is at least one higher-priority frame pending, the application doesn't move the new one to the controller's transmission mailboxes, it remains in the queue. Otherwise, if the new frame has a higher priority level than all of the pending frames, it is pushed to the controller's transmission mailboxes and removed from the queue. In the latter case, if a lower-priority frame loses arbitration, the controller would postpone its transmission and try transmitting the higher-priority one instead. That resolves the problem.

There is an interesting extreme case, however. Imagine a controller equipped with N transmission mailboxes. Suppose the application needs to emit N frames in the increasing order of priority, which leads to all of the transmission mailboxes of the controller being occupied. Now, if all of the conditions below are satisfied, the system ends up with a priority inversion condition nevertheless, despite the measures described above:

- The highest-priority pending CAN frame cannot be transmitted due to the bus being saturated with a higher-priority traffic.
- The application needs to emit a new frame which has a higher priority than that which saturates the bus.

If both hold, a priority inversion is afoot because there is no free transmission mailbox to inject the new higher-priority frame into. The scenario is extremely unlikely, however; it is also possible to construct the application in a way that would preclude the problem, e.g., by limiting the number of simultaneously used distinct CAN ID values.

The following pseudocode demonstrates the principles explained above:

```
// Returns the index of the TX mailbox that can be used for the transmission of the newFrame
// If none are available, returns -1.
getFreeMailboxIndex(newFrame)
{
    chosen_mailbox = -1     // By default, assume that no mailboxes are available

    for i = 0...NumberOfTxMailboxes
    {
        if isTxMailboxFree(i)
        {
            chosen_mailbox = i
            // Note: cannot break here, shall check all other mailboxes as well.
        }
        else
        {
            if not isFramePriorityHigher(newFrame, getFrameFromTxMailbox(i))
            {
                chosen_mailbox = -1
                break   // Denied - shall wait until this mailbox has finished transmitting
            }
        }
    }

    return chosen_mailbox
}
```

4.2.4.4 *Automatic hardware acceptance filter configuration*

Most CAN controllers are equipped with hardware acceptance filters. Hardware acceptance filters reduce the application workload by ignoring irrelevant CAN frames on the bus by comparing their ID values against the set of relevant ID values configured by the application.

There exist two common approaches to CAN hardware filtering: list-based and mask-based. In the case of the list-based approach, every CAN frame detected on the bus is compared against the set of reference CAN ID values provided by the application; only those frames that are found in the reference set are accepted. Due to the complex structure of the CAN ID field used by UAVCAN, usage of the list-based filtering method with this protocol is impractical.

Most CAN controller vendors implement mask-based filters, where the behavior of each filter is defined by two parameters: the mask M and the reference ID R. Then, such filter accepts only those CAN frames for which the following bitwise logical condition holds true[a]:

$$((X \wedge M) \oplus R) \leftrightarrow 0$$

where X is the CAN ID value of the evaluated frame.

Complex UAVCAN applications are often required to operate with more distinct transfers than there are acceptance filters available in the hardware. That creates the challenge of finding the optimal configuration of the available filters that meets the following criteria:

- All CAN frames needed by the application are accepted.
- The number of irrelevant frames (i.e., not used by the application) accepted from the bus is minimized.

The optimal configuration is a function of the number of available hardware filters, the set of distinct transfers needed by the application, and the expected frequency of occurrence of all possible distinct transfers on the bus. The latter is important because if there are to be irrelevant transfers, it makes sense to optimize the configuration so that the acceptance of less common irrelevant transfers is preferred over the more common irrelevant transfers, as that reduces the processing load on the application.

The optimal configuration depends on the properties of the network the node is connected to. In the absence of the information about the network, or if the properties of the network are expected to change frequently, it is possible to resort to a quasi-optimal configuration which assumes that the occurrence of all possible irrelevant transfers is equally probable. As such, the quasi-optimal configuration is a function of only the number of available hardware filters and the set of distinct transfers needed by the application.

The quasi-optimal configuration can be easily found automatically. Certain implementations of the UAVCAN protocol stack include this functionality, allowing the application to easily adjust the configuration of the hardware acceptance filters using a very simple API.

A quasi-optimal hardware acceptance filter configuration algorithm is described below. The approach was first proposed by P. Kirienko and I. Sheremet in 2015.

First, the bitwise *filter merge* operation is defined on filter configurations A and B. The set of CAN frames accepted by the merged filter configuration is a superset of those accepted by A and B. The definition is as follows:

$$m_M(R_A, R_B, M_A, M_B) = M_A \wedge M_B \wedge \neg(R_A \oplus R_B)$$
$$m_R(R_A, R_B, M_A, M_B) = R_A \wedge m_M(R_A, R_B, M_A, M_B)$$

The *filter rank* is a function of the mask of the filter. The rank of a filter is a unitless quantity that defines in relative terms how selective the filter configuration is. The rank of a filter is proportional to the likelihood that the filter will reject a random CAN ID. In the context of hardware filtering, this quantity is conveniently representable via the number of bits set in the filter mask parameter (also known as *population count*):

$$r(M) = \begin{cases} 0 & \mid M < 1 \\ r(\lfloor \frac{M}{2} \rfloor) & \mid M \bmod 2 = 0 \\ r(\lfloor \frac{M}{2} \rfloor) + 1 & \mid M \bmod 2 \neq 0 \end{cases}$$

Having the low-level operations defined, we can proceed to define the whole algorithm. First, construct the initial set of CAN acceptance filter configurations according to the requirements of the application. Then, as long as the number of configurations in the set exceeds the number of available hardware acceptance filters, repeat the following:

1. Find the pair A, B of configurations in the set for which $r(m_M(R_A, R_B, M_A, M_B))$ is maximized.
2. Remove A and B from the set of configurations.
3. Add a new configuration X to the set of configurations, where $M_X = m_M(R_A, R_B, M_A, M_B)$, and $R_X = m_R(R_A, R_B, M_A, M_B)$.

The algorithm reduces the number of filter configurations by one at each iteration, until the number of available hardware filters is sufficient to accommodate the whole set of configurations.

[a] Notation: $\wedge$ – bitwise logical AND, $\oplus$ – bitwise logical XOR, $\neg$ – bitwise logical NOT.

5 Application layer

Previous chapters of this specification define a set of basic concepts that are the foundation of the protocol: they allow one to define data types and exchange data objects over the bus in a robust and deterministic manner. This chapter is focused on higher-level concepts: rules, conventions, and standard functions that are to be respected by applications utilizing UAVCAN to maximize cross-vendor compatibility, avoid ambiguities, and prevent some common design pitfalls.

The rules, conventions, and standard functions defined in this chapter are designed to be an acceptable middle ground for any sensible aerospace or robotic system. UAVCAN favors no particular domain or kind of system among targeted applications.

- Section 5.1 contains a set of mandatory rules that shall be followed by all UAVCAN implementations.
- Section 5.2 contains a set of conventions and recommendations that are not mandatory to follow. Every deviation, however, should be justified and well-documented.
- Section 5.3 contains a full list of high-level functions defined on top of UAVCAN. Formal specification of such functions is provided in the DSDL data type definition files that those functions are based on (see chapter 6).

5.1 Application-level requirements

This section describes a set of high-level rules that shall be obeyed by all UAVCAN implementations.

5.1.1 Port identifier distribution

An overview of related concepts is provided in chapter 2.

The subject and service identifier values are segregated into three ranges:

- unregulated port identifiers that can be freely chosen by users and integrators (both fixed and non-fixed);
- regulated fixed identifiers for non-standard data type definitions that are assigned by the UAVCAN maintainers for publicly released data types;
- regulated identifiers of the standard data types that are an integral part of the UAVCAN specification.

More information on the subject of data type regulation is provided in section 2.1.2.2.

The ranges are summarized in table 5.1.1. The ranges may be expanded, but not contracted, in a future version of the document.

Subject-ID	Service-ID	Purpose
[0, 6143]	[0, 255]	Unregulated identifiers (both fixed and non-fixed).
[6144, 7167]	[256, 383]	Non-standard fixed regulated identifiers (i.e., vendor-specific).
[7168, 8191]	[384, 511]	Standard fixed regulated identifiers.

Table 5.1: Port identifier distribution

5.1.2 Port compatibility

The system integrator shall ensure that nodes participating in data exchange via a given port[91] use data type definitions that are sufficiently congruent so that the resulting behavior of the involved nodes is predictable and the possibility of unintended behaviors caused by misinterpretation of exchanged serialized objects is eliminated.

Let there be type A:

```
void1
uint7 demand_factor_pct  # [percent]
# Values above 100% are not allowed.
```

And type B:

```
uint8 demand_factor_pct  # [percent]
# Values above 100% indicate overload.
```

The data types are not semantically compatible, but they can be used on the same subject nevertheless: a subscriber expecting B can accept A. The reverse is not true, however.

This example shows that even semantically incompatible types can facilitate behaviorally correct interoperability of nodes.

Compatibility of subjects and services is completely independent from the names of the involved data types. Data types can be moved between namespaces and freely renamed and re-versioned without breaking compatibility with existing deployments. Nodes provided by different vendors that utilize differently named data types may still interoperate if such data types happen to be compatible. The duty of ensuring the compatibility lies on the system integrator.

5.1.3 Standard namespace

An overview of related concepts is provided in chapter 3.

This specification defines a set of standard regulated DSDL data types located under the root namespace named "`uavcan`" (section 6).

Vendor-specific, user-specific, or any other data types not defined by this specification shall not be defined inside the standard root namespace[92].

[91] I.e., subject or service.
[92] Custom data type definitions shall be located inside vendor-specific or user-specific namespaces instead.

5.2 Application-level conventions

This section describes a set of high-level conventions designed to enhance compatibility of applications leveraging UAVCAN. The conventions described here are not mandatory to follow; however, every deviation should be justified and documented.

5.2.1 Node identifier distribution

An overview of related concepts is provided in chapter 2.

Valid values of node-ID range from 0 up to a transport-specific upper boundary which is guaranteed to be above 127 for any transport.

The two uppermost node-ID values are reserved for diagnostic and debugging tools; these node-ID values should not be used in fielded systems.

5.2.2 Service latency

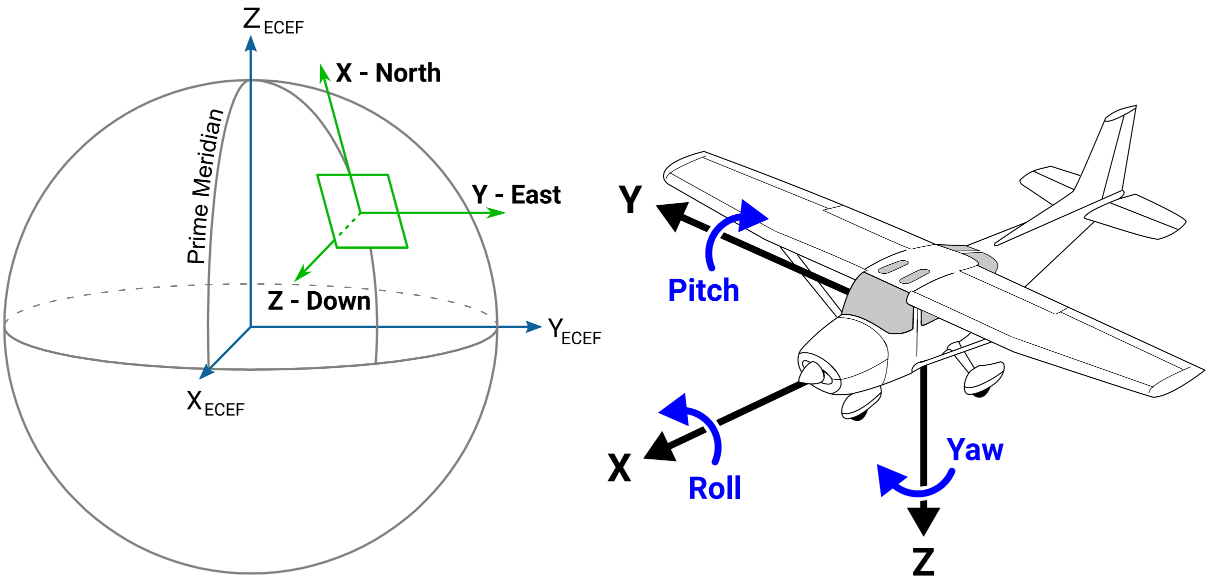
If the server uses a significant part of the timeout period to process the request, the client might drop the request before receiving the response. Servers should minimize the request processing time; that is, the time between reception of a service request transfer and the transmission of the corresponding service response transfer.

The worst-case request processing time should be documented for each server-side service port.

5.2.3 Coordinate frames

UAVCAN follows the conventions that are widely accepted in relevant applications. Adherence to the coordinate frame conventions described here maximizes compatibility and reduces the amount of computations for conversions between incompatible coordinate systems and representations. It is recognized, however, that some applications may find the advised conventions unsuitable, in which case deviations are permitted. Any such deviations shall be explicitly documented.

All coordinate systems defined in this section are right-handed. If application-specific coordinate systems are introduced, they should be right-handed as well.



North-East-Down (NED) frame and body frame conventions. All systems are right-handed.

Figure 5.1: Coordinate frame conventions

5.2.3.1 World frame

For world fixed frames, the *North-East-Down* (NED) right-handed notation is preferred: X – northward, Y – eastward, Z – downward.

5.2.3.2 Body frame

In relation to a body, the convention is as follows, right-handed[93]: X – forward, Y – rightward, Z – downward.

[93] This convention is widely used in aeronautic applications.

5.2.3.3 *Optical frame*

In the case of cameras, the following right-handed convention is preferred[94]: X – rightward, Y – downward, Z – towards the scene along the optical axis.

5.2.4 Rotation representation

All applications should represent rotations using quaternions with the elements ordered as follows[95]: W, X, Y, Z. Other forms of rotation representation should be avoided.

Angular velocities should be represented using the right-handed, fixed-axis (extrinsic) convention: X (roll), Y (pitch), Z (yaw).

Quaternions are considered to offer the optimal trade-off between bandwidth efficiency, computation complexity, and explicitness:

- Euler angles are not self-contained, requiring applications to agree on a particular convention beforehand; a convention would be difficult to establish considering different demands of various use cases.
- Euler angles and fixed axis rotations typically cannot be used for computations directly due to angular interpolation issues and singularities; thus, to make use of such representations, one often has to convert them to a different form (e.g., quaternion); such conversions are computationally heavy.
- Rotation matrices are highly redundant.

5.2.5 Matrix representation

5.2.5.1 *General*

Matrices should be represented as flat arrays in the row-major order.

$$\begin{bmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \end{bmatrix} \rightarrow (x_{11}, x_{12}, x_{13}, x_{21}, x_{22}, x_{23})$$

5.2.5.2 *Square matrices*

There are standard compressed representations of an $n \times n$ square matrix.

An array of size n^2 represents a full square matrix. This is equivalent to the general case reviewed above.

An array of $\frac{(1+n)n}{2}$ elements represents a symmetric matrix, where array members represent the elements of the upper-right triangle arranged in the row-major order.

$$\begin{bmatrix} a & b & c \\ b & d & e \\ c & e & f \end{bmatrix} \rightarrow (a, b, c, d, e, f)$$

This form is well-suited for covariance matrix representation.

An array of n elements represents a diagonal matrix, where an array member at position i (where $i = 1$ for the first element) represents the matrix element $x_{i,i}$ (where $x_{1,1}$ is the upper-left element).

$$\begin{bmatrix} a & 0 & 0 \\ 0 & b & 0 \\ 0 & 0 & c \end{bmatrix} \rightarrow (a, b, c)$$

An array of one element represents a scalar matrix.

$$\begin{bmatrix} a & 0 & 0 \\ 0 & a & 0 \\ 0 & 0 & a \end{bmatrix} \rightarrow a$$

An empty array represents a zero matrix.

[94] This convention is widely used in various applications involving computer vision systems.

[95] Assuming $w + x\boldsymbol{i} + y\boldsymbol{j} + z\boldsymbol{k}$.

5.2.5.3 Covariance matrices

A zero covariance matrix represents an unknown covariance[96].

Infinite error variance means that the associated value is undefined.

5.2.6 Physical quantity representation

5.2.6.1 Units

All units should be SI[97] units (base or derived). Usage of any other units is strongly discouraged.

When defining data types, fields and constants that represent unscaled quantities in SI units should not have suffixes indicating the unit, since that would be redundant.

On the other hand, fields and constants that contain quantities in non-SI units[98] or scaled SI units[99] should be suffixed with the standard abbreviation of the unit[100] and its metric prefix[101] (if any), maintaining the proper letter case of the abbreviation. In other words, the letter case of the suffix is independent of the letter case of the attribute it is attached to.

Scaling coefficients should not be chosen arbitrarily; instead, the choice should be limited to the standard metric prefixes defined by the International System of Units.

All standard metric prefixes have well-defined abbreviations that are constructed from ASCII characters, except for one: the micro prefix is abbreviated as a Greek letter "µ" (mu). When defining data types, "µ" should be replaced with the lowercase Latin letter "u".

Irrespective of the suffix, it is recommended to always specify units for every field in the comments.

```
float16 temperature          # [kelvin] Suffix not needed because an unscaled SI unit is used here.

uint24 delay_us              # [microsecond] Scaled SI unit, suffix required. Mu replaced with "u".
uint24 MAX_DELAY_us = 600000 # [microsecond] Notice the letter case.

float32 kinetic_energy_GJ    # [gigajoule] Notice the letter case.

float16 estimated_charge_mAh # [milliampere hour] Scaled non-SI unit. Discouraged, use coulomb.
float16 MAX_CHARGE_mAh = 1e4 # [milliampere hour] Notice the letter case.
```

5.2.6.2 Enhanced type safety

In the interest of improving type safety and reducing the possibility of a human error, it is recommended to avoid reliance on raw scalar types (such as `float32`) when defining fields containing physical quantities. Instead, the explicitly typed alternatives defined in the standard DSDL namespace `uavcan.si.unit` (section 6.38 on page 132) (also see section 5.3.6.1) should be used.

```
float32[4] kinetic_energy                           # [joule] Not recommended.
uavcan.si.unit.energy.Scalar.1.0[4] kinetic_energy  # This is the recommended practice.
# Kinetic energy of four bodies.

float32[3] velocity                                 # [meter/second] Not recommended.
uavcan.si.unit.velocity.Vector3.1.0                 # This is the recommended practice.
# 3D velocity vector.
```

[96] As described above, an empty array represents a zero matrix, from which follows that an empty array represents unknown covariance.
[97] International System of Units.
[98] E.g., degree Celsius instead of kelvin.
[99] E.g., microsecond instead of second.
[100] E.g., kg for kilogram, J for joule.
[101] E.g., M for mega, n for nano.

5.3 Application-level functions

This section documents the high-level functionality defined by UAVCAN. The common high-level functions defined by the specification span across different application domains. All of the functions defined in this section are optional (not mandatory to implement), except for the node heartbeat feature (section 5.3.2), which is mandatory for all UAVCAN nodes.

The detailed specifications for each function are provided in the DSDL comments of the data type definitions they are built upon, whereas this section serves as a high-level overview and index.

5.3.1 Node initialization

UAVCAN does not require that nodes undergo any specific initialization upon connection to the bus — a node is free to begin functioning immediately once it is powered up. The operating mode of the node (such as initialization, normal operation, maintenance, and so on) is to be reflected via the mandatory heartbeat message described in section 5.3.2.

5.3.2 Node heartbeat

Every non-anonymous UAVCAN node shall report its status and presence by periodically publishing messages of type `uavcan.node.Heartbeat` (section 6.4.4 on page 101) at a fixed rate specified in the message definition using the fixed subject-ID. Anonymous nodes shall not publish to this subject.

This is the only high-level protocol function that UAVCAN nodes are required to support. All other data types and application-level functions are optional.

The DSDL source text of `uavcan.node.Heartbeat` version 1.0 (this is the only version) with a fixed subject ID 7509 is provided below. More information is available in section 6.4.4 on page 101.

```
# Abstract node status information.
# This is the only high-level function that shall be implemented by all nodes.
#
# All UAVCAN nodes that have a node-ID are required to publish this message to its fixed subject periodically.
# Nodes that do not have a node-ID (also known as "anonymous nodes") shall not publish to this subject.
#
# The default subject-ID 7509 is 1110101010101 in binary. The alternating bit pattern at the end helps transceiver
# synchronization (e.g., on CAN-based networks) and on some transports permits automatic bit rate detection.
#
# Network-wide health monitoring can be implemented by subscribing to the fixed subject.

uint16 MAX_PUBLICATION_PERIOD = 1   # [second]
# The publication period shall not exceed this limit.
# The period should not change while the node is running.

uint16 OFFLINE_TIMEOUT = 3          # [second]
# If the last message from the node was received more than this amount of time ago, it should be considered offline.

uint32 uptime                       # [second]
# The uptime seconds counter should never overflow. The counter will reach the upper limit in ~136 years,
# upon which time it should stay at 0xFFFFFFFF until the node is restarted.
# Other nodes may detect that a remote node has restarted when this value leaps backwards.

Health.1.0 health
# The abstract health status of this node.

Mode.1.0 mode
# The abstract operating mode of the publishing node.
# This field indicates the general level of readiness that can be further elaborated on a per-activity basis
# using various specialized interfaces.

uint8 vendor_specific_status_code
# Optional, vendor-specific node status code, e.g. a fault code or a status bitmask.

@assert _offset_ % 8 == {0}
@assert _offset_ == {56}  # Fits into a single-frame Classic CAN transfer (least capable transport, smallest MTU).
@extent 12 * 8
```

5.3.3 Generic node information

The service `uavcan.node.GetInfo` (section 6.4.2 on page 99) can be used to obtain generic information about the node, such as the structured name of the node (which includes the name of its vendor), a 128-bit globally unique identifier, the version information about its hardware and software, version of the UAVCAN specification implemented on the node, and the optional certificate of authenticity.

While the service is, strictly speaking, optional, omitting its support is highly discouraged, since it is instrumental for network discovery, firmware update, and various maintenance and diagnostic needs.

The DSDL source text of `uavcan.node.GetInfo` version 1.0 (this is the only version) with a fixed service ID 430 is provided below. More information is available in section 6.4.2 on page 99.

```
# Full node info request.
# All of the returned information shall be static (unchanged) while the node is running.
```

```
# It is highly recommended to support this service on all nodes.

@sealed

---

Version.1.0 protocol_version
# The UAVCAN protocol version implemented on this node, both major and minor.
# Not to be changed while the node is running.

Version.1.0 hardware_version
Version.1.0 software_version
# The version information shall not be changed while the node is running.
# The correct hardware version shall be reported at all times, excepting software-only nodes, in which
# case it should be set to zeros.
# If the node is equipped with a UAVCAN-capable bootloader, the bootloader should report the software
# version of the installed application, if there is any; if no application is found, zeros should be reported.

uint64 software_vcs_revision_id
# A version control system (VCS) revision number or hash. Not to be changed while the node is running.
# For example, this field can be used for reporting the short git commit hash of the current
# software revision.
# Set to zero if not used.

uint8[16] unique_id
# The unique-ID (UID) is a 128-bit long sequence that is likely to be globally unique per node.
# The vendor shall ensure that the probability of a collision with any other node UID globally is negligibly low.
# UID is defined once per hardware unit and should never be changed.
# All zeros is not a valid UID.
# If the node is equipped with a UAVCAN-capable bootloader, the bootloader shall use the same UID.

@assert _offset_ == {30 * 8}
# Manual serialization note: only fixed-size fields up to this point. The following fields are dynamically sized.

uint8[<=50] name
# Human-readable non-empty ASCII node name. An empty name is not permitted.
# The name shall not be changed while the node is running.
# Allowed characters are: a-z (lowercase ASCII letters) 0-9 (decimal digits) . (dot) - (dash) _ (underscore).
# Node name is a reversed Internet domain name (like Java packages), e.g. "com.manufacturer.project.product".

uint64[<=1] software_image_crc
# The value of an arbitrary hash function applied to the software image. Not to be changed while the node is running.
# This field can be used to detect whether the software or firmware running on the node is an exact
# same version as a certain specific revision. This field provides a very strong identity guarantee,
# unlike the version fields above, which can be the same for different builds of the software.
# As can be seen from its definition, this field is optional.
#
# The exact hash function and the methods of its application are implementation-defined.
# However, implementations are recommended to adhere to the following guidelines, fully or partially:
#   - The hash function should be CRC-64-WE.
#   - The hash function should be applied to the entire application image padded to 8 bytes.
#   - If the computed image CRC is stored within the software image itself, the value of
#     the hash function becomes ill-defined, because it becomes recursively dependent on itself.
#     In order to circumvent this issue, while computing or checking the CRC, its value stored
#     within the image should be zeroed out.

uint8[<=222] certificate_of_authenticity
# The certificate of authenticity (COA) of the node, 222 bytes max, optional. This field can be used for
# reporting digital signatures (e.g., RSA-1776, or ECDSA if a higher degree of cryptographic strength is desired).
# Leave empty if not used. Not to be changed while the node is running.

@assert _offset_ % 8 == {0}
@assert _offset_.max == (313 * 8)     # At most five CAN FD frames
@extent 448 * 8
```

5.3.4 Bus data flow monitoring

Interfaces defined in the namespace `uavcan.node.port` (section 6.5 on page 102) (see table 5.2) facilitate network inspection and monitoring.

By comparing the data obtained with the help of these interfaces from each node on the bus, the caller can reconstruct the data exchange graph for the entire bus (assuming that every node on the bus supports the services in question; they are not mandatory).

Table 5.2: Index of the nested namespace "uavcan.node.port"

Namespace tree	Ver.	FPID	max(BLS) bytes	Extent bytes	Full name
uavcan					
node					
port					
List	0.1	7510	8466	*sealed*	uavcan.node.port.List
ID	1.0		3	*sealed*	uavcan.node.port.ID
ServiceID	1.0		2	*sealed*	uavcan.node.port.ServiceID
ServiceIDList	0.1		132	128	uavcan.node.port.ServiceIDList
SubjectID	1.0		2	*sealed*	uavcan.node.port.SubjectID
SubjectIDList	0.1		4101	4097	uavcan.node.port.SubjectIDList

5.3.5 Network-wide time synchronization

UAVCAN provides a simple and robust method of time synchronization[102] that is built upon the work "Implementing a Distributed High-Resolution Real-Time Clock using the CAN-Bus" published by M. Gergeleit and H. Streich[103]. The detailed specification of the time synchronization algorithm is provided in the documentation for the message type `uavcan.time.Synchronization` (section 6.9.2 on page 114).

`uavcan.time.GetSynchronizationMasterInfo` (section 6.9.1 on page 114) provides nodes with information about the currently used time system and related data like the number of leap seconds added.

Redundant time synchronization masters are supported for the benefit of high-reliability applications.

> Time synchronization with explicit sensor feed timestamping should be preferred over inferior alternatives such as sensor lag reporting that are sometimes found in simpler systems because such alternatives are difficult to scale and they do not account for the delays introduced by communication interfaces.
>
> It is the duty of every node that publishes timestamped data to account for its own internal delays; for example, if the data latency of a local sensor is known, it needs to be accounted for in the reported timestamp value.

Table 5.3: Index of the nested namespace "uavcan.time"

Namespace tree	Ver.	FPID	max(BLS) bytes	Extent bytes	Full name
uavcan					
time					
GetSynchronizationMasterInfo	0.1	510	52 ⇌ 196	48 ⇌ 192	uavcan.time.GetSynchronizationMasterInfo
Synchronization	1.0	7168	7	*sealed*	uavcan.time.Synchronization
SynchronizedTimestamp	1.0		7	*sealed*	uavcan.time.SynchronizedTimestamp
TAIInfo	0.1		2	*sealed*	uavcan.time.TAIInfo
TimeSystem	0.1		1	*sealed*	uavcan.time.TimeSystem

5.3.6 Primitive types and physical quantities

The namespaces `uavcan.si` (section 6.17 on page 126) and `uavcan.primitive` (section 6.14 on page 122) included in the standard data type set are designed to provide a generic and flexible method of real-time data exchange. However, these are not bandwidth-efficient.

Generally, applications where the bus bandwidth and latency are important should minimize their reliance on these generic data types and favor more specialized types instead that are custom-designed for their particular use cases; e.g., vendor-specific types or application-specific types, either designed in-house, published by third parties[104], or supplied by vendors of COTS equipment used in the application.

Vendors of COTS equipment are recommended to ensure that some minimal functionality is available via these generic types without reliance on their vendor-specific types (if there are any). This is important for reusability, because some of the systems where such COTS nodes are to be integrated may not be able to easily support vendor-specific types.

5.3.6.1 SI namespace

The `si` namespace is named after the International System of Units (SI). The namespace contains a collection of scalar and vector value types that describe most commonly used physical quantities in SI; for example, velocity, mass, energy, angle, and time. The objective of these types is to permit construction of arbitrarily complex distributed control systems without reliance on any particular vendor-specific data types.

The namespace `uavcan.si.unit` (section 6.38 on page 132) contains basic units that can be used as type-safe wrappers over `float32` and other scalar and array types. The namespace `uavcan.si.sample` (section 6.17 on page 126) contains time-stamped versions of the same.

Each message type defined in the namespace `uavcan.si.sample` contains a timestamp field of type `uavcan.time.SynchronizedTimestamp` (section 6.9.3 on page 116). Every emitted message should be timestamped in order to allow subscribers to identify which of the messages relate to the same event or to the same instant. Messages that are emitted in bulk in relation to the same event or the same instant should contain exactly the same value of the timestamp to simplify the task of timestamp matching for the subscribers.

The exact strategy of matching related messages by timestamp employed by subscribers is entirely implementation-defined. The specification does not concern itself with this matter because it is expected

[102] The ability to accurately synchronize time between nodes is instrumental for building distributed real-time data processing systems such as various robotic applications, autopilots, autonomous driving solutions, and so on.
[103] Proceedings of the 1st international CAN-Conference 94, Mainz, 13.-14. Sep. 1994, CAN in Automation e.V., Erlangen.
[104] As long as the license permits.

that different applications will opt for different design trade-offs and different tactics to suit their constraints. Such diversity is not harmful, because its effects are always confined to the local node and cannot affect operation of other nodes or their compatibility.

Tables 5.4 and 5.5 provide a high-level overview of the SI namespace. Please follow the references for details.

Table 5.4: Index of the nested namespace "uavcan.si.unit"

Namespace tree	Ver.	FPID	max(BLS) bytes	Extent bytes	Full name
uavcan					
si					
unit					
acceleration					
Scalar	1.0		4	*sealed*	uavcan.si.unit.acceleration.Scalar
Vector3	1.0		12	*sealed*	uavcan.si.unit.acceleration.Vector3
angle					
Quaternion	1.0		16	*sealed*	uavcan.si.unit.angle.Quaternion
Scalar	1.0		4	*sealed*	uavcan.si.unit.angle.Scalar
angular_acceleration					
Scalar	1.0		4	*sealed*	uavcan.si.unit.angular_acceleration.Scalar
Vector3	1.0		12	*sealed*	uavcan.si.unit.angular_acceleration.Vector3
angular_velocity					
Scalar	1.0		4	*sealed*	uavcan.si.unit.angular_velocity.Scalar
Vector3	1.0		12	*sealed*	uavcan.si.unit.angular_velocity.Vector3
duration					
Scalar	1.0		4	*sealed*	uavcan.si.unit.duration.Scalar
WideScalar	1.0		8	*sealed*	uavcan.si.unit.duration.WideScalar
electric_charge					
Scalar	1.0		4	*sealed*	uavcan.si.unit.electric_charge.Scalar
electric_current					
Scalar	1.0		4	*sealed*	uavcan.si.unit.electric_current.Scalar
energy					
Scalar	1.0		4	*sealed*	uavcan.si.unit.energy.Scalar
force					
Scalar	1.0		4	*sealed*	uavcan.si.unit.force.Scalar
Vector3	1.0		12	*sealed*	uavcan.si.unit.force.Vector3
frequency					
Scalar	1.0		4	*sealed*	uavcan.si.unit.frequency.Scalar
length					
Scalar	1.0		4	*sealed*	uavcan.si.unit.length.Scalar
Vector3	1.0		12	*sealed*	uavcan.si.unit.length.Vector3
WideScalar	1.0		8	*sealed*	uavcan.si.unit.length.WideScalar
WideVector3	1.0		24	*sealed*	uavcan.si.unit.length.WideVector3
magnetic_field_strength					
Scalar	1.0		4	*sealed*	uavcan.si.unit.magnetic_field_strength.Scalar
Vector3	1.0		12	*sealed*	uavcan.si.unit.magnetic_field_strength.Vector3
mass					
Scalar	1.0		4	*sealed*	uavcan.si.unit.mass.Scalar
power					
Scalar	1.0		4	*sealed*	uavcan.si.unit.power.Scalar
pressure					
Scalar	1.0		4	*sealed*	uavcan.si.unit.pressure.Scalar
temperature					
Scalar	1.0		4	*sealed*	uavcan.si.unit.temperature.Scalar
torque					
Scalar	1.0		4	*sealed*	uavcan.si.unit.torque.Scalar
Vector3	1.0		12	*sealed*	uavcan.si.unit.torque.Vector3
velocity					
Scalar	1.0		4	*sealed*	uavcan.si.unit.velocity.Scalar
Vector3	1.0		12	*sealed*	uavcan.si.unit.velocity.Vector3
voltage					
Scalar	1.0		4	*sealed*	uavcan.si.unit.voltage.Scalar
volume					
Scalar	1.0		4	*sealed*	uavcan.si.unit.volume.Scalar
volumetric_flow_rate					
Scalar	1.0		4	*sealed*	uavcan.si.unit.volumetric_flow_rate.Scalar

Table 5.5: Index of the nested namespace "uavcan.si.sample"

Namespace tree	Ver.	FPID	max(BLS) bytes	Extent bytes	Full name
uavcan					
si					
sample					
acceleration					
Scalar	1.0		11	*sealed*	uavcan.si.sample.acceleration.Scalar
Vector3	1.0		19	*sealed*	uavcan.si.sample.acceleration.Vector3
angle					
Quaternion	1.0		23	*sealed*	uavcan.si.sample.angle.Quaternion
Scalar	1.0		11	*sealed*	uavcan.si.sample.angle.Scalar
angular_acceleration					
Scalar	1.0		11	*sealed*	uavcan.si.sample.angular_acceleration.Scalar
Vector3	1.0		19	*sealed*	uavcan.si.sample.angular_acceleration.Vector3
angular_velocity					
Scalar	1.0		11	*sealed*	uavcan.si.sample.angular_velocity.Scalar
Vector3	1.0		19	*sealed*	uavcan.si.sample.angular_velocity.Vector3
duration					
Scalar	1.0		11	*sealed*	uavcan.si.sample.duration.Scalar
WideScalar	1.0		15	*sealed*	uavcan.si.sample.duration.WideScalar
electric_charge					
Scalar	1.0		11	*sealed*	uavcan.si.sample.electric_charge.Scalar
electric_current					
Scalar	1.0		11	*sealed*	uavcan.si.sample.electric_current.Scalar
energy					
Scalar	1.0		11	*sealed*	uavcan.si.sample.energy.Scalar
force					
Scalar	1.0		11	*sealed*	uavcan.si.sample.force.Scalar
Vector3	1.0		19	*sealed*	uavcan.si.sample.force.Vector3
frequency					
Scalar	1.0		11	*sealed*	uavcan.si.sample.frequency.Scalar
length					
Scalar	1.0		11	*sealed*	uavcan.si.sample.length.Scalar
Vector3	1.0		19	*sealed*	uavcan.si.sample.length.Vector3
WideScalar	1.0		15	*sealed*	uavcan.si.sample.length.WideScalar
WideVector3	1.0		31	*sealed*	uavcan.si.sample.length.WideVector3
magnetic_field_strength					
Scalar	1.0		11	*sealed*	uavcan.si.sample.magnetic_field_strength.Scalar
Vector3	1.0		19	*sealed*	uavcan.si.sample.magnetic_field_strength.Vector3
mass					
Scalar	1.0		11	*sealed*	uavcan.si.sample.mass.Scalar
power					
Scalar	1.0		11	*sealed*	uavcan.si.sample.power.Scalar
pressure					
Scalar	1.0		11	*sealed*	uavcan.si.sample.pressure.Scalar
temperature					
Scalar	1.0		11	*sealed*	uavcan.si.sample.temperature.Scalar
torque					
Scalar	1.0		11	*sealed*	uavcan.si.sample.torque.Scalar
Vector3	1.0		19	*sealed*	uavcan.si.sample.torque.Vector3
velocity					
Scalar	1.0		11	*sealed*	uavcan.si.sample.velocity.Scalar
Vector3	1.0		19	*sealed*	uavcan.si.sample.velocity.Vector3
voltage					
Scalar	1.0		11	*sealed*	uavcan.si.sample.voltage.Scalar
volume					
Scalar	1.0		11	*sealed*	uavcan.si.sample.volume.Scalar
volumetric_flow_rate					
Scalar	1.0		11	*sealed*	uavcan.si.sample.volumetric_flow_rate.Scalar

5.3.6.2 *Primitive namespace*

The primitive namespace contains a collection of primitive types: integer types, floating point types, bit flag, string, raw block of bytes, and an empty value. Integer, floating point, and bit flag types are available in two categories: scalar and array; the latter are limited so that their serialized representation is never larger than 257 bytes.

The primitive types are designed to complement the SI namespace with an even simpler set of basic types that do not make any assumptions about the meaning of the data they describe. The primitive types provide a very high degree of flexibility, but due to their lack of semantic information, their use carries the risk of creating suboptimal interfaces that are difficult to use, validate, and scale.

Normally, the use of primitive types should be limited to very basic vendor-neutral interfaces for COTS equipment and software, debug and diagnostic purposes, and whenever there is a need to exchange data the type of which cannot be determined statically.[105]

Table 5.6 provides a high-level overview of the primitive namespace. Please follow the references for details.

[105] An example of the latter use case is the register protocol described in section 5.3.10.

Table 5.6: Index of the nested namespace "uavcan.primitive"

Namespace tree	Ver.	FPID	max(BLS) bytes	Extent bytes	Full name
uavcan					
primitive					
Empty	1.0		0	*sealed*	uavcan.primitive.Empty
String	1.0		258	*sealed*	uavcan.primitive.String
Unstructured	1.0		258	*sealed*	uavcan.primitive.Unstructured
array					
Bit	1.0		258	*sealed*	uavcan.primitive.array.Bit
Integer8	1.0		258	*sealed*	uavcan.primitive.array.Integer8
Integer16	1.0		257	*sealed*	uavcan.primitive.array.Integer16
Integer32	1.0		257	*sealed*	uavcan.primitive.array.Integer32
Integer64	1.0		257	*sealed*	uavcan.primitive.array.Integer64
Natural8	1.0		258	*sealed*	uavcan.primitive.array.Natural8
Natural16	1.0		257	*sealed*	uavcan.primitive.array.Natural16
Natural32	1.0		257	*sealed*	uavcan.primitive.array.Natural32
Natural64	1.0		257	*sealed*	uavcan.primitive.array.Natural64
Real16	1.0		257	*sealed*	uavcan.primitive.array.Real16
Real32	1.0		257	*sealed*	uavcan.primitive.array.Real32
Real64	1.0		257	*sealed*	uavcan.primitive.array.Real64
scalar					
Bit	1.0		1	*sealed*	uavcan.primitive.scalar.Bit
Integer8	1.0		1	*sealed*	uavcan.primitive.scalar.Integer8
Integer16	1.0		2	*sealed*	uavcan.primitive.scalar.Integer16
Integer32	1.0		4	*sealed*	uavcan.primitive.scalar.Integer32
Integer64	1.0		8	*sealed*	uavcan.primitive.scalar.Integer64
Natural8	1.0		1	*sealed*	uavcan.primitive.scalar.Natural8
Natural16	1.0		2	*sealed*	uavcan.primitive.scalar.Natural16
Natural32	1.0		4	*sealed*	uavcan.primitive.scalar.Natural32
Natural64	1.0		8	*sealed*	uavcan.primitive.scalar.Natural64
Real16	1.0		2	*sealed*	uavcan.primitive.scalar.Real16
Real32	1.0		4	*sealed*	uavcan.primitive.scalar.Real32
Real64	1.0		8	*sealed*	uavcan.primitive.scalar.Real64

5.3.7 Remote file system interface

The set of standard data types contains a collection of services for manipulation of remote file systems (namespace `uavcan.file` (section 6.2 on page 88), see table 5.7). All basic file system operations are supported, including file reading and writing, directory listing, metadata retrieval (size, modification time, etc.), moving, renaming, creating, and deleting.

The set of supported operations may be extended in future versions of the protocol.

Implementers of file servers are strongly advised to always support services `Read` and `GetInfo`, as that allows clients to make assumptions about the minimal degree of available service. If write operations are required, all of the defined services should be supported.

Table 5.7: Index of the nested namespace "uavcan.file"

Namespace tree	Ver.	FPID	max(BLS) bytes	Extent bytes	Full name
uavcan					
file					
GetInfo	0.2	405	304 ⇌ 52	300 ⇌ 48	uavcan.file.GetInfo
older version	0.1	405	304 ⇌ 52	300 ⇌ 48	···
List	0.2	406	304 ⇌ 304	300 ⇌ 300	uavcan.file.List
older version	0.1	406	304 ⇌ 304	300 ⇌ 300	···
Modify	1.1	407	604 ⇌ 52	600 ⇌ 48	uavcan.file.Modify
older version	1.0	407	604 ⇌ 52	600 ⇌ 48	···
Read	1.1	408	304 ⇌ 304	300 ⇌ 300	uavcan.file.Read
older version	1.0	408	304 ⇌ 304	300 ⇌ 300	···
Write	1.1	409	604 ⇌ 52	600 ⇌ 48	uavcan.file.Write
older version	1.0	409	604 ⇌ 52	600 ⇌ 48	···
Error	1.0		2	*sealed*	uavcan.file.Error
Path	2.0		256	*sealed*	uavcan.file.Path
older version	1.0		113	*sealed*	···

5.3.8 Generic node commands

Commonly used node-level application-agnostic auxiliary commands (such as: restart, power off, factory reset, emergency stop, etc.) are supported via the standard service `uavcan.node.ExecuteCommand` (section 6.4.1 on page 98). The service also allows vendors to define vendor-specific commands alongside the standard ones.

It is recommended to support this service in all nodes.

5.3.9 Node software update

A simple software[106] update protocol is defined on top of the remote file system interface (section 5.3.7) and the generic node commands (section 5.3.8).

The software update process involves the following data types:

- `uavcan.node.ExecuteCommand` (section 6.4.1 on page 98) – used to initiate the software update process.
- `uavcan.file.Read` (section 6.2.4 on page 91) – used to transfer the software image file(s) from the file server to the updated node.

The software update protocol logic is described in detail in the documentation for the data type `uavcan.node.ExecuteCommand` (section 6.4.1 on page 98). The protocol is considered simple enough to be usable in embedded bootloaders with small memory-constrained microcontrollers.

5.3.10 Register interface

UAVCAN defines the concept of *named register* – a scalar, vector, or string value with an associated human-readable name that is stored on a UAVCAN node locally and is accessible via UAVCAN[107] for reading and/or modification by other nodes on the bus.

Named registers are designed to serve the following purposes:

Node configuration parameter management — Named registers can be used to expose persistently stored values that define behaviors of the local node.

Diagnostics and monitoring — Named registers can be used to expose internal states (variables) of the node's decision-making and data processing logic (implemented in software or hardware) to provide insights about its inner processes.

Advanced node information reporting — Named registers can store any invariants provided by the vendor, such as calibration coefficients or unique identifiers.

Special functions — Non-persistent named registers can be used to trigger specific behaviors or start predefined operations when written.

Advanced debugging — Registers following a specific naming pattern can be used to provide direct read and write access to the local node's application software to facilitate in-depth debugging and monitoring.

The register protocol rests upon two service types defined in the namespace `uavcan.register` (section 6.8 on page 110); the namespace index is shown in table 5.8. Data types supported by the register protocol are defined in the nested data structure `uavcan.register.Value` (section 6.8.4 on page 113).

The UAVCAN specification defines several standard naming patterns to facilitate cross-vendor compatibility and provide a framework of common basic functionality.

Table 5.8: Index of the nested namespace "uavcan.register"

Namespace tree	Ver.	FPID	max(BLS) bytes	Extent bytes	Full name
`uavcan`					
`register`					
`Access`	1.0	384	515 ⥫ 267	*sealed* ⥫ *sealed*	`uavcan.register.Access`
`List`	1.0	385	2 ⥫ 256	*sealed* ⥫ *sealed*	`uavcan.register.List`
`Name`	1.0		256	*sealed*	`uavcan.register.Name`
`Value`	1.0		259	*sealed*	`uavcan.register.Value`

5.3.11 Diagnostics and event logging

The message type `uavcan.diagnostic.Record` (section 6.1.1 on page 86) is designed to facilitate emission of human-readable diagnostic messages and event logging, both for the needs of real-time display[108] and for long-term storage[109].

5.3.12 Plug-and-play nodes

Every UAVCAN node shall have a node-ID that is unique within the network (excepting anonymous nodes). Normally, such identifiers are assigned by the network designer, integrator, some automatic external tool, or another entity that is external to the network. However, there exist circumstances where such manual assignment is either difficult or undesirable.

[106] Or firmware – UAVCAN does not distinguish between the two.
[107] And, possibly, other interfaces.
[108] E.g., messages displayed to a human operator/pilot in real time.
[109] E.g., flight data recording.

Nodes that can join any UAVCAN network automatically without any prior manual configuration are called *plug-and-play nodes* (or *PnP nodes* for brevity).

Plug-and-play nodes automatically obtain a node-ID and deduce all necessary parameters of the physical layer such as the bit rate.

UAVCAN defines an automatic node-ID allocation protocol that is built on top of the data types defined in the namespace `uavcan.pnp` (section 6.6 on page 105) (where *pnp* stands for "plug-and-play") (see table 5.9). The protocol is described in the documentation for the data type `uavcan.pnp.NodeIDAllocationData` (section 6.6.1 on page 105).

The plug-and-play node-ID allocation protocol relies on anonymous messages reviewed in section 4.1.1.4. Remember that the plug-and-play feature is entirely optional and it is expected that applications where a high degree of determinism and robustness is expected are unlikely to benefit from it.

This feature derives from the work "In search of an understandable consensus algorithm"[110] by Diego Ongaro and John Ousterhout.

Table 5.9: Index of the nested namespace "uavcan.pnp"

Namespace tree	Ver.	FPID	max(BLS) bytes	Extent bytes	Full name
uavcan					
pnp					
NodeIDAllocationData	2.0	8165	52	48	uavcan.pnp.NodeIDAllocationData
older version	1.0	8166	9	*sealed*	…
cluster					
AppendEntries	1.0	390	100 ⥫ 52	96 ⥫ 48	uavcan.pnp.cluster.AppendEntries
Discovery	1.0	8164	100	96	uavcan.pnp.cluster.Discovery
RequestVote	1.0	391	52 ⥫ 52	48 ⥫ 48	uavcan.pnp.cluster.RequestVote
Entry	1.0		22	*sealed*	uavcan.pnp.cluster.Entry

5.3.13 Internet/LAN forwarding interface

Data types defined in the namespace `uavcan.internet` (section 6.3 on page 93) (see table 5.10) are designed for establishing robust direct connectivity between local UAVCAN nodes and hosts on the Internet or on a local area network (LAN) using *modem nodes*[111] (possibly redundant).

This basic support for world-wide communication provided at the protocol level allows any component of a vehicle equipped with modem nodes to reach external resources or exchange arbitrary data globally without depending on an application-specific means of communication[112].

The set of supported Internet/LAN protocols may be extended in future revisions of the specification.

Some of the major applications for this feature are as follows:

1. Direct telemetry transmission from UAVCAN nodes to a remote data collection server.
2. Implementation of remote API for on-board equipment (e.g., web interface).
3. Reception of real-time correction data streams (e.g., RTCM RC-104) for precise positioning applications.
4. Automatic upgrades directly from the vendor's Internet resources.

Table 5.10: Index of the nested namespace "uavcan.internet"

Namespace tree	Ver.	FPID	max(BLS) bytes	Extent bytes	Full name
uavcan					
internet					
udp					
HandleIncomingPacket	0.2	500	604 ⥫ 67	600 ⥫ 63	uavcan.internet.udp.HandleIncomingPacket
older version	0.1	500	604 ⥫ 67	600 ⥫ 63	…
OutgoingPacket	0.2	8174	604	600	uavcan.internet.udp.OutgoingPacket
older version	0.1	8174	604	600	…

5.3.14 Meta-transport

Data types defined in the namespace `uavcan.metatransport` (section 6.10 on page 117) (see table 5.11) are designed for tunneling transport frames[113] over UAVCAN subjects, as well as logging UAVCAN traffic in the form of serialized UAVCAN message objects.

110 Proceedings of USENIX Annual Technical Conference, p. 305-320, 2014.
111 Usually such modem nodes are implemented using on-board cellular, radio frequency, or satellite communication hardware.
112 Information security and other security-related concerns are outside of the scope of this specification.
113 Section 4.1.1.

Table 5.11: Index of the nested namespace "uavcan.metatransport"

Namespace tree	Ver.	FPID	max(BLS) bytes	Extent bytes	Full name
uavcan					
metatransport					
can					
ArbitrationID	0.1		5	*sealed*	uavcan.metatransport.can.ArbitrationID
BaseArbitrationID	0.1		4	*sealed*	uavcan.metatransport.can.BaseArbitrationID
DataClassic	0.1		14	*sealed*	uavcan.metatransport.can.DataClassic
DataFD	0.1		70	*sealed*	uavcan.metatransport.can.DataFD
Error	0.1		4	*sealed*	uavcan.metatransport.can.Error
ExtendedArbitrationID	0.1		4	*sealed*	uavcan.metatransport.can.ExtendedArbitrationID
Frame	0.2		71	*sealed*	uavcan.metatransport.can.Frame
older version	0.1		78	*sealed*	…
Manifestation	0.1		71	*sealed*	uavcan.metatransport.can.Manifestation
RTR	0.1		5	*sealed*	uavcan.metatransport.can.RTR
ethernet					
EtherType	0.1		2	*sealed*	uavcan.metatransport.ethernet.EtherType
Frame	0.1		9232	*sealed*	uavcan.metatransport.ethernet.Frame
serial					
Fragment	0.2		2050	*sealed*	uavcan.metatransport.serial.Fragment
older version	0.1		265	*sealed*	…
udp					
Endpoint	0.1		32	*sealed*	uavcan.metatransport.udp.Endpoint
Frame	0.1		10244	10240	uavcan.metatransport.udp.Frame

6 List of standard data types

This chapter contains the full list of standard data types defined by the UAVCAN specification. The source text of the DSDL data type definitions provided here is also available via the official project website at uavcan.org.

Regulated non-standard definitions[114] are not included in this list.

In the table, *BLS* stands for bit length set. The extent is not shown for sealed entities – that would be redundant because sealing implies that the extent equals the maximum bit length set. For service types, the parameters pertaining to the request and response are shown separately.

The index table 6.1 is provided before the definitions for ease of navigation.

[114] I.e., public definitions contributed by vendors and other users of the specification, as explained in section 2.1.2.2.

Table 6.1: Index of the root namespace "uavcan"

Namespace tree	Ver.	FPID	max(BLS) bytes	Extent bytes	Full name
uavcan					
diagnostic					
Record	1.1	8184	304	300	uavcan.diagnostic.Record
older version	1.0	8184	304	300	···
Severity	1.0		1	*sealed*	uavcan.diagnostic.Severity
file					
GetInfo	0.2	405	304 ⇌ 52	300 ⇌ 48	uavcan.file.GetInfo
older version	0.1	405	304 ⇌ 52	300 ⇌ 48	···
List	0.2	406	304 ⇌ 304	300 ⇌ 300	uavcan.file.List
older version	0.1	406	304 ⇌ 304	300 ⇌ 300	···
Modify	1.1	407	604 ⇌ 52	600 ⇌ 48	uavcan.file.Modify
older version	1.0	407	604 ⇌ 52	600 ⇌ 48	···
Read	1.1	408	304 ⇌ 304	300 ⇌ 300	uavcan.file.Read
older version	1.0	408	304 ⇌ 304	300 ⇌ 300	···
Write	1.1	409	604 ⇌ 52	600 ⇌ 48	uavcan.file.Write
older version	1.0	409	604 ⇌ 52	600 ⇌ 48	···
Error	1.0		2	*sealed*	uavcan.file.Error
Path	2.0		256	*sealed*	uavcan.file.Path
older version	1.0		113	*sealed*	···
internet					
udp					
HandleIncomingPacket	0.2	500	604 ⇌ 67	600 ⇌ 63	uavcan.internet.udp.HandleIncomingPacket
older version	0.1	500	604 ⇌ 67	600 ⇌ 63	···
OutgoingPacket	0.2	8174	604	600	uavcan.internet.udp.OutgoingPacket
older version	0.1	8174	604	600	···
node					
ExecuteCommand	1.1	435	304 ⇌ 52	300 ⇌ 48	uavcan.node.ExecuteCommand
older version	1.0	435	304 ⇌ 52	300 ⇌ 48	···
GetInfo	1.0	430	0 ⇌ 452	*sealed* ⇌ 448	uavcan.node.GetInfo
GetTransportStatistics	0.1	434	0 ⇌ 196	*sealed* ⇌ 192	uavcan.node.GetTransportStatistics
Heartbeat	1.0	7509	16	12	uavcan.node.Heartbeat
Health	1.0		1	*sealed*	uavcan.node.Health
ID	1.0		2	*sealed*	uavcan.node.ID
IOStatistics	0.1		15	*sealed*	uavcan.node.IOStatistics
Mode	1.0		1	*sealed*	uavcan.node.Mode
Version	1.0		2	*sealed*	uavcan.node.Version
port					
List	0.1	7510	8466	*sealed*	uavcan.node.port.List
ID	1.0		3	*sealed*	uavcan.node.port.ID
ServiceID	1.0		2	*sealed*	uavcan.node.port.ServiceID
ServiceIDList	0.1		132	128	uavcan.node.port.ServiceIDList
SubjectID	1.0		2	*sealed*	uavcan.node.port.SubjectID
SubjectIDList	0.1		4101	4097	uavcan.node.port.SubjectIDList
pnp					
NodeIDAllocationData	2.0	8165	52	48	uavcan.pnp.NodeIDAllocationData
older version	1.0	8166	9	*sealed*	···
cluster					
AppendEntries	1.0	390	100 ⇌ 52	96 ⇌ 48	uavcan.pnp.cluster.AppendEntries
Discovery	1.0	8164	100	96	uavcan.pnp.cluster.Discovery
RequestVote	1.0	391	52 ⇌ 52	48 ⇌ 48	uavcan.pnp.cluster.RequestVote
Entry	1.0		22	*sealed*	uavcan.pnp.cluster.Entry
register					
Access	1.0	384	515 ⇌ 267	*sealed* ⇌ *sealed*	uavcan.register.Access
List	1.0	385	2 ⇌ 256	*sealed* ⇌ *sealed*	uavcan.register.List
Name	1.0		256	*sealed*	uavcan.register.Name
Value	1.0		259	*sealed*	uavcan.register.Value
time					
GetSynchronizationMasterInfo	0.1	510	52 ⇌ 196	48 ⇌ 192	uavcan.time.GetSynchronizationMasterInfo
Synchronization	1.0	7168	7	*sealed*	uavcan.time.Synchronization
SynchronizedTimestamp	1.0		7	*sealed*	uavcan.time.SynchronizedTimestamp
TAIInfo	0.1		2	*sealed*	uavcan.time.TAIInfo
TimeSystem	0.1		1	*sealed*	uavcan.time.TimeSystem
metatransport					
can					
ArbitrationID	0.1		5	*sealed*	uavcan.metatransport.can.ArbitrationID
BaseArbitrationID	0.1		4	*sealed*	uavcan.metatransport.can.BaseArbitrationID
DataClassic	0.1		14	*sealed*	uavcan.metatransport.can.DataClassic
DataFD	0.1		70	*sealed*	uavcan.metatransport.can.DataFD
Error	0.1		4	*sealed*	uavcan.metatransport.can.Error
ExtendedArbitrationID	0.1		4	*sealed*	uavcan.metatransport.can.ExtendedArbitrationID
Frame	0.2		71	*sealed*	uavcan.metatransport.can.Frame
older version	0.1		78	*sealed*	···
Manifestation	0.1		71	*sealed*	uavcan.metatransport.can.Manifestation
RTR	0.1		5	*sealed*	uavcan.metatransport.can.RTR
ethernet					
EtherType	0.1		2	*sealed*	uavcan.metatransport.ethernet.EtherType
Frame	0.1		9232	*sealed*	uavcan.metatransport.ethernet.Frame
serial					
Fragment	0.2		2050	*sealed*	uavcan.metatransport.serial.Fragment
older version	0.1		265	*sealed*	···

udp				
Endpoint	0.1	32	*sealed*	uavcan.metatransport.udp.Endpoint
Frame	0.1	10244	10240	uavcan.metatransport.udp.Frame
primitive				
Empty	1.0	0	*sealed*	uavcan.primitive.Empty
String	1.0	258	*sealed*	uavcan.primitive.String
Unstructured	1.0	258	*sealed*	uavcan.primitive.Unstructured
array				
Bit	1.0	258	*sealed*	uavcan.primitive.array.Bit
Integer8	1.0	258	*sealed*	uavcan.primitive.array.Integer8
Integer16	1.0	257	*sealed*	uavcan.primitive.array.Integer16
Integer32	1.0	257	*sealed*	uavcan.primitive.array.Integer32
Integer64	1.0	257	*sealed*	uavcan.primitive.array.Integer64
Natural8	1.0	258	*sealed*	uavcan.primitive.array.Natural8
Natural16	1.0	257	*sealed*	uavcan.primitive.array.Natural16
Natural32	1.0	257	*sealed*	uavcan.primitive.array.Natural32
Natural64	1.0	257	*sealed*	uavcan.primitive.array.Natural64
Real16	1.0	257	*sealed*	uavcan.primitive.array.Real16
Real32	1.0	257	*sealed*	uavcan.primitive.array.Real32
Real64	1.0	257	*sealed*	uavcan.primitive.array.Real64
scalar				
Bit	1.0	1	*sealed*	uavcan.primitive.scalar.Bit
Integer8	1.0	1	*sealed*	uavcan.primitive.scalar.Integer8
Integer16	1.0	2	*sealed*	uavcan.primitive.scalar.Integer16
Integer32	1.0	4	*sealed*	uavcan.primitive.scalar.Integer32
Integer64	1.0	8	*sealed*	uavcan.primitive.scalar.Integer64
Natural8	1.0	1	*sealed*	uavcan.primitive.scalar.Natural8
Natural16	1.0	2	*sealed*	uavcan.primitive.scalar.Natural16
Natural32	1.0	4	*sealed*	uavcan.primitive.scalar.Natural32
Natural64	1.0	8	*sealed*	uavcan.primitive.scalar.Natural64
Real16	1.0	2	*sealed*	uavcan.primitive.scalar.Real16
Real32	1.0	4	*sealed*	uavcan.primitive.scalar.Real32
Real64	1.0	8	*sealed*	uavcan.primitive.scalar.Real64
si				
sample				
acceleration				
Scalar	1.0	11	*sealed*	uavcan.si.sample.acceleration.Scalar
Vector3	1.0	19	*sealed*	uavcan.si.sample.acceleration.Vector3
angle				
Quaternion	1.0	23	*sealed*	uavcan.si.sample.angle.Quaternion
Scalar	1.0	11	*sealed*	uavcan.si.sample.angle.Scalar
angular_acceleration				
Scalar	1.0	11	*sealed*	uavcan.si.sample.angular_acceleration.Scalar
Vector3	1.0	19	*sealed*	uavcan.si.sample.angular_acceleration.Vector3
angular_velocity				
Scalar	1.0	11	*sealed*	uavcan.si.sample.angular_velocity.Scalar
Vector3	1.0	19	*sealed*	uavcan.si.sample.angular_velocity.Vector3
duration				
Scalar	1.0	11	*sealed*	uavcan.si.sample.duration.Scalar
WideScalar	1.0	15	*sealed*	uavcan.si.sample.duration.WideScalar
electric_charge				
Scalar	1.0	11	*sealed*	uavcan.si.sample.electric_charge.Scalar
electric_current				
Scalar	1.0	11	*sealed*	uavcan.si.sample.electric_current.Scalar
energy				
Scalar	1.0	11	*sealed*	uavcan.si.sample.energy.Scalar
force				
Scalar	1.0	11	*sealed*	uavcan.si.sample.force.Scalar
Vector3	1.0	19	*sealed*	uavcan.si.sample.force.Vector3
frequency				
Scalar	1.0	11	*sealed*	uavcan.si.sample.frequency.Scalar
length				
Scalar	1.0	11	*sealed*	uavcan.si.sample.length.Scalar
Vector3	1.0	19	*sealed*	uavcan.si.sample.length.Vector3
WideScalar	1.0	15	*sealed*	uavcan.si.sample.length.WideScalar
WideVector3	1.0	31	*sealed*	uavcan.si.sample.length.WideVector3
magnetic_field_strength				
Scalar	1.0	11	*sealed*	uavcan.si.sample.magnetic_field_strength.Scalar
Vector3	1.0	19	*sealed*	uavcan.si.sample.magnetic_field_strength.Vector3
mass				
Scalar	1.0	11	*sealed*	uavcan.si.sample.mass.Scalar
power				
Scalar	1.0	11	*sealed*	uavcan.si.sample.power.Scalar
pressure				
Scalar	1.0	11	*sealed*	uavcan.si.sample.pressure.Scalar
temperature				
Scalar	1.0	11	*sealed*	uavcan.si.sample.temperature.Scalar
torque				
Scalar	1.0	11	*sealed*	uavcan.si.sample.torque.Scalar
Vector3	1.0	19	*sealed*	uavcan.si.sample.torque.Vector3
velocity				
Scalar	1.0	11	*sealed*	uavcan.si.sample.velocity.Scalar
Vector3	1.0	19	*sealed*	uavcan.si.sample.velocity.Vector3

voltage				
Scalar	1.0	11	*sealed*	uavcan.si.sample.voltage.Scalar
volume				
Scalar	1.0	11	*sealed*	uavcan.si.sample.volume.Scalar
volumetric_flow_rate				
Scalar	1.0	11	*sealed*	uavcan.si.sample.volumetric_flow_rate.Scalar
unit				
acceleration				
Scalar	1.0	4	*sealed*	uavcan.si.unit.acceleration.Scalar
Vector3	1.0	12	*sealed*	uavcan.si.unit.acceleration.Vector3
angle				
Quaternion	1.0	16	*sealed*	uavcan.si.unit.angle.Quaternion
Scalar	1.0	4	*sealed*	uavcan.si.unit.angle.Scalar
angular_acceleration				
Scalar	1.0	4	*sealed*	uavcan.si.unit.angular_acceleration.Scalar
Vector3	1.0	12	*sealed*	uavcan.si.unit.angular_acceleration.Vector3
angular_velocity				
Scalar	1.0	4	*sealed*	uavcan.si.unit.angular_velocity.Scalar
Vector3	1.0	12	*sealed*	uavcan.si.unit.angular_velocity.Vector3
duration				
Scalar	1.0	4	*sealed*	uavcan.si.unit.duration.Scalar
WideScalar	1.0	8	*sealed*	uavcan.si.unit.duration.WideScalar
electric_charge				
Scalar	1.0	4	*sealed*	uavcan.si.unit.electric_charge.Scalar
electric_current				
Scalar	1.0	4	*sealed*	uavcan.si.unit.electric_current.Scalar
energy				
Scalar	1.0	4	*sealed*	uavcan.si.unit.energy.Scalar
force				
Scalar	1.0	4	*sealed*	uavcan.si.unit.force.Scalar
Vector3	1.0	12	*sealed*	uavcan.si.unit.force.Vector3
frequency				
Scalar	1.0	4	*sealed*	uavcan.si.unit.frequency.Scalar
length				
Scalar	1.0	4	*sealed*	uavcan.si.unit.length.Scalar
Vector3	1.0	12	*sealed*	uavcan.si.unit.length.Vector3
WideScalar	1.0	8	*sealed*	uavcan.si.unit.length.WideScalar
WideVector3	1.0	24	*sealed*	uavcan.si.unit.length.WideVector3
magnetic_field_strength				
Scalar	1.0	4	*sealed*	uavcan.si.unit.magnetic_field_strength.Scalar
Vector3	1.0	12	*sealed*	uavcan.si.unit.magnetic_field_strength.Vector3
mass				
Scalar	1.0	4	*sealed*	uavcan.si.unit.mass.Scalar
power				
Scalar	1.0	4	*sealed*	uavcan.si.unit.power.Scalar
pressure				
Scalar	1.0	4	*sealed*	uavcan.si.unit.pressure.Scalar
temperature				
Scalar	1.0	4	*sealed*	uavcan.si.unit.temperature.Scalar
torque				
Scalar	1.0	4	*sealed*	uavcan.si.unit.torque.Scalar
Vector3	1.0	12	*sealed*	uavcan.si.unit.torque.Vector3
velocity				
Scalar	1.0	4	*sealed*	uavcan.si.unit.velocity.Scalar
Vector3	1.0	12	*sealed*	uavcan.si.unit.velocity.Vector3
voltage				
Scalar	1.0	4	*sealed*	uavcan.si.unit.voltage.Scalar
volume				
Scalar	1.0	4	*sealed*	uavcan.si.unit.volume.Scalar
volumetric_flow_rate				
Scalar	1.0	4	*sealed*	uavcan.si.unit.volumetric_flow_rate.Scalar

6.1 uavcan.diagnostic

6.1.1 Record

Full message type name: **`uavcan.diagnostic.Record`**

6.1.1.1 Version 1.1, fixed subject ID 8184

Size without delimiter header: 9...264 bytes; extent 300 bytes.

```
# Generic human-readable text message for logging and displaying purposes.
# Generally, it should be published at the lowest priority level.

uavcan.time.SynchronizedTimestamp.1.0 timestamp
# Optional timestamp in the network-synchronized time system; zero if undefined.
# The timestamp value conveys the exact moment when the reported event took place.

Severity.1.0 severity

uint8[<256] text
# Message text.
# Normally, messages should be kept as short as possible, especially those of high severity.

@extent 300 * 8
```

6.1.1.2 Version 1.0, fixed subject ID 8184, DEPRECATED

Size without delimiter header: 9...121 bytes; extent 300 bytes.

```
# Generic human-readable text message for logging and displaying purposes.
# Generally, it should be published at the lowest priority level.

@deprecated

uavcan.time.SynchronizedTimestamp.1.0 timestamp
# Optional timestamp in the network-synchronized time system; zero if undefined.
# The timestamp value conveys the exact moment when the reported event took place.

Severity.1.0 severity

uint8[<=112] text
# Message text.
# Normally, messages should be kept as short as possible, especially those of high severity.

@extent 300 * 8
@assert _offset_.max <= (124 * 8)     # Two CAN FD frames max
```

6.1.2 Severity

Full message type name: **`uavcan.diagnostic.Severity`**

6.1.2.1 Version 1.0

Size 1 bytes; sealed.

```
# Generic message severity representation.

uint3 value
# The severity level ranging from 0 to 7, where low values represent low-severity (unimportant) messages, and
# high values represent high-severity (important) messages. Several mnemonics for the severity levels are
# defined below. Nodes are advised to implement output filtering mechanisms, allowing users to select
# the minimal severity for emitted messages; messages of the selected and higher severity levels will
# be published, and messages of lower severity will be suppressed (discarded).

uint3 TRACE = 0
# Messages of this severity can be used only during development.
# They shall not be used in a fielded operational system.

uint3 DEBUG = 1
# Messages that can aid in troubleshooting.
# Messages of this severity and lower should be disabled by default.

uint3 INFO = 2
# General informational messages of low importance.
# Messages of this severity and lower should be disabled by default.

uint3 NOTICE = 3
# General informational messages of high importance.
# Messages of this severity and lower should be disabled by default.

uint3 WARNING = 4
# Messages reporting abnormalities and warning conditions.
# Messages of this severity and higher should be enabled by default.

uint3 ERROR = 5
# Messages reporting problems and error conditions.
# Messages of this severity and higher should be enabled by default.

uint3 CRITICAL = 6
# Messages reporting serious problems and critical conditions.
# Messages of this severity and higher should be always enabled.
```

```

uint3 ALERT = 7
# Notifications of dangerous circumstances that demand immediate attention.
# Messages of this severity should be always enabled.

@sealed
```

6.2 uavcan.file

6.2.1 GetInfo

Full service type name: **`uavcan.file.GetInfo`**

6.2.1.1 Version 0.2, fixed service ID 405

- Request: Size without delimiter header: 1…256 bytes; extent 300 bytes.
- Response: Size without delimiter header: 13 bytes; extent 48 bytes.

```
# Information about a remote file system entry (file, directory, etc).

Path.2.0 path

@extent 300 * 8

---

Error.1.0 error
# Result of the operation.

truncated uint40 size
# File size in bytes. Should be set to zero for directories.

truncated uint40 unix_timestamp_of_last_modification
# The UNIX Epoch time when the entry was last modified. Zero if unknown.

bool is_file_not_directory  # True if file, false if directory.
bool is_link                # This is a link to another entry; the above flag indicates the type of the target.
bool is_readable            # The item can be read by the caller (applies to files and directories).
bool is_writeable           # The item can be written by the caller (applies to files and directories).
# If such entry does not exist, all flags should be cleared/ignored.
void4

@extent 48 * 8
```

6.2.1.2 Version 0.1, fixed service ID 405, DEPRECATED

- Request: Size without delimiter header: 1…113 bytes; extent 300 bytes.
- Response: Size without delimiter header: 13 bytes; extent 48 bytes.

```
# Information about a remote file system entry (file, directory, etc).

@deprecated

Path.1.0 path

@extent 300 * 8

---

Error.1.0 error
# Result of the operation.

truncated uint40 size
# File size in bytes. Should be set to zero for directories.

truncated uint40 unix_timestamp_of_last_modification
# The UNIX Epoch time when the entry was last modified. Zero if unknown.

bool is_file_not_directory  # True if file, false if directory.
bool is_link                # This is a link to another entry; the above flag indicates the type of the target.
bool is_readable            # The item can be read by the caller (applies to files and directories).
bool is_writeable           # The item can be written by the caller (applies to files and directories).
# If such entry does not exist, all flags should be cleared/ignored.
void4

@extent 48 * 8
```

6.2.2 List

Full service type name: **`uavcan.file.List`**

6.2.2.1 Version 0.2, fixed service ID 406

- Request: Size without delimiter header: 9…264 bytes; extent 300 bytes.
- Response: Size without delimiter header: 5…260 bytes; extent 300 bytes.

```
# This service can be used to list a remote directory, one entry per request.
#
# The client should query each entry independently, iterating 'entry_index' from 0 until the last entry.
# When the index reaches the number of elements in the directory, the server will report that there is
# no such entry by returning an empty name.
#
# The field entry_index shall be applied to an ordered list of directory entries (e.g. alphabetically ordered).
# The exact sorting criteria does not matter as long as it provides the same ordering for subsequent service calls.
#
```

```
# Observe that this listing operation is fundamentally non-atomic. The caller shall beware of possible race conditions
# and is responsible for handling them properly. Particularly, consider what happens if a new item is inserted into
# the directory between two subsequent calls: if the item happened to be inserted at the index that is lower than the
# index of the next request, the next returned item (or several, if more items were inserted) will repeat the ones
# that were listed earlier. The caller should handle that properly, either by ignoring the repeated items or by
# restarting the listing operation from the beginning (index 0).

uint32 entry_index

void32                  # Reserved for future use.

Path.2.0 directory_path

@extent 300 * 8

---

void32                  # Reserved for future use.

Path.2.0 entry_base_name
# The base name of the referenced entry, i.e., relative to the outer directory.
# The outer directory path is not included to conserve bandwidth.
# Empty if such entry does not exist.
#
# For example, suppose there is a file "/foo/bar/baz.bin". Listing the directory with the path "/foo/bar/" (the slash
# at the end is optional) at the index 0 will return "baz.bin". Listing the same directory at the index 1 (or any
# higher) will return an empty name "", indicating that the caller has reached the end of the list.

@extent 300 * 8
```

6.2.2.2 *Version 0.1, fixed service ID 406, DEPRECATED*

- Request: Size without delimiter header: 9...121 bytes; extent 300 bytes.
- Response: Size without delimiter header: 5...117 bytes; extent 300 bytes.

```
# This service can be used to list a remote directory, one entry per request.
#
# The client should query each entry independently, iterating 'entry_index' from 0 until the last entry.
# When the index reaches the number of elements in the directory, the server will report that there is
# no such entry by returning an empty name.
#
# The field entry_index shall be applied to an ordered list of directory entries (e.g. alphabetically ordered).
# The exact sorting criteria does not matter as long as it provides the same ordering for subsequent service calls.
#
# Observe that this listing operation is fundamentally non-atomic. The caller shall beware of possible race conditions
# and is responsible for handling them properly. Particularly, consider what happens if a new item is inserted into
# the directory between two subsequent calls: if the item happened to be inserted at the index that is lower than the
# index of the next request, the next returned item (or several, if more items were inserted) will repeat the ones
# that were listed earlier. The caller should handle that properly, either by ignoring the repeated items or by
# restarting the listing operation from the beginning (index 0).

@deprecated

uint32 entry_index

void32                  # Reserved for future use.

Path.1.0 directory_path

@extent 300 * 8

---

void32                  # Reserved for future use.

Path.1.0 entry_base_name
# The base name of the referenced entry, i.e., relative to the outer directory.
# The outer directory path is not included to conserve bandwidth.
# Empty if such entry does not exist.
#
# For example, suppose there is a file "/foo/bar/baz.bin". Listing the directory with the path "/foo/bar/" (the slash
# at the end is optional) at the index 0 will return "baz.bin". Listing the same directory at the index 1 (or any
# higher) will return an empty name "", indicating that the caller has reached the end of the list.

@extent 300 * 8
```

6.2.3 Modify

Full service type name: **`uavcan.file.Modify`**

6.2.3.1 *Version 1.1, fixed service ID 407*

- Request: Size without delimiter header: 6...516 bytes; extent 600 bytes.
- Response: Size without delimiter header: 2 bytes; extent 48 bytes.

```
# Manipulate a remote file system entry. Applies to files, directories, and links alike.
# If the remote entry is a directory, all nested entries will be affected, too.
#
# The server should perform all operations atomically, unless atomicity is not supported by
# the underlying file system.
#
# Atomic copying can be effectively employed by remote nodes before reading or after writing
```

```
# the file to minimize the possibility of race conditions.
# For example, before reading a large file from the server, the cilent might opt to create
# a temporary copy of it first, then read the copy, and delete it upon completion. Likewise,
# a similar strategy can be employed for writing, where the file is first written at a
# temporary location, and then moved to its final destination. These approaches, however,
# may lead to creation of dangling temporary files if the client failed to dispose of them
# properly, so that risk should be taken into account.
#
# Move/Copy
#   Specify the source path and the destination path.
#   If the source does not exist, the operation will fail.
#   Set the preserve_source flag to copy rather than move.
#   If the destination exists and overwrite_destination is not set, the operation will fail.
#   If the target path includes non-existent directories, they will be created (like "mkdir -p").
#
# Touch
#   Specify the destination path and make the source path empty.
#   If the path exists (file/directory/link), its modification time will be updated.
#   If the path does not exist, an empty file will be created.
#   If the target path includes non-existent directories, they will be created (like "mkdir -p").
#   Flags are ignored.
#
# Remove
#   Specify the source path (file/directory/link) and make the destination path empty.
#   Fails if the path does not exist.
#   Flags are ignored.

bool preserve_source           # Do not remove the source. Used to copy instead of moving.
bool overwrite_destination     # If the destination exists, remove it beforehand.
void30

Path.2.0 source
Path.2.0 destination

@extent 600 * 8

---

Error.1.0 error

@extent 48 * 8
```

6.2.3.2 *Version 1.0, fixed service ID 407, DEPRECATED*

- Request: Size without delimiter header: 6…230 bytes; extent 600 bytes.
- Response: Size without delimiter header: 2 bytes; extent 48 bytes.

```
# Manipulate a remote file system entry. Applies to files, directories, and links alike.
# If the remote entry is a directory, all nested entries will be affected, too.
#
# The server should perform all operations atomically, unless atomicity is not supported by
# the underlying file system.
#
# Atomic copying can be effectively employed by remote nodes before reading or after writing
# the file to minimize the possibility of race conditions.
# For example, before reading a large file from the server, the cilent might opt to create
# a temporary copy of it first, then read the copy, and delete it upon completion. Likewise,
# a similar strategy can be employed for writing, where the file is first written at a
# temporary location, and then moved to its final destination. These approaches, however,
# may lead to creation of dangling temporary files if the client failed to dispose of them
# properly, so that risk should be taken into account.
#
# Move/Copy
#   Specify the source path and the destination path.
#   If the source does not exist, the operation will fail.
#   Set the preserve_source flag to copy rather than move.
#   If the destination exists and overwrite_destination is not set, the operation will fail.
#   If the target path includes non-existent directories, they will be created (like "mkdir -p").
#
# Touch
#   Specify the destination path and make the source path empty.
#   If the path exists (file/directory/link), its modification time will be updated.
#   If the path does not exist, an empty file will be created.
#   If the target path includes non-existent directories, they will be created (like "mkdir -p").
#   Flags are ignored.
#
# Remove
#   Specify the source path (file/directory/link) and make the destination path empty.
#   Fails if the path does not exist.
#   Flags are ignored.

@deprecated

bool preserve_source           # Do not remove the source. Used to copy instead of moving.
bool overwrite_destination     # If the destination exists, remove it beforehand.
void30

Path.1.0 source
Path.1.0 destination

@extent 600 * 8

---

Error.1.0 error

@extent 48 * 8
```

6.2.4 Read

Full service type name: **uavcan.file.Read**

6.2.4.1 Version 1.1, fixed service ID 408

- Request: Size without delimiter header: 6…261 bytes; extent 300 bytes.
- Response: Size without delimiter header: 4…260 bytes; extent 300 bytes.

```
# Read file from a remote node.
#
# There are two possible outcomes of a successful call:
#  1. Data array size equals its capacity. This means that the end of the file is not reached yet.
#  2. Data array size is less than its capacity, possibly zero. This means that the end of the file is reached.
#
# Thus, if the client needs to fetch the entire file, it should repeatedly call this service while increasing the
# offset, until a non-full data array is returned.
#
# If the object pointed by 'path' cannot be read (e.g. it is a directory or it does not exist), an appropriate error
# code will be returned, and the data array will be empty.
#
# It is easy to see that this protocol is prone to race conditions because the remote file can be modified
# between read operations which might result in the client obtaining a damaged file. To combat this,
# application designers are recommended to adhere to the following convention. Let every file whose integrity
# is of interest have a hash or a digital signature, which is stored in an adjacent file under the same name
# suffixed with the appropriate extension according to the type of hash or digital signature used.
# For example, let there be file "image.bin", integrity of which shall be ensured by the client upon downloading.
# Suppose that the file is hashed using SHA-256, so the appropriate file extension for the hash would be
# ".sha256". Following this convention, the hash of "image.bin" would be stored in "image.bin.sha256".
# After downloading the file, the client would read the hash (being small, the hash can be read in a single
# request) and check it against a locally computed value. Some servers may opt to generate such hash files
# automatically as necessary; for example, if such file is requested but it does not exist, the server would
# compute the necessary signature or hash (the type of hash/signature can be deduced from the requested file
# extension) and return it as if the file existed. Obviously, this would be impractical for very large files;
# in that case, hash/signature should be pre-computed and stored in a real file. If this approach is followed,
# implementers are advised to use only SHA-256 for hashing, in order to reduce the number of fielded
# incompatible implementations.

truncated uint40 offset

Path.2.0 path

@extent 300 * 8

---

Error.1.0 error

uavcan.primitive.Unstructured.1.0 data

@extent 300 * 8
```

6.2.4.2 Version 1.0, fixed service ID 408, DEPRECATED

- Request: Size without delimiter header: 6…118 bytes; extent 300 bytes.
- Response: Size without delimiter header: 4…260 bytes; extent 300 bytes.

```
# Read file from a remote node.
#
# There are two possible outcomes of a successful call:
#  1. Data array size equals its capacity. This means that the end of the file is not reached yet.
#  2. Data array size is less than its capacity, possibly zero. This means that the end of the file is reached.
#
# Thus, if the client needs to fetch the entire file, it should repeatedly call this service while increasing the
# offset, until a non-full data array is returned.
#
# If the object pointed by 'path' cannot be read (e.g. it is a directory or it does not exist), an appropriate error
# code will be returned, and the data array will be empty.
#
# It is easy to see that this protocol is prone to race conditions because the remote file can be modified
# between read operations which might result in the client obtaining a damaged file. To combat this,
# application designers are recommended to adhere to the following convention. Let every file whose integrity
# is of interest have a hash or a digital signature, which is stored in an adjacent file under the same name
# suffixed with the appropriate extension according to the type of hash or digital signature used.
# For example, let there be file "image.bin", integrity of which shall be ensured by the client upon downloading.
# Suppose that the file is hashed using SHA-256, so the appropriate file extension for the hash would be
# ".sha256". Following this convention, the hash of "image.bin" would be stored in "image.bin.sha256".
# After downloading the file, the client would read the hash (being small, the hash can be read in a single
# request) and check it against a locally computed value. Some servers may opt to generate such hash files
# automatically as necessary; for example, if such file is requested but it does not exist, the server would
# compute the necessary signature or hash (the type of hash/signature can be deduced from the requested file
# extension) and return it as if the file existed. Obviously, this would be impractical for very large files;
# in that case, hash/signature should be pre-computed and stored in a real file. If this approach is followed,
# implementers are advised to use only SHA-256 for hashing, in order to reduce the number of fielded
# incompatible implementations.

@deprecated

truncated uint40 offset

Path.1.0 path

@extent 300 * 8

```

```
---

Error.1.0 error

uint8[<=256] data

@extent 300 * 8
```

6.2.5 Write

Full service type name: **uavcan.file.Write**

6.2.5.1 Version 1.1, fixed service ID 409

- Request: Size without delimiter header: 8...519 bytes; extent 600 bytes.
- Response: Size without delimiter header: 2 bytes; extent 48 bytes.

```
# Write into a remote file.
# The server shall place the contents of the field 'data' into the file pointed by 'path' at the offset specified by
# the field 'offset'.
#
# When writing a file, the client should repeatedly call this service with data while advancing the offset until the
# file is written completely. When the write sequence is completed, the client shall call the service one last time,
# with the offset set to the size of the file and with the data field empty, which will signal the server that the
# transfer is finished.
#
# When the write operation is complete, the server shall truncate the resulting file past the specified offset.

truncated uint40 offset

Path.2.0 path

uavcan.primitive.Unstructured.1.0 data

@extent 600 * 8

---

Error.1.0 error

@extent 48 * 8
```

6.2.5.2 Version 1.0, fixed service ID 409, DEPRECATED

- Request: Size without delimiter header: 7...311 bytes; extent 600 bytes.
- Response: Size without delimiter header: 2 bytes; extent 48 bytes.

```
# Write into a remote file.
# The server shall place the contents of the field 'data' into the file pointed by 'path' at the offset specified by
# the field 'offset'.
#
# When writing a file, the client should repeatedly call this service with data while advancing the offset until the
# file is written completely. When the write sequence is completed, the client shall call the service one last time,
# with the offset set to the size of the file and with the data field empty, which will signal the server that the
# transfer is finished.
#
# When the write operation is complete, the server shall truncate the resulting file past the specified offset.

@deprecated

truncated uint40 offset

Path.1.0 path

uint8[<=192] data   # 192 = 128 + 64; the write protocol permits usage of smaller chunks.

@extent 600 * 8

---

Error.1.0 error

@extent 48 * 8
```

6.2.6 Error

Full message type name: **uavcan.file.Error**

6.2.6.1 Version 1.0

Size 2 bytes; sealed.

```
# Nested type.
# Result of a file system operation.

uint16 OK                = 0
uint16 UNKNOWN_ERROR     = 65535

```

```
uint16 NOT_FOUND         = 2
uint16 IO_ERROR          = 5
uint16 ACCESS_DENIED     = 13
uint16 IS_DIRECTORY      = 21  # I.e., attempted read/write on a path that points to a directory
uint16 INVALID_VALUE     = 22  # E.g., file name is not valid for the target file system
uint16 FILE_TOO_LARGE    = 27
uint16 OUT_OF_SPACE      = 28
uint16 NOT_SUPPORTED     = 38

uint16 value

@sealed
```

6.2.7 Path

Full message type name: **uavcan.file.Path**

6.2.7.1 Version 2.0

Size 1...256 bytes; sealed.

```
# Nested type.
# A file system path encoded in UTF8. The only valid separator is the forward slash "/".
# A single slash ("/") refers to the root directory (the location of which is defined by the server).
# Relative references (e.g. "..") are not defined and not permitted (although this may change in the future).
# Conventions (not enforced):
#   - A path pointing to a file or a link to file should not end with a separator.
#   - A path pointing to a directory or to a link to directory should end with a separator.

uint8 SEPARATOR = '/'
uint8 MAX_LENGTH = 2 ** 8 - 1

uint8[<=MAX_LENGTH] path

@sealed
```

6.2.7.2 Version 1.0, DEPRECATED

Size 1...113 bytes; sealed.

```
# Nested type.
# A file system path encoded in UTF8. The only valid separator is the forward slash "/".
# A single slash ("/") refers to the root directory (the location of which is defined by the server).
# Relative references (e.g. "..") are not defined and not permitted (although this may change in the future).
# Conventions (not enforced):
#   - A path pointing to a file or a link to file should not end with a separator.
#   - A path pointing to a directory or to a link to directory should end with a separator.
#
# The maximum path length limit is chosen as a trade-off between compatibility with deep directory structures and
# the worst-case transfer length. The limit is 112 bytes, which allows all transfers containing a single instance
# of path and no other large data chunks to fit into two CAN FD frames.

@deprecated

uint8 SEPARATOR = '/'
uint8 MAX_LENGTH = 112

uint8[<=MAX_LENGTH] path

@sealed
```

6.3 uavcan.internet.udp

6.3.1 HandleIncomingPacket

Full service type name: **uavcan.internet.udp.HandleIncomingPacket**

6.3.1.1 Version 0.2, fixed service ID 500

- Request: Size without delimiter header: 4...512 bytes; extent 600 bytes.
- Response: Size without delimiter header: 0 bytes; extent 63 bytes.

```
# This message carries UDP packets sent from a remote host on the Internet or a LAN to a node on the local UAVCAN bus.
# Please refer to the definition of the message type OutgoingPacket for a general overview of the packet forwarding
# logic.
#
# This data type has been made a service type rather than a message type in order to make its transfers addressable,
# allowing nodes to employ hardware acceptance filters for filtering out forwarded datagrams that are not addressed
# to them. Additionally, requiring the destination nodes to always respond upon reception of the forwarded datagram
# opens interesting opportunities for future extensions of the forwarding protocol. If the service invocation times
# out, the modem node is permitted to remove the corresponding entry from the NAT table immediately, not waiting
# for its TTL to expire.
#
# It should be noted that this data type definition intentionally leaves out the source address. This is done in
# order to simplify the implementation, reduce the bus traffic overhead, and because the nature of the
# communication patterns proposed by this set of messages does not provide a valid way to implement server hosts
# on the local UAVCAN bus. It is assumed that local nodes can be only clients, and therefore, they will be able to
# determine the address of the sender simply by mapping the field session_id to their internally maintained states.
```

```
# Furthermore, it is uncertain what is the optimal way of representing the source address for
# client nodes: it is assumed that the local nodes will mostly use DNS names rather than IP addresses, so if there
# was a source address field, modem nodes would have to perform reverse mapping from the IP address they received
# the datagram from to the corresponding DNS name that was used by the local node with the outgoing message. This
# approach creates a number of troubling corner cases and adds a fair amount of hidden complexities to the
# implementation of modem nodes.
#
# It is recommended to perform service invocations at the same transfer priority level as was used for broadcasting
# the latest matching message of type OutgoingPacket. However, meeting this recommendation would require the modem
# node to implement additional logic, which may be undesirable. Therefore, implementers are free to deviate from
# this recommendation and resort to a fixed priority level instead. In the case of a fixed priority level, it is
# advised to use the lowest transfer priority level.

uint16 session_id
# This field shall contain the same value that was used by the local node when sending the corresponding outgoing
# packet using the message type OutgoingPacket. This value will be used by the local node to match the response
# with its local context.

uint8[<=508] payload
# Effective payload. This data will be forwarded from the remote host verbatim.
# UDP packets that contain more than 508 bytes of payload may be dropped by some types of
# communication equipment. Refer to RFC 791 and 2460 for an in-depth review.
# Datagrams that exceed the capacity of this field should be discarded by the modem node.

@extent 600 * 8

---

@extent 63 * 8
```

6.3.1.2 *Version 0.1, fixed service ID 500, DEPRECATED*

- Request: Size without delimiter header: 4...313 bytes; extent 600 bytes.
- Response: Size without delimiter header: 0 bytes; extent 63 bytes.

```
# This message carries UDP packets sent from a remote host on the Internet or a LAN to a node on the local UAVCAN bus.
# Please refer to the definition of the message type OutgoingPacket for a general overview of the packet forwarding
# logic.
#
# This data type has been made a service type rather than a message type in order to make its transfers addressable,
# allowing nodes to employ hardware acceptance filters for filtering out forwarded datagrams that are not addressed
# to them. Additionally, requiring the destination nodes to always respond upon reception of the forwarded datagram
# opens interesting opportunities for future extensions of the forwarding protocol. If the service invocation times
# out, the modem node is permitted to remove the corresponding entry from the NAT table immediately, not waiting
# for its TTL to expire.
#
# It should be noted that this data type definition intentionally leaves out the source address. This is done in
# order to simplify the implementation, reduce the bus traffic overhead, and because the nature of the
# communication patterns proposed by this set of messages does not provide a valid way to implement server hosts
# on the local UAVCAN bus. It is assumed that local nodes can be only clients, and therefore, they will be able to
# determine the address of the sender simply by mapping the field session_id to their internally maintained states.
# Furthermore, it is uncertain what is the optimal way of representing the source address for
# client nodes: it is assumed that the local nodes will mostly use DNS names rather than IP addresses, so if there
# was a source address field, modem nodes would have to perform reverse mapping from the IP address they received
# the datagram from to the corresponding DNS name that was used by the local node with the outgoing message. This
# approach creates a number of troubling corner cases and adds a fair amount of hidden complexities to the
# implementation of modem nodes.
#
# It is recommended to perform service invocations at the same transfer priority level as was used for broadcasting
# the latest matching message of type OutgoingPacket. However, meeting this recommendation would require the modem
# node to implement additional logic, which may be undesirable. Therefore, implementers are free to deviate from
# this recommendation and resort to a fixed priority level instead. In the case of a fixed priority level, it is
# advised to use the lowest transfer priority level.

@deprecated

uint16 session_id
# This field shall contain the same value that was used by the local node when sending the corresponding outgoing
# packet using the message type OutgoingPacket. This value will be used by the local node to match the response
# with its local context.

uint8[<=309] payload
# Effective payload. This data will be forwarded from the remote host verbatim.
# UDP packets that contain more than 508 bytes of payload may be dropped by some types of
# communication equipment. Refer to RFC 791 and 2460 for an in-depth review.
# UAVCAN further limits the maximum packet size to reduce the memory and traffic burden on the nodes.
# Datagrams that exceed the capacity of this field should be discarded by the modem node.

@extent 600 * 8
@assert _offset_ % 8 == {0}
@assert _offset_.max == (313 * 8)      # At most five CAN FD frames

---

@extent 63 * 8
```

6.3.2 OutgoingPacket

Full message type name: **`uavcan.internet.udp.OutgoingPacket`**

6.3.2.1 *Version 0.2, fixed subject ID 8174*

Size without delimiter header: 8...561 bytes; extent 600 bytes.

```
# This message carries UDP packets from a node on the local bus to a remote host on the Internet or a LAN.
#
# Any node can broadcast a message of this type.
#
# All nodes that are capable of communication with the Internet or a LAN should subscribe to messages
# of this type and forward the payload to the indicated host and port using exactly one UDP datagram
# per message (i.e. additional fragmentation is to be avoided). Such nodes will be referred to as
# "modem nodes".
#
# It is expected that some systems will have more than one modem node available.
# Each modem node is supposed to forward every message it sees, which will naturally create
# some degree of modular redundancy and fault tolerance. The remote host should therefore be able to
# properly handle possibly duplicated messages from different source addresses, in addition to
# possible duplications introduced by the UDP/IP protocol itself. There are at least two obvious
# strategies that can be employed by the remote host:
#
#   - Accept only the first message, ignore duplicates. This approach requires that the UDP stream
#     should contain some metadata necessary for the remote host to determine the source and ordering
#     of each received datum. This approach works best for periodic data, such as telemetry, where
#     the sender does not expect any responses.
#
#   - Process all messages, including duplicates. This approach assumes that the remote host acts
#     as a server, processing all received requests and providing responses to each. This arrangement
#     implies that the client may receive duplicated responses. It is therefore the client's
#     responsibility to resolve the possible ambiguity. An obvious solution is to accept the first
#     arrived response and ignore the later ones.
#
# Applications are free to choose whatever redundancy management strategy works best for them.
#
# If the source node expects that the remote host will send some data back, it shall explicitly notify
# the modem nodes about this, so that they could prepare to perform reverse forwarding when the
# expected data arrives from the remote host. The technique of reverse forwarding is known in
# networking as IP Masquerading, or (in general) Network Address Translation (NAT). The notification
# is performed by means of setting one of the corresponding flags defined below.
#
# In order to be able to match datagrams received from remote hosts and the local nodes they should
# be forwarded to, modem nodes are required to keep certain metadata about outgoing datagrams. Such
# metadata is stored in a data structure referred to as "NAT table", where every entry would normally
# contain at least the following fields:
#   - The local UDP port number that was used to send the outgoing datagram from.
#     Per RFC 4787, the port number is chosen by the modem node automatically.
#   - The node-ID of the local node that has sent the outgoing datagram.
#   - Value of the field session_id defined below.
#   - Possibly some other data, depending on the implementation.
#
# The modem nodes are required to keep each NAT table entry for at least NAT_ENTRY_MIN_TTL seconds
# since the last reverse forwarding action was performed. Should the memory resources of the modem node
# be exhausted, it is allowed to remove old NAT entries earlier, following the policy of least recent use.
#
# Having received a UDP packet from a remote host, the modem node would check the NAT table in order
# to determine where on the UAVCAN bus the received data should be forwarded to. If the NAT table
# contains no matches, the received data should be silently dropped. If a match is found, the
# modem node will forward the data to the recipient node using the service HandleIncomingPacket.
# If the service invocation times out, the modem node is permitted to remove the corresponding entry from
# the NAT table immediately (but it is not required). This will ensure that the modem nodes will not be
# tasked with translations for client nodes that are no longer online or are unreachable.
# Additionally, client nodes will be able to hint the modem nodes to remove translation entries they no
# longer need by simply refusing to respond to the corresponding service invocation. Please refer to
# the definition of that service data type for a more in-depth review of the reverse forwarding process.
#
# Modem nodes can also perform traffic shaping, if needed, by means of delaying or dropping UDP
# datagrams that exceed the quota.
#
# To summarize, a typical data exchange occurrence should amount to the following actions:
#
#   - A local UAVCAN node broadcasts a message of type OutgoingPacket with the payload it needs
#     to forward. If the node expects the remote host to send any data back, it sets the masquerading flag.
#
#   - Every modem node on the bus receives the message and performs the following actions:
#
#       - The domain name is resolved, unless the destination address provided in the message
#         is already an IP address, in which case this step should be skipped.
#
#       - The domain name to IP address mapping is added to the local DNS cache, although this
#         part is entirely implementation defined and is not required.
#
#       - The masquerading flag is checked. If it is set, a new entry is added to the NAT table.
#         If such entry already existed, its expiration timeout is reset. If no such entry existed
#         and a new one cannot be added because of memory limitations, the least recently used
#         (i.e. oldest) entry of the NAT table is replaced with the new one.
#
#       - The payload is forwarded to the determined IP address.
#
#   - At this point, direct forwarding is complete. Should any of the modem nodes receive an incoming
#     packet, they would attempt to perform a reverse forwarding according to the above provided algorithm.
#
# It is recommended to use the lowest transport priority level when broadcasting messages of this type,
# in order to avoid interference with a real-time traffic on the bus. Usage of higher priority levels is
# unlikely to be practical because the latency and throughput limitations introduced by the on-board radio
# communication equipment are likely to vastly exceed those of the local CAN bus.

uint32 NAT_ENTRY_MIN_TTL = 24 * 60 * 60  # [second]
# Modem nodes are required to keep the NAT table entries alive for at least this amount of time, unless the
# table is overflowed, in which case they are allowed to remove least recently used entries in favor of
# newer ones. Modem nodes are required to be able to accommodate at least 100 entries in the NAT table.

uint16 session_id
# This field is set to an arbitrary value by the transmitting node in order to be able to match the response
# with the locally kept context. The function of this field is virtually identical to that of UDP/IP port
# numbers. This value can be set to zero safely if the sending node does not have multiple contexts to
# distinguish between.
```

```

uint16 destination_port
# UDP destination port number.

uint8[<=45] destination_address
# Domain name or IP address where the payload should be forwarded to.
# Note that broadcast addresses are allowed here, for example, 255.255.255.255.
# Broadcasting with masquerading enabled works the same way as unicasting with masquerading enabled: the modem
# node should take care to channel all traffic arriving at the opened port from any source to the node that
# requested masquerading.
# The full domain name length may not exceed 253 octets, according to the DNS specification.
# UAVCAN imposes a stricter length limit in order to reduce the memory and traffic burden on the bus: 45 characters.
# 45 characters is the amount of space that is required to represent the longest possible form of an IPv6 address
# (an IPv4-mapped IPv6 address). Examples:
#   "forum.uavcan.org"                              - domain name
#   "192.168.1.1"                                   - IPv4 address
#   "2001:0db8:85a3:0000:0000:8a2e:0370:7334"       - IPv6 address, full form
#   "2001:db8:85a3::8a2e:370:7334"                  - IPv6 address, same as above, short form (preferred)
#   "ABCD:ABCD:ABCD:ABCD:ABCD:ABCD:192.168.158.190" - IPv4-mapped IPv6, full form (length limit, 45 characters)

@assert _offset_ % 8 == {0}

bool use_masquerading   # Expect data back (i.e., instruct the modem to use the NAT table).
bool use_dtls           # Use Datagram Transport Layer Security. Drop the packet if DTLS is not supported.
# Option flags.
void6

uint8[<=508] payload
# Effective payload. This data will be forwarded to the remote host verbatim.
# UDP packets that contain more than 508 bytes of payload may be dropped by some types of
# communication equipment. Refer to RFC 791 and 2460 for an in-depth review.

@extent 600 * 8
```

6.3.2.2 *Version 0.1, fixed subject ID 8174, DEPRECATED*

Size without delimiter header: 8…313 bytes; extent 600 bytes.

```
# This message carries UDP packets from a node on the local bus to a remote host on the Internet or a LAN.
#
# Any node can broadcast a message of this type.
#
# All nodes that are capable of communication with the Internet or a LAN should subscribe to messages
# of this type and forward the payload to the indicated host and port using exactly one UDP datagram
# per message (i.e. additional fragmentation is to be avoided). Such nodes will be referred to as
# "modem nodes".
#
# It is expected that some systems will have more than one modem node available.
# Each modem node is supposed to forward every message it sees, which will naturally create
# some degree of modular redundancy and fault tolerance. The remote host should therefore be able to
# properly handle possibly duplicated messages from different source addresses, in addition to
# possible duplications introduced by the UDP/IP protocol itself. There are at least two obvious
# strategies that can be employed by the remote host:
#
#   - Accept only the first message, ignore duplicates. This approach requires that the UDP stream
#     should contain some metadata necessary for the remote host to determine the source and ordering
#     of each received datum. This approach works best for periodic data, such as telemetry, where
#     the sender does not expect any responses.
#
#   - Process all messages, including duplicates. This approach assumes that the remote host acts
#     as a server, processing all received requests and providing responses to each. This arrangement
#     implies that the client may receive duplicated responses. It is therefore the client's
#     responsibility to resolve the possible ambiguity. An obvious solution is to accept the first
#     arrived response and ignore the later ones.
#
# Applications are free to choose whatever redundancy management strategy works best for them.
#
# If the source node expects that the remote host will send some data back, it shall explicitly notify
# the modem nodes about this, so that they could prepare to perform reverse forwarding when the
# expected data arrives from the remote host. The technique of reverse forwarding is known in
# networking as IP Masquerading, or (in general) Network Address Translation (NAT). The notification
# is performed by means of setting one of the corresponding flags defined below.
#
# In order to be able to match datagrams received from remote hosts and the local nodes they should
# be forwarded to, modem nodes are required to keep certain metadata about outgoing datagrams. Such
# metadata is stored in a data structure referred to as "NAT table", where every entry would normally
# contain at least the following fields:
#   - The local UDP port number that was used to send the outgoing datagram from.
#     Per RFC 4787, the port number is chosen by the modem node automatically.
#   - The node-ID of the local node that has sent the outgoing datagram.
#   - Value of the field session_id defined below.
#   - Possibly some other data, depending on the implementation.
#
# The modem nodes are required to keep each NAT table entry for at least NAT_ENTRY_MIN_TTL seconds
# since the last reverse forwarding action was performed. Should the memory resources of the modem node
# be exhausted, it is allowed to remove old NAT entries earlier, following the policy of least recent use.
#
# Having received a UDP packet from a remote host, the modem node would check the NAT table in order
# to determine where on the UAVCAN bus the received data should be forwarded to. If the NAT table
# contains no matches, the received data should be silently dropped. If a match is found, the
# modem node will forward the data to the recipient node using the service HandleIncomingPacket.
# If the service invocation times out, the modem node is permitted to remove the corresponding entry from
# the NAT table immediately (but it is not required). This will ensure that the modem nodes will not be
# tasked with translations for client nodes that are no longer online or are unreachable.
# Additionally, client nodes will be able to hint the modem nodes to remove translation entries they no
# longer need by simply refusing to respond to the corresponding service invocation. Please refer to
# the definition of that service data type for a more in-depth review of the reverse forwarding process.
#
```

```
# Modem nodes can also perform traffic shaping, if needed, by means of delaying or dropping UDP
# datagrams that exceed the quota.
#
# To summarize, a typical data exchange occurrence should amount to the following actions:
#
#   - A local UAVCAN node broadcasts a message of type OutgoingPacket with the payload it needs
#     to forward. If the node expects the remote host to send any data back, it sets the masquerading flag.
#
#   - Every modem node on the bus receives the message and performs the following actions:
#
#       - The domain name is resolved, unless the destination address provided in the message
#         is already an IP address, in which case this step should be skipped.
#
#       - The domain name to IP address mapping is added to the local DNS cache, although this
#         part is entirely implementation defined and is not required.
#
#       - The masquerading flag is checked. If it is set, a new entry is added to the NAT table.
#         If such entry already existed, its expiration timeout is reset. If no such entry existed
#         and a new one cannot be added because of memory limitations, the least recently used
#         (i.e. oldest) entry of the NAT table is replaced with the new one.
#
#       - The payload is forwarded to the determined IP address.
#
#   - At this point, direct forwarding is complete. Should any of the modem nodes receive an incoming
#     packet, they would attempt to perform a reverse forwarding according to the above provided algorithm.
#
# It is recommended to use the lowest transport priority level when broadcasting messages of this type,
# in order to avoid interference with a real-time traffic on the bus. Usage of higher priority levels is
# unlikely to be practical because the latency and throughput limitations introduced by the on-board radio
# communication equipment are likely to vastly exceed those of the local CAN bus.

@deprecated

uint32 NAT_ENTRY_MIN_TTL = 24 * 60 * 60  # [second]
# Modem nodes are required to keep the NAT table entries alive for at least this amount of time, unless the
# table is overflowed, in which case they are allowed to remove least recently used entries in favor of
# newer ones. Modem nodes are required to be able to accommodate at least 100 entries in the NAT table.

uint16 session_id
# This field is set to an arbitrary value by the transmitting node in order to be able to match the response
# with the locally kept context. The function of this field is virtually identical to that of UDP/IP port
# numbers. This value can be set to zero safely if the sending node does not have multiple contexts to
# distinguish between.

uint16 destination_port
# UDP destination port number.

uint8[<=45] destination_address
# Domain name or IP address where the payload should be forwarded to.
# Note that broadcast addresses are allowed here, for example, 255.255.255.255.
# Broadcasting with masquerading enabled works the same way as unicasting with masquerading enabled: the modem
# node should take care to channel all traffic arriving at the opened port from any source to the node that
# requested masquerading.
# The full domain name length may not exceed 253 octets, according to the DNS specification.
# UAVCAN imposes a stricter length limit in order to reduce the memory and traffic burden on the bus: 45 characters.
# 45 characters is the amount of space that is required to represent the longest possible form of an IPv6 address
# (an IPv4-mapped IPv6 address). Examples:
#   "forum.uavcan.org"                              - domain name
#   "192.168.1.1"                                   - IPv4 address
#   "2001:0db8:85a3:0000:0000:8a2e:0370:7334"       - IPv6 address, full form
#   "2001:db8:85a3::8a2e:370:7334"                  - IPv6 address, same as above, short form (preferred)
#   "ABCD:ABCD:ABCD:ABCD:ABCD:ABCD:192.168.158.190" - IPv4-mapped IPv6, full form (length limit, 45 characters)

@assert _offset_ % 8 == {0}

bool use_masquerading   # Expect data back (i.e., instruct the modem to use the NAT table).
bool use_dtls           # Use Datagram Transport Layer Security. Drop the packet if DTLS is not supported.
# Option flags.
void6

uint8[<=260] payload
# Effective payload. This data will be forwarded to the remote host verbatim.
# UDP packets that contain more than 508 bytes of payload may be dropped by some types of
# communication equipment. Refer to RFC 791 and 2460 for an in-depth review.
# UAVCAN further limits the maximum packet size to reduce the memory and traffic burden on the nodes.

@extent 600 * 8
@assert _offset_ % 8 == {0}
@assert _offset_.max / 8 == 313
```

6.4 uavcan.node

6.4.1 ExecuteCommand

Full service type name: **uavcan.node.ExecuteCommand**

6.4.1.1 Version 1.1, fixed service ID 435

- Request: Size without delimiter header: 3…258 bytes; extent 300 bytes.
- Response: Size without delimiter header: 1 bytes; extent 48 bytes.

```
# Instructs the server node to execute or commence execution of a simple predefined command.
# All standard commands are optional; i.e., not guaranteed to be supported by all nodes.

uint16 command
# Standard pre-defined commands are at the top of the range (defined below).
# Vendors can define arbitrary, vendor-specific commands in the bottom part of the range (starting from zero).
# Vendor-specific commands shall not use identifiers above 32767.

uint16 COMMAND_RESTART = 65535
# Reboot the node.
# Note that some standard commands may or may not require a restart in order to take effect; e.g., factory reset.

uint16 COMMAND_POWER_OFF = 65534
# Shut down the node; further access will not be possible until the power is turned back on.

uint16 COMMAND_BEGIN_SOFTWARE_UPDATE = 65533
# Begin the software update process using uavcan.file.Read. This command makes use of the "parameter" field below.
# The parameter contains the path to the new software image file to be downloaded by the server from the client
# using the standard service uavcan.file.Read. Observe that this operation swaps the roles of the client and
# the server.
#
# Upon reception of this command, the server (updatee) will evaluate whether it is possible to begin the
# software update process. If that is deemed impossible, the command will be rejected with one of the
# error codes defined in the response section of this definition (e.g., BAD_STATE if the node is currently
# on-duty and a sudden interruption of its activities is considered unsafe, and so on).
# If an update process is already underway, the updatee should abort the process and restart with the new file,
# unless the updatee can determine that the specified file is the same file that is already being downloaded,
# in which case it is allowed to respond SUCCESS and continue the old update process.
# If there are no other conditions precluding the requested update, the updatee will return a SUCCESS and
# initiate the file transfer process by invoking the standard service uavcan.file.Read repeatedly until the file
# is transferred fully (please refer to the documentation for that data type for more information about its usage).
#
# While the software is being updated, the updatee should set its mode (the field "mode" in uavcan.node.Heartbeat)
# to MODE_SOFTWARE_UPDATE. Please refer to the documentation for uavcan.node.Heartbeat for more information.
#
# It is recognized that most systems will have to interrupt their normal services to perform the software update
# (unless some form of software hot swapping is implemented, as is the case in some high-availability systems).
#
# Microcontrollers that are requested to update their firmware may need to stop execution of their current firmware
# and start the embedded bootloader (although other approaches are possible as well). In that case,
# while the embedded bootloader is running, the mode reported via the message uavcan.node.Heartbeat should be
# MODE_SOFTWARE_UPDATE as long as the bootloader is runing, even if no update-related activities
# are currently underway. For example, if the update process failed and the bootloader cannot load the software,
# the same mode MODE_SOFTWARE_UPDATE will be reported.
# It is also recognized that in a microcontroller setting, the application that served the update request will have
# to pass the update-related metadata (such as the node-ID of the server and the firmware image file path) to
# the embedded bootloader. The tactics of that transaction lie outside of the scope of this specification.

uint16 COMMAND_FACTORY_RESET = 65532
# Return the node's configuration back to the factory default settings (may require restart).
# Due to the uncertainty whether a restart is required, generic interfaces should always force a restart.

uint16 COMMAND_EMERGENCY_STOP = 65531
# Cease activities immediately, enter a safe state until restarted.
# Further operation may no longer be possible until a restart command is executed.

uint16 COMMAND_STORE_PERSISTENT_STATES = 65530
# This command instructs the node to store the current configuration parameter values and other persistent states
# to the non-volatile storage. Nodes are allowed to manage persistent states automatically, obviating the need for
# this command by committing all such data to the non-volatile memory automatically as necessary. However, some
# nodes may lack this functionality, in which case this parameter should be used. Generic interfaces should always
# invoke this command in order to ensure that the data is stored even if the node doesn't implement automatic
# persistence management.

uint8[<=uavcan.file.Path.2.0.MAX_LENGTH] parameter
# A string parameter supplied to the command. The format and interpretation is command-specific.
# The standard commands do not use this field (ignore it), excepting the following:
#   - COMMAND_BEGIN_SOFTWARE_UPDATE

@extent 300 * 8

---

uint8 STATUS_SUCCESS        = 0     # Started or executed successfully
uint8 STATUS_FAILURE        = 1     # Could not start or the desired outcome could not be reached
uint8 STATUS_NOT_AUTHORIZED = 2     # Denied due to lack of authorization
uint8 STATUS_BAD_COMMAND    = 3     # The requested command is not known or not supported
uint8 STATUS_BAD_PARAMETER  = 4     # The supplied parameter cannot be used with the selected command
uint8 STATUS_BAD_STATE      = 5     # The current state of the node does not permit execution of this command
uint8 STATUS_INTERNAL_ERROR = 6     # The operation should have succeeded but an unexpected failure occurred
uint8 status
# The result of the request.

@extent 48 * 8
```

6.4.1.2 *Version 1.0, fixed service ID 435, DEPRECATED*

- Request: Size without delimiter header: 3 . . . 115 bytes; extent 300 bytes.
- Response: Size without delimiter header: 1 bytes; extent 48 bytes.

```
# Instructs the server node to execute or commence execution of a simple predefined command.
# All standard commands are optional; i.e., not guaranteed to be supported by all nodes.

@deprecated

uint16 command
# Standard pre-defined commands are at the top of the range (defined below).
# Vendors can define arbitrary, vendor-specific commands in the bottom part of the range (starting from zero).
# Vendor-specific commands shall not use identifiers above 32767.

uint16 COMMAND_RESTART = 65535
# Reboot the node.
# Note that some standard commands may or may not require a restart in order to take effect; e.g., factory reset.

uint16 COMMAND_POWER_OFF = 65534
# Shut down the node; further access will not be possible until the power is turned back on.

uint16 COMMAND_BEGIN_SOFTWARE_UPDATE = 65533
# Begin the software update process using uavcan.file.Read. This command makes use of the "parameter" field below.
# The parameter contains the path to the new software image file to be downloaded by the server from the client
# using the standard service uavcan.file.Read. Observe that this operation swaps the roles of the client and
# the server.
#
# Upon reception of this command, the server (updatee) will evaluate whether it is possible to begin the
# software update process. If that is deemed impossible, the command will be rejected with one of the
# error codes defined in the response section of this definition (e.g., BAD_STATE if the node is currently
# on-duty and a sudden interruption of its activities is considered unsafe, and so on).
# If an update process is already underway, the updatee should abort the process and restart with the new file,
# unless the updatee can determine that the specified file is the same file that is already being downloaded,
# in which case it is allowed to respond SUCCESS and continue the old update process.
# If there are no other conditions precluding the requested update, the updatee will return a SUCCESS and
# initiate the file transfer process by invoking the standard service uavcan.file.Read repeatedly until the file
# is transferred fully (please refer to the documentation for that data type for more information about its usage).
#
# While the software is being updated, the updatee should set its mode (the field "mode" in uavcan.node.Heartbeat)
# to MODE_SOFTWARE_UPDATE. Please refer to the documentation for uavcan.node.Heartbeat for more information.
#
# It is recognized that most systems will have to interrupt their normal services to perform the software update
# (unless some form of software hot swapping is implemented, as is the case in some high-availability systems).
#
# Microcontrollers that are requested to update their firmware may need to stop execution of their current firmware
# and start the embedded bootloader (although other approaches are possible as well). In that case,
# while the embedded bootloader is running, the mode reported via the message uavcan.node.Heartbeat should be
# MODE_SOFTWARE_UPDATE as long as the bootloader is runing, even if no update-related activities
# are currently underway. For example, if the update process failed and the bootloader cannot load the software,
# the same mode MODE_SOFTWARE_UPDATE will be reported.
# It is also recognized that in a microcontroller setting, the application that served the update request will have
# to pass the update-related metadata (such as the node-ID of the server and the firmware image file path) to
# the embedded bootloader. The tactics of that transaction lie outside of the scope of this specification.

uint16 COMMAND_FACTORY_RESET = 65532
# Return the node's configuration back to the factory default settings (may require restart).
# Due to the uncertainty whether a restart is required, generic interfaces should always force a restart.

uint16 COMMAND_EMERGENCY_STOP = 65531
# Cease activities immediately, enter a safe state until restarted.
# Further operation may no longer be possible until a restart command is executed.

uint16 COMMAND_STORE_PERSISTENT_STATES = 65530
# This command instructs the node to store the current configuration parameter values and other persistent states
# to the non-volatile storage. Nodes are allowed to manage persistent states automatically, obviating the need for
# this command by committing all such data to the non-volatile memory automatically as necessary. However, some
# nodes may lack this functionality, in which case this parameter should be used. Generic interfaces should always
# invoke this command in order to ensure that the data is stored even if the node doesn't implement automatic
# persistence management.

uint8[<=uavcan.file.Path.1.0.MAX_LENGTH] parameter
# A string parameter supplied to the command. The format and interpretation is command-specific.
# The standard commands do not use this field (ignore it), excepting the following:
#   - COMMAND_BEGIN_SOFTWARE_UPDATE

@assert _offset_ % 8 == {0}
@assert _offset_.max <= (124 * 8)     # Two CAN FD frames max
@extent 300 * 8

---

uint8 STATUS_SUCCESS        = 0     # Started or executed successfully
uint8 STATUS_FAILURE        = 1     # Could not start or the desired outcome could not be reached
uint8 STATUS_NOT_AUTHORIZED = 2     # Denied due to lack of authorization
uint8 STATUS_BAD_COMMAND    = 3     # The requested command is not known or not supported
uint8 STATUS_BAD_PARAMETER  = 4     # The supplied parameter cannot be used with the selected command
uint8 STATUS_BAD_STATE      = 5     # The current state of the node does not permit execution of this command
uint8 STATUS_INTERNAL_ERROR = 6     # The operation should have succeeded but an unexpected failure occurred
uint8 status
# The result of the request.

@extent 48 * 8
```

6.4.2 GetInfo

Full service type name: **`uavcan.node.GetInfo`**

6.4.2.1 *Version 1.0, fixed service ID 430*

- Request: Size 0 bytes; sealed.
- Response: Size without delimiter header: 33…313 bytes; extent 448 bytes.

```
# Full node info request.
# All of the returned information shall be static (unchanged) while the node is running.
# It is highly recommended to support this service on all nodes.

@sealed

---

Version.1.0 protocol_version
# The UAVCAN protocol version implemented on this node, both major and minor.
# Not to be changed while the node is running.

Version.1.0 hardware_version
Version.1.0 software_version
# The version information shall not be changed while the node is running.
# The correct hardware version shall be reported at all times, excepting software-only nodes, in which
# case it should be set to zeros.
# If the node is equipped with a UAVCAN-capable bootloader, the bootloader should report the software
# version of the installed application, if there is any; if no application is found, zeros should be reported.

uint64 software_vcs_revision_id
# A version control system (VCS) revision number or hash. Not to be changed while the node is running.
# For example, this field can be used for reporting the short git commit hash of the current
# software revision.
# Set to zero if not used.

uint8[16] unique_id
# The unique-ID (UID) is a 128-bit long sequence that is likely to be globally unique per node.
# The vendor shall ensure that the probability of a collision with any other node UID globally is negligibly low.
# UID is defined once per hardware unit and should never be changed.
# All zeros is not a valid UID.
# If the node is equipped with a UAVCAN-capable bootloader, the bootloader shall use the same UID.

@assert _offset_ == {30 * 8}
# Manual serialization note: only fixed-size fields up to this point. The following fields are dynamically sized.

uint8[<=50] name
# Human-readable non-empty ASCII node name. An empty name is not permitted.
# The name shall not be changed while the node is running.
# Allowed characters are: a-z (lowercase ASCII letters) 0-9 (decimal digits) . (dot) - (dash) _ (underscore).
# Node name is a reversed Internet domain name (like Java packages), e.g. "com.manufacturer.project.product".

uint64[<=1] software_image_crc
# The value of an arbitrary hash function applied to the software image. Not to be changed while the node is running.
# This field can be used to detect whether the software or firmware running on the node is an exact
# same version as a certain specific revision. This field provides a very strong identity guarantee,
# unlike the version fields above, which can be the same for different builds of the software.
# As can be seen from its definition, this field is optional.
#
# The exact hash function and the methods of its application are implementation-defined.
# However, implementations are recommended to adhere to the following guidelines, fully or partially:
#   - The hash function should be CRC-64-WE.
#   - The hash function should be applied to the entire application image padded to 8 bytes.
#   - If the computed image CRC is stored within the software image itself, the value of
#     the hash function becomes ill-defined, because it becomes recursively dependent on itself.
#     In order to circumvent this issue, while computing or checking the CRC, its value stored
#     within the image should be zeroed out.

uint8[<=222] certificate_of_authenticity
# The certificate of authenticity (COA) of the node, 222 bytes max, optional. This field can be used for
# reporting digital signatures (e.g., RSA-1776, or ECDSA if a higher degree of cryptographic strength is desired).
# Leave empty if not used. Not to be changed while the node is running.

@assert _offset_ % 8 == {0}
@assert _offset_.max == (313 * 8)     # At most five CAN FD frames
@extent 448 * 8
```

6.4.3 GetTransportStatistics

Full service type name: **`uavcan.node.GetTransportStatistics`**

6.4.3.1 *Version 0.1, fixed service ID 434*

- Request: Size 0 bytes; sealed.
- Response: Size without delimiter header: 16…61 bytes; extent 192 bytes.

```
# Returns a set of general low-level transport statistical counters.
# Servers are encouraged but not required to sample the data atomically.

@sealed

---

uint8 MAX_NETWORK_INTERFACES = 3
# UAVCAN supports up to triply modular redundant interfaces.

IOStatistics.0.1 transfer_statistics
# UAVCAN transfer performance statistics:
# the number of UAVCAN transfers successfully sent, successfully received, and failed.
```

```
# The methods of error counting are implementation-defined.

IOStatistics.0.1[<=MAX_NETWORK_INTERFACES] network_interface_statistics
# Network interface statistics, separate per interface.
# E.g., for a doubly redundant transport, this array would contain two elements,
# the one at the index zero would apply to the first interface, the other to the second interface.
# The methods of counting are implementation-defined.

@assert _offset_.max <= (63 * 8)      # One CAN FD frame
@extent 192 * 8
```

6.4.4 Heartbeat

Full message type name: **uavcan.node.Heartbeat**

6.4.4.1 Version 1.0, fixed subject ID 7509

Size without delimiter header: 7 bytes; extent 12 bytes.

```
# Abstract node status information.
# This is the only high-level function that shall be implemented by all nodes.
#
# All UAVCAN nodes that have a node-ID are required to publish this message to its fixed subject periodically.
# Nodes that do not have a node-ID (also known as "anonymous nodes") shall not publish to this subject.
#
# The default subject-ID 7509 is 1110101010101 in binary. The alternating bit pattern at the end helps transceiver
# synchronization (e.g., on CAN-based networks) and on some transports permits automatic bit rate detection.
#
# Network-wide health monitoring can be implemented by subscribing to the fixed subject.

uint16 MAX_PUBLICATION_PERIOD = 1   # [second]
# The publication period shall not exceed this limit.
# The period should not change while the node is running.

uint16 OFFLINE_TIMEOUT = 3          # [second]
# If the last message from the node was received more than this amount of time ago, it should be considered offline.

uint32 uptime                       # [second]
# The uptime seconds counter should never overflow. The counter will reach the upper limit in ~136 years,
# upon which time it should stay at 0xFFFFFFFF until the node is restarted.
# Other nodes may detect that a remote node has restarted when this value leaps backwards.

Health.1.0 health
# The abstract health status of this node.

Mode.1.0 mode
# The abstract operating mode of the publishing node.
# This field indicates the general level of readiness that can be further elaborated on a per-activity basis
# using various specialized interfaces.

uint8 vendor_specific_status_code
# Optional, vendor-specific node status code, e.g. a fault code or a status bitmask.

@assert _offset_ % 8 == {0}
@assert _offset_ == {56}  # Fits into a single-frame Classic CAN transfer (least capable transport, smallest MTU).
@extent 12 * 8
```

6.4.5 Health

Full message type name: **uavcan.node.Health**

6.4.5.1 Version 1.0

Size 1 bytes; sealed.

```
# Abstract component health information. If the node performs multiple activities (provides multiple network services),
# its health status should reflect the status of the worst-performing activity (network service).
# Follows:
#   https://www.law.cornell.edu/cfr/text/14/23.1322
#   https://www.faa.gov/documentLibrary/media/Advisory_Circular/AC_25.1322-1.pdf section 6

uint2 value

uint2 NOMINAL = 0
# The component is functioning properly (nominal).

uint2 ADVISORY = 1
# A critical parameter went out of range or the component encountered a minor failure that does not prevent
# the subsystem from performing any of its real-time functions.

uint2 CAUTION = 2
# The component encountered a major failure and is performing in a degraded mode or outside of its designed limitations.

uint2 WARNING = 3
# The component suffered a fatal malfunction and is unable to perform its intended function.

@sealed
```

6.4.6 ID

Full message type name: **uavcan.node.ID**

6.4.6.1 Version 1.0

Size 2 bytes; sealed.

```
# Defines a node-ID.
# The maximum valid value is dependent on the underlying transport layer.
# Values lower than 128 are always valid for all transports.
# Refer to the specification for more info.

uint16 value

@sealed
@assert _offset_ == {16}
```

6.4.7 IOStatistics

Full message type name: **`uavcan.node.IOStatistics`**

6.4.7.1 Version 0.1

Size 15 bytes; sealed.

```
# A standard set of generic input/output statistical counters that generally should not overflow.
# If a 40-bit counter is incremented every millisecond, it will overflow in ~35 years.
# If an overflow occurs, the value will wrap over to zero.
#
# The values should not be reset while the node is running.

truncated uint40 num_emitted
# The number of successfully emitted entities.

truncated uint40 num_received
# The number of successfully received entities.

truncated uint40 num_errored
# How many errors have occurred.
# The exact definition of "error" and how they are counted are implementation-defined,
# unless specifically defined otherwise.

@sealed
```

6.4.8 Mode

Full message type name: **`uavcan.node.Mode`**

6.4.8.1 Version 1.0

Size 1 bytes; sealed.

```
# The operating mode of a node.
# Reserved values can be used in future revisions of the specification.

uint3 value

uint3 OPERATIONAL = 0
# Normal operating mode.

uint3 INITIALIZATION = 1
# Initialization is in progress; this mode is entered immediately after startup.

uint3 MAINTENANCE = 2
# E.g., calibration, self-test, etc.

uint3 SOFTWARE_UPDATE = 3
# New software/firmware is being loaded or the bootloader is running.

@sealed
```

6.4.9 Version

Full message type name: **`uavcan.node.Version`**

6.4.9.1 Version 1.0

Size 2 bytes; sealed.

```
# A shortened semantic version representation: only major and minor.
# The protocol generally does not concern itself with the patch version.

uint8 major
uint8 minor

@sealed
```

6.5 uavcan.node.port

6.5.1 List

Full message type name: **uavcan.node.port.List**

6.5.1.1 Version 0.1, fixed subject ID 7510

Size 16…8466 bytes; sealed.

```
# A list of ports that this node is using:
# - Subjects published by this node (whether periodically or ad-hoc).
# - Subjects that this node is subscribed to (a datalogger or a debugger would typically subscribe to all subjects).
# - RPC services consumed by this node (i.e., service clients).
# - RPC services provided by this node (i.e., service servers).
#
# All nodes should implement this capability to provide network introspection and diagnostic capabilities.
# This message should be published using the fixed subject-ID as follows:
# - At the OPTIONAL priority level at least every MAX_PUBLICATION_PERIOD seconds.
# - At the OPTIONAL or SLOW priority level within MAX_PUBLICATION_PERIOD after the port configuration is changed.

uint8 MAX_PUBLICATION_PERIOD = 10  # [seconds]
# If the port configuration is not updated in this amount of time, the node should publish this message anyway.

SubjectIDList.0.1 publishers
SubjectIDList.0.1 subscribers
ServiceIDList.0.1 clients
ServiceIDList.0.1 servers

@sealed
```

6.5.2 ID

Full message type name: **uavcan.node.port.ID**

6.5.2.1 Version 1.0

Size 3 bytes; sealed.

```
# Used to refer either to a Service or to a Subject.
# The chosen tag identifies the kind of the port, then the numerical ID identifies the port within the kind.

@union

SubjectID.1.0 subject_id
ServiceID.1.0 service_id

@sealed
@assert _offset_ == {24}
```

6.5.3 ServiceID

Full message type name: **uavcan.node.port.ServiceID**

6.5.3.1 Version 1.0

Size 2 bytes; sealed.

```
# Service-ID. The ranges are defined by the specification.

uint9 MAX = 511

uint9 value

@sealed
```

6.5.4 ServiceIDList

Full message type name: **uavcan.node.port.ServiceIDList**

6.5.4.1 Version 0.1

Size without delimiter header: 64 bytes; extent 128 bytes.

```
# A list of service identifiers.
# This is a trivial constant-size bitmask with some reserved space in case the range of service-ID is increased
# in a future revision of the protocol.

uint16 CAPACITY = ServiceID.1.0.MAX + 1

bool[CAPACITY] mask
# The index represents the identifier value. True -- present/used. False -- absent/unused.

@extent 1024  # Reserve space in case the range is extended in the future.

@assert CAPACITY % 8 == 0
@assert _offset_ == {CAPACITY}
```

6.5.5 SubjectID

Full message type name: **`uavcan.node.port.SubjectID`**

6.5.5.1 Version 1.0

Size 2 bytes; sealed.

```
# Subject-ID. The ranges are defined by the specification.

uint13 MAX = 8191

uint13 value

@sealed
```

6.5.6 SubjectIDList

Full message type name: **`uavcan.node.port.SubjectIDList`**

6.5.6.1 Version 0.1

Size without delimiter header: 1…1025 bytes; extent 4097 bytes.

```
# A list of subject identifiers.
# The range of subject-ID is large, so using a fixed-size bitmask would make this type difficult to handle on
# resource-constrained systems. To address that, we provide two extra options: a simple variable-length list,
# and a special case that indicates that every subject-ID is in use.

@union

uint16 CAPACITY = SubjectID.1.0.MAX + 1

bool[CAPACITY] mask
# The index represents the identifier value. True -- present/used. False -- absent/unused.

SubjectID.1.0[<256] sparse_list
# A list of identifiers that can be used instead of the mask if most of the identifiers are unused.

uavcan.primitive.Empty.1.0 total
# A special case indicating that all identifiers are in use.

@extent 8 + 2 ** 15  # Reserve space in case the range is extended in the future.
```

6.6 uavcan.pnp

6.6.1 NodeIDAllocationData

Full message type name: **`uavcan.pnp.NodeIDAllocationData`**

6.6.1.1 Version 2.0, fixed subject ID 8165

Size without delimiter header: 18 bytes; extent 48 bytes.

```
# In order to be able to operate in a UAVCAN network, a node shall have a node-ID that is unique within the network.
# Typically, a valid node-ID can be configured manually for each node; however, in certain use cases the manual
# approach is either undesirable or impossible, therefore UAVCAN defines the high-level feature of plug-and-play
# nodes that allows nodes to obtain a node-ID value automatically upon connection to the network. When combined
# with automatic physical layer configuration (such as auto bit rate detection), this feature allows one to implement
# nodes that can join a UAVCAN network without any prior manual configuration whatsoever. Such nodes are referred to
# as "plug-and-play nodes" (or "PnP nodes" for brevity).
#
# The feature is fundamentally non-deterministic and is likely to be unfit for some high-reliability systems;
# the designers need to carefully consider the trade-offs involved before deciding to rely on this feature.
# Normally, static node-ID settings should be preferred.
#
# This feature relies on the concept of "anonymous message transfers", please consult with the UAVCAN transport
# layer specification for details.
#
# The process of plug-and-play node-ID allocation always involves two types of nodes: "allocators", which serve
# allocation requests; and "allocatees", which request PnP node-ID from allocators. A UAVCAN network may implement
# the following configurations of allocators:
#
#   - Zero allocators, in which case plug-and-play node-ID allocation cannot be used, only nodes with statically
#     configured node-ID can communicate.
#
#   - One allocator, in which case the feature of plug-and-play node-ID allocation will become unavailable
#     if the allocator fails. In this configuration, the role of the allocator can be performed even by a very
#     resource-constrained system, e.g., a low-end microcontroller.
#
#   - Three allocators, in which case the allocators will be using a replicated allocation table via a
#     distributed consensus algorithm. In this configuration, the network can tolerate the loss of one
#     allocator and continue to serve allocation requests. This configuration requires the allocators to
#     maintain large data structures for the needs of the distributed consensus algorithm, and may therefore
#     require a slightly more sophisticated computational platform, e.g., a high-end microcontroller.
#
#   - Five allocators, it is the same as the three allocator configuration reviewed above except that the network
#     can tolerate the loss of two allocators and still continue to serve allocation requests.
#
# In order to get a PnP node-ID, an allocatee shall have a globally unique 128-bit integer identifier, known as
# unique-ID (where "globally unique" means that the probability of having two nodes anywhere in the world that share
# the same unique-ID is negligibly low). This is the same value that is used in the field unique_id of the data type
# uavcan.node.GetInfo. All PnP nodes shall support the service uavcan.node.GetInfo, and they shall use the same
# unique-ID value when requesting node-ID allocation and when responding to the GetInfo requests (there may exist
# other usages of the unique-ID value, but they lie outside of the scope of the PnP protocol).
#
# During allocation, the allocatee communicates its unique-ID to the allocator (or allocators in the case of a
# redundant allocator configuration), which then use it to produce an appropriate allocation response. Unique-ID
# values are kept by allocators in the "allocation table" - a data structure that contains the mapping between
# unique-ID and the corresponding node-ID values. The allocation table is a write-only data structure that can
# only expand. When a new allocatee requests a PnP node-ID, its unique-ID is recorded in the allocation table,
# and all subsequent allocation requests from the same allocatee will be served with the same node-ID value.
#
# In configurations with redundant allocators, every allocator maintains a replica of the same allocation table
# (a UAVCAN network cannot contain more than one allocation table, regardless of the number of allocators employed).
# While the allocation table is a write-only data structure that can only grow, it is still possible to wipe the
# table completely (as long as it is removed from all redundant allocators on the network simultaneously),
# forcing the allocators to forget known nodes and perform all future allocations anew.
#
# In the context of the following description, nodes that use a manually-configured node-ID will be referred to as
# "static nodes". It is assumed that allocators are always static nodes themselves since there is no other authority
# on the network that can grant a PnP node-ID, so allocators are unable to request a PnP node-ID for themselves.
# Excepting allocators, it is not recommended to mix PnP and static nodes on the same network; i.e., normally,
# a UAVCAN network should contain either all static nodes, or all PnP nodes (excepting allocators). If this
# recommendation cannot be followed, the following rules of safe co-existence of PnP nodes with static nodes should
# be adopted:
#   - It is safe to connect PnP nodes to the bus at any time.
#   - A static node can be connected to the bus if the allocator (allocators) is (are) already aware of it.
#     I.e., the static node is already listed in the allocation table.
#   - A new static node (i.e., a node that does not meet the above criterion) can be connected to the bus only if
#     no PnP allocation requests are happening at the moment.
#
# Due to the possibility of coexistence of static nodes with PnP nodes, allocators are tasked with monitoring
# the nodes present in the network. If the allocator detects an online node in the network the node-ID of which is
# not found in the allocation table (or the local copy thereof in the case of redundant allocators), the allocator
# shall create a new mock entry where the node-ID matches that of the newly detected node and the unique-ID is set to
# zero (i.e., a 128-bit long sequence of zero bits). This behavior ensures that PnP nodes will never be granted
# node-ID values that are already taken by static nodes. Allocators are allowed to request the true unique-ID of the
# newly detected nodes by issuing requests uavcan.node.GetInfo instead of using mock zero unique-IDs, but this is not
# required for the sake of simplicity and determinism (some nodes may fail to respond to the GetInfo request, e.g.,
# if this service is not supported). Note that in the case of redundant allocators, some of them may be relieved of
# this task due to the intrinsic properties of the distributed consensus algorithm; please refer to the documentation
# for the data type uavcan.pnp.cluster.AppendEntries for more information.
#
# The unique-ID & node-ID pair of each allocator shall be kept in the allocation table as well. It is allowed to replace
# the unique-ID values of allocators with zeros at the discretion of the implementer.
#
# As can be inferred from the above, the process of PnP node-ID allocation involves up to two types of communications:
#
#   - "Allocatee-allocator exchange" - this communication is used when an allocatee requests a PnP node-ID from the
#     allocator (or redundant allocators), and also when the allocator transmits a response back to the allocatee.
#     This communication is invariant to the allocator configuration used, i.e., the allocatees are not aware of
```

```
#     how many allocators are available on the network and how they are configured. In configurations with
#     non-redundant (i.e., single) allocator, this is the only type of PnP allocation exchanges.
#
#   - "Allocator-allocator exchange" - this communication is used by redundant allocators for the maintenance of
#     the replicated allocation table and for other needs of the distributed consensus algorithm. Allocatees are
#     completely isolated and are unaware of these exchanges. This communication is not used with the single-allocator
#     configuration, since there is only one server and the allocation table is not distributed. The data types
#     used for the allocator-allocator exchanges are defined in the namespace uavcan.pnp.cluster.
#
# As has been said earlier, the logic used for communication between allocators (for the needs of the maintenance of
# the distributed allocation table) is completely unrelated to the allocatees. The allocatees are unaware of these
# exchanges, and they are also unaware of the allocator configuration used on the network: single or redundant.
# As such, the documentation you're currently reading does not describe the logic and requirements of the
# allocator-allocator exchanges for redundant configurations; for that, please refer to the documentation for the
# data type uavcan.pnp.cluster.AppendEntries.
#
# Allocatee-allocator exchanges are performed using only this message type uavcan.pnp.NodeIDAllocationData. Allocators
# use it with regular message transfers; allocatees use it with anonymous message transfers. The specification and
# usage info for this data type is provided below.
#
# The general idea of the allocatee-allocator exchanges is that the allocatee communicates to the allocator its
# unique-ID and, if applicable, the preferred node-ID value that it would like to have. The allocatee uses
# anonymous message transfers of this type. The allocator performs the allocation and sends a response using
# the same message type, where the field for unique-ID is populated with the unique-ID of the requesting node
# and the field for node-ID is populated with the allocated node-ID. All exchanges from allocatee to allocator use
# single-frame transfers only (see the specification for more information on the limitations of anonymous messages).
#
# The allocatee-allocator exchange logic differs between allocators and allocatees. For allocators, the logic is
# trivial: upon reception of a request, the allocator performs an allocation and sends a response back. If the
# allocation could not be performed for any reason (e.g., the allocation table is full, or there was a failure),
# no response is sent back (i.e., the request is simply ignored); the recommended strategy for the allocatee is to
# continue sending new allocation requests until a response is granted or a higher-level system (e.g., a maintenance
# technician or some automation) intervenes to rectify the problem (e.g., by purging the allocation table).
# The allocator that could not complete an allocation for any reason is recommended to emit a diagnostic message
# with a human-readable description of the problem. For allocatees, the logic is described below.
#
# This message is used for PnP node-ID allocation on all transports where the maximum transmission unit size is
# sufficiently large. For low-MTU transports such as Classic CAN there is an older version of the definition (v1)
# that takes the low MTU into account (the unique-ID value is replaced with a short hash in order to fit the data
# into one 7-byte-long transfer).
#
# Generally, the randomly chosen values of the request period (Trequest) should be in the range from 0 to 1 seconds.
# Applications that are not concerned about the allocation time are recommended to pick higher values, as it will
# reduce interference with other nodes where faster allocations may be desirable. The random interval shall be chosen
# anew per transmission, whereas the pseudo node-ID value is allowed to stay constant per node.
#
# The source of random data for Trequest shall be likely to yield different values for participating nodes, avoiding
# common sequences. This implies that the time since boot alone is not a sufficiently robust source of randomness,
# as that would be probable to cause nodes powered up at the same time to emit colliding messages repeatedly.
#
# The response timeout is not explicitly defined for this protocol, as the allocatee will request a new allocation
# Trequest units of time later again, unless an allocation has been granted. Since the request and response messages
# are fully idempotent, accidentally repeated messages (e.g., due to benign race conditions that are inherent to this
# protocol) are harmless.
#
# On the allocatee's side the protocol is defined through the following set of rules:
#
#   Rule A. On initialization:
#     1. The allocatee subscribes to this message.
#     2. The allocatee starts the Request Timer with a random interval of Trequest.
#
#   Rule B. On expiration of the Request Timer:
#     1. Request Timer restarts with a random interval of Trequest (chosen anew).
#     2. The allocatee broadcasts an allocation request message, where the fields are populated as follows:
#        node_id   - the preferred node-ID, or the highest valid value if the allocatee doesn't have any preference.
#        unique_id - the 128-bit unique-ID of the allocatee, same value that is reported via uavcan.node.GetInfo.
#
#   Rule C. On an allocation message WHERE (source node-ID is non-anonymous, i.e., regular allocation response)
#                                 AND   (the field unique_id matches the allocatee's unique-ID):
#     1. Request Timer stops.
#     2. The allocatee initializes its node-ID with the received value.
#     3. The allocatee terminates its subscription to allocation messages.
#     4. Exit.
#
# As can be seen, the algorithm assumes that the allocatee will continue to emit requests at random intervals
# until an allocation is granted or the allocatee is disconnected.

uavcan.node.ID.1.0 node_id
# If the message transfer is anonymous (i.e., allocation request), this is the preferred ID.
# If the message transfer is non-anonymous (i.e., allocation response), this is the allocated ID.
#
# If the allocatee does not have any preference, it should request the highest possible node-ID. Keep in mind that
# the two highest node-ID values are reserved for network maintenance tools; requesting those is not prohibited,
# but the allocator is recommended to avoid granting these node-ID, using nearest available lower value instead.
# The allocator will traverse the allocation table starting from the preferred node-ID upward,
# until a free node-ID is found (or the first ID reserved for network maintenance tools is reached).
# If a free node-ID could not be found, the allocator will restart the search from the preferred node-ID
# downward, until a free node-ID is found.

uint8[16] unique_id
# The unique-ID of the allocatee. This is the SAME value that is reported via uavcan.node.GetInfo.
# The value is subjected to the same set of constraints; e.g., it can't be changed while the node is running,
# and the same value should be unlikely to be used by any two different nodes anywhere in the world.
#
# If this is a non-anonymous transfer (i.e., allocation response), allocatees will match this value against their
# own unique-ID, and ignore the message if there is no match. If the IDs match, then the field node_id contains
# the allocated node-ID value for this node.

@assert _offset_.max / 8 == 18
@extent 48 * 8
```

6.6.1.2 Version 1.0, fixed subject ID 8166

Size 7…9 bytes; sealed.

```
# This definition of the allocation message is intended for use with transports where anonymous transfers are limited
# to 7 bytes of payload, such as Classic CAN. The definition is carried over from the original UAVCAN v0 specification
# with some modifications. For transports other than Classic CAN (e.g., CAN FD, serial, etc.) there is a more
# general, more capable definition NodeIDAllocationData v2.0. The PnP protocol itself is described in the documentation
# for the v2 definition. The documentation provided here builds upon the general case, so read that first please.
#
# The full 128-bit unique-ID can't be accommodated in a single-frame anonymous message transfer over Classic CAN, so
# this definition substitutes the full 128-bit ID with a smaller 48-bit hash of it. The 48-bit hash is obtained by
# applying an arbitrary hash function to the unique-ID that outputs at least 48 bit of data. The recommended hash
# function is the standard CRC-64WE where only the lowest 48 bit of the result are used.
#
# Allocators that support allocation messages of different versions should maintain a shared allocation table for all.
# Requests received via the v1 message obviously do not contain the full unique-ID; the allocators are recommended
# to left-zero-pad the small 48-bit hash in order to obtain a "pseudo unique-ID", and use this value in the
# allocation table as a substitute for the real unique-ID. It is recognized that this behavior will have certain
# side effects, such as the same allocatee obtaining different allocated node-ID values depending on which version
# of the message is used, but they are considered tolerable.
#
# Allocatees that may need to operate over Classic CAN along with high-MTU transports may choose to use
# only this constrained method of allocation for consistency and simplification.
#
# In order to save space for the hash, the preferred node-ID is removed from the request. The allocated node-ID
# is provided in the response, however; this is achieved by means of an optional field that is not populated in
# the request but is populated in the response. This implies that the response may be a multi-frame transfer,
# which is acceptable since responses are sent by allocators, which are regular nodes, and therefore they are
# allowed to use regular message transfers rather than being limited to anonymous message transfers as allocatees are.
#
# On the allocatee's side the protocol is defined through the following set of rules:
#
#   Rule A. On initialization:
#     1. The allocatee subscribes to this message.
#     2. The allocatee starts the Request Timer with a random interval of Trequest.
#
#   Rule B. On expiration of the Request Timer (started as per rules A, B, or C):
#     1. Request Timer restarts with a random interval of Trequest (chosen anew).
#     2. The allocatee broadcasts an allocation request message, where the fields are populated as follows:
#        unique_id_hash    - a 48-bit hash of the unique-ID of the allocatee.
#        allocated_node_id - empty (not populated).
#
#   Rule C. On any allocation message, even if other rules also match:
#     1. Request Timer restarts with a random interval of Trequest (chosen anew).
#
#   Rule D. On an allocation message WHERE (source node-ID is non-anonymous, i.e., regular allocation response)
#                                    AND   (the field unique_id_hash matches the allocatee's 48-bit unique-ID hash)
#                                    AND   (the field allocated_node_id is populated):
#     1. Request Timer stops.
#     2. The allocatee initializes its node-ID with the received value.
#     3. The allocatee terminates its subscription to allocation messages.
#     4. Exit.

truncated uint48 unique_id_hash
# An arbitrary 48-bit hash of the unique-ID of the local node.

uavcan.node.ID.1.0[<=1] allocated_node_id
# Shall be empty in request messages.
# Shall be populated in response messages.

@sealed
@assert _offset_.min / 8 == 7        # This is for requests only.
@assert _offset_.max / 8 == 9        # Responses are non-anonymous, so they can be multi-frame.
```

6.7 uavcan.pnp.cluster

6.7.1 AppendEntries

Full service type name: **`uavcan.pnp.cluster.AppendEntries`**

6.7.1.1 Version 1.0, fixed service ID 390

- Request: Size without delimiter header: 13…35 bytes; extent 96 bytes.
- Response: Size without delimiter header: 5 bytes; extent 48 bytes.

```
# This type is a part of the Raft consensus algorithm. The Raft consensus is used for the maintenance of the
# distributed allocation table between redundant allocators. The following description is focused on the exchanges
# between redundant PnP node-ID allocators. It does not apply to the case of non-redundant allocators, because
# in that case the allocation table is stored locally and the process of node-ID allocation is trivial and fully local.
# Exchanges between allocatees and allocators are documented in the appropriate message type definition.
#
# The algorithm used for replication of the allocation table across redundant allocators is a fairly direct
# implementation of the Raft consensus algorithm, as published in the paper
# "In Search of an Understandable Consensus Algorithm (Extended Version)" by Diego Ongaro and John Ousterhout.
# The following text assumes that the reader is familiar with the paper.
#
# The Raft log contains entries of type Entry (in the same namespace), where every entry contains the Raft term
# number, the unique-ID, and the corresponding node-ID value (or zeros if it could not be requested from a static
# node). Therefore, the Raft log is the allocation table itself.
#
# Since the maximum number of entries in the allocation table is limited by the range of node-ID values, the log
# capacity is bounded. Therefore, the snapshot transfer and log compaction functions are not required,
# so they are not used in this implementation of the Raft algorithm.
```

```
#
# When an allocator becomes the leader of the Raft cluster, it checks if the Raft log contains an entry for its own
# node-ID, and if it doesn't, the leader adds its own allocation entry to the log (the unique-ID can be replaced with
# zeros at the discretion of the implementer). This behavior guarantees that the Raft log always contains at least
# one entry, therefore it is not necessary to support negative log indices, as proposed by the Raft paper.
#
# Since the log is write-only and limited in growth, all allocations are permanent. This restriction is acceptable,
# since UAVCAN is a vehicle bus, and configuration of vehicle's components is not expected to change frequently.
# Old allocations can be removed in order to free node-IDs for new allocations by clearing the Raft log on all
# allocators; such clearing shall be performed simultaneously while the network is down, otherwise the Raft cluster
# will automatically attempt to restore the lost state on the allocators where the table was cleared.
#
# The allocators need to be aware of each other's node-ID in order to form a cluster. In order to learn each other's
# node-ID values, the allocators broadcast messages of type Discovery (in the same namespace) until the cluster is
# fully discovered and all allocators know of each other's node-ID. This extension to the Raft algorithm makes the
# cluster almost configuration-free - the only parameter that shall be configured on all allocators of the cluster
# is the number of nodes in the cluster (everything else will be auto-detected).
#
# Runtime cluster membership changes are not supported, since they are not needed for a vehicle bus.
#
# As has been explained in the general description of the PnP node-ID allocation feature, allocators shall watch for
# unknown static nodes appearing on the bus. In the case of a non-redundant allocator, the task is trivial, since the
# allocation table can be updated locally. In the case of a Raft cluster, however, the network monitoring task shall
# be performed by the leader only, since other cluster members cannot commit to the shared allocation table (i.e.,
# the Raft log) anyway. Redundant allocators should not attempt to obtain the true unique-ID of the newly detected
# static nodes (use zeros instead), because the allocation table is write-only: if the unique-ID of a static node
# ever changes (e.g., a replacement unit is installed, or network configuration is changed manually), the change
# will be impossible to reflect in the allocation table.
#
# Only the current Raft leader can process allocation requests and engage in communication with allocatees.
# An allocator is allowed to send allocation responses only if both conditions are met:
#
#   - The allocator is currently the Raft leader.
#   - Its replica of the Raft log does not contain uncommitted entries (i.e. the last allocation request has been
#     completed successfully).
#
# All cluster maintenance traffic should normally use either the lowest or the next-to-lowest transfer priority level.

uint8 DEFAULT_MIN_ELECTION_TIMEOUT = 2      # [second]
uint8 DEFAULT_MAX_ELECTION_TIMEOUT = 4      # [second]
# Given the minimum election timeout and the cluster size,
# the maximum recommended request interval can be derived as follows:
#
#   max recommended request interval = (min election timeout) / 2 requests / (cluster size - 1)
#
# The equation assumes that the Leader requests one Follower at a time, so that there's at most one pending call
# at any moment. Such behavior is optimal as it creates a uniform bus load, although it is implementation-specific.
# Obviously, the request interval can be lower than that if needed, but higher values are not recommended as they may
# cause Followers to initiate premature elections in case of frame losses or delays.
#
# The timeout value is randomized in the range (MIN, MAX], according to the Raft paper. The randomization granularity
# should be at least one millisecond or higher.

uint32 term
uint32 prev_log_term
uint16 prev_log_index
uint16 leader_commit
# Refer to the Raft paper for explanation.

Entry.1.0[<=1] entries
# Worst case replication time per Follower can be computed as:
#
#   worst replication time = (node-ID capacity) * (2 trips of next_index) * (request interval per Follower)
#
# E.g., given the request interval of 0.5 seconds, the worst case replication time for CAN bus is:
#
#   128 nodes * 2 trips * 0.5 seconds = 128 seconds.
#
# This is the amount of time it will take for a new Follower to reconstruct a full replica of the distributed log.

@assert _offset_ % 8 == {0}
@extent 96 * 8

---

uint32 term
bool success
# Refer to the Raft paper for explanation.

@extent 48 * 8
```

6.7.2 Discovery

Full message type name: **`uavcan.pnp.cluster.Discovery`**

6.7.2.1 Version 1.0, fixed subject ID 8164

Size without delimiter header: 2 . . . 12 bytes; extent 96 bytes.

```
# This message is used by redundant allocators to find each other's node-ID.
# Please refer to the type AppendEntries for details.
#
# An allocator should stop publishing this message as soon as it has discovered all other allocators in the cluster.
#
# An exception applies: when an allocator receives a Discovery message where the list of known nodes is incomplete
# (i.e. len(known_nodes) < configured_cluster_size), it shall publish a Discovery message once. This condition
# # allows other allocators to quickly re-discover the cluster after a restart.
```

```

uint8 BROADCASTING_PERIOD = 1      # [second]
# This message should be broadcasted by the allocator at this interval until all other allocators are discovered.

uint3 MAX_CLUSTER_SIZE = 5
# The redundant allocator cluster cannot contain more than 5 allocators.

uint3 configured_cluster_size
# The number of allocators in the cluster as configured on the sender.
# This value shall be the same across all allocators.

void5

uavcan.node.ID.1.0[<=5] known_nodes
# Node-IDs of the allocators that are known to the publishing allocator, including the publishing allocator itself.

@assert _offset_ % 8 == {0}
@extent 96 * 8
```

6.7.3 RequestVote

Full service type name: **`uavcan.pnp.cluster.RequestVote`**

6.7.3.1 Version 1.0, fixed service ID 391

- Request: Size without delimiter header: 10 bytes; extent 48 bytes.
- Response: Size without delimiter header: 5 bytes; extent 48 bytes.

```
# This type is a part of the Raft consensus algorithm. Please refer to the type AppendEntries for details.

uint32 term
uint32 last_log_term
uint16 last_log_index
# Refer to the Raft paper for explanation.

@extent 48 * 8

---

uint32 term
bool vote_granted
# Refer to the Raft paper for explanation.

@extent 48 * 8
```

6.7.4 Entry

Full message type name: **`uavcan.pnp.cluster.Entry`**

6.7.4.1 Version 1.0

Size 22 bytes; sealed.

```
# One PnP node-ID allocation entry.
# This type is a part of the Raft consensus algorithm. Please refer to the type AppendEntries for details.

uint32 term                     # Refer to the Raft paper for explanation.

uint8[16] unique_id             # Unique-ID of this allocation; zero if unknown.

uavcan.node.ID.1.0 node_id      # Node-ID of this allocation.

@sealed
@assert _offset_ % 8 == {0}
```

6.8 uavcan.register

6.8.1 Access

Full service type name: **uavcan.register.Access**

6.8.1.1 Version 1.0, fixed service ID 384

- Request: Size 2...515 bytes; sealed.
- Response: Size 9...267 bytes; sealed.

```
# Registers are strongly-typed named values used to store the configuration parameters of a node.
# This service is used to write and read a register.
#
#
#   READ/WRITE BEHAVIORS
#
# The write operation is performed first, unless skipped by sending an empty value in the request.
# The server may attempt to convert the type of the supplied value to the correct type if there is a type mismatch
# (e.g. uint8 may be converted to uint16); however, servers are not required to perform implicit type conversion,
# and the rules of such conversion are not explicitly specified, so this behavior should not be relied upon.
#
# On the next step the register will be read regardless of the outcome of the write operation. As such, if the write
# operation could not be performed (e.g. due to a type mismatch or any other issue), the register will retain its old
# value. By evaluating the response the caller can determine whether the register was written successfully.
#
# The write-read sequence is not guaranteed to be atomic, meaning that external influences may cause the register to
# change its value between the write and the subsequent read operation. The caller is responsible for handling that
# case properly.
#
# The timestamp provided in the response corresponds to the time when the register was read. The timestamp may
# be empty if the server does not support timestamping or its clock is not (yet) synchronized with the network.
#
# If only read is desired, but not write, the caller shall provide a value of type 'empty'. That will signal the server
# that the write operation shall be skipped, and it will proceed to read the register immediately.
#
# If the requested register does not exist, the write operation will have no effect and the returned value will be
# empty. Existing registers should not return 'empty' when read since that would make them indistinguishable from
# nonexistent registers.
#
#
#   REGISTER DEFINITION REQUIREMENTS
#
# Registers shall never change their type or flags as long as the server is running. Meaning that:
#   - Mutability and persistence flags cannot change their states.
#   - Read operations shall always return values of the same type and same dimensionality.
#     The dimensionality requirement does not apply to inherently variable-length values such as strings and
#     unstructured chunks.
#
# Register name should contain only:
#   - Lowercase ASCII alphanumeric characters (a-z, 0-9)
#   - Full stop (.)
#   - Low line (underscore) (_)
# With the following limitations/recommendations:
#   - The name shall not begin with a decimal digit (0-9).
#   - The name shall neither begin nor end with a full stop.
#   - A low line shall not be followed by a non-alphanumeric character.
#   - The name should contain at least one full stop character.
# Other patterns and ASCII characters are reserved for special function registers (introduced below).
#
#
#   ENVIRONMENT VARIABLES
#
# This section applies only to software nodes executed in a high-level operating system that supports environment
# variables or an equivalent mechanism.
#
# When a software node is launched, it is usually necessary to provide some of its configuration information early,
# particularly that which is related to UAVCAN networking, before the node is started. Environment variables offer
# a convenient way of addressing this. Software nodes that support the register interface should evaluate the
# available environment variables during initialization and update their registers (whether they are stored in
# a persistent storage or in memory) accordingly. This should be completed before the first register read access.
#
# A register name is mapped to an environment variable name as follows:
#   - the name is upper-cased;
#   - full stop characters are replaced with double low line characters.
# For example: 'motor.inductance_dq' is mapped to 'MOTOR__INDUCTANCE_DQ'.
#
# Register values are represented in environment variables as follows:
#   - string:                         utf-8 or platform-specific
#   - unstructured:                   as-is
#   - bit, integer*, natural*, real*: space-separated decimals
#
# If an environment variable matches the name of an existing register but its value cannot be converted to the
# register's type, an error should be raised.
#
# If an environment variable does not match the name of any register, it may be ignored. However, if the implementation
# can reliably deduce the type and purpose of the register, it may create one automatically. This provision is to
# support applications where the register schema may be altered by configuration.
#
#
#   SPECIAL FUNCTION REGISTERS
#
# The following optional special function register names are defined:
#   - suffix '<' is used to define an immutable persistent value that contains the maximum value
#     of the respective register.
#   - suffix '>' is like above, used to define the minimum value of the respective register.
#   - suffix '=' is like above, used to define the default value of the respective register.
```

```
#   - prefix '*' is reserved for raw memory access (to be defined later).
# Examples:
#   - register name "system.parameter"
#   - maximum value is contained in the register named "system.parameter<" (optional)
#   - minimum value is contained in the register named "system.parameter>" (optional)
#   - default value is contained in the register named "system.parameter=" (optional)
#
# The type and dimensionality of the special function registers containing the minimum, maximum, and the default
# value of a register shall be the same as those of the register they relate to.
#
# If a written value exceeds the minimum/maximum specified by the respective special function registers,
# the server may either adjust the value automatically, or to retain the old value, depending on which behavior
# suits the objectives of the application better.
# The values of registers containing non-scalar numerical entities should be compared elementwise.
#
#
#   STANDARD REGISTERS
#
# The following table specifies the register name patterns that are reserved by the specification for
# common functions. These conventions are not mandatory to follow, but implementers are recommended to adhere because
# they enable enhanced introspection capabilities and simplify device configuration and diagnostics.
#
#   REGISTER NAME PATTERN                               TYPE            FLAGS                   RECOMMENDED DEFAULT
# =====================================================================================================================
#
#   uavcan.node.id                                      natural16[1]    mutable, persistent     65535 (unset/PnP)
#
# Contains the node-ID of the local node. Values above the maximum valid node-ID for the current transport
# indicate that the node-ID is not set; if plug-and-play is supported, it will be used by the node to obtain an
# automatic node-ID. Invalid values other than 65535 should be avoided for consistency.
#
# ---------------------------------------------------------------------------------------------------------------------
#
#   uavcan.node.description                             string          mutable, persistent     (empty)
#
# User/integrator-defined, human-readable description of this specific node.
# This is intended for use by a system integrator and should not be set by the manufacturer of a component.
# For example: on a quad-rotor drone this might read "motor 2" for one of the ESC nodes.
#
# ---------------------------------------------------------------------------------------------------------------------
#
#   uavcan.pub.PORT_NAME.id                             natural16[1]    mutable, persistent     65535 (unset, invalid)
#   uavcan.sub.PORT_NAME.id                             ditto           ditto                   ditto
#   uavcan.cln.PORT_NAME.id                             ditto           ditto                   ditto
#   uavcan.srv.PORT_NAME.id                             ditto           ditto                   ditto
#
# Publication/subscription/client/server port-ID, respectively. These registers are configured by the system integrator
# or an autoconfiguration authority when the node is first connected to a network.
#
# The "PORT_NAME" defines the human-friendly name of the port, which is related to the corresponding function
# or a network service supported by the node. The name shall match the following POSIX ERE expression:
#
#   [a-zA-Z_][a-zA-Z0-9_]*
#
# The names are defined by the vendor of the node. The user/integrator is expected to understand their meaning and
# relation to the functional capabilities of the node by reading the technical documentation provided by the vendor.
#
# A port whose port-ID register is unset (invalid value) remains inactive (unused); the corresponding function may
# be disabled. For example, a register named "uavcan.pub.measurement.id" defines the subject-ID of a measurement
# published by this node; if the register contains an invalid value (above the maximum valid subject-ID),
# said measurement is not published.
#
# The same name is used in other similar registers defined below. Network introspection and autoconfiguration tools
# will expect to find a register of this form for every configurable port supported by the node.
#
# ---------------------------------------------------------------------------------------------------------------------
#
#   uavcan.pub.PORT_NAME.type                           string          immutable, persistent   N/A
#   uavcan.sub.PORT_NAME.type                           ditto           ditto                   ditto
#   uavcan.cln.PORT_NAME.type                           ditto           ditto                   ditto
#   uavcan.srv.PORT_NAME.type                           ditto           ditto                   ditto
#
# Publication/subscription/client/server full data type name and dot-separated version numbers, respectively.
# These registers are set by the vendor once and typically they are to remain unchanged (hence "immutable").
# The "PORT_NAME" defines the human-friendly name of the port as specified above.
# For example, a register named "uavcan.pub.measurement.type" may contain "uavcan.si.sample.angle.Quaternion.1.0".
#
# ---------------------------------------------------------------------------------------------------------------------
#
#   uavcan.diagnostic.*
#
# Prefix reserved for future use.
#
# ---------------------------------------------------------------------------------------------------------------------
#
#   uavcan.can.bitrate                                  natural32[2]    implementation-defined  implementation-defined
#   uavcan.can.iface                                    string          mutable, persistent     implementation-defined
#
# These registers are only relevant for nodes that support UAVCAN/CAN.
#
# uavcan.can.bitrate defines the CAN bus bit rate: the first value is the arbitration bit rate, the second is the
# data phase bit rate. Nodes that support only Classic CAN should ignore the second value. Nodes that support CAN FD
# should initialize in the Classic CAN mode (MTU 8 bytes, BRS flag not set) if the values are equal. If CAN bitrate
# is not configurable or is always auto-detected, this register may be omitted or made immutable; otherwise it should
# be mutable and persistent.
#
# uavcan.can.iface is only relevant for software nodes or nodes that are capable of using different CAN interfaces.
# The value is a space-separated list of CAN interface names to use. The name format is implementation-defined
# (for example, "can0").
#
# ---------------------------------------------------------------------------------------------------------------------
```

```
#
#   uavcan.udp.*
#
# Prefix reserved for future use.
#
# --------------------------------------------------------------------------------------------------------------------#
#
#   uavcan.serial.*
#
# Prefix reserved for future use.
#
# --------------------------------------------------------------------------------------------------------------------

Name.1.0 name
# The name of the accessed register. Shall not be empty.
# Use the List service to obtain the list of registers on the node.

Value.1.0 value
# Value to be written. Empty if no write is required.

@sealed

---

uavcan.time.SynchronizedTimestamp.1.0 timestamp
# The moment of time when the register was read (not written).
# Zero if the server does not support timestamping.

bool mutable
# Mutable means that the register can be written using this service.
# Immutable registers cannot be written, but that doesn't imply that their values are constant (unchanging).

bool persistent
# Persistence means that the register retains its value permanently across power cycles or any other changes
# in the state of the server, until it is explicitly overwritten (either via UAVCAN, any other interface,
# or by the device itself).
#
# The server is recommended to manage persistence automatically by committing changed register values to a
# non-volatile storage automatically as necessary. If automatic persistence management is not implemented, it
# can be controlled manually via the standard service uavcan.node.ExecuteCommand. The same service can be used
# to return the configuration to a factory-default state. Please refer to its definition for more information.
#
# Consider the following examples:
#   - Configuration parameters are usually both mutable and persistent.
#   - Diagnostic values are usually immutable and non-persisient.
#   - Registers that trigger an activity when written are typically mutable but non-persisient.
#   - Registers that contain factory-programmed values such as calibration coefficients that can't
#     be changed are typically immutable but persistent.

void6

Value.1.0 value
# The value of the register when it was read (beware of race conditions).
# Registers never change their type and dimensionality while the node is running.
# Empty value means that the register does not exist (in this case the flags should be cleared/ignored).
# By comparing the returned value against the write request the caller can determine whether the register
# was written successfully, unless write was not requested.
# An empty value shall never be returned for an existing register.

@sealed
```

6.8.2 List

Full service type name: **`uavcan.register.List`**

6.8.2.1 Version 1.0, fixed service ID 385

- Request: Size 2 bytes; sealed.
- Response: Size 1…256 bytes; sealed.

```
# This service allows the caller to discover the names of all registers available on the server
# by iterating the index field from zero until an empty name is returned.
#
# The ordering of the registers shall remain constant while the server is running.
# The ordering is not guaranteed to remain unchanged when the server node is restarted.

uint16 index

@sealed

---

Name.1.0 name
# Empty name in response means that the index is out of bounds, i.e., discovery is finished.

@sealed
```

6.8.3 Name

Full message type name: **`uavcan.register.Name`**

6.8.3.1 Version 1.0

Size 1…256 bytes; sealed.

```
# An UTF8-encoded register name.

uint8[<256] name

@sealed
```

6.8.4 Value

Full message type name: **uavcan.register.Value**

6.8.4.1 Version 1.0

Size 1...259 bytes; sealed.

```
# This union contains all possible value types supported by the register protocol.
# Numeric types can be either scalars or arrays; the former is a special case of the latter.

@union

uavcan.primitive.Empty.1.0        empty         # Tag 0     Used to represent an undefined value
uavcan.primitive.String.1.0       string        # Tag 1     UTF-8 encoded text
uavcan.primitive.Unstructured.1.0 unstructured  # Tag 2     Raw unstructured binary image
uavcan.primitive.array.Bit.1.0    bit           # Tag 3     Bit array

uavcan.primitive.array.Integer64.1.0 integer64  # Tag 4
uavcan.primitive.array.Integer32.1.0 integer32  # Tag 5
uavcan.primitive.array.Integer16.1.0 integer16  # Tag 6
uavcan.primitive.array.Integer8.1.0  integer8   # Tag 7

uavcan.primitive.array.Natural64.1.0 natural64  # Tag 8
uavcan.primitive.array.Natural32.1.0 natural32  # Tag 9
uavcan.primitive.array.Natural16.1.0 natural16  # Tag 10
uavcan.primitive.array.Natural8.1.0  natural8   # Tag 11

uavcan.primitive.array.Real64.1.0 real64        # Tag 12    Exactly representable integers: [-2**53,    +2**53]
uavcan.primitive.array.Real32.1.0 real32        # Tag 13    Exactly representable integers: [-16777216, +16777216]
uavcan.primitive.array.Real16.1.0 real16        # Tag 14    Exactly representable integers: [-2048,     +2048]

@sealed
@assert _offset_.min == 8                       # Empty and the tag
@assert _offset_.max == 258 * 8 + 8             # 258 bytes per field max and the tag
```

6.9 uavcan.time

6.9.1 GetSynchronizationMasterInfo

Full service type name: **uavcan.time.GetSynchronizationMasterInfo**

6.9.1.1 *Version 0.1, fixed service ID 510*

- Request: Size without delimiter header: 0 bytes; extent 48 bytes.
- Response: Size without delimiter header: 7 bytes; extent 192 bytes.

```
# Every node that acts as a time synchronization master, or is capable of acting as such,
# should support this service.
# Its objective is to provide information about which time system is currently used in the network.
#
# Once a time system is chosen, it cannot be changed as long as at least one node on the network is running.
# In other words, the time system cannot be changed while the network is operating.
# An implication of this is that if there are redundant time synchronization masters, they all shall
# use the same time system always.

@extent 48 * 8

---

float32 error_variance      # [second^2]
# Error variance, in second^2, of the time value reported by this master.
# This value is allowed to change freely while the master is running.
# For example, if the master's own clock is synchronized with a GNSS, the error variance is expected to increase
# as signal reception deteriorates. If the signal is lost, this value is expected to grow steadily, the rate of
# growth would be dependent on the quality of the time keeping hardware available locally (bad hardware yields
# faster growth). Once the signal is regained, this value would drop back to nominal.

TimeSystem.0.1 time_system
# Time system currently in use by the master.
# Cannot be changed while the network is operating.

TAIInfo.0.1 tai_info
# Actual information about TAI provided by this master, if supported.
# The fields in this data type are optional.

@extent 192 * 8
```

6.9.2 Synchronization

Full message type name: **uavcan.time.Synchronization**

6.9.2.1 *Version 1.0, fixed subject ID 7168*

Size 7 bytes; sealed.

```
# Network-wide time synchronization message.
# Any node that publishes timestamped data should use this time reference.
#
# The time synchronization algorithm is based on the work
# "Implementing a Distributed High-Resolution Real-Time Clock using the CAN-Bus" by M. Gergeleit and H. Streich.
# The general idea of the algorithm is to have one or more nodes that periodically publish a message of this type
# containing the exact timestamp of the PREVIOUS transmission of this message.
# A node that publishes this message periodically is referred to as a "time synchronization master",
# whereas nodes that synchronize their clocks with the master are referred to as "time synchronization slaves".
#
# Once a time base is chosen, it cannot be changed as long as at least one node on the network is running.
# In other words, the time base cannot be changed while the network is operating.
# An implication of this is that if there are redundant time synchronization masters, they all shall
# use the same time base.
#
# The resolution is dependent on the transport and its physical layer, but generally it can be assumed
# to be close to one bit time but not better than one microsecond (e.g., for a 500 kbps CAN bus,
# the resolution is two microseconds). The maximum accuracy is achievable only if the transport layer
# supports precise timestamping in hardware; otherwise, the accuracy may be degraded.
#
# This algorithm allows the slaves to precisely estimate the difference (i.e., phase error) between their
# local time and the master clock they are synchronized with. The algorithm for clock rate adjustment
# is entirely implementation-defined (for example, a simple phase-locked loop or a PID rate controller can be used).
#
# The network can accommodate more than one time synchronization master for purposes of increased reliability:
# if one master fails, the others will continue to provide the network with accurate and consistent time information.
# The risk of undesirable transients while the masters are swapped is mitigated by the requirement that all masters
# use the same time base at all times, as described above.
#
# The master with the lowest node-ID is called the "dominant master". The current dominant master ceases to be one
# if its last synchronization message was published more than 3X seconds ago, where X is the time interval
# between the last and the previous messages published by it. In this case, the master with the next-higher node-ID
# will take over as the new dominant master. The current dominant master will be displaced immediately as soon as
# the first message from a new master with a lower node-ID is seen on the bus.
#
# In the presence of multiple masters, they all publish their time synchronization messages concurrently at all times.
# The slaves shall listen to the master with the lowest node-ID and ignore the messages published by masters with
# higher node-ID values.
#
# Currently, there is a work underway to develop and validate a highly robust fault-operational time synchronization
# algorithm where the slaves select the median time base among all available masters rather than using only the
# one with the lowest node-ID value. Follow the work at https://forum.uavcan.org. When complete, this algorithm
# will be added in a backward-compatible way as an option for high-reliability systems.
```

```
#
# For networks with redundant transports, the timestamp value published on different interfaces is likely to be
# different, since different transports are generally not expected to be synchronized. Synchronization slaves
# are allowed to use any of the available redundant interfaces for synchronization at their discretion.
#
# The following pseudocode shows the logic of a time synchronization master. This example assumes that the master
# does not need to synchronize its own clock with other masters on the bus, which is the case if the current master
# is the only master, or if all masters synchronize their clocks with a robust external source, e.g., a GNSS system.
# If several masters need to synchronize their clock through the bus, their logic will be extended with the
# slave-side behavior explained later.
#
#       // State variables
#       transfer_id := 0;
#       previous_tx_timestamp_per_iface[NUM_IFACES] := {0};
#
#       // This function publishes a message with a specified transfer-ID using only one transport interface.
#       function publishMessage(transfer_id, iface_index, msg);
#
#       // This callback is invoked when the transport layer completes the transmission of a time sync message.
#       // Observe that the time sync message is always a single-frame message by virtue of its small size.
#       // The tx_timestamp argument contains the exact timestamp when the transport frame was delivered to the bus.
#       function messageTxTimestampCallback(iface_index, tx_timestamp)
#       {
#           previous_tx_timestamp_per_iface[iface_index] := tx_timestamp;
#       }
#
#       // Publishes messages of type uavcan.time.Synchronization to each available transport interface.
#       // It is assumed that this function is invoked with a fixed frequency not lower than 1 hertz.
#       function publishTimeSync()
#       {
#           for (i := 0; i < NUM_IFACES; i++)
#           {
#               message := uavcan.time.Synchronization();
#               message.previous_transmission_timestamp_usec := previous_tx_timestamp_per_iface[i];
#               previous_tx_timestamp_per_iface[i] := 0;
#               publishMessage(transfer_id, i, message);
#           }
#           transfer_id++; // Overflow shall be handled correctly
#       }
#
# (end of the master-side logic pseudocode)
# The following pseudocode describes the logic of a time synchronization slave.
#
#       // State variables:
#       previous_rx_real_timestamp := 0;            // This clock is being synchronized
#       previous_rx_monotonic_timestamp := 0;       // Monotonic time -- doesn't leap or change rate
#       previous_transfer_id := 0;
#       state := STATE_UPDATE;                      // Variants: STATE_UPDATE, STATE_ADJUST
#       master_node_id := -1;                       // Invalid value
#       iface_index := -1;                          // Invalid value
#
#       // This function adjusts the local clock by the specified amount
#       function adjustLocalTime(phase_error);
#
#       function adjust(message)
#       {
#           // Clock adjustment will be performed every second message
#           local_time_phase_error := previous_rx_real_timestamp - msg.previous_transmission_timestamp_microsecond;
#           adjustLocalTime(local_time_phase_error);
#           state := STATE_UPDATE;
#       }
#
#       function update(message)
#       {
#           // A message is assumed to have two timestamps:
#           //   Real      - sampled from the clock that is being synchronized
#           //   Monotonic - clock that never leaps and never changes rate
#           previous_rx_real_timestamp := message.rx_real_timestamp;
#           previous_rx_monotonic_timestamp := message.rx_monotonic_timestamp;
#           master_node_id := message.source_node_id;
#           iface_index := message.iface_index;
#           previous_transfer_id := message.transfer_id;
#           state := STATE_ADJUST;
#       }
#
#       // Accepts the message of type uavcan.time.Synchronization
#       function handleReceivedTimeSyncMessage(message)
#       {
#           time_since_previous_msg := message.monotonic_timestamp - previous_rx_monotonic_timestamp;
#
#           needs_init := (master_node_id < 0) or (iface_index < 0);
#           switch_master := message.source_node_id < master_node_id;
#
#           // The value publisher_timeout is computed as described in the specification (3x interval)
#           publisher_timed_out := time_since_previous_msg > publisher_timeout;
#
#           if (needs_init or switch_master or publisher_timed_out)
#           {
#               update(message);
#           }
#           else if ((message.iface_index == iface_index) and (message.source_node_id == master_node_id))
#           {
#               // Revert the state to STATE_UPDATE if needed
#               if (state == STATE_ADJUST)
#               {
#                   msg_invalid := message.previous_transmission_timestamp_microsecond == 0;
#                   // Overflow shall be handled correctly
#                   wrong_tid := message.transfer_id != (previous_transfer_id + 1);
#                   wrong_timing := time_since_previous_msg > MAX_PUBLICATION_PERIOD;
#                   if (msg_invalid or wrong_tid or wrong_timing)
#                   {
```

```
#                       state := STATE_UPDATE;
#                   }
#               }
#               // Handle the current state
#               if (state == STATE_ADJUST)
#               {
#                   adjust(message);
#               }
#               else
#               {
#                   update(message);
#               }
#           }   // else ignore
#       }
#
# (end of the slave-side logic pseudocode)

uint8 MAX_PUBLICATION_PERIOD = 1                 # [second]
# Publication period limits.
# A master should not change its publication period while running.

uint8 PUBLISHER_TIMEOUT_PERIOD_MULTIPLIER = 3
# Synchronization slaves should normally switch to a new master if the current master was silent
# for thrice the interval between the reception of the last two messages published by it.
# For example, imagine that the last message was received at the time X, and the previous message
# was received at the time (X - 0.5 seconds); the period is 0.5 seconds, and therefore the publisher
# timeout is (0.5 seconds * 3) = 1.5 seconds. If there was no message from the current master in
# this amount of time, all slaves will synchronize with another master with the next-higher node-ID.

truncated uint56 previous_transmission_timestamp_microsecond
# The time when the PREVIOUS message was transmitted from the current publisher, in microseconds.
# If this message is published for the first time, or if the previous transmission was more than
# one second ago, this field shall be zero.

@sealed
@assert _offset_ % 8 == {0}
@assert _offset_.max <= 56    # Shall fit into one CAN 2.0 frame (least capable transport, smallest MTU)
```

6.9.3 SynchronizedTimestamp

Full message type name: **`uavcan.time.SynchronizedTimestamp`**

6.9.3.1 Version 1.0

Size 7 bytes; sealed.

```
# Nested data type used for representing a network-wide synchronized timestamp with microsecond resolution.
# This data type is highly recommended for use both in standard and vendor-specific messages alike.

uint56 UNKNOWN = 0  # Zero means that the time is not known.

truncated uint56 microsecond
# The number of microseconds that have passed since some arbitrary moment in the past.
# The moment of origin (i.e., the time base) is defined per-application. The current time base in use
# can be requested from the time synchronization master, see the corresponding service definition.
#
# This value is to never overflow. The value is 56-bit wide because:
#
#   - 2^56 microseconds is about 2285 years, which is plenty. A 64-bit microsecond counter would be
#     unnecessarily wide and its overflow interval of 585 thousand years induces a mild existential crisis.
#
#   - Classic-CAN (not FD) transports carry up to 7 bytes of payload per frame.
#     Time sync messages shall use single-frame transfers, which means that the value can't be wider than 56 bits.

@sealed
```

6.9.4 TAIInfo

Full message type name: **`uavcan.time.TAIInfo`**

6.9.4.1 Version 0.1

Size 2 bytes; sealed.

```
# This data types defines constants and runtime values pertaining to the International Atomic Time, also known as TAI.
# See https://en.wikipedia.org/wiki/International_Atomic_Time.
#
# The relationship between the three major time systems -- TAI, GPS, and UTC -- is as follows:
#
#   TAI = GPS + 19 seconds
#   TAI = UTC + LS + 10 seconds
#
# Where "LS" is the current number of leap seconds: https://en.wikipedia.org/wiki/Leap_second.
#
# UAVCAN applications should only rely on TAI whenever a global time system is needed.
# GPS time is strongly discouraged for reasons of consistency across different positioning systems and applications.

uint8 DIFFERENCE_TAI_MINUS_GPS = 19    # [second]
# The fixed difference, in seconds, between TAI and GPS time. Does not change ever.
# Systems that use GPS time as a reference should convert that to TAI by adding this difference.

uint10 DIFFERENCE_TAI_MINUS_UTC_UNKNOWN = 0
uint10 difference_tai_minus_utc
# The current difference between TAI and UTC, if known. If unknown, set to zero.
```

```
#
# This value may change states between known and unknown while the master is running,
# depending on its ability to obtain robust values from external sources.
#
# This value may change twice a year, possibly while the system is running; https://en.wikipedia.org/wiki/Leap_second.
# Since the rotation of Earth is decelerating, this value may only be positive. Do not use outside Earth.
#
# For reference, here is the full list of recorded TAI-UTC difference values, valid at the time of writing:
#
#      Date     | TAI-UTC difference [second]
#     ----------|-----------------------------
#      Jan 1972 | 10
#      Jul 1972 | 11
#      Jan 1973 | 12
#      Jan 1974 | 13
#      Jan 1975 | 14
#      Jan 1976 | 15
#      Jan 1977 | 16
#      Jan 1978 | 17
#      Jan 1979 | 18
#      Jan 1980 | 19
#      Jul 1981 | 20
#      Jul 1982 | 21
#      Jul 1983 | 22
#      Jul 1985 | 23
#      Jan 1988 | 24
#      Jan 1990 | 25
#      Jan 1991 | 26
#      Jul 1992 | 27
#      Jul 1993 | 28
#      Jul 1994 | 29
#      Jan 1996 | 30
#      Jul 1997 | 31
#      Jan 1999 | 32
#      Jan 2006 | 33
#      Jan 2009 | 34
#      Jul 2012 | 35
#      Jul 2015 | 36
#      Jan 2017 | 37
#
# As of 2020, the future of the leap second and the relation between UTC and TAI remains uncertain.

@sealed
```

6.9.5 TimeSystem

Full message type name: **uavcan.time.TimeSystem**

6.9.5.1 Version 0.1

Size 1 bytes; sealed.

```
# Time system enumeration.
# The time system shall be the same for all masters in the network.
# It cannot be changed while the network is running.

truncated uint4 value

uint4 MONOTONIC_SINCE_BOOT = 0
# Monotonic time since boot.
# Monotonic time is a time reference that doesn't change rate or make leaps.

uint4 TAI = 1
# International Atomic Time; https://en.wikipedia.org/wiki/International_Atomic_Time.
# The timestamp value contains the number of microseconds elapsed since 1970-01-01T00:00:00Z TAI.
# TAI is always a fixed integer number of seconds ahead of GPS time.
# Systems that use GPS time as a reference should convert that to TAI by adding the fixed difference.
# GPS time is not supported for reasons of consistency across different positioning systems and applications.

uint4 APPLICATION_SPECIFIC = 15
# Application-specific time system of unknown properties.

@sealed
```

6.10 uavcan.metatransport.can

6.10.1 ArbitrationID

Full message type name: **uavcan.metatransport.can.ArbitrationID**

6.10.1.1 Version 0.1

Size 5 bytes; sealed.

```
# CAN frame arbitration field.

@union

BaseArbitrationID.0.1     base
ExtendedArbitrationID.0.1 extended

@sealed
@assert _offset_ == {40}
```

6.10.2 BaseArbitrationID

Full message type name: **uavcan.metatransport.can.BaseArbitrationID**

6.10.2.1 Version 0.1

Size 4 bytes; sealed.

```
# 11-bit identifier.

truncated uint11 value
void21

@sealed
```

6.10.3 DataClassic

Full message type name: **uavcan.metatransport.can.DataClassic**

6.10.3.1 Version 0.1

Size 6…14 bytes; sealed.

```
# Classic data frame payload.

ArbitrationID.0.1 arbitration_id
uint8[<=8] data

@sealed
@assert _offset_.min == 48
@assert _offset_ % 8 == {0}
```

6.10.4 DataFD

Full message type name: **uavcan.metatransport.can.DataFD**

6.10.4.1 Version 0.1

Size 6…70 bytes; sealed.

```
# CAN FD data frame payload.

ArbitrationID.0.1 arbitration_id
uint8[<=64] data

@sealed
@assert _offset_.min == 48
@assert _offset_ % 8 == {0}
```

6.10.5 Error

Full message type name: **uavcan.metatransport.can.Error**

6.10.5.1 Version 0.1

Size 4 bytes; sealed.

```
# CAN bus error report: either an intentionally generated error frame or a disturbance.

void32

@sealed
```

6.10.6 ExtendedArbitrationID

Full message type name: **uavcan.metatransport.can.ExtendedArbitrationID**

6.10.6.1 Version 0.1

Size 4 bytes; sealed.

```
# 29-bit identifier.

truncated uint29 value
void3

@sealed
```

6.10.7 Frame

Full message type name: **uavcan.metatransport.can.Frame**

6.10.7.1 Version 0.2

Size 5…71 bytes; sealed.

```
# Classic CAN or CAN FD frame representation. This is the top-level data type in its namespace.

@union

Error.0.1       error                         # CAN error (intentional or disturbance)
DataFD.0.1      data_fd                       # Bit rate switch flag active
DataClassic.0.1 data_classic                  # Bit rate switch flag not active
RTR.0.1         remote_transmission_request   # Bit rate switch flag not active

@sealed                         # Sealed because the structure is rigidly dictated by an external standard.
@assert _offset_.min == 8 + 32
@assert _offset_.max == 8 + 8 + 32 + 8 + 64 * 8
```

6.10.7.2 Version 0.1, DEPRECATED

Size 12…78 bytes; sealed.

```
# CAN 2.0 or CAN FD frame representation. This is the top-level data type in its namespace.

@deprecated  # See next version.

uavcan.time.SynchronizedTimestamp.1.0 timestamp

Manifestation.0.1 manifestation

@sealed                         # Sealed because the structure is rigidly dictated by an external standard.
@assert _offset_ % 8 == {0}
@assert _offset_.min == 12 * 8
@assert _offset_.max == 78 * 8
```

6.10.8 Manifestation

Full message type name: **`uavcan.metatransport.can.Manifestation`**

6.10.8.1 Version 0.1, DEPRECATED

Size 5…71 bytes; sealed.

```
# CAN frame properties that can be manifested on the bus.

@deprecated     # See Frame.0.2 as a replacement
@union

Error.0.1       error                         # CAN error (intentional or disturbance)
DataFD.0.1      data_fd                       # Bit rate switch flag active
DataClassic.0.1 data_classic                  # Bit rate switch flag not active
RTR.0.1         remote_transmission_request   # Bit rate switch flag not active

@sealed
@assert _offset_.min == 8 + 32
@assert _offset_.max == 8 + 8 + 32 + 8 + 64 * 8
@assert _offset_ % 8 == {0}
```

6.10.9 RTR

Full message type name: **`uavcan.metatransport.can.RTR`**

6.10.9.1 Version 0.1

Size 5 bytes; sealed.

```
# Classic remote transmission request (not defined for CAN FD).

ArbitrationID.0.1 arbitration_id

@sealed
@assert _offset_ == {40}
```

6.11 uavcan.metatransport.ethernet

6.11.1 EtherType

Full message type name: **`uavcan.metatransport.ethernet.EtherType`**

6.11.1.1 Version 0.1

Size 2 bytes; sealed.

```
# Standard EtherType constants as defined by IEEE Registration Authority and IANA.
# This list is only a small subset of constants that are considered to be relevant for UAVCAN.
```

```
uint16 value

uint16 IP_V4 = 0x0800
uint16 ARP   = 0x0806
uint16 IP_V6 = 0x86DD

@sealed
```

6.11.2 Frame

Full message type name: **`uavcan.metatransport.ethernet.Frame`**

6.11.2.1 Version 0.1

Size 16...9232 bytes; sealed.

```
# IEEE 802.3 Ethernet frame encapsulation.
# In terms of libpcap/tcpdump, the corresponding link type is LINKTYPE_ETHERNET/DLT_EN10MB.

uint8[6] destination
uint8[6] source

EtherType.0.1 ethertype

uint8[<=9216] payload  # Supports conventional jumbo frames (up to 9 KiB).

@sealed  # Sealed because the format is defined by external specifications.
```

6.12 uavcan.metatransport.serial

6.12.1 Fragment

Full message type name: **`uavcan.metatransport.serial.Fragment`**

6.12.1.1 Version 0.2

Size 2...2050 bytes; sealed.

```
# A chunk of raw bytes exchanged over a serial transport. Serial links do not support framing natively.
# The chunk may be of arbitrary size.
#
# If this data type is used to encapsulate UAVCAN/serial, then it is recommended to ensure that each message
# contains at most one UAVCAN/serial transport frame (frames are separated by zero-valued delimiter bytes).

uint12 CAPACITY_BYTES = 2048
uint8[<=CAPACITY_BYTES] data

@sealed
```

6.12.1.2 Version 0.1, DEPRECATED

Size 9...265 bytes; sealed.

```
# A chunk of raw bytes exchanged over a serial transport. Serial links do not support framing natively.
# The chunk may be of arbitrary size.

@deprecated  # See next version.

uavcan.time.SynchronizedTimestamp.1.0 timestamp

uint9 CAPACITY_BYTES = 256
uint8[<=CAPACITY_BYTES] data

@sealed
@assert _offset_ % 8 == {0}
@assert _offset_.max / 8 <= 313
```

6.13 uavcan.metatransport.udp

6.13.1 Endpoint

Full message type name: **`uavcan.metatransport.udp.Endpoint`**

6.13.1.1 Version 0.1, DEPRECATED

Size 32 bytes; sealed.

```
# A UDP/IP endpoint address specification.

@deprecated  # Replaced by uavcan.metatransport.ethernet

uint8[16] ip_address
# The IP address of the host in the network byte order (big endian).
```

```
# IPv6 addresses are represented as-is.
# IPv4 addresses are represented using IPv4-mapped IPv6 addresses.

uint8[6] mac_address
# MAC address of the host in the network byte order (big endian).

uint16 port
# The UDP port number.

void64

@sealed
@assert _offset_ == {32} * 8
```

6.13.2 Frame

Full message type name: **uavcan.metatransport.udp.Frame**

6.13.2.1 Version 0.1, DEPRECATED

Size without delimiter header: 74…9262 bytes; extent 10240 bytes.

```
# A generic UDP/IP frame.
# Jumboframes are supported in the interest of greater application compatibility.

@deprecated  # Replaced by uavcan.metatransport.ethernet

uavcan.time.SynchronizedTimestamp.1.0 timestamp

void8
@assert _offset_ % 64 == {0}

Endpoint.0.1 source
Endpoint.0.1 destination

@assert _offset_ % 64 == {0}

uint14 MTU = 1024 * 9 - 20 - 8  # Max jumbo frame 9 KiB, IP header min 20 B, UDP header 8 B.
uint8[<=MTU] data

@extent 1024 * 10 * 8           # The auto-deduced extent would be unreasonably large for this type.
@assert _offset_ % 8 == {0}
```

6.14 uavcan.primitive

6.14.1 Empty

Full message type name: **uavcan.primitive.Empty**

6.14.1.1 Version 1.0

Size 0 bytes; sealed.

```
@sealed
```

6.14.2 String

Full message type name: **uavcan.primitive.String**

6.14.2.1 Version 1.0

Size 2…258 bytes; sealed.

```
# A UTF8-encoded string of text.
# Since the string is represented as a dynamic array of bytes, it is not null-terminated. Like Pascal string.

uint8[<=256] value

@sealed
@assert _offset_ % 8 == {0}
@assert _offset_.max / 8 == 258
```

6.14.3 Unstructured

Full message type name: **uavcan.primitive.Unstructured**

6.14.3.1 Version 1.0

Size 2…258 bytes; sealed.

```
# An unstructured collection of bytes, e.g., raw binary image.

uint8[<=256] value

@sealed
@assert _offset_ % 8 == {0}
@assert _offset_.max / 8 == 258
```

6.15 uavcan.primitive.array

6.15.1 Bit

Full message type name: **uavcan.primitive.array.Bit**

6.15.1.1 Version 1.0

Size 2…258 bytes; sealed.

```
# 2048 bits + 11 bit length + 4 bit padding = 2064 bits = 258 bytes

bool[<=2048] value

@sealed
@assert _offset_.min == 16
@assert _offset_.max / 8 == 258
```

6.15.2 Integer8

Full message type name: **uavcan.primitive.array.Integer8**

6.15.2.1 Version 1.0

Size 2…258 bytes; sealed.

```
int8[<=256] value
@assert _offset_ % 8 == {0}
@assert _offset_.max / 8 == 258
@sealed
```

6.15.3 Integer16

Full message type name: **uavcan.primitive.array.Integer16**

6.15.3.1 Version 1.0

Size 1...257 bytes; sealed.

```
int16[<=128] value
@assert _offset_ % 8 == {0}
@assert _offset_.max / 8 == 257
@sealed
```

6.15.4 Integer32

Full message type name: **`uavcan.primitive.array.Integer32`**

6.15.4.1 Version 1.0

Size 1...257 bytes; sealed.

```
int32[<=64] value
@assert _offset_ % 8 == {0}
@assert _offset_.max / 8 == 257
@sealed
```

6.15.5 Integer64

Full message type name: **`uavcan.primitive.array.Integer64`**

6.15.5.1 Version 1.0

Size 1...257 bytes; sealed.

```
int64[<=32] value
@assert _offset_ % 8 == {0}
@assert _offset_.max / 8 == 257
@sealed
```

6.15.6 Natural8

Full message type name: **`uavcan.primitive.array.Natural8`**

6.15.6.1 Version 1.0

Size 2...258 bytes; sealed.

```
uint8[<=256] value
@assert _offset_ % 8 == {0}
@assert _offset_.max / 8 == 258
@sealed
```

6.15.7 Natural16

Full message type name: **`uavcan.primitive.array.Natural16`**

6.15.7.1 Version 1.0

Size 1...257 bytes; sealed.

```
uint16[<=128] value
@assert _offset_ % 8 == {0}
@assert _offset_.max / 8 == 257
@sealed
```

6.15.8 Natural32

Full message type name: **`uavcan.primitive.array.Natural32`**

6.15.8.1 Version 1.0

Size 1...257 bytes; sealed.

```
uint32[<=64] value
@assert _offset_ % 8 == {0}
@assert _offset_.max / 8 == 257
@sealed
```

6.15.9 Natural64

Full message type name: **`uavcan.primitive.array.Natural64`**

6.15.9.1 Version 1.0

Size 1...257 bytes; sealed.

```
uint64[<=32] value
@assert _offset_ % 8 == {0}
@assert _offset_.max / 8 == 257
@sealed
```

6.15.10 Real16

Full message type name: **uavcan.primitive.array.Real16**

6.15.10.1 Version 1.0

Size 1...257 bytes; sealed.

```
# Exactly representable integers: [-2048, +2048]

float16[<=128] value

@sealed
@assert _offset_ % 8 == {0}
@assert _offset_.max / 8 == 257
```

6.15.11 Real32

Full message type name: **uavcan.primitive.array.Real32**

6.15.11.1 Version 1.0

Size 1...257 bytes; sealed.

```
# Exactly representable integers: [-16777216, +16777216]

float32[<=64] value

@sealed
@assert _offset_ % 8 == {0}
@assert _offset_.max / 8 == 257
```

6.15.12 Real64

Full message type name: **uavcan.primitive.array.Real64**

6.15.12.1 Version 1.0

Size 1...257 bytes; sealed.

```
# Exactly representable integers: [-2**53, +2**53]

float64[<=32] value

@sealed
@assert _offset_ % 8 == {0}
@assert _offset_.max / 8 == 257
```

6.16 uavcan.primitive.scalar

6.16.1 Bit

Full message type name: **uavcan.primitive.scalar.Bit**

6.16.1.1 Version 1.0

Size 1 bytes; sealed.

```
bool value
@sealed
```

6.16.2 Integer8

Full message type name: **uavcan.primitive.scalar.Integer8**

6.16.2.1 Version 1.0

Size 1 bytes; sealed.

```
int8 value
@sealed
```

6.16.3 Integer16

Full message type name: **`uavcan.primitive.scalar.Integer16`**

6.16.3.1 Version 1.0

Size 2 bytes; sealed.

```
int16 value
@sealed
```

6.16.4 Integer32

Full message type name: **`uavcan.primitive.scalar.Integer32`**

6.16.4.1 Version 1.0

Size 4 bytes; sealed.

```
int32 value
@sealed
```

6.16.5 Integer64

Full message type name: **`uavcan.primitive.scalar.Integer64`**

6.16.5.1 Version 1.0

Size 8 bytes; sealed.

```
int64 value
@sealed
```

6.16.6 Natural8

Full message type name: **`uavcan.primitive.scalar.Natural8`**

6.16.6.1 Version 1.0

Size 1 bytes; sealed.

```
uint8 value
@sealed
```

6.16.7 Natural16

Full message type name: **`uavcan.primitive.scalar.Natural16`**

6.16.7.1 Version 1.0

Size 2 bytes; sealed.

```
uint16 value
@sealed
```

6.16.8 Natural32

Full message type name: **`uavcan.primitive.scalar.Natural32`**

6.16.8.1 Version 1.0

Size 4 bytes; sealed.

```
uint32 value
@sealed
```

6.16.9 Natural64

Full message type name: **`uavcan.primitive.scalar.Natural64`**

6.16.9.1 Version 1.0

Size 8 bytes; sealed.

```
uint64 value
@sealed
```

6.16.10 Real16

Full message type name: **uavcan.primitive.scalar.Real16**

6.16.10.1 Version 1.0

Size 2 bytes; sealed.

```
float16 value       # Exactly representable integers: [-2048, +2048]
@sealed
```

6.16.11 Real32

Full message type name: **uavcan.primitive.scalar.Real32**

6.16.11.1 Version 1.0

Size 4 bytes; sealed.

```
float32 value       # Exactly representable integers: [-16777216, +16777216]
@sealed
```

6.16.12 Real64

Full message type name: **uavcan.primitive.scalar.Real64**

6.16.12.1 Version 1.0

Size 8 bytes; sealed.

```
float64 value       # Exactly representable integers: [-2**53, +2**53]
@sealed
```

6.17 uavcan.si.sample.acceleration

6.17.1 Scalar

Full message type name: **uavcan.si.sample.acceleration.Scalar**

6.17.1.1 Version 1.0

Size 11 bytes; sealed.

```
uavcan.time.SynchronizedTimestamp.1.0 timestamp
float32 meter_per_second_per_second
@sealed
```

6.17.2 Vector3

Full message type name: **uavcan.si.sample.acceleration.Vector3**

6.17.2.1 Version 1.0

Size 19 bytes; sealed.

```
uavcan.time.SynchronizedTimestamp.1.0 timestamp
float32[3] meter_per_second_per_second
@sealed
```

6.18 uavcan.si.sample.angle

6.18.1 Quaternion

Full message type name: **uavcan.si.sample.angle.Quaternion**

6.18.1.1 Version 1.0

Size 23 bytes; sealed.

```
uavcan.time.SynchronizedTimestamp.1.0 timestamp
float32[4] wxyz
@sealed
```

6.18.2 Scalar

Full message type name: **uavcan.si.sample.angle.Scalar**

6.18.2.1 Version 1.0

Size 11 bytes; sealed.

```
uavcan.time.SynchronizedTimestamp.1.0 timestamp
float32 radian
@sealed
```

6.19 uavcan.si.sample.angular_acceleration

6.19.1 Scalar

Full message type name: **`uavcan.si.sample.angular_acceleration.Scalar`**

6.19.1.1 Version 1.0

Size 11 bytes; sealed.

```
uavcan.time.SynchronizedTimestamp.1.0 timestamp
float32 radian_per_second_per_second
@sealed
```

6.19.2 Vector3

Full message type name: **`uavcan.si.sample.angular_acceleration.Vector3`**

6.19.2.1 Version 1.0

Size 19 bytes; sealed.

```
uavcan.time.SynchronizedTimestamp.1.0 timestamp
float32[3] radian_per_second_per_second
@sealed
```

6.20 uavcan.si.sample.angular_velocity

6.20.1 Scalar

Full message type name: **`uavcan.si.sample.angular_velocity.Scalar`**

6.20.1.1 Version 1.0

Size 11 bytes; sealed.

```
uavcan.time.SynchronizedTimestamp.1.0 timestamp
float32 radian_per_second
@sealed
```

6.20.2 Vector3

Full message type name: **`uavcan.si.sample.angular_velocity.Vector3`**

6.20.2.1 Version 1.0

Size 19 bytes; sealed.

```
uavcan.time.SynchronizedTimestamp.1.0 timestamp
float32[3] radian_per_second
@sealed
```

6.21 uavcan.si.sample.duration

6.21.1 Scalar

Full message type name: **`uavcan.si.sample.duration.Scalar`**

6.21.1.1 Version 1.0

Size 11 bytes; sealed.

```
uavcan.time.SynchronizedTimestamp.1.0 timestamp
float32 second
@sealed
```

6.21.2 WideScalar

Full message type name: **`uavcan.si.sample.duration.WideScalar`**

6.21.2.1 Version 1.0

Size 15 bytes; sealed.

```
uavcan.time.SynchronizedTimestamp.1.0 timestamp
float64 second
@sealed
```

6.22 uavcan.si.sample.electric_charge

6.22.1 Scalar

Full message type name: **uavcan.si.sample.electric_charge.Scalar**

6.22.1.1 Version 1.0

Size 11 bytes; sealed.

```
uavcan.time.SynchronizedTimestamp.1.0 timestamp
float32 coulomb
@sealed
```

6.23 uavcan.si.sample.electric_current

6.23.1 Scalar

Full message type name: **uavcan.si.sample.electric_current.Scalar**

6.23.1.1 Version 1.0

Size 11 bytes; sealed.

```
uavcan.time.SynchronizedTimestamp.1.0 timestamp
float32 ampere
@sealed
```

6.24 uavcan.si.sample.energy

6.24.1 Scalar

Full message type name: **uavcan.si.sample.energy.Scalar**

6.24.1.1 Version 1.0

Size 11 bytes; sealed.

```
uavcan.time.SynchronizedTimestamp.1.0 timestamp
float32 joule
@sealed
```

6.25 uavcan.si.sample.force

6.25.1 Scalar

Full message type name: **uavcan.si.sample.force.Scalar**

6.25.1.1 Version 1.0

Size 11 bytes; sealed.

```
uavcan.time.SynchronizedTimestamp.1.0 timestamp
float32 newton
@sealed
```

6.25.2 Vector3

Full message type name: **uavcan.si.sample.force.Vector3**

6.25.2.1 Version 1.0

Size 19 bytes; sealed.

```
uavcan.time.SynchronizedTimestamp.1.0 timestamp
float32[3] newton
@sealed
```

6.26 uavcan.si.sample.frequency

6.26.1 Scalar

Full message type name: **`uavcan.si.sample.frequency.Scalar`**

6.26.1.1 Version 1.0

Size 11 bytes; sealed.

```
uavcan.time.SynchronizedTimestamp.1.0 timestamp
float32 hertz
@sealed
```

6.27 uavcan.si.sample.length

6.27.1 Scalar

Full message type name: **`uavcan.si.sample.length.Scalar`**

6.27.1.1 Version 1.0

Size 11 bytes; sealed.

```
uavcan.time.SynchronizedTimestamp.1.0 timestamp
float32 meter
@sealed
```

6.27.2 Vector3

Full message type name: **`uavcan.si.sample.length.Vector3`**

6.27.2.1 Version 1.0

Size 19 bytes; sealed.

```
uavcan.time.SynchronizedTimestamp.1.0 timestamp
float32[3] meter
@sealed
```

6.27.3 WideScalar

Full message type name: **`uavcan.si.sample.length.WideScalar`**

6.27.3.1 Version 1.0

Size 15 bytes; sealed.

```
uavcan.time.SynchronizedTimestamp.1.0 timestamp
float64 meter
@sealed
```

6.27.4 WideVector3

Full message type name: **`uavcan.si.sample.length.WideVector3`**

6.27.4.1 Version 1.0

Size 31 bytes; sealed.

```
uavcan.time.SynchronizedTimestamp.1.0 timestamp
float64[3] meter
@sealed
```

6.28 uavcan.si.sample.magnetic_field_strength

6.28.1 Scalar

Full message type name: **`uavcan.si.sample.magnetic_field_strength.Scalar`**

6.28.1.1 Version 1.0

Size 11 bytes; sealed.

```
uavcan.time.SynchronizedTimestamp.1.0 timestamp
float32 tesla
@sealed
```

6.28.2 Vector3

Full message type name: **`uavcan.si.sample.magnetic_field_strength.Vector3`**

6.28.2.1 Version 1.0

Size 19 bytes; sealed.

```
uavcan.time.SynchronizedTimestamp.1.0 timestamp
float32[3] tesla
@sealed
```

6.29 uavcan.si.sample.mass

6.29.1 Scalar

Full message type name: **`uavcan.si.sample.mass.Scalar`**

6.29.1.1 Version 1.0

Size 11 bytes; sealed.

```
uavcan.time.SynchronizedTimestamp.1.0 timestamp
float32 kilogram
@sealed
```

6.30 uavcan.si.sample.power

6.30.1 Scalar

Full message type name: **`uavcan.si.sample.power.Scalar`**

6.30.1.1 Version 1.0

Size 11 bytes; sealed.

```
uavcan.time.SynchronizedTimestamp.1.0 timestamp
float32 watt
@sealed
```

6.31 uavcan.si.sample.pressure

6.31.1 Scalar

Full message type name: **`uavcan.si.sample.pressure.Scalar`**

6.31.1.1 Version 1.0

Size 11 bytes; sealed.

```
uavcan.time.SynchronizedTimestamp.1.0 timestamp
float32 pascal
@sealed
```

6.32 uavcan.si.sample.temperature

6.32.1 Scalar

Full message type name: **`uavcan.si.sample.temperature.Scalar`**

6.32.1.1 Version 1.0

Size 11 bytes; sealed.

```
uavcan.time.SynchronizedTimestamp.1.0 timestamp
float32 kelvin
@sealed
```

6.33 uavcan.si.sample.torque

6.33.1 Scalar

Full message type name: **`uavcan.si.sample.torque.Scalar`**

6.33.1.1 Version 1.0

Size 11 bytes; sealed.

```
uavcan.time.SynchronizedTimestamp.1.0 timestamp
float32 newton_meter
@sealed
```

6.33.2 Vector3

Full message type name: **`uavcan.si.sample.torque.Vector3`**

6.33.2.1 Version 1.0

Size 19 bytes; sealed.

```
uavcan.time.SynchronizedTimestamp.1.0 timestamp
float32[3] newton_meter
@sealed
```

6.34 uavcan.si.sample.velocity

6.34.1 Scalar

Full message type name: **`uavcan.si.sample.velocity.Scalar`**

6.34.1.1 Version 1.0

Size 11 bytes; sealed.

```
uavcan.time.SynchronizedTimestamp.1.0 timestamp
float32 meter_per_second
@sealed
```

6.34.2 Vector3

Full message type name: **`uavcan.si.sample.velocity.Vector3`**

6.34.2.1 Version 1.0

Size 19 bytes; sealed.

```
uavcan.time.SynchronizedTimestamp.1.0 timestamp
float32[3] meter_per_second
@sealed
```

6.35 uavcan.si.sample.voltage

6.35.1 Scalar

Full message type name: **`uavcan.si.sample.voltage.Scalar`**

6.35.1.1 Version 1.0

Size 11 bytes; sealed.

```
uavcan.time.SynchronizedTimestamp.1.0 timestamp
float32 volt
@sealed
```

6.36 uavcan.si.sample.volume

6.36.1 Scalar

Full message type name: **`uavcan.si.sample.volume.Scalar`**

6.36.1.1 Version 1.0

Size 11 bytes; sealed.

```
uavcan.time.SynchronizedTimestamp.1.0 timestamp
float32 cubic_meter
@sealed
```

6.37 uavcan.si.sample.volumetric_flow_rate

6.37.1 Scalar

Full message type name: **uavcan.si.sample.volumetric_flow_rate.Scalar**

6.37.1.1 Version 1.0

Size 11 bytes; sealed.

```
uavcan.time.SynchronizedTimestamp.1.0 timestamp
float32 cubic_meter_per_second
@sealed
```

6.38 uavcan.si.unit.acceleration

6.38.1 Scalar

Full message type name: **uavcan.si.unit.acceleration.Scalar**

6.38.1.1 Version 1.0

Size 4 bytes; sealed.

```
float32 meter_per_second_per_second
@sealed
```

6.38.2 Vector3

Full message type name: **uavcan.si.unit.acceleration.Vector3**

6.38.2.1 Version 1.0

Size 12 bytes; sealed.

```
float32[3] meter_per_second_per_second
@sealed
```

6.39 uavcan.si.unit.angle

6.39.1 Quaternion

Full message type name: **uavcan.si.unit.angle.Quaternion**

6.39.1.1 Version 1.0

Size 16 bytes; sealed.

```
float32[4] wxyz
@sealed
```

6.39.2 Scalar

Full message type name: **uavcan.si.unit.angle.Scalar**

6.39.2.1 Version 1.0

Size 4 bytes; sealed.

```
float32 radian
@sealed
```

6.40 uavcan.si.unit.angular_acceleration

6.40.1 Scalar

Full message type name: **uavcan.si.unit.angular_acceleration.Scalar**

6.40.1.1 Version 1.0

Size 4 bytes; sealed.

```
float32 radian_per_second_per_second
@sealed
```

6.40.2 Vector3

Full message type name: **uavcan.si.unit.angular_acceleration.Vector3**

6.40.2.1 Version 1.0

Size 12 bytes; sealed.

```
float32[3] radian_per_second_per_second
@sealed
```

6.41 uavcan.si.unit.angular_velocity

6.41.1 Scalar

Full message type name: **uavcan.si.unit.angular_velocity.Scalar**

6.41.1.1 Version 1.0

Size 4 bytes; sealed.

```
float32 radian_per_second
@sealed
```

6.41.2 Vector3

Full message type name: **uavcan.si.unit.angular_velocity.Vector3**

6.41.2.1 Version 1.0

Size 12 bytes; sealed.

```
float32[3] radian_per_second
@sealed
```

6.42 uavcan.si.unit.duration

6.42.1 Scalar

Full message type name: **uavcan.si.unit.duration.Scalar**

6.42.1.1 Version 1.0

Size 4 bytes; sealed.

```
float32 second
@sealed
```

6.42.2 WideScalar

Full message type name: **uavcan.si.unit.duration.WideScalar**

6.42.2.1 Version 1.0

Size 8 bytes; sealed.

```
float64 second
@sealed
```

6.43 uavcan.si.unit.electric_charge

6.43.1 Scalar

Full message type name: **uavcan.si.unit.electric_charge.Scalar**

6.43.1.1 Version 1.0

Size 4 bytes; sealed.

```
float32 coulomb
@sealed
```

6.44 uavcan.si.unit.electric_current

6.44.1 Scalar

Full message type name: **uavcan.si.unit.electric_current.Scalar**

6.44.1.1 Version 1.0

Size 4 bytes; sealed.

```
float32 ampere
@sealed
```

6.45 uavcan.si.unit.energy

6.45.1 Scalar

Full message type name: **`uavcan.si.unit.energy.Scalar`**

6.45.1.1 Version 1.0

Size 4 bytes; sealed.

```
float32 joule
@sealed
```

6.46 uavcan.si.unit.force

6.46.1 Scalar

Full message type name: **`uavcan.si.unit.force.Scalar`**

6.46.1.1 Version 1.0

Size 4 bytes; sealed.

```
float32 newton
@sealed
```

6.46.2 Vector3

Full message type name: **`uavcan.si.unit.force.Vector3`**

6.46.2.1 Version 1.0

Size 12 bytes; sealed.

```
float32[3] newton
@sealed
```

6.47 uavcan.si.unit.frequency

6.47.1 Scalar

Full message type name: **`uavcan.si.unit.frequency.Scalar`**

6.47.1.1 Version 1.0

Size 4 bytes; sealed.

```
float32 hertz
@sealed
```

6.48 uavcan.si.unit.length

6.48.1 Scalar

Full message type name: **`uavcan.si.unit.length.Scalar`**

6.48.1.1 Version 1.0

Size 4 bytes; sealed.

```
float32 meter
@sealed
```

6.48.2 Vector3

Full message type name: **`uavcan.si.unit.length.Vector3`**

6.48.2.1 Version 1.0

Size 12 bytes; sealed.

```
float32[3] meter
@sealed
```

6.48.3 WideScalar

Full message type name: **`uavcan.si.unit.length.WideScalar`**

6.48.3.1 Version 1.0

Size 8 bytes; sealed.

```
float64 meter
@sealed
```

6.48.4 WideVector3

Full message type name: **`uavcan.si.unit.length.WideVector3`**

6.48.4.1 Version 1.0

Size 24 bytes; sealed.

```
float64[3] meter
@sealed
```

6.49 uavcan.si.unit.magnetic_field_strength

6.49.1 Scalar

Full message type name: **`uavcan.si.unit.magnetic_field_strength.Scalar`**

6.49.1.1 Version 1.0

Size 4 bytes; sealed.

```
float32 tesla
@sealed
```

6.49.2 Vector3

Full message type name: **`uavcan.si.unit.magnetic_field_strength.Vector3`**

6.49.2.1 Version 1.0

Size 12 bytes; sealed.

```
float32[3] tesla
@sealed
```

6.50 uavcan.si.unit.mass

6.50.1 Scalar

Full message type name: **`uavcan.si.unit.mass.Scalar`**

6.50.1.1 Version 1.0

Size 4 bytes; sealed.

```
float32 kilogram
@sealed
```

6.51 uavcan.si.unit.power

6.51.1 Scalar

Full message type name: **`uavcan.si.unit.power.Scalar`**

6.51.1.1 Version 1.0

Size 4 bytes; sealed.

```
float32 watt
@sealed
```

6.52 uavcan.si.unit.pressure

6.52.1 Scalar

Full message type name: **`uavcan.si.unit.pressure.Scalar`**

6.52.1.1 Version 1.0

Size 4 bytes; sealed.

```
float32 pascal
@sealed
```

6.53 uavcan.si.unit.temperature

6.53.1 Scalar

Full message type name: **`uavcan.si.unit.temperature.Scalar`**

6.53.1.1 Version 1.0

Size 4 bytes; sealed.

```
float32 kelvin
@sealed
```

6.54 uavcan.si.unit.torque

6.54.1 Scalar

Full message type name: **`uavcan.si.unit.torque.Scalar`**

6.54.1.1 Version 1.0

Size 4 bytes; sealed.

```
float32 newton_meter
@sealed
```

6.54.2 Vector3

Full message type name: **`uavcan.si.unit.torque.Vector3`**

6.54.2.1 Version 1.0

Size 12 bytes; sealed.

```
float32[3] newton_meter
@sealed
```

6.55 uavcan.si.unit.velocity

6.55.1 Scalar

Full message type name: **`uavcan.si.unit.velocity.Scalar`**

6.55.1.1 Version 1.0

Size 4 bytes; sealed.

```
float32 meter_per_second
@sealed
```

6.55.2 Vector3

Full message type name: **`uavcan.si.unit.velocity.Vector3`**

6.55.2.1 Version 1.0

Size 12 bytes; sealed.

```
float32[3] meter_per_second
@sealed
```

6.56 uavcan.si.unit.voltage

6.56.1 Scalar

Full message type name: **`uavcan.si.unit.voltage.Scalar`**

6.56.1.1 Version 1.0

Size 4 bytes; sealed.

```
float32 volt
@sealed
```

6.57 uavcan.si.unit.volume

6.57.1 Scalar

Full message type name: **`uavcan.si.unit.volume.Scalar`**

6.57.1.1 Version 1.0

Size 4 bytes; sealed.

```
float32 cubic_meter
@sealed
```

6.58 uavcan.si.unit.volumetric_flow_rate

6.58.1 Scalar

Full message type name: **`uavcan.si.unit.volumetric_flow_rate.Scalar`**

6.58.1.1 Version 1.0

Size 4 bytes; sealed.

```
float32 cubic_meter_per_second
@sealed
```