UAVCAN

Specification v1.0

Revision 2019-05-08

Overview

UAVCAN is an open lightweight protocol designed for reliable communication in aerospace and robotic applications via robust vehicle bus networks.

Features:

- Democratic network – no bus master, no single point of failure.
- Publish/subscribe and request/response (RPC[1]) exchange 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.
- The specification and high quality reference implementations in popular programming languages are free, open source, and available for commercial use (MIT license).

Support and feedback

Information, documentation, and discussions related to UAVCAN are available via the official website at uavcan.org.

License

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

[1] Remote procedure call.

Disclaimer of warranty

NOTE WELL: This Specification is provided on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE.

Limitation of liability

In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any author of this Specification be liable for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising from, out of, or in connection with the Specification or the implementation, deployment, or other use of the Specification (including but not limited to damages for loss of goodwill, work stoppage, equipment failure or malfunction, injuries to persons, or any and all other commercial damages or losses), even if such author has been advised of the possibility of such damages.

Table of contents

List of tables

List of figures

1 Introduction

UAVCAN is a lightweight protocol designed to provide a highly reliable communication method for aerospace and robotic applications via robust vehicle bus networks.

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

1.1 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.

```python
# This is a source code listing.
print('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 parametrization 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: $\text{BADC0FFEE}_{16} = 50159747054$, 10101_2.

1.2 Design principles

Democratic network — There will be no master node. All nodes in the network will have the same communication rights; there must 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.

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[3], network-wide time synchronization, plug-and-play node support, etc.

Atomic data abstractions — Nodes must 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.

[2] This is a footnote.

[3] Which naturally grows into a vehicle-wide health monitoring solution.

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

Support for redundant interfaces and redundant nodes — UAVCAN must 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.

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

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.

1.3 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. More information is provided in chapter 3.

UAVCAN supports 32768 message subject identifiers for publish/subscribe exchanges, 512 service identifiers for remote procedure call exchanges, and at least 8 anonymous message subject identifiers for certain special features such as plug-and-play support. A small subset of these identifiers is reserved for the core standard and for publicly released vendor-specific types. More information is provided in chapter 5.

UAVCAN supports at least[5] eight distinct communication priority levels, defined in section 4.1.4. Within each priority level different types of transfers, messages, and services are prioritized in a well-defined, deterministic manner.

The list of transport protocols supported by UAVCAN is provided in chapter 4. Non-redundant, doubly-redundant and triply-redundant transports are supported. More information on the physical layer and standardized physical connectivity options is provided in chapter 7.

1.4 Public regulated data types

This section talks about general management policies. The related technical aspects are covered in chapters 2 and 3.

The UAVCAN maintainers are charged with maintaining and advancing the set of public regulated data types based on the input from adopters. This feedback is gathered via the online discussion and collaboration websites which are open to the general public and available through uavcan.org.

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. See chapter 5 for more information.

Within the same major version of the specification, the set of public regulated data type definitions can be modified only in the following ways:

- A new data type can be added, as long as it doesn't conflict with any of the existing data types.
- An existing data type can be modified, as long as its version number is updated accordingly and all backward compatibility guarantees are respected.
- An existing data type or a particular major version of it can be declared deprecated.
  - Once declared deprecated, the data type will be maintained for at least two more years. After this period, its regulated fixed port-ID (if defined) may be reused for an incompatible data type definition. The maintainers will be striving to postpone the reuse of regulated port identifiers as much as possible in order to minimize the possibility of unintended conflicts.

[5] Depending on the transport protocol.

[6] Commercial off-the-shelf equipment.

- Deprecation will be indicated in the DSDL definition and announced via the discussion forum.

A link to the repository containing the set of default DSDL definitions can be found on the official website.

1.5 Referenced sources

The UAVCAN specification contains references to the following sources:

- CiA 801 — Application note — Automatic bit rate detection.
- CiA 103 — Intrinsically safe capable physical layer.
- CiA 303 — Recommendation — Part 1: Cabling and connector pin assignment.
- IEEE 754 — Standard for binary floating-point arithmetic.
- 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++.
- "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.
- semver.org - Semantic versioning specification.
- 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.

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 use same message data type down to the major version (minor versions are allowed to differ) and pertain to 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 use same service data type down to the major version (minor versions are allowed to differ; as a consequence, the minor data type version number of a service response may differ from that of its corresponding request) and pertain to the same function.

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 port-ID used in a given UAVCAN network shall not be used with different data type definitions unless they share the same full name and the same major version number[9].

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 data types are defined using the *data structure description language* (DSDL) (chapter 3). A DSDL definition specifies the name, major version, minor version, the data schemas, and an optional fixed port-ID of the data type among other less important properties. Message data types always define exactly

[7] Here and elsewhere in this specification, *ID* and *identifier* are used interchangeably unless specifically indicated otherwise.

[8] Like remote procedure call (RPC).

[9] More on data type versioning in section 3.8.

one data schema, whereas service data types contain two independent schema definitions: 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 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*[10].

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 must not be used with public unregulated data types, since that is likely to cause unresolvable port identifier collisions[11]. This restriction shall be followed at all times by all compliant implementations and systems[12].

Table 2.1: Data type taxonomy

	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”*.

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

Each of the two sets of valid port identifiers (which are subject identifiers and service identifiers) are segregated into three categories (the ranges are documented in chapter 5):

- 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 runtime-assigned), 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

[10] 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.

[11] 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.

[12] 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.

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[13].

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

Serialized message and service objects[14] are exchanged by means of the transport layer (chapter 4), which implements automatic decomposition of long transfers into several transport frames[15] 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. For more information see chapter 5.

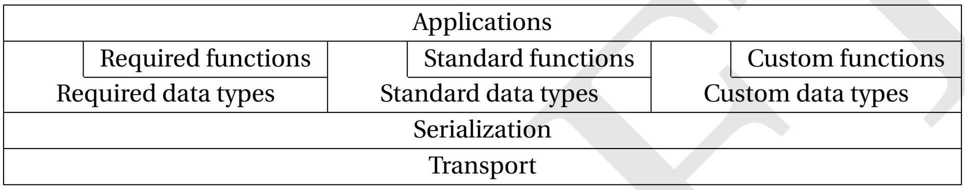


Figure 2.1: UAVCAN architectural diagram.

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 the table 2.2.

Table 2.2: Published message properties

Property	Description
Payload	The serialized message object.
Subject-ID	Numerical identifier that indicates how the information should be interpreted.
Source node-ID	The node-ID of the transmitting node (excepting anonymous messages).
Transfer-ID	A small overflowing integer that increments with every transfer of this message type from a given node. Used for message sequence monitoring, multi-frame transfer reassembly, and elimination of transport frame duplication errors for single-frame transfers. Additionally, Transfer-ID is crucial for automatic management of redundant transport interfaces. Its properties are explained in detail in the chapter 4.

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, and that it can't be decomposed across several transport frames.

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 particularly useful for the plug-and-play feature, which is explored in detail in chapter 5.

[13] 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.

[14] Here and elsewhere, an *object* means a value that is an instance of a well-defined type.

[15] Here and elsewhere, a *transport frame* means a block of data that can be atomically exchanged over the transport layer network, e.g., a CAN frame.

Anonymous messages cannot be decomposed into multiple transport frames, meaning that their payload capacity is limited to that of a single transport frame. More info is provided in chapter 4.

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[16]:

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

Information contained in service requests and responses is summarized in the table 2.3.

Table 2.3: Service request/response properties

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	A small overflowing integer that increments with every call of this service type from a given node. Used for request/response matching, multi-frame transfer reassembly, and elimination of transport frame duplication errors for single-frame transfers. Additionally, Transfer-ID is crucial for automatic management of redundant transport interfaces. Its properties are explained in detail in the chapter 4.

Both the request and the response contain same values for all listed fields except payload, where the content is application-defined. Clients match responses with corresponding requests using the following fields: service-ID, client node-ID, server node-ID, and transfer-ID.

[16] 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.

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[17].

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 message data type defines one data schema of the message object, and a service data type contains two independent data schema definitions: one of them applies to the request object (transferred from client to server), and the other applies to the response object (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 on the figure 3.1.

[17] 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.

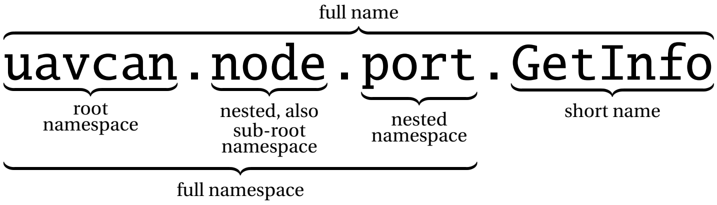


Figure 3.1: Data type name structure.

A set of full namespace names and a set of full data type names must not intersect[18].

Data type names and namespace names are case-sensitive. However, names that differ only in letter case are not permitted[19]. 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 must not be a digit. A name component must not match any of the reserved word patterns, which are listed in the table 3.5.

The length of a full data type name must not exceed 50 characters[20].

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 must 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 must 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[21] 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).

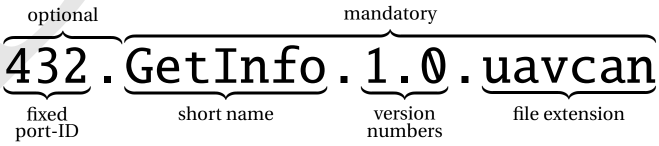


Figure 3.2: Data type definition file name structure.

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

[18] 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.

[19] Because that may cause problems with case-insensitive file systems.

[20] This includes the name component separators.

[21] Chapter 2.

cannot be spread across different directories sharing the same name. One directory cannot define more than one level of nesting[22].

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). As explained above, one data type definition contains either one or two data schema definitions, for message types and service types, respectively.

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

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.

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 (or a data schema) refers to the set of bit length values of all possible serialized representations of objects that are instances of the data type (or that follow the data schema).

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.

[22] For example, "`foo.bar`" is not a valid directory name. The valid representation would be "`bar`" nested in "`foo`".

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 notation used in the following definition is a variant of the extended Backus–Naur form[23]. The rule definition patterns are specified in the table 3.1. The text of the formal definition contains comments which begin with an octothorp and last until the end of the line.

Table 3.1: Notation used in the formal grammar definition

Pattern	Description
`"korovan"`	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.
`korovan / paukan / excavator`	Alternatives. The leftmost matching alternative is accepted.
`korovan?`	Optional greedy match.
`paukan*`	Zero or more expressions, greedy match.
`excavator+`	One or more expressions, greedy match.
`~r"regex-pattern"`	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-pattern’`	As above, with single quotes instead of double quotes.
`(korovan paukan)`	Parentheses are used for grouping.

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.

[23] This notation is a subset of the notation defined in a Python parsing library titled Parsimonious. Parsimonious is 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 must 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 must 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.

Table 3.2: Unary operators

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

Table 3.3: Binary 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

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 3.4.2) 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 the table 3.4 which must follow one of the specified therein forms. An escape sequence begins with the ASCII backslash character “\”.

Table 3.4: String literal escape sequences

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.

```
@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) parametrized by the type of the contained elements which is determined automatically.

A set literal declaration must 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 the table 3.5 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.

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

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?`	`lpt`	Compatibility with Microsoft Windows
`_.*_`	`_offset_`	Special-purpose intrinsic entities

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[24]. 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.

[24] 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.

Table 3.6: Operators defined on instances of rational numbers

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.

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.

Table 3.7: Operators defined on instances of Unicode strings

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.

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 must 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 must contain at least one element.

The attributes and operators defined on set instances are listed in the tables 3.8 and 3.9, where E represents the set element type.

Table 3.8: Attributes defined on instances of sets

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

Table 3.9: Operators defined on instances of sets

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

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

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 the section 3.7.

3.4.1 Void types

Void types are used for padding purposes. As will be explained in later sections, it is desirable to align field attributes at byte boundaries; void types can be used to facilitate that.

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.

3.4.2 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.

3.4.2.1 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 3.10.
    - **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 3.11.

Table 3.10: Properties of integer types

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 3.11: Properties of floating point types

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

3.4.2.2 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 the table 3.12. Cases that are marked as illegal are not permitted in DSDL definitions.

Table 3.12: Lossy assignment rules per cast mode

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.

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.

3.4.2.3 *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.

Table 3.13: Operators defined on instances of type boolean

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.

3.4.2.4 *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

3.4.3 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[25].

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 must yield a positive integer of type "`rational`" upon its evaluation; any other value or type renders the current DSDL definition invalid.

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 must 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.

3.4.4 Composite types

3.4.4.1 Kinds

There are two kinds of composite data type definitions: message types and service types. All versions of a data type must be of the same kind[26].

A message data type defines one data schema.

A service data type contains two data schema definitions: one for service request object, and one for service response object, in that order. The two schemas are separated by the service response marker ("`---`") on a separate line.

[25] 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.

[26] For example, if a data type version 0.1 is of a message kind, all later versions of it must be messages, too.

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.

3.4.4.2 *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 the figure 3.4.

To facilitate look-up of external dependencies, implementations are expected to obtain from external sources[27] the list of directories that are the roots of namespaces containing the referred dependencies.

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

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 the 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.

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

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 must be marked deprecated as well[28]. If a non-deprecated definition refers to a deprecated definition, the referring definition is malformed[29].

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.

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

3.4.4.3 *Unions*

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

A tagged union schema 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.

3.5 Attributes

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

[27] For example, from user-provided configuration options.

[28] Deprecation is indicated with the help of a special directive, as explained in section 3.6

[29] This tainting behavior is designed to prevent unexpected breakage of type hierarchies when one of the deprecated dependencies reaches its end of life.

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 must be unique within its data schema definition and it shall not match any of the reserved name patterns specified in the table 3.5. 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 must 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 must be of the void category. A padding field attribute may not have a name.

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 must 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 must 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 the table 3.14.

Table 3.14: Permitted constant attribute value initialization patterns

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.

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 the 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 schema definition. The representation of the referred attribute in the context of the referring DSDL expression is specified in the table 3.15.

Table 3.15: Local attribute representation

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

```
uint8 FOO = 123
uint16 BAR = FOO ** 2
@assert BAR == 15129
---  # The request data schema definition 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
```

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 $\texttt{set}_{\texttt{<rational>}}$.

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 data schema definition.

If the current data schema definition belongs to the tagged union category, the referring statement must 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:

$$\texttt{\_offset\_}_{\text{union}} = \bigcup_{i=1}^{n} \left\{ \lceil \log_2 n \rceil + b : b \in B_i \right\}$$

where n is the number of fields defined in the current union definition, and B_i is the bit length set of the data type of the field number i.

If the current data schema definition does not belong to the tagged union category, the referring statement may be located anywhere within the current data schema definition. The value of the offset attribute is defined as:

$$\texttt{\_offset\_} = \begin{cases} \left\{ \sum s : s \in \prod_{i=1}^{n} B_i \right\} & \mid n > 0 \\ \{0\} & \mid n = 0 \end{cases}$$

where n is the number of fields defined in statements preceding the referring statement (see section 3.2 on statement ordering), B_i is the bit length set of the data type of the field number i, and $\prod$ is the Cartesian product operator.

```
@union
uint8 a
#@print _offset_  # Would fail: it's a tagged union, _offset_ is undefined until after the last field
uint16 b
@assert _offset_ == {1 + 8,  1 + 16}
---
@assert _offset_ == {0}
float16 a
@assert _offset_ == {16}
void4
@assert _offset_ == {20}
int4 b
@assert _offset_ == {24}
uint8[<4] c
@assert _offset_ == 2 + {24,  32,  40,  48}
void6
# 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 below. 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.
@assert _offset_ % 8 == {0}
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.

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 schema definition containing this directive belongs to the tagged union category (section 3.4.4.3).

Usage of this directive is subject to the following constraints:

- The directive shall not be used more than once per data schema definition.
- The directive shall be placed before the first composite type attribute definition in the current data schema.

```
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 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 the 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 first data schema definition.

```
@deprecated         # Applies to the entire definition
uint8 FOO = 123
#@deprecated        # Would fail - must be placed before the first attribute definition
---
#@deprecated        # Would fail - must be placed in the first data schema definition
```

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.

3.6.3 Assertion check

The identifier of the assertion check directive is “`assert`”. The assertion check directive expects an expression which must yield a value of type “`bool`” (section 3.4.2) 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}
```

3.6.4 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.

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 most significant bit is considered first (0-th index), and the least 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.

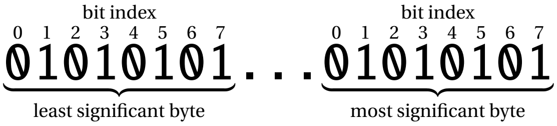


Figure 3.6: Bit and byte ordering.

3.7.1.3 *Implicit padding*

Excepting one edge case reviewed below, serialized representations of DSDL entities never include implicit data padding. Unaligned data may lead to suboptimal serialization and deserialization performance; therefore, the data type designer should manually align elements as necessary to prevent performance degradation. It is guaranteed, however, that unaligned data cannot result in unspecified or otherwise unexpected behavior of the data handling routines. The manual approach to data alignment allows the data type designer to trade-off serialization efficiency over network bandwidth utilization and data transfer latency as necessary without compromising functional safety.

The exceptional edge case mentioned above when implicit padding is introduced is as follows. Serialized representations of DSDL entities operate at the bit level, whereas the transport protocols supported by UAVCAN[30] use byte as the smallest atomic data element. The resulting mismatch of the data granularity levels is resolved by appending the serialized representation of the top-level composite type object with zero (0) bits until the length of its bit sequence is an integer multiple of eight (8). The term *top-level object* denotes an object that is not nested inside another DSDL entity. In other words, padding bits may only be added before a fully constructed serialized representation is handed over for transmission over the UAVCAN network (or any other destination which does not support bit-level data granularity).

Additionally, the transport layer may introduce extra padding at the end of the serialized representation, as described in relevant sections of chapter 4; however, such padding behavior falls outside of the scope of the DSDL specification as it belongs to the domain of network transports rather than data presentation.

When a serialized representation is deconstructed, the value of the trailing padding bits must be ignored.

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 decoded, 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 schema will be serialized as a sequence of three zero bits 000_2:

```
void3
```

The following bit sequences are valid serialized representations of the schema: 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
```

[30] As well as the majority of network protocols in general.

3.7.3 Primitive types

3.7.3.1 *General principles*

Implementations where native data formats are incompatible with those adopted by UAVCAN must perform implicit conversions between the native formats and the corresponding UAVCAN formats during serialization and deserialization. Implementations shall strive to 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[31]. If the bit width of the value is not an integer multiple of eight (8), the rightmost byte (i.e., the most significant byte) is shifted left until the most significant bit of the value is aligned with the most significant bit of the byte.

> 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:
>
> $$\overbrace{\underbrace{\texttt{11011010}}_{\text{Least significant bits/byte}}\texttt{,}\underbrace{\texttt{1110}}_{\text{Most significant bits}}}^{\text{The value }3802_{10}}\underbrace{\texttt{????}}_{\text{Next object or zero padding bits}}$$

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 123 of type `uint9` is 001111011_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 -123 of type `int9` is 110000101_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 must be able to model positive infinity, negative infinity, signaling NaN[32], and quiet NaN.

[31] Also known as "little endian".

[32] 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 schema 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 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 determined as follows:

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

where c is the capacity (i.e., the maximum number of elements) of the variable-length array. The number of elements n contained in the variable-length array is encoded in the serialized representation of the implicit length field[33] as described in section 3.7.3.3.

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

It is recommended to manually align variable-length arrays by prepending them with padding field attributes so that the first element is byte-aligned, as that facilitates more efficient serialization and deserialization.

```
void2                       # Padding - good practice but not required
AnyType.1.0[<42] array      # The implicit length field is 6 bits wide
@assert _offset_.min == 8   # Ensuring that the first element is byte-aligned
```

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

```
void2                       # Padding - good practice but not required
uint6 implicit_length_field # Set to 3, because the array contains three elements
AnyType.1.0 item_0
AnyType.1.0 item_1
AnyType.1.0 item_2
```

3.7.5 Composite types

As explained in the section 3.4.4, a data type of the service kind contains two data schema definitions: one for request object, one for response object. Unless explicitly specified otherwise, any reference to serialized representations of a service type implies either or both of its data schema definitions, depending on context.

A serialized representation of an object of a composite type is an ordered set of serialized representations of its field attribute values joined into a bit sequence. The ordering of the serialized representations of the field attribute values follows the order of field attribute declaration.

[33] Seeing as $n \leq c$ always holds, bit sequences where the implicit length field contains values greater than c do not belong to the set of serialized representations of the array.

Consider the following data schema 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
```

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, where bytes (octets) are comma-separated:

$$\overbrace{\texttt{11011010,1110}}^{\texttt{first}}\underbrace{\texttt{111}}_{\texttt{second}}\overbrace{\texttt{1,011}}^{\texttt{third}}\underbrace{\texttt{11}}_{\texttt{fourth}}\overbrace{\texttt{100,0}}^{\texttt{fifth}}\underbrace{\texttt{???????}}_{\substack{\text{Next object or}\\ \text{zero padding bits}}}$$

3.7.5.1 *Tagged unions*

Similar to variable-length arrays, a serialized representation of a 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:

$$\lceil \log_2 n \rceil$$

where n is the number of field attributes in the union, $n \geq 2$.

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[34] as described in section 3.7.3.3.

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

[34] Seeing as $i < n$ always holds, where i is the index of the current field attribute, 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.

Consider the following example:

```
@union           # In this case, the implicit union tag is 2 bit 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 encode the field b, the implicit union tag shall be assigned the value 1. The following data schema will have an identical layout:

```
uint2 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:

$$\underbrace{\texttt{01}}_{\substack{\text{union}\\\text{tag}}}\underbrace{\texttt{00000111}}_{\text{field b}}$$

It is recommended to manually align tagged unions when they are nested into outer objects by prepending them with a padding field attribute so that the value contained in the union is byte-aligned after the tag, as that facilitates more efficient serialization and deserialization.

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.
```

Consider the following union schema definition:

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

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

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

3.8 Data type 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 Compatibility

3.8.2.1 Bit compatibility

A data type or schema A is bit-compatible with a data type or schema B if and only if the set of serialized representations[35] of A is a superset of the set of serialized representations of B.

A and B are said to be *mutually bit-compatible* if their sets of serialized representations are equal.

A *variable-length* data type or schema is a serializable data type or schema whose set of serialized representations contains bit sequences of different lengths. Conversely, any data type or schema that is not variable-length is *fixed-length*.

[35] The serialization rules are reviewed in detail in the section 3.7.

The following two definitions are mutually bit-compatible:

```
uint32 a
uint32 b
```

```
uint64 c
```

Consider the following example data type definition; assume that its full data type name is `demo.Pair`:

```
# demo.Pair.1.0
float16 first
float16 second
```

Further, let the following define a data type named `demo.PairVector`:

```
# demo.PairVector.1.0
demo.Pair.1.0[3] vector
```

Then the following two definitions are bit-compatible:

```
demo.PairVector.1.0 pair_vector
```

```
float16 first_0     # pair_vector.vector[0].first
float16 second_0    # pair_vector.vector[0].second
float16 first_1     # pair_vector.vector[1].first
float16 second_1    # pair_vector.vector[1].second
float16 first_2     # pair_vector.vector[2].first
float16 second_2    # pair_vector.vector[2].second
```

The latter definition in the example above is a flattened unrolled form of the former definition. As such, in that particular example, both definitions can be used interchangeably; an object serialized using one definition can be deserialized using the other definition. However, it is also possible to construct bit-compatible definitions that are not functionally equivalent:

```
float16 a
float32 b
```

```
float32 a
float16 b
```

Even though the above definitions are bit-compatible, one cannot be substituted with the other. The problem of functional equivalency is addressed by the concept of semantic compatibility, explored in the section 3.8.2.2.

Complicated scenarios are possible when a bit belonging to a primitive-typed field attribute is handed over to a constrained field such as an implicit array length field or an implicit union tag field. Some interesting examples are shown in the table 3.16, together with a set of serialized representation patterns. Remember that the bits belonging to void-typed field attributes are ignored during deserialization.

Table 3.16: Complex bit compatibility examples

	A	**B**	**C**	**D**	**E**

Definition	`void1`
	`bool[<3] a`
		`bool x`
		`bool[<3] a`
			`void1`
			`bool[<4] a`
				`bool x`
				`bool[<4] a`
					`bool[<5] a`

Serialized representations	`000`
	`001a`
	`010aa`
	
	`100`
	`101a`
	`110aa`
			`000`
			`001a`
			`010aa`
			`011aaa`
			`100`
			`101a`
			`110aa`
			`111aaa`
					`000`
					`001a`
					`010aa`
					`011aaa`
					`100aaaa`

Bit-compatible with	B	A	A, B, D	A, B, C	*(none)*

3.8.2.2 *Semantic compatibility*

A data type A is semantically compatible with a data type B if an application that correctly uses A exhibits a functionally equivalent behavior to an application that correctly uses B. The property of semantic compatibility is commutative.

> Despite using different binary layouts, the following two definitions are semantically compatible and also bit-compatible:
>
> ```
> uint16 FLAG_A = 1
> uint16 FLAG_B = 256
> uint16 flags
> ```
>
> ```
> uint8 FLAG_A = 1
> uint8 FLAG_B = 1
> uint8 flags_a
> uint8 flags_b
> ```
>
> Therefore, the definitions can be used interchangeably. 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.

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, UAVCAN requires that all data type definitions that share the same full name and the same major version number must be semantically compatible with each other and mutually bit-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 must be assigned “1.0” (major 1, minor 0).

In order to ensure predictability and functional safety of applications that leverage UAVCAN, the standard requires that once a data type definition is released, its DSDL source text, name, version numbers, fixed port-ID, 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[36].

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

[36] 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 must be exactly one or zero definitions per combination of data type name and version number pair.

All data types under the same name must 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 must also be of the message kind.

3.8.3.3 *Port identifier assignment constraints*

A port identifier of a given kind (subject or service)[37] shall only be used with one major version of one data type in a given UAVCAN network.

The mapping from port-ID to its data type may be non-injective. In other words, a data type may be simultaneously used with more than one port-ID.

The following constraints apply to fixed port-ID assignments:

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

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[38].

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]

# Status flags in the lower bits
uint8 FLAG_COOLING_SYSTEM_A_ACTIVE = 1
```

[37] The kind is specified explicitly due to the fact that the sets of subject-ID and service-ID are orthogonal. In other words, the numeric value of a port-ID may refer to different data types if they are of different kinds.

[38] 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_B_ACTIVE = 2
# Error flags in the higher bits
uint8 FLAG_PSU_MALFUNCTION = 32
uint8 FLAG_OVERHEATING     = 64
uint8 FLAG_CRYOBOX_BREACH  = 128
# Storage for the above defined flags (this is not the recommended practice)
uint8 flags
```

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 must 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 neither semantically compatible nor bit-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

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

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 mutually 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

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

void3   # Reserved for other flags

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

Since the definitions v1.0 and v1.1 are mutually bit-compatible and 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 bit 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

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

void3   # Reserved for other flags

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

Nodes using v1.0, v1.1, and v2.0 definitions can still 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 v2.0).

[a] The topic of data serialization is explored in detail in the 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 attribute 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`.
- Avoid short names, unnecessary abbreviations, and uncommon acronyms.

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.

Trailing comments (i.e., comments that are located on the same line with a statement) should be separated from the preceding text with at least two spaces.

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, prefixed with padding bits as necessary to retain byte alignment.

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 bit-compatible and 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:

```
void7                           # It is recommended to ensure byte alignment.
MyType[<=1] optional_field
```

Union-based optional field:

```
@union                          # The implicit tag is one bit 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`):

```
void7                           # It is recommended to ensure byte alignment.
MaybeMyType optional_field
```

The shown approaches are mutually bit-compatible and semantically compatible.

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[39] 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
```

[39] Which are popular in programming.

4 Transport layer

This chapter defines the transport layer of UAVCAN. First, general implementation-agnostic concepts are introduced. Afterwards, they are further defined for each supported transport medium, e.g., CAN FD.

As the specification is extended to add support for new transport protocols, some of the generic aspects may be pushed to lower-level transport-specific sections if they are found to map poorly on the newly added transports. Such changes are guaranteed to preserve full backward compatibility of the existing transport protocols.

4.1 Core concepts

4.1.1 Transfer

A *transfer* is an act of data transmission between nodes.

4.1.1.1 Broadcast and unicast transfers

A transfer that is addressed to any interested node except the source node is a *broadcast transfer.* A transfer that is addressed to one particular node is a *unicast transfer.*

In the case of broadcast transfers, the sending node makes the data widely available on the bus, allowing any interested node to freely opt-in and process it[40]. The decision of whether to process any given transfer or not is made by receiving nodes.

In the case of unicast transfers, the addressing logic is inverted: the sending node decides which particular remote node should receive the transfer. All other nodes remain unaffected by such transmission and take no part in the addressing process.

4.1.1.2 Message and service transfers

A *message transfer* is a broadcast transfer that contains a serialized message and its metadata[41].

A *service transfer* is a unicast transfer that contains either a service request or a service response with related metadata.

4.1.1.3 Single-frame and multi-frame transfers

Both message and service transfers can be further distinguished between single-frame and multi-frame transfers.

A *single-frame transfer* is a transfer that is entirely contained in a single transport frame. The amount of data that can be exchanged using single-frame transfers is dependent on the transport protocol in use.

A *multi-frame transfer* is a transfer that has its payload distributed over multiple transport frames. The UAVCAN protocol stack handles transfer decomposition and reassembly automatically.

The choice between single-frame and multi-frame transfers is made by the UAVCAN protocol logic on the transmitting node based on the amount of payload data to be transferred. The application does not have any control over the type of transfer that will be used except limiting the amount of payload data. UAVCAN protocol implementations must always choose single-frame transfers if possible; multi-frame transfers can be used only if all of the requested payload cannot be allocated in one transport frame.

4.1.1.4 Common properties

The properties listed in the table 4.1 are common to all types of transfers.

[40] The word “broadcast” should not lead one to believe that every node is required to process such transfers. The opt-in logic is facilitated by automatic acceptance filtering features implemented on the transport layer.

[41] Such as the subject-ID and the source node-ID.

Table 4.1: Common transfer properties

Property	Description
Payload	The serialized object.
Port-ID	A numerical identifier that indicates how the data should be processed. This is the subject-ID for message transfers and service-ID for service transfers.
Source node-ID	The node-ID of the transmitting node (excepting anonymous message transfers).
Priority	A non-negative integer value that defines the transfer urgency. Higher priority transfers can preempt lower priority transfers.
Transfer-ID	A small overflowing integer that increments with every transfer of this data type from a given node.

4.1.2 Message publication

Message publication is the main method of communication between UAVCAN nodes.

A published message is carried by a single message transfer that contains the serialized message object. A published message does not contain any additional fields besides those listed in the table 4.1.

In order to publish a message, the publishing node must have a node-ID that is unique within the network. An exception applies to *anonymous message publications*.

4.1.2.1 Anonymous message publication

An anonymous message transfer is a transfer that can be sent from a node that does not have a node-ID. This kind of message transfer is especially useful for facilitation of *plug-and-play nodes* (a high-level concept that is reviewed in detail in chapter 5).

A node that does not have a node-ID is said to be in *passive mode*. Passive nodes are unable to initiate regular data exchanges, but they can listen to the transfers exchanged over the bus, and they can emit anonymous message transfers.

An anonymous message has the same properties as a regular message, except for the source node-ID.

An anonymous transfer can only be a single-frame transfer. Multi-frame anonymous message transfers are not allowed. This restriction must be kept in mind when designing message data types intended for use with anonymous message transfers: when used with anonymous transfers, the whole message must fit into a single transport frame; however, the same data type can be used with multi-frame regular (non-anonymous) transfers, if desired.

Anonymous messages may require special handling logic depending on the transport layer in use.

4.1.2.2 Message timing requirements

Generally, a message transmission should be aborted if it cannot be completed in 1 second. Applications are allowed to deviate from this recommendation, provided that every such deviation is explicitly documented. It is expected that high-frequency high-priority messages may opt for lower timeout values, whereas low-priority delayable data may opt for higher timeout values to account for network congestion.

4.1.3 Service invocation

A service invocation sequence consists of two related service transfers: *service request transfer* and *service response transfer*.

A service request transfer is sent from the invoking node – *client node* – to the node that provides the service – *server node*. Upon handling the request, the server node responds to the client node with a service response transfer. The client will match the response with the corresponding request by comparing the following values: server node-ID, service-ID, and the transfer-ID.

The tables 4.2 and 4.3 describe the properties of service request and service response transfers, respectively.

Both the client and the server must have node-ID values that are unique within the network; service invocation is not available to passive nodes. The client and the server must be two distinct nodes.

Table 4.2: Service request transfer properties

Property	Description
Payload	The serialized service request object.
Service-ID	See the table 4.1.
Source node-ID	The node-ID of the client (the invoking node).
Destination node-ID	The node-ID of the server (the invoked node).
Priority	See the table 4.1.
Transfer-ID	An integer value that: 1. allows the server to distinguish the request from other requests from the same client; 2. allows the client to match the response with its request.

Table 4.3: Service response transfer properties

Property	Description
Payload	The serialized service response object.
Service-ID	Same value as in the request transfer.
Source node-ID	The node-ID of the server (the invoked node).
Destination node-ID	The node-ID of the client (the invoking node).
Priority	Same value as in the request transfer.
Transfer-ID	Same value as in the request transfer.

4.1.3.1 *Service timing requirements*

Applications are recommended to follow the service invocation timing recommendations specified below. Applications are allowed to deviate from these recommendations, provided that every such deviation is explicitly documented.

- Service transfer transmission should be aborted if does not complete in 1 second.
- The client should stop waiting for a response from the server if one has not arrived within 1 second.

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. It is recommended to ensure that the server will be able to process any request in less than 0.5 seconds.

4.1.4 Transfer priority

UAVCAN transfers are prioritized by means of the transfer priority property, which allows at least 8 (eight) different priority levels for all types of transfers (some transports may support more than eight priority levels). Transfers with higher priority levels preempt transfers with lower priority levels, delaying their transmission until there are no more higher priority transfers to exchange.

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 must 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.

4.1.5 Transfer descriptor

Transfer emission and reception processes rely on the concept of *transfer descriptor*.

A transfer descriptor is a set of properties that identify a particular set of transfers that originate from the same source node, share the same port-ID, same kind (message or service), and are addressed to the same destination node (the latter applies only to unicast transfers).

The properties that constitute a transfer descriptor are listed below:

- Transfer kind (message or service).
- Port-ID (subject-ID for message transfers, service-ID for service transfers).
- Source node-ID.
- Destination node-ID (only for service transfers).

For convenience, two derived definitions are introduced. Their objective is to simplify the description of transfer reception and emission logic that appears later in this specification.

Emitted transfer descriptor – a transfer descriptor where the source node-ID equals the local node's ID.

Received transfer descriptor – a transfer descriptor where the destination node-ID equals the local node's ID (for service transfers) or is not defined (for message transfers).

4.1.5.1 Hard real-time considerations

Hard real-time applications require a predictable and deterministic data processing time. The concept of transfer descriptor plays an important role in communication; hence, its contribution to the worst case data processing load should be carefully analyzed.

From the above definition of transfer descriptor it is easy to derive that for any message subject-ID or any service subject-ID the maximum number of transfer descriptors that can be observed by the local node will never exceed the number of nodes on the bus minus one[a]. If the number of nodes on the bus cannot be known in advance, it can be considered to equal 128, which is the maximum node capacity of a UAVCAN bus.

The total number of distinct transfer descriptors that can be observed by a node on any valid UAVCAN bus is a product of the number of distinct port-ID values utilized by the node and the number of other nodes on the bus.

The transport emission and reception logic defined later in this specification relies on data structures indexed by transfer descriptor values. Elements of such structures can be easily accessed via constant-complexity static look-up tables because the worst case number of elements is always statically known.

[a] The local node cannot exchange data with itself, hence minus one.

4.2 Transfer emission

4.2.1 Transfer-ID computation

The *transfer-ID* is a small unsigned integer value in the range from 0 to 31, inclusive, that is provided for every transfer. This value is crucial for many aspects of UAVCAN communication[42]; specifically:

[42] 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 of time. Such use is strongly discouraged because it is

Message sequence monitoring - the continuously increasing transfer-ID allows receiving nodes to detect lost messages and detect when a message stream from any remote node is interrupted.

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

Transport frame deduplication - for single-frame transfers, the transfer-ID allows receiving nodes to work around the transport frame duplication problem[43] (multi-frame transfers combat the frame duplication problem using the toggle bit, which is introduced later).

Multi-frame transfer reassembly - more info is provided in section 4.3.

Automatic management of redundant interfaces - the transfer-ID parameter allows the UAVCAN protocol stack to perform 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 physical transports into a highly reliable single virtual communication channel.

For message transfers and service request transfers the ID value should be computed as described below. For service response transfers this value must be directly copied from the corresponding service request transfer.

Every node that is interested in emitting transfers must maintain a mapping (or a similar functionally equivalent static structure[44]) from emitted transfer descriptors (section 4.1.5) to transfer-ID counters. This mapping is referred to as the *emitted transfer-ID map*.

Whenever a node needs to emit a transfer, it will query its transfer-ID map for the appropriate transfer descriptor. If the map does not contain such entry, a new entry will be created with the transfer-ID counter initialized to zero. The node will use the current value of the transfer-ID from the map for the transfer, and then the value stored in the map will be incremented by one. When the stored transfer-ID exceeds its maximum value, it will roll over to zero.

It is expected that some nodes will need to emit certain transfers aperiodically or on an ad-hoc basis, thereby creating unused entries in the emitted transfer-ID map. If such aperiodic or ad-hoc transfers are of interest, the worst case number of unused entries can be determined statically as a function of the number of port identifiers used and the number of addressed nodes on the bus (the latter applies to services only). Nodes are not allowed to remove any entries from the transfer-ID map as long as they are running.

4.2.2 Single frame transfers

If the size of the entire transfer payload does not exceed the space available for payload in a single transport frame, the whole transfer will be contained in one transport frame. Such transfer is called a *single-frame transfer*.

Single frame transfers are more efficient than multi-frame transfers in terms of throughput, latency, and data overhead.

4.2.3 Multi-frame transfers

Multi-frame transfers are used when the size of the transfer payload exceeds the space available for payload in a single transport frame.

Two new concepts are introduced in the context of multi-frame transfers, both of which are reviewed below in detail:

- Transfer CRC[45].
- Toggle bit.

In order to emit a multi-frame transfer, the node must first compute the CRC for the entirety of the transfer payload. The node appends the resulting CRC value at the end of the transfer payload in the big-endian byte order, and then emits the resulting byte set in chunks as an ordered sequence of transport frames, where the

impossible to detect if one node is more than 32 messages behind another. If temporal synchronization is necessary, explicit time stamping should be used instead.

[43] This is a well-known issue that can be observed with certain transports such as 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 the transfer-ID or the toggle bit (both of which are used in UAVCAN) allow applications to circumvent this problem.

[44] For example, simple static variables.

[45] CRC stands for “cyclic redundancy check”, an error-detecting code added to data transmissions to reduce the likelihood of undetected data corruption.

first transport frame contains the beginning of the payload bytes, and the last transport frame contains the last bytes of the payload (possibly none) plus the transfer CRC.

The data field of all transport frames of a multi-frame transfer, except the last one, should be fully utilized. Applications are allowed to limit the maximum amount of data transferred per transport frame in order to improve the preemption granularity, thus reducing the worst case latency of higher priority transfers[46]. Receiving nodes must be prepared to reconstruct multi-frame transfers that utilize the available payload space partially.

All frames of a multi-frame transfer should be pushed to the transmission queue at once, in the proper order from the first frame to the last frame. Explicit gap time between transport frames belonging to the same transfer should not be introduced; rather, implementations always should strive to minimize it. Re-ordering of frames belonging to the same multi-frame transfer is prohibited.

4.2.3.1 *Transfer CRC*

Transfer CRC allows receiving nodes to ensure that a received multi-frame transfer has been reassembled correctly.

It should be understood that the transfer CRC is not intended for bit-level data integrity checks, as that must be managed by the transport layer implementation on a per-frame basis[47]. As such, the transfer CRC allows receiving nodes to ensure that all of the frames of a multi-frame transfer were received, all of the received frames were reassembled in the correct order, and that all of the received frames belong to the same multi-frame transfer.

The transfer CRC is computed over the entire payload of the transfer. Certain transport implementations[48] may require a short sequence of padding bytes to be added at the end of the transfer payload due to the low granularity of the frame payload length property; in that case, the padding bytes must be included in the CRC computation as well, as if they were part of the useful payload.

The resulting CRC value is appended to the transfer in the *big-endian byte order* (most significant byte first), in order to take advantage of the CRC residue check intrinsic to the used algorithm.

The transfer CRC algorithm specification is provided in the table 4.4.

Table 4.4: Transfer CRC algorithm parameters

Property	Value
Name	CRC-16/CCITT-FALSE
Initial value	FFFF_{16}
Polynomial	1021_{16}
Reverse	No
Output XOR	0
Residue	0
Check	$(49, 50, \ldots, 56, 57) \to \mathrm{29B1}_{16}$

The following code snippet provides a basic implementation of the transfer CRC algorithm in C++.

[46] For example, some CAN FD applications may choose to restrict the maximum payload size to 32 bytes rather than the protocol limit of 64 bytes, as that provides more opportunities for higher-priority frames to take over the bus. The trade-off is that smaller frames lead to higher transfer fragmentation, increase the bus load, and increase the overall average latency.

[47] Bit-level errors at the transport frame level may compromise the error-detecting properties of the transfer CRC.

[48] Such as CAN FD.

```
// UAVCAN transfer CRC algorithm implementation in C++.
// License: CC0, no copyright reserved.

#include <iostream>
#include <cstdint>
#include <cstddef>

class TransferCRC
{
    std::uint16_t value_ = 0xFFFFU;

public:
    void add(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_; }
};

int main()
{
    TransferCRC crc;
    crc.add(reinterpret_cast<const std::uint8_t*>("123456789"), 9);
    std::cout << std::hex << "0x" << crc.get() << std::endl;  // Outputs 0x29B1
    return 0;
}
```

4.2.3.2 *Toggle bit*

The toggle bit is a property defined at the transport frame level. Its purpose is to detect and avoid transport frame duplication errors in multi-frame transfers[49].

The toggle bit of the first transport frame of a multi-frame transfer must be set to one. The toggle bits of the following transport frames of the transfer must alternate, i.e., the toggle bit of the second transport frame must be zero, the toggle bit of the third transport frame must be one, and so on.

For single-frame transfers, the toggle bit must be set to one or removed completely, whichever option works best for the particular transport.

Transfers where the initial value of the toggle bit is zero must be ignored. The initial state of the toggle bit may be inverted in the future revisions of the protocol to facilitate automatic protocol version detection.

4.2.4 Redundant interface support

In configurations with redundant bus interfaces, nodes are required to submit every outgoing transfer to the transmission queues of all available redundant interfaces simultaneously. It is recognized that perfectly simultaneous transmission may not be possible due to different utilization rates of the redundant interfaces and different phasing of their traffic; however, that is not an issue for UAVCAN. If perfectly simultaneous frame submission is not possible, interfaces with lower numerical index should be handled in the first order.

An exception to the above rule applies if the payload of the transfer depends on some properties of the interface through which the transfer is emitted. An example of such a special case is the time synchronization algorithm leveraged by UAVCAN (documented in chapter 5 of the specification).

[49] In single-frame transfers, transport frame deduplication is based on the transfer-ID counter.

Redundant interfaces are used for increased fault tolerance, not for load sharing reasons. Whenever a node is connected to an interface the likelihood of the interface failing is increased. This suggests that backup interfaces may only interconnect with mission-critical equipment, unless a homogeneous network architecture is desired[50]. See section 7.2.1.

4.3 Transfer reception

4.3.1 Transfer-ID comparison

The following explanation relies on the concept of the *transfer-ID forward distance*. Transfer-ID forward distance F is a function of two transfer-ID values, A and B, that defines the number of increment operations that need to be applied to A so that $A' = B$, assuming modulo 32 arithmetic[51]:

$$A + F = B \pmod{32}$$

The *half range* of transfer-ID is 16.

The following code sample provides an example implementation of the transfer-ID comparison algorithm in C++.

```cpp
// UAVCAN transfer-ID forward distance computation algorithm implemented in C++.
// License: CC0, no copyright reserved.

#include <cstdint>
#include <iostream>
#include <cassert>

constexpr std::uint8_t TransferIDBitLength = 5;  // Defined by the specification

[[nodiscard]]
constexpr std::uint8_t computeForwardDistance(std::uint8_t a, std::uint8_t b)
{
    constexpr std::uint8_t MaxValue = (1U << TransferIDBitLength) - 1U;
    assert((a <= MaxValue) && (b <= MaxValue));

    std::int16_t d = static_cast<std::int16_t>(b) - static_cast<std::int16_t>(a);
    if (d < 0)
    {
        d += 1U << TransferIDBitLength;
    }

    assert(d >= 0);
    assert(d <= MaxValue);
    assert(((a + d) & MaxValue) == b);
    return static_cast<std::uint8_t>(d);
}

int main()
{
    assert(0  == computeForwardDistance(0, 0));
    assert(1  == computeForwardDistance(0, 1));
    assert(7  == computeForwardDistance(0, 7));
    assert(0  == computeForwardDistance(7, 7));
    assert(31 == computeForwardDistance(31, 30)); // overflow
    assert(1  == computeForwardDistance(31, 0));  // overflow
    return 0;
}
```

4.3.2 State variables

4.3.2.1 Main principles

Nodes that receive transfers must keep a certain set of state variables for each received transfer descriptor (section 4.1.5).

The set of state variables as documented in the table 4.5 will be referred to as the *receiver state*. For the purposes of this specification, it is assumed that the node will maintain a mapping from transfer descriptors to receiver states, which will be referred to as the *receiver map*. It is understood that implementations might prefer different architectures, which is permitted as long as the resulting behavior of the node observable at the protocol level is functionally equivalent.

[50] Heterogeneous transport configuration complicates the analysis of the network, which might make it impractical in safety-critical deployments. In that case, a simpler configuration where each available redundant bus is connected to every node may be preferred.

[51] For example: $A = 0, B = 0, F \to 0$; $A = 0, B = 5, F \to 5$; $A = 5, B = 0, F \to 27$; $A = 31, B = 30, F \to 31$; $A = 31, B = 0, F \to 1$.

Whenever a node receives a transfer, it will query its receiver map for the matching received transfer descriptor. If the matching state does not exist, the node will add a new receiver state to the map and initialize it as defined in section 4.3.2.2. The node then will proceed with the procedure of *receiver state update*, which is defined in section 4.3.3 for redundant transports and section 4.3.4 for non-redundant transports.

It is expected that some transfers will be aperiodic or ad-hoc, which implies that the receiver map may over time accumulate receiver states that are no longer used. Therefore, nodes are allowed, but not required, to remove any receiver state from the receiver map as soon as the state reaches the *transfer-ID timeout condition*[52], as defined in section 4.3.2.3.

Receiver state can only be modified when a new transport frame of a matching transfer is received. This guarantee simplifies implementation, as it implies that the receiver states will not require any periodic background maintenance activities.

Table 4.5: Transfer reception state variables

State	Description
Transfer payload	Useful payload byte sequence; extended upon reception of new matching transport frames.
Transfer-ID	The transfer-ID value of the next expected transport frame. Section 4.2.1.
Next toggle bit	Expected value of the toggle bit in the next transport frame. Section 4.2.3.2.
Transfer timestamp	The local monotonic timestamp sampled when the first frame of the transfer arrived. Here, "monotonic" means that the reference clock does not change its rate or make leaps.
Interface index	Only in the case of redundant transport interfaces.

4.3.2.2 *Initial state*

The initial state is reached when a new entry of the receiver map is created or an existing entry is reset. Like any other state update, an entry can be created or reset only synchronously with the reception of a matching transport frame.

Upon reset, the receiver state will meet the following conditions:

- The transfer payload buffer is empty.
- The transfer-ID state matches the actual transfer-ID value from the newly received transfer, unless this is a non-first frame of a multi-frame transfer. In the latter case, the transfer-ID state will match the received transfer-ID value incremented by one.
- The toggle bit is set to its initial state (section 4.2.3.2).
- The transfer timestamp matches the reception timestamp from the transport frame.
- The interface index matches the index of the interface that the new frame was received from (for nodes with redundant interfaces only).

A receiver state must be reset when any of the following conditions are met:

- A new receiver state instance is created.
- A transfer-ID timeout condition is reached (section 4.3.2.3).
- A first frame of a transfer (either a multi-frame or a single-frame; in the latter case, the same frame would also be the last frame of the transfer) is received from the same interface as the previous frame (does not apply to non-redundantly interfaced nodes), and the transfer-ID forward distance (section 4.3.1) from the received transfer-ID to the stored transfer-ID is greater than one.
- Only for redundantly interfaced nodes: A first frame of a transfer is received, an interface switchover condition is reached (section 4.3.2.4), and the transfer-ID forward distance from the stored transfer-ID to the received transfer-ID is less than the transfer-ID half range (section 4.3.1).

4.3.2.3 *Transfer-ID timeout condition*

A state is said to have reached the transfer-ID timeout condition if the last matching transfer was seen more than 2 (two) seconds ago. When this condition is reached, the receiver must accept the next transfer disregarding its transfer-ID value.

Nodes are allowed to use different timeout values, if that is believed to benefit the application. If a different timeout value is used, it must be explicitly documented.

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

High timeout values increase the risk of an undetected transfer loss when a remote node suffers an emitted

[52] Such behavior is not recommended for hard real-time applications, where deterministic static look-up tables should be preferred instead.

transfer-ID map state loss (e.g., due to the whole node being restarted). However, the effects of such a transfer loss caused by a loss of state on a remote node are always constrained to the first transfer only.

4.3.2.4 *Interface switchover condition*

This condition is only applicable for configurations with redundant transport interfaces. It means that the node is allowed to receive the next transfer from an interface that is not the same the previous transfer was received from.

The condition is reached when the last matching transfer was successfully received more than T_{switch} seconds ago. The value of T_{switch} should not exceed the reception transfer ID timeout, as defined in section 4.3.2.3, because if T_{switch} were to exceed the transfer-ID timeout, an interface switchover would be performed by the normal receiver state reset procedure, rendering T_{switch} useless.

The actual value of T_{switch} can be either a constant chosen by the designer according to the application requirements (e.g., the maximum recovery time in the event of an interface failure), or the protocol stack can estimate this value automatically by analyzing the transfer intervals.

Nodes are required to let the first interface time out before using the next one because the transfer-ID field is expected to wrap around frequently (every 32 transfers). Different interfaces are expected to exhibit different latencies even in a properly functioning system, especially if the system contains both redundantly-interfaced and non-redundantly-interfaced nodes. If the latency of a backup interface relative to the primary interface exceeds 32 transfer intervals, and receiving nodes were to be allowed to switch between interfaces freely disregarding the timeout, the receiving node would skip the whole period of transfer-IDs (32 transfers will be lost). The problem would primarily affect low-priority transfers where large latencies are more likely.

4.3.3 State update in a redundant interface configuration

The following pseudocode demonstrates the transfer reception process for a configuration with redundant transport interfaces.

```
// Constants:
tid_timeout := 2 seconds;
tid_half_range := 16;
iface_switch_delay := UserDefinedConstant; // Or autodetect

// State variables:
initialized := 0;
payload;
this_transfer_timestamp;
current_transfer_id;
iface_index;
toggle;

function receiveFrame(frame)
{
    // Resolving the state flags:
    tid_timed_out := (frame.timestamp - this_transfer_timestamp) > tid_timeout;
    same_iface := frame.iface_index == iface_index;
    first_frame := frame.start_of_transfer;
    non_wrapped_tid := computeForwardDistance(current_transfer_id, frame.transfer_id) < tid_half_range;
    not_previous_tid := computeForwardDistance(frame.transfer_id, current_transfer_id) > 1;
    iface_switch_allowed := (frame.timestamp - this_transfer_timestamp) > iface_switch_delay;
    // Using the state flags from above, deciding whether we need to reset:
    need_restart :=
        (!initialized) or
        (tid_timed_out) or
        (same_iface and first_frame and not_previous_tid) or
        (iface_switch_allowed and first_frame and non_wrapped_tid);

    if (need_restart)
    {
        initialized := 1;
        iface_index := frame.iface_index;
        current_transfer_id := frame.transfer_id;
        payload.clear();
        toggle := 0;
        if (!first_frame)
        {
            current_transfer_id.increment();
            return;         // Ignore this frame, since the start of the transfer has already been missed
        }
    }

    if (frame.iface_index != iface_index)
    {
        return;  // Wrong interface, ignore
    }

    if (frame.toggle != toggle)
    {
        return;  // Unexpected toggle bit, ignore
    }

    if (frame.transfer_id != current_transfer_id)
    {
        return;  // Unexpected transfer-ID, ignore
    }

    if (first_frame)
    {
        this_transfer_timestamp := frame.timestamp;
    }

    toggle := !toggle;
    payload.append(frame.data);

    if (frame.last_frame)
    {
        // CRC validation for multi-frame transfers is intentionally omitted for brevity
        processTransfer(payload, ...);
        current_transfer_id.increment();
        toggle := 0;
        payload.clear();
    }
}
```

4.3.4 State update in a non-redundant interface configuration

The following pseudocode demonstrates the transfer reception process for a configuration with a non-redundant transport interface. This is a specialization of the more general algorithm defined for redundant transport.

```
// Constants:
tid_timeout := 2 seconds;

// State variables:
initialized := 0;
payload;
this_transfer_timestamp;
current_transfer_id;
toggle;

function receiveFrame(frame)
{
    // Resolving the state flags:
    tid_timed_out := (frame.timestamp - this_transfer_timestamp) > tid_timeout;
    first_frame := frame.start_of_transfer;
    not_previous_tid := computeForwardDistance(frame.transfer_id, current_transfer_id) > 1;
    // Using the state flags from above, deciding whether we need to reset:
    need_restart :=
        (!initialized) or
        (tid_timed_out) or
        (first_frame and not_previous_tid);

    if (need_restart)
    {
        initialized := 1;
        current_transfer_id := frame.transfer_id;
        payload.clear();
        toggle := 0;
        if (!first_frame)
        {
            current_transfer_id.increment();
            return; // Ignore this frame, since the start of the transfer has already been missed
        }
    }

    if (frame.toggle != toggle)
    {
        return;  // Unexpected toggle bit, ignore
    }

    if (frame.transfer_id != current_transfer_id)
    {
        return;  // Unexpected transfer-ID, ignore
    }

    if (first_frame)
    {
        this_transfer_timestamp := frame.timestamp;
    }

    toggle := !toggle;
    payload.append(frame.data);

    if (frame.last_frame)
    {
        // CRC validation for multi-frame transfers is intentionally omitted for brevity
        processTransfer(payload, ...);
        current_transfer_id.increment();
        toggle := 0;
        payload.clear();
    }
}
```

4.4 CAN bus transport layer specification

This section specifies the CAN-based transport layer of UAVCAN.

Here and in the following parts of this section, "CAN" implies both CAN 2.0 and CAN FD, unless specifically noted otherwise. CAN FD should be considered the primary transport protocol.

UAVCAN utilizes only extended CAN frames with 29-bit identifiers. UAVCAN can share the same bus with other protocols based on standard (non-extended) CAN frames with 11-bit identifiers. However, future revisions of UAVCAN may utilize 11-bit identifiers as well; therefore, backward compatibility with other protocols is not guaranteed.

4.4.1 CAN ID structure

UAVCAN utilizes two different CAN ID formats for message transfers and service transfers. The structure is summarized on the figure 4.1.

The fields are described in detail in the following sections. The tables 4.6 and 4.7 summarize the purpose of the fields and their permitted values for message transfers and service transfers, respectively.

Message	Priority			Service, not message	Anonymous message	Subject-ID																Source node-ID							Protocol version
Values	[0,7]			0	𝔹	[0,32767]																[0,127]							1
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	Priority			Service, not message	Request, not response	Service-ID									Destination node-ID							Source node-ID							Protocol version
Values	[0,7]			1	𝔹	[0,511]									[0,127]							[0,127]							1
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 4.1: CAN ID structure

Table 4.6: CAN ID fields for message transfers

Field	Width	Permitted values	Description
Transfer priority	3	[0,7] (any)	Section 4.1.4.
Service not message	1	0	Always zero for message transfers.
Anonymous message	1	{0,1} (any)	Zero for regular (non-anonymous) message transfers. One for anonymous message transfers.
Subject-ID	16	[0,32767]	Subject identifier of the current message. The most significant bit is always zero.
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 4.4.1.2.
Protocol version	1	1	Protocol version, ignore frame (or use according to a newer specification) if this field is not 1.

Table 4.7: CAN ID fields for service transfers

Field	Width	Permitted values	Description
Transfer priority	3	[0,7] (any)	Section 4.1.4.
Service not message	1	1	Always one for service transfers.
Request not response	1	{0,1} (any)	1 for service request, 0 for service response.
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 (i.e., server for requests, client for responses).
Source node-ID	7	[0,127] (any)	Node-ID of the origin (i.e., client for requests, server for responses).
Protocol version	1	1	Protocol version, ignore frame (or use according to a newer specification) if this field is not 1.

4.4.1.1 Transfer priority

Valid values for priority range from 0 to 7, inclusively, where 0 corresponds to the highest priority, and 7 corresponds to the lowest priority.

In multi-frame transfers, the value of the priority field must 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.
2. Anonymous message transfers.
3. Service response transfers.
4. Service request transfers.

Message transfers take precedence over service transfers because message publication is the primary method of communication in UAVCAN networks. Service responses take precedence over service requests in order to make service invocations more atomic and reduce the number of pending states in the system.

Within the same type and the same priority level, transfers are prioritized according to the port-ID[53]: transfers with lower port-ID values preempt those with higher port-ID values.

Mnemonics for transfer priority levels are provided in section 4.1.4, and their mapping to the priority field for CAN bus is shown in the table 4.8.

Table 4.8: CAN transfer priority level mapping

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

4.4.1.2 *Source node-ID field in anonymous message transfers*

CAN bus does not allow different nodes to transmit CAN frames with different data field values under the same CAN ID. Owing to the fact that the CAN ID includes the node-ID value of the transmitting node, this restriction does not affect regular UAVCAN transfers. However, anonymous message transfers would violate this restriction, because they don't have a unique node-ID.

In order to work around this problem, UAVCAN requires that the source node-ID field of anonymous messages[54] is initialized with a pseudorandom *pseudo-ID value*, and defines special logic for handling CAN bus errors during transmission of anonymous frames.

The source of the pseudorandom data used for the pseudo-ID must be likely to produce different values for different payloads[55]. A possible way of initializing the source node pseudo-ID value is to apply the transfer CRC function (as defined in the section 4.2.3.1) to the payload of the anonymous message, and then use the seven least significant bits of the result. Nodes that adopt this approach will be using the same pseudo-ID value for identical messages, which is acceptable since this will not trigger an error on the bus.

Since the pseudo-ID is only seven bits long (128 possible values), a collision where multiple nodes emit CAN frames with different data but same CAN ID is likely to happen despite the randomization measures described earlier. Therefore, the protocol must account for possible errors on the CAN bus triggered by CAN ID collisions. In order to comply with this requirement, UAVCAN requires all nodes to immediately abort transmission of all anonymous transfers once an error on the CAN bus is detected[56]. This measure allows the protocol to prevent the bus deadlock that may occur if the automatic retransmission on bus error is not suppressed. Some method of media access control[57] should be used at the application level for further conflict resolution[58].

Nodes that receive anonymous transfers must ignore their source node identifiers.

[53] Subject-ID or service-ID.

[54] Source node identifier is not defined for anonymous message transfers; see table 4.1.

[55] As explained in section 4.1.2.1, anonymous message transfers are always single-frame transfers.

[56] I.e., if the CAN controller uses automatic retransmission, it should be disabled for anonymous frames.

[57] E.g., CSMA/CD (carrier-sense multiple access with collision detection). 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.

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

4.4.2 CAN frame data

The CAN frame data field may contain the following segments, in the listed order:

1. The useful payload (serialized object). This segment may be empty.
2. Possible padding bytes. Padding bytes may be necessary if the transport layer does not provide byte-level granularity of the data field length (e.g., CAN FD).
3. The last frame of multi-frame transfers always contains the transfer CRC (section 4.2.3.1).
4. The last byte of the data field always contains the *tail byte*.

The segments are documented below in this section.

4.4.2.1 Tail byte

UAVCAN adds one byte of overhead to every CAN frame irrespective of the type of the transfer. The extra byte contains certain metadata for the needs of the transport layer. It is named the *tail byte*, and as the name suggests, it is always situated at the very last byte of the data field of every CAN frame. The tail byte contains four fields: *start of transfer*, *end of transfer*, *toggle bit*, and the transfer-ID (described earlier in the section 4.2.1). The placement of the fields and their usage for single-frame and multi-frame transfers are documented in the table 4.9.

Table 4.9: 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 4.2.3.2.
4	Transfer-ID		
3		Modulo 32 (range [0, 31])	
2		section 4.2.1	
1			
0		(least significant bit)	

The transfer-ID field is populated according to the specification provided in the section 4.2.1. The usage of this field is independent of the type of the transfer.

For single-frame transfers, the fields start-of-transfer, end-of-transfer, and the toggle bit are all set to 1.

For multi-frame transfers, the fields start-of-transfer and end-of-transfer are used to state the boundaries of the current transfer as described in the table. The transfer-ID value is identical for all frames of a multi-frame transfer.

The toggle bit, as described in the section 4.2.3.2, serves two main purposes: CAN frame deduplication and protocol version detection.

4.4.2.2 Padding bytes

Certain transports (such as CAN FD) may not provide byte-level granularity of the CAN data field length. In that case, the useful payload is to be padded with the minimal number of padding bytes required to bring the total length of the CAN data field to a value that can satisfy the length granularity constraints.

When transmitting, each padding byte must be set to $85 = 55_{\text{hex}} = 0101\,0101_{\text{bin}}$. This specific padding value is chosen to avoid stuff bits and to facilitate CAN controller synchronization.

When receiving, the values of the padding bytes must be ignored. In other words, receiving nodes must not make any assumptions about the values of the padding bytes.

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. Therefore, nodes must ignore the trailing unused data bytes at the end of serialized byte sequences; a length mismatch is only to be considered an error if the received byte sequence is shorter than expected by the deserialization routine.

4.4.2.3 Single-frame transfers

For single-frame transfers, the data field of the CAN frame contains two or three segments: the useful payload (which is the serialized object, may be empty), possible padding bytes, and the tail byte (the last byte of the data field).

The resulting data field segmentation is shown in the table 4.10.

Table 4.10: CAN frame data segments for single-frame transfers

Offset	Length	Segment
0	$L_{\text{payload}} \geq 0$	Useful payload (serialized object).
L_{payload}	$L_{\text{padding}} \geq 0$	Padding bytes (if necessary).
$L_{\text{payload}} + L_{\text{padding}}$	1	Tail byte.

4.4.2.4 *Multi-frame transfers*

For multi-frame transfers, all frames except the last one contain only a fragment of the useful payload and the tail byte. Notice that the padding bytes are not used in multi-frame transfers, excepting the last frame; instead of padding, every frame except the last one must use the number of payload bytes that satisfies the length granularity constraints.

The useful payload is fragmented in the forward order: the first CAN frame of a multi-frame transfer contains the beginning of the payload (the first fragment), the following frames contain the subsequent fragments of the useful payload. The last CAN frame of a multi-frame transfer contains the last fragment, unless the last fragment was fully accommodated by the second-to-last CAN frame of the transfer. In the latter case, the last CAN frame will contain only the metadata, as specified below in this section.

Each CAN frame of a multi-frame transfer except the last one should use the maximum CAN data length permitted by the transport. Observe that this is not a hard requirement; some systems that utilize CAN FD may opt for shorter CAN frames in order to reduce the worst case preemption latency, as explained in section 4.2.3. Therefore, UAVCAN implementations must be able to correctly process incoming multi-frame transfers with arbitrary CAN frame data lengths.

The resulting data field segmentation for all frames of a multi-frame transfer except the last one is shown in the table 4.11.

Table 4.11: CAN frame data segments for multi-frame transfers (except the last CAN frame of the transfer)

Offset	Length	Segment
0	$L_{\text{payload}} > 0$	A fragment of the useful payload (serialized object). This segment occupies the entirety of the CAN data field except the last byte, which is used by the tail byte. No padding is allowed.
L_{payload}	1	Tail byte.

The last CAN frame of a multi-frame transfer contains one or two additional segments: the padding bytes (if necessary) and the transfer CRC. The padding rules are identical to those of single-frame transfers. The transfer CRC is to be allocated in the big-endian byte order[59] immediately before the tail byte. The resulting data field segmentation is shown in the table 4.12.

Table 4.12: CAN frame data segments for multi-frame transfers (the last CAN frame of the transfer)

Offset	Length	Segment
0	$L_{\text{payload}} \geq 0$	The last fragment of the useful payload (serialized object).
L_{payload}	$L_{\text{padding}} \geq 0$	Padding bytes (if necessary).
$L_{\text{payload}} + L_{\text{padding}}$	2	Transfer CRC, high byte. Transfer CRC, low byte.
$L_{\text{payload}} + L_{\text{padding}} + 2$	1	Tail byte.

4.4.3 Software design considerations

4.4.3.1 *Ordered transmission*

Multi-frame transfers use identical CAN ID for all frames of the transfer, and UAVCAN requires that all frames of a multi-frame transfer should be transmitted in the correct order. Therefore, the CAN controller driver software must ensure that CAN frames with identical CAN ID values must be transmitted in their order of appearance in the transmission queue. Some CAN controllers will not meet this requirement by default, so the designer must take special care to ensure the correct behavior, and apply workarounds if necessary.

[59] Most significant byte first. This byte order is used to allow faster CRC residue checks; more info in section 4.2.3.1.

4.4.3.2 *Transmission timestamping*

Certain advanced features of UAVCAN may require the driver to timestamp outgoing transport frames, e.g., the time synchronization feature.

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.

4.4.3.3 *Inner priority inversion*

Implementations should take necessary precautions against the problem of inner priority inversion. The following non-normative section provides an overview of the inner priority inversion problem and suggests a possible solution.

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 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 retransmitted, 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 proper 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, must check all other mailboxes as well.
        }
        else
        {
            if not isFramePriorityHigher(newFrame, getFrameFromTxMailbox(i))
            {
                chosen_mailbox = -1
                break   // Denied - must wait until this mailbox has finished transmitting
            }
        }
    }

    return chosen_mailbox
}
```

4.4.3.4 *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.

The quasi-optimal hardware acceptance filter configuration algorithm is defined below.

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:

$$r(M) = \begin{cases} 0 & \mid M < 1 \\ r(\lfloor \frac{M}{2} \rfloor) & \mid M \bmod 2 = 0 \\ r(\lfloor \frac{M}{2} \rfloor) + 1 & \mid M \bmod 2 \neq 0 \end{cases}$$

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 $X_M = m_M(R_A, R_B, M_A, M_B)$, and $X_R = m_R(R_A, R_B, M_A, M_B)$.

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 does not attempt to favor any particular domain or kind of systems among targeted applications.

- Section 5.1 contains a set of mandatory rules that must 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 upon (see chapter 6).

5.1 Application-level requirements

This section describes a set of high-level rules that must 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; and 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 the table 5.1. Unused gaps are reserved for future expansion of adjacent ranges.

Table 5.1: Port identifier distribution

Subject-ID	Service-ID	Purpose
$[0, 24575]$	$[0, 127]$	Unregulated identifiers (both fixed and non-fixed).
$[28672, 29695]$	$[256, 319]$	Non-standard fixed regulated identifiers (i.e., vendor-specific).
$[31744, 32767]$	$[384, 511]$	Standard fixed regulated identifiers.

5.1.2 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 must not be defined inside the standard root namespace[60].

[60] 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 adequately documented.

5.2.1 Node identifier distribution

An overview of related concepts is provided in chapter 2.

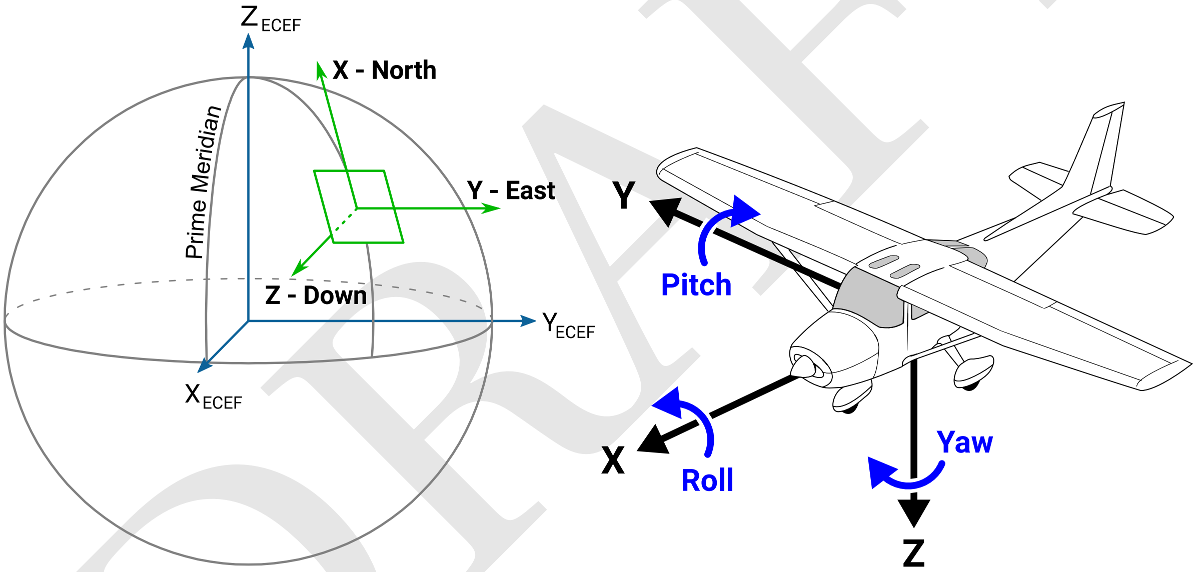
Valid values of node-ID range from 0 up to a transport-specific upper boundary which is guaranteed to be above 127 for any transport.

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 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 must 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.

5.2.2.1 World frame

For world fixed frames, the *North-East-Down* (NED) right handed notation should be preferred.

X — northward;

Y — eastward;

Z — down.

5.2.2.2 Body frame

In relation to a body, the convention is as defined below, right handed. This convention is widely used in aeronautic applications.

X — forward;

Y — right;

Z — down.

5.2.2.3 *Optical frame*

In the case of cameras, the right handed convention specified below should be preferred. It is widely used in various applications involving computer vision systems.

X — right;

Y — down;

Z — towards the scene along the optical axis.

5.2.3 Rotation representation

All applications should represent rotations using quaternions with the elements ordered as follows[61]: 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; such 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, in order to make use of such representations, one often has to convert them to a different form (e.g., quaternion); such conversions are relatively computationally heavy.
> - Rotation matrices are highly redundant.

5.2.4 Matrix representation

5.2.4.1 *General*

Matrices should be represented as flat arrays in the row-major order.

$$\begin{bmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \end{bmatrix} \rightarrow (x_{11}, x_{12}, x_{13}, x_{21}, x_{22}, x_{23})$$

5.2.4.2 *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.

$$\begin{bmatrix} a & b & c \\ b & d & e \\ c & e & f \end{bmatrix} \rightarrow \left(a, b, c, d, e, f\right)$$

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.

$$\begin{bmatrix} a & 0 & 0 \\ 0 & a & 0 \\ 0 & 0 & a \end{bmatrix} \rightarrow a$$

[61] Assuming $w + x\boldsymbol{i} + y\boldsymbol{j} + z\boldsymbol{k}$.

An empty array represents a zero matrix.

5.2.4.3 Covariance matrices

A zero covariance matrix represents an unknown covariance[62].

Infinite error variance means that the associated value is undefined.

5.2.5 Physical quantity representation

All units should be SI[63] 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[64] or scaled SI units[65] should be suffixed with the standard abbreviation of the unit[66] and its metric prefix[67] (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.
```

[62] As described above, an empty array represents a zero matrix, from which follows that an empty array represents unknown covariance.

[63] International System of Units.

[64] E.g., degree Celsius instead of kelvin.

[65] E.g., microsecond instead of second.

[66] E.g., kg for kilogram, J for joule.

[67] 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 or an 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 UAVCAN node must report its status and presence by publishing messages of type `uavcan.node.Heartbeat` (section 6.4.4 on page 82) at a fixed rate specified in the message definition.

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 32085 is provided below. More information is available in section 6.4.4 on page 82.

```
#
# Abstract node status information.
#
# All UAVCAN nodes are required to publish this message periodically.
# This is the only high-level function that must be implemented by all nodes.
#
# The default subject ID 32085 is 111110101010101 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.
#

# The publication period must not exceed this limit.
# The period should not change while the node is running.
uint16 MAX_PUBLICATION_PERIOD = 1   # [second]

# If the last message from the node was received more than this amount of time ago, it should be considered offline.
uint16 OFFLINE_TIMEOUT = 3          # [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.
uint32 uptime                       # [second]

# Abstract node health information. See constants below.
# 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
truncated uint2 health

# The node is functioning properly (nominal).
uint2 HEALTH_NOMINAL  = 0

# A critical parameter went out of range or the node encountered a minor failure that does not prevent
# the subsystem from performing any of its real-time functions.
uint2 HEALTH_ADVISORY = 1

# The node encountered a major failure and is performing in a degraded mode or outside of its designed limitations.
uint2 HEALTH_CAUTION  = 2

# The node suffered a fatal malfunction and is unable to perform its intended function.
uint2 HEALTH_WARNING  = 3

# The current operating mode. See constants below.
#
# The mode OFFLINE can be used for informing other network participants that the sending node has ceased its
# activities or about to do so. In this case, other nodes will not have to wait for the OFFLINE_TIMEOUT to
# expire before detecting that the sending node is no longer available.
#
# Reserved values can be used in future revisions of the specification.
truncated uint3 mode

# Normal operating mode.
uint3 MODE_OPERATIONAL      = 0

# Initialization is in progress; this mode is entered immediately after startup.
uint3 MODE_INITIALIZATION   = 1

# E.g. calibration, self-test, etc.
uint3 MODE_MAINTENANCE      = 2

# New software/firmware is being loaded or the bootloader is running.
```

```
uint3 MODE_SOFTWARE_UPDATE  = 3

# The node is no longer available.
uint3 MODE_OFFLINE          = 7

# Optional, vendor-specific node status code, e.g. a fault code or a status bitmask.
truncated uint19 vendor_specific_status_code

@assert _offset_ % 8 == {0}
@assert _offset_.max <= 56    # Must fit into one CAN 2.0 frame (least capable transport, smallest MTU)
```

5.3.3 Generic node information

The service `uavcan.node.GetInfo` (section 6.4.2 on page 81) 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 0.1 (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 81.

```
#
# Full node info request.
# All of the returned information must be static (unchanged) while the node is running.
# It is highly recommended to support this service on all nodes.
#

---

# The UAVCAN protocol version implemented on this node, both major and minor.
# Not to be changed while the node is running.
Version.1.0 protocol_version

# The version information must not be changed while the node is running.
# The correct hardware version must 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.
Version.1.0 hardware_version
Version.1.0 software_version

# 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.
uint64 software_vcs_revision_id

# The unique node ID is a 128-bit long sequence that is likely to be globally unique per node.
# The vendor must 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 must use the same UID.
uint8[16] unique_id

# Manual serialization note: only fixed-size fields up to this point. The following fields are dynamically sized.
@assert _offset_ == {30 * 8}

# Human-readable non-empty ASCII node name. An empty name is not permitted.
# The name must 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".
void2
uint8[<=50] name

# 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.
void7
uint64[<=1] software_image_crc

# 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.
uint8[<=222] certificate_of_authenticity

@assert _offset_ % 8 == {0}
@assert _offset_.max == (313 * 8)     # At most five CAN FD frames
```

5.3.4 Bus data flow monitoring

The combination of the following three services defined in the namespace `uavcan.node.port` (section 6.5 on page 83) (see table 5.2) enables a highly capable tool of network inspection and monitoring:

- `uavcan.node.port.List` (section 6.5.3 on page 84) — designed for obtaining the full set of subjects and services implemented by the server node.
- `uavcan.node.port.GetInfo` (section 6.5.1 on page 83) — returns the static (unchanging or infrequently changing) information about the selected subject or service.
- `uavcan.node.port.GetStatistics` (section 6.5.2 on page 84) — returns the transfer event counters of the selected subject or service.

The first service `List` allows the caller to construct a list of all subjects and services used by the server node (i.e., the node that the request was sent to). The second service `GetInfo` allows the caller to map each subject or service to a particular data type, and understand the role of the server node in relation to said subject or service (publisher, subscriber, or server).

By comparing the data obtained with the help of these two services from each node on the bus, the caller can reconstruct the data exchange graph for the entire bus, thus enabling advanced network monitoring and diagnostics (assuming that every node on the bus supports the services in question; they are not mandatory).

The last service `GetStatistics` can be used to sample the number of transfers and errors observed on the specified port. When invoked periodically, this service allows the caller to observe the real time intensity of data exchange for each port independently. In combination with the data exchange graph reconstruction described earlier, this service allows the caller to build a sophisticated real-time view of the bus.

Table 5.2: Index of the nested namespace "uavcan.node.port"

Namespace tree	Ver.	FPID	Max bytes	Page sec.	Full name and kind (message/service)
uavcan					
node					
port					
GetInfo	0.1	432	3 54	83 6.5.1	⇄ uavcan.node.port.GetInfo
GetStatistics	0.1	433	3 15	84 6.5.2	⇄ uavcan.node.port.GetStatistics
List	0.1	431	3 257	84 6.5.3	⇄ uavcan.node.port.List
ID	1.0		3	85 6.5.4	✉ uavcan.node.port.ID
ServiceID	1.0		2	85 6.5.5	✉ uavcan.node.port.ServiceID
SubjectID	1.0		2	85 6.5.6	✉ uavcan.node.port.SubjectID

5.3.5 Network-wide time synchronization

UAVCAN provides a simple and robust method of time synchronization[68] 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[69]. 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 95).

An important service type `uavcan.time.GetSynchronizationMasterInfo` (section 6.9.1 on page 95) is designed to provide nodes with information about the currently used time system and related data such as the number of leap seconds.

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 a 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 bytes	Page sec.	Full name and kind (message/service)
uavcan					
time					
GetSynchronizationMasterInfo	0.1	510	0 7	95 6.9.1	⇄ uavcan.time.GetSynchronizationMasterInfo
Synchronization	1.0	31744	7	95 6.9.2	✉ uavcan.time.Synchronization
SynchronizedAmbiguousTimestamp	1.0		3	97 6.9.3	✉ uavcan.time.SynchronizedAmbiguousTimestamp
SynchronizedTimestamp	1.0		7	97 6.9.4	✉ uavcan.time.SynchronizedTimestamp
TimeSystem	0.1		1	98 6.9.5	✉ uavcan.time.TimeSystem

[68] 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.

[69] Proceedings of the 1st international CAN-Conference 94, Mainz, 13.-14. Sep. 1994, CAN in Automation e.V., Erlangen.

5.3.6 Primitive types and physical quantities

The namespaces `uavcan.si` (section 6.13 on page 103) and `uavcan.primitive` (section 6.10 on page 99) included in the standard data type set are designed to provide a very generic and flexible, albeit neither bandwidth-efficient nor low-latency, method of real-time data exchange.

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[70], or supplied by vendors of COTS equipment used in the application.

Vendors of COTS equipment should always ensure that at least 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 it is expected that some of the systems where such COTS nodes are to be integrated into 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 the SI system of units; for example, velocity, mass, energy, angle, time, and so on. The objective of these types is to permit construction of arbitrarily complex distributed control systems without reliance on any particular vendor-specific data types.

Each message type defined in the SI namespace contains a short overflowing timestamp field of type `uavcan.time.SynchronizedAmbiguousTimestamp` (section 6.9.3 on page 97). 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, in order 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 that different applications will opt for different design trade-offs and different tactics to suit their constraints best. 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.

The table 5.4 provides a high-level overview of the SI namespace. Please follow the references for details.

[70] As long as the license permits.

Table 5.4: Index of the nested namespace “uavcan.si”

Namespace tree	Ver.	FPID	Max bytes	Page sec.	Full name and kind (message/service)
uavcan					
si					
acceleration					
Scalar	1.0		7	103 6.13.1	✉ uavcan.si.acceleration.Scalar
Vector3	1.0		15	103 6.13.2	✉ uavcan.si.acceleration.Vector3
angle					
Quaternion	1.0		19	103 6.14.1	✉ uavcan.si.angle.Quaternion
Scalar	1.0		7	104 6.14.2	✉ uavcan.si.angle.Scalar
angular_velocity					
Scalar	1.0		7	104 6.15.1	✉ uavcan.si.angular_velocity.Scalar
Vector3	1.0		15	104 6.15.2	✉ uavcan.si.angular_velocity.Vector3
duration					
Scalar	1.0		7	104 6.16.1	✉ uavcan.si.duration.Scalar
WideScalar	1.0		11	104 6.16.2	✉ uavcan.si.duration.WideScalar
electric_charge					
Scalar	1.0		7	104 6.17.1	✉ uavcan.si.electric_charge.Scalar
electric_current					
Scalar	1.0		7	105 6.18.1	✉ uavcan.si.electric_current.Scalar
energy					
Scalar	1.0		7	105 6.19.1	✉ uavcan.si.energy.Scalar
length					
Scalar	1.0		7	105 6.20.1	✉ uavcan.si.length.Scalar
Vector3	1.0		15	105 6.20.2	✉ uavcan.si.length.Vector3
WideVector3	1.0		27	105 6.20.3	✉ uavcan.si.length.WideVector3
magnetic_field_strength					
Scalar	1.0		7	106 6.21.1	✉ uavcan.si.magnetic_field_strength.Scalar
Vector3	1.0		15	106 6.21.2	✉ uavcan.si.magnetic_field_strength.Vector3
mass					
Scalar	1.0		7	106 6.22.1	✉ uavcan.si.mass.Scalar
power					
Scalar	1.0		7	106 6.23.1	✉ uavcan.si.power.Scalar
pressure					
Scalar	1.0		7	106 6.24.1	✉ uavcan.si.pressure.Scalar
temperature					
Scalar	1.0		7	107 6.25.1	✉ uavcan.si.temperature.Scalar
velocity					
Scalar	1.0		7	107 6.26.1	✉ uavcan.si.velocity.Scalar
Vector3	1.0		15	107 6.26.2	✉ uavcan.si.velocity.Vector3
voltage					
Scalar	1.0		7	107 6.27.1	✉ uavcan.si.voltage.Scalar
volume					
Scalar	1.0		7	107 6.28.1	✉ uavcan.si.volume.Scalar
volumetric_flow_rate					
Scalar	1.0		7	108 6.29.1	✉ uavcan.si.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 usage carries the risk of creating suboptimal interfaces that are difficult to use, validate, and scale.

Normally, the usage 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 (an example of the latter use case is the register protocol described in section 5.3.10).

The table 5.5 provides a high-level overview of the primitive namespace. Please follow the references for details.

Table 5.5: Index of the nested namespace "uavcan.primitive"

Namespace tree	Ver.	FPID	Max bytes	Page sec.	Full name and kind (message/service)
uavcan					
primitive					
Empty	1.0		0	99 6.10.1	✉ uavcan.primitive.Empty
String	1.0		256	99 6.10.2	✉ uavcan.primitive.String
Unstructured	1.0		256	99 6.10.3	✉ uavcan.primitive.Unstructured
array					
Bit	1.0		256	99 6.11.1	✉ uavcan.primitive.array.Bit
Integer8	1.0		256	99 6.11.2	✉ uavcan.primitive.array.Integer8
Integer16	1.0		257	100 6.11.3	✉ uavcan.primitive.array.Integer16
Integer32	1.0		257	100 6.11.4	✉ uavcan.primitive.array.Integer32
Integer64	1.0		257	100 6.11.5	✉ uavcan.primitive.array.Integer64
Natural8	1.0		256	100 6.11.6	✉ uavcan.primitive.array.Natural8
Natural16	1.0		257	100 6.11.7	✉ uavcan.primitive.array.Natural16
Natural32	1.0		257	100 6.11.8	✉ uavcan.primitive.array.Natural32
Natural64	1.0		257	101 6.11.9	✉ uavcan.primitive.array.Natural64
Real16	1.0		257	101 6.11.10	✉ uavcan.primitive.array.Real16
Real32	1.0		257	101 6.11.11	✉ uavcan.primitive.array.Real32
Real64	1.0		257	101 6.11.12	✉ uavcan.primitive.array.Real64
scalar					
Bit	1.0		1	101 6.12.1	✉ uavcan.primitive.scalar.Bit
Integer8	1.0		1	101 6.12.2	✉ uavcan.primitive.scalar.Integer8
Integer16	1.0		2	102 6.12.3	✉ uavcan.primitive.scalar.Integer16
Integer32	1.0		4	102 6.12.4	✉ uavcan.primitive.scalar.Integer32
Integer64	1.0		8	102 6.12.5	✉ uavcan.primitive.scalar.Integer64
Natural8	1.0		1	102 6.12.6	✉ uavcan.primitive.scalar.Natural8
Natural16	1.0		2	102 6.12.7	✉ uavcan.primitive.scalar.Natural16
Natural32	1.0		4	102 6.12.8	✉ uavcan.primitive.scalar.Natural32
Natural64	1.0		8	102 6.12.9	✉ uavcan.primitive.scalar.Natural64
Real16	1.0		2	103 6.12.10	✉ uavcan.primitive.scalar.Real16
Real32	1.0		4	103 6.12.11	✉ uavcan.primitive.scalar.Real32
Real64	1.0		8	103 6.12.12	✉ 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 74), see the table 5.6). All basic file system operations are supported, including file reading and writing, directory listing, metadata retrieval (size, modification time, etc.), moving, renaming, creation, and deletion.

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 5.6: Index of the nested namespace "uavcan.file"

Namespace tree	Ver.	FPID	Max bytes	Page sec.	Full name and kind (message/service)
uavcan					
file					
GetInfo	0.1	405	113 21	74 6.2.1	⇄ uavcan.file.GetInfo
List	0.1	406	121 117	74 6.2.2	⇄ uavcan.file.List
Modify	1.0	407	230 2	75 6.2.3	⇄ uavcan.file.Modify
Read	1.0	408	118 260	75 6.2.4	⇄ uavcan.file.Read
Write	1.0	409	247 2	76 6.2.5	⇄ uavcan.file.Write
Error	1.0		2	76 6.2.6	✉ uavcan.file.Error
Path	1.0		113	76 6.2.7	✉ uavcan.file.Path

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 80). 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[71] 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 process of software update involves the following data types:

- `uavcan.node.ExecuteCommand` (section 6.4.1 on page 80) – used to initiate the software update process.

[71] Or firmware – UAVCAN does not distinguish between the two.

- `uavcan.file.Read` (section 6.2.4 on page 75) – used to transfer the software image file (or multiple files) 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 80). 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[72] 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 commence 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 92); the namespace index is shown in the table 5.7. Data types supported by the register protocol are defined in the nested data structure `uavcan.register.Value` (section 6.8.4 on page 93).

The UAVCAN specification defines several standard naming patterns to facilitate cross-vendor compatibility and provide a framework of common basic functionality.

Table 5.7: Index of the nested namespace "uavcan.register"

Namespace tree	Ver.	FPID	Max bytes		Page sec.		Full name and kind (message/service)
uavcan							
register							
Access	0.1	384	309	265	92	6.8.1	⇄ uavcan.register.Access
List	0.1	385	2	51	93	6.8.2	⇄ uavcan.register.List
Name	0.1		51		93	6.8.3	🖂 uavcan.register.Name
Value	0.1		258		93	6.8.4	🖂 uavcan.register.Value

5.3.11 Diagnostics and event logging

The message type `uavcan.diagnostic.Record` (section 6.1.1 on page 73) is designed to facilitate emission of human-readable diagnostic messages and event logging, both for the needs of real-time display[73] and for long-term storage[74].

5.3.12 Plug-and-play nodes

Every UAVCAN node must have a node-ID that is unique within the network. 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.

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 86) (where *pnp* stands for "plug-and-play") (see table 5.8). The

[72] And, possibly, other interfaces.
[73] E.g., messages displayed to a human operator in real time.
[74] E.g., flight data recording.

protocol is described in the documentation for the data type `uavcan.pnp.NodeIDAllocationData` (section 6.6.1 on page 86).

The plug-and-play node-ID allocation protocol relies on anonymous messages reviewed in the section 4.1.2.1. Remember that the plug-and-play feature is entirely optional and it is expected that some 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"[75] by Diego Ongaro and John Ousterhout.

Table 5.8: Index of the nested namespace "uavcan.pnp"

Namespace tree	Ver.	FPID	Max bytes		Page sec.		Full name and kind (message/service)
`uavcan`							
`pnp`							
`NodeIDAllocationData`	0.1	32741	18		86	6.6.1	✉ `uavcan.pnp.NodeIDAllocationData`
`NodeIDAllocationDataMTU8`	0.1	32742	9		88	6.6.2	✉ `uavcan.pnp.NodeIDAllocationDataMTU8`
`cluster`							
`AppendEntries`	1.0	390	33	5	89	6.7.1	⇄ `uavcan.pnp.cluster.AppendEntries`
`Discovery`	1.0	32740	11		90	6.7.2	✉ `uavcan.pnp.cluster.Discovery`
`RequestVote`	1.0	391	9	5	90	6.7.3	⇄ `uavcan.pnp.cluster.RequestVote`
`Entry`	1.0		22		90	6.7.4	✉ `uavcan.pnp.cluster.Entry`

5.3.13 Internet/LAN forwarding interface

Data types defined in the namespace `uavcan.internet` (section 6.3 on page 77) (see table 5.9) are designed for establishing robust direct connectivity between local UAVCAN nodes and hosts on the Internet or on a local area network (LAN) by means of so called *modem nodes*[76] (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 dependency on application-specific means of communication[77].

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.
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 5.9: Index of the nested namespace "uavcan.internet"

Namespace tree	Ver.	FPID	Max bytes		Page sec.		Full name and kind (message/service)
`uavcan`							
`internet`							
`udp`							
`HandleIncomingPacket`	0.1	500	313	7	77	6.3.1	⇄ `uavcan.internet.udp.HandleIncomingPacket`
`OutgoingPacket`	0.1	32750	307		78	6.3.2	✉ `uavcan.internet.udp.OutgoingPacket`

[75] Proceedings of USENIX Annual Technical Conference, p. 305-320, 2014.

[76] Normally, a modem node would be implemented using on-board cellular, radio frequency, or satellite communication hardware.

[77] Information security and other security-related concerns are outside of the scope of this specification.

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[78] are not included in this list.

Each definition is provided with a length information table for convenience, where the minimum and maximum serialized length is shown in several units: bits, bytes (octets), and transport frames for some of the supported transport protocols. The acronym *MTU* used in the length information tables stands for *maximum transmission unit*, which is the maximum amount of data, in bytes (octets, eight bits per byte), that fits into one transport frame for the specific transport protocol.

The index table 6.1 is provided before the definitions for ease of navigation.

[78] 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 bytes		Page sec.		Full name and kind (message/service)
uavcan							
diagnostic							
Record	1.0	32760	121		73	6.1.1	✉ uavcan.diagnostic.Record
Severity	1.0		1		73	6.1.2	✉ uavcan.diagnostic.Severity
file							
GetInfo	0.1	405	113	21	74	6.2.1	⇄ uavcan.file.GetInfo
List	0.1	406	121	117	74	6.2.2	⇄ uavcan.file.List
Modify	1.0	407	230	2	75	6.2.3	⇄ uavcan.file.Modify
Read	1.0	408	118	260	75	6.2.4	⇄ uavcan.file.Read
Write	1.0	409	247	2	76	6.2.5	⇄ uavcan.file.Write
Error	1.0		2		76	6.2.6	✉ uavcan.file.Error
Path	1.0		113		76	6.2.7	✉ uavcan.file.Path
internet							
udp							
HandleIncomingPacket	0.1	500	313	7	77	6.3.1	⇄ uavcan.internet.udp.HandleIncomingPacket
OutgoingPacket	0.1	32750	307		78	6.3.2	✉ uavcan.internet.udp.OutgoingPacket
node							
ExecuteCommand	0.1	435	115	7	80	6.4.1	⇄ uavcan.node.ExecuteCommand
GetInfo	0.1	430	0	313	81	6.4.2	⇄ uavcan.node.GetInfo
GetTransportStatistics	0.1	434	0	61	81	6.4.3	⇄ uavcan.node.GetTransportStatistics
Heartbeat	1.0	32085	7		82	6.4.4	✉ uavcan.node.Heartbeat
ID	1.0		2		83	6.4.5	✉ uavcan.node.ID
IOStatistics	0.1		15		83	6.4.6	✉ uavcan.node.IOStatistics
Version	1.0		2		83	6.4.7	✉ uavcan.node.Version
port							
GetInfo	0.1	432	3	54	83	6.5.1	⇄ uavcan.node.port.GetInfo
GetStatistics	0.1	433	3	15	84	6.5.2	⇄ uavcan.node.port.GetStatistics
List	0.1	431	3	257	84	6.5.3	⇄ uavcan.node.port.List
ID	1.0		3		85	6.5.4	✉ uavcan.node.port.ID
ServiceID	1.0		2		85	6.5.5	✉ uavcan.node.port.ServiceID
SubjectID	1.0		2		85	6.5.6	✉ uavcan.node.port.SubjectID
pnp							
NodeIDAllocationData	0.1	32741	18		86	6.6.1	✉ uavcan.pnp.NodeIDAllocationData
NodeIDAllocationDataMTU8	0.1	32742	9		88	6.6.2	✉ uavcan.pnp.NodeIDAllocationDataMTU8
cluster							
AppendEntries	1.0	390	33	5	89	6.7.1	⇄ uavcan.pnp.cluster.AppendEntries
Discovery	1.0	32740	11		90	6.7.2	✉ uavcan.pnp.cluster.Discovery
RequestVote	1.0	391	9	5	90	6.7.3	⇄ uavcan.pnp.cluster.RequestVote
Entry	1.0		22		90	6.7.4	✉ uavcan.pnp.cluster.Entry
register							
Access	0.1	384	309	265	92	6.8.1	⇄ uavcan.register.Access
List	0.1	385	2	51	93	6.8.2	⇄ uavcan.register.List
Name	0.1		51		93	6.8.3	✉ uavcan.register.Name
Value	0.1		258		93	6.8.4	✉ uavcan.register.Value
time							
GetSynchronizationMasterInfo	0.1	510	0	7	95	6.9.1	⇄ uavcan.time.GetSynchronizationMasterInfo
Synchronization	1.0	31744	7		95	6.9.2	✉ uavcan.time.Synchronization
SynchronizedAmbiguousTimestamp	1.0		3		97	6.9.3	✉ uavcan.time.SynchronizedAmbiguousTimestamp
SynchronizedTimestamp	1.0		7		97	6.9.4	✉ uavcan.time.SynchronizedTimestamp
TimeSystem	0.1		1		98	6.9.5	✉ uavcan.time.TimeSystem
primitive							
Empty	1.0		0		99	6.10.1	✉ uavcan.primitive.Empty
String	1.0		256		99	6.10.2	✉ uavcan.primitive.String
Unstructured	1.0		256		99	6.10.3	✉ uavcan.primitive.Unstructured
array							
Bit	1.0		256		99	6.11.1	✉ uavcan.primitive.array.Bit
Integer8	1.0		256		99	6.11.2	✉ uavcan.primitive.array.Integer8
Integer16	1.0		257		100	6.11.3	✉ uavcan.primitive.array.Integer16
Integer32	1.0		257		100	6.11.4	✉ uavcan.primitive.array.Integer32
Integer64	1.0		257		100	6.11.5	✉ uavcan.primitive.array.Integer64
Natural8	1.0		256		100	6.11.6	✉ uavcan.primitive.array.Natural8
Natural16	1.0		257		100	6.11.7	✉ uavcan.primitive.array.Natural16
Natural32	1.0		257		100	6.11.8	✉ uavcan.primitive.array.Natural32
Natural64	1.0		257		101	6.11.9	✉ uavcan.primitive.array.Natural64
Real16	1.0		257		101	6.11.10	✉ uavcan.primitive.array.Real16
Real32	1.0		257		101	6.11.11	✉ uavcan.primitive.array.Real32
Real64	1.0		257		101	6.11.12	✉ uavcan.primitive.array.Real64

scalar					
Bit	1.0	1	101	6.12.1	✉ uavcan.primitive.scalar.Bit
Integer8	1.0	1	101	6.12.2	✉ uavcan.primitive.scalar.Integer8
Integer16	1.0	2	102	6.12.3	✉ uavcan.primitive.scalar.Integer16
Integer32	1.0	4	102	6.12.4	✉ uavcan.primitive.scalar.Integer32
Integer64	1.0	8	102	6.12.5	✉ uavcan.primitive.scalar.Integer64
Natural8	1.0	1	102	6.12.6	✉ uavcan.primitive.scalar.Natural8
Natural16	1.0	2	102	6.12.7	✉ uavcan.primitive.scalar.Natural16
Natural32	1.0	4	102	6.12.8	✉ uavcan.primitive.scalar.Natural32
Natural64	1.0	8	102	6.12.9	✉ uavcan.primitive.scalar.Natural64
Real16	1.0	2	103	6.12.10	✉ uavcan.primitive.scalar.Real16
Real32	1.0	4	103	6.12.11	✉ uavcan.primitive.scalar.Real32
Real64	1.0	8	103	6.12.12	✉ uavcan.primitive.scalar.Real64
si					
acceleration					
Scalar	1.0	7	103	6.13.1	✉ uavcan.si.acceleration.Scalar
Vector3	1.0	15	103	6.13.2	✉ uavcan.si.acceleration.Vector3
angle					
Quaternion	1.0	19	103	6.14.1	✉ uavcan.si.angle.Quaternion
Scalar	1.0	7	104	6.14.2	✉ uavcan.si.angle.Scalar
angular_velocity					
Scalar	1.0	7	104	6.15.1	✉ uavcan.si.angular_velocity.Scalar
Vector3	1.0	15	104	6.15.2	✉ uavcan.si.angular_velocity.Vector3
duration					
Scalar	1.0	7	104	6.16.1	✉ uavcan.si.duration.Scalar
WideScalar	1.0	11	104	6.16.2	✉ uavcan.si.duration.WideScalar
electric_charge					
Scalar	1.0	7	104	6.17.1	✉ uavcan.si.electric_charge.Scalar
electric_current					
Scalar	1.0	7	105	6.18.1	✉ uavcan.si.electric_current.Scalar
energy					
Scalar	1.0	7	105	6.19.1	✉ uavcan.si.energy.Scalar
length					
Scalar	1.0	7	105	6.20.1	✉ uavcan.si.length.Scalar
Vector3	1.0	15	105	6.20.2	✉ uavcan.si.length.Vector3
WideVector3	1.0	27	105	6.20.3	✉ uavcan.si.length.WideVector3
magnetic_field_strength					
Scalar	1.0	7	106	6.21.1	✉ uavcan.si.magnetic_field_strength.Scalar
Vector3	1.0	15	106	6.21.2	✉ uavcan.si.magnetic_field_strength.Vector3
mass					
Scalar	1.0	7	106	6.22.1	✉ uavcan.si.mass.Scalar
power					
Scalar	1.0	7	106	6.23.1	✉ uavcan.si.power.Scalar
pressure					
Scalar	1.0	7	106	6.24.1	✉ uavcan.si.pressure.Scalar
temperature					
Scalar	1.0	7	107	6.25.1	✉ uavcan.si.temperature.Scalar
velocity					
Scalar	1.0	7	107	6.26.1	✉ uavcan.si.velocity.Scalar
Vector3	1.0	15	107	6.26.2	✉ uavcan.si.velocity.Vector3
voltage					
Scalar	1.0	7	107	6.27.1	✉ uavcan.si.voltage.Scalar
volume					
Scalar	1.0	7	107	6.28.1	✉ uavcan.si.volume.Scalar
volumetric_flow_rate					
Scalar	1.0	7	108	6.29.1	✉ uavcan.si.volumetric_flow_rate.Scalar

6.1 uavcan.diagnostic

6.1.1 Record

Full message type name: **`uavcan.diagnostic.Record`**

6.1.1.1 Version 1.0, fixed subject ID 32760

Length unit	Bit	Byte (octet)	CAN MTU 8	CAN MTU 64
Message length	[72, 968]	[9, 121]	[2, 18]	[1, 2]

```
#
# Generic human-readable text message for logging and displaying purposes.
# Generally it should be published at the lowest priority level.
#

# Optional timestamp in the network-synchronized time system; zero if undefined.
# The timestamp value conveys the exact moment when the reported event took place.
uavcan.time.SynchronizedTimestamp.1.0 timestamp

# Standard severity, 3 bit wide.
Severity.1.0 severity

# Message text.
# Normally, messages should be kept as short as possible, especially those of high severity.
void6
uint8[<=112] text

@assert _offset_ % 8 == {0}
@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

Length unit	Bit	Byte (octet)	CAN MTU 8	CAN MTU 64
Message length	3	1	1	1

```
#
# Generic message severity representation.
#

# 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 value

# Messages of this severity can be used only during development.
# They must not be used in a fielded operational system.
uint3 TRACE = 0

# Messages that can aid in troubleshooting.
# Messages of this severity and lower should be disabled by default.
uint3 DEBUG = 1

# General informational messages of low importance.
# Messages of this severity and lower should be disabled by default.
uint3 INFO = 2

# General informational messages of high importance.
# Messages of this severity and lower should be disabled by default.
uint3 NOTICE = 3

# Messages reporting abnormalities and warning conditions.
# Messages of this severity and higher should be enabled by default.
uint3 WARNING = 4

# Messages reporting problems and error conditions.
# Messages of this severity and higher should be enabled by default.
uint3 ERROR = 5

# Messages reporting serious problems and critical conditions.
# Messages of this severity and higher should be always enabled.
uint3 CRITICAL = 6

# Notifications of dangerous circumstances that demand immediate attention.
# Messages of this severity should be always enabled.
uint3 ALERT = 7
```

6.2 uavcan.file

6.2.1 GetInfo

Full service type name: **`uavcan.file.GetInfo`**

6.2.1.1 Version 0.1, fixed service ID 405

Length unit	Bit	Byte (octet)	CAN MTU 8	CAN MTU 64
Request length	[8, 904]	[1, 113]	[1, 17]	[1, 2]
Response length	168	21	4	1

```
#
# Information about a remote file system entry (file, directory, etc).
#

Path.1.0 path

---

# Result of the operation
Error.1.0 error

# File size in bytes. Should be set to zero for directories.
truncated uint40 size

# The UNIX Epoch time when the entry was last modified. Zero if unknown.
truncated uint40 unix_timestamp_of_last_modification

# If such entry does not exist, all flags should be cleared/ignored.
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)
void4

# Reserved for future use
void64

@assert _offset_ % 8 == {0}
```

6.2.2 List

Full service type name: **`uavcan.file.List`**

6.2.2.1 Version 0.1, fixed service ID 406

Length unit	Bit	Byte (octet)	CAN MTU 8	CAN MTU 64
Request length	[72, 968]	[9, 121]	[2, 18]	[1, 2]
Response length	[40, 936]	[5, 117]	[1, 17]	[1, 2]

```
#
# 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 must 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.1.0 directory_path

@assert _offset_ % 8 == {0}

---

void32                  # Reserved for future use

# 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.
Path.1.0 entry_base_name

@assert _offset_ % 8 == {0}
```

6.2.3 Modify

Full service type name: **uavcan.file.Modify**

6.2.3.1 Version 1.0, fixed service ID 407

Length unit	Bit	Byte (octet)	CAN MTU 8	CAN MTU 64
Request length	[48, 1840]	[6, 230]	[1, 34]	[1, 4]
Response length	16	2	1	1

```
#
# 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.1.0 source
Path.1.0 destination

@assert _offset_ % 8 == {0}

---

Error.1.0 error

@assert _offset_ % 8 == {0}
```

6.2.4 Read

Full service type name: **uavcan.file.Read**

6.2.4.1 Version 1.0, fixed service ID 408

Length unit	Bit	Byte (octet)	CAN MTU 8	CAN MTU 64
Request length	[48, 944]	[6, 118]	[1, 18]	[1, 2]
Response length	[32, 2080]	[4, 260]	[1, 38]	[1, 5]

```
#
# 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 must 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.1.0 path

@assert _offset_ % 8 == {0}

---

Error.1.0 error

void7
uint8[<=256] data

@assert _offset_ % 8 == {0}
```

6.2.5 Write

Full service type name: **uavcan.file.Write**

6.2.5.1 Version 1.0, fixed service ID 409

Length unit	Bit	Byte (octet)	CAN MTU 8	CAN MTU 64
Request length	[56, 1976]	[7, 247]	[1, 36]	[1, 4]
Response length	16	2	1	1

```
#
# 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.1.0 path

uint8[<=128] data

@assert _offset_ % 8 == {0}
@assert _offset_.max / 8 <= 300

---

Error.1.0 error
```

6.2.6 Error

Full message type name: **uavcan.file.Error**

6.2.6.1 Version 1.0

Length unit	Bit	Byte (octet)	CAN MTU 8	CAN MTU 64
Message length	16	2	1	1

```
#
# 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
```

6.2.7 Path

Full message type name: **uavcan.file.Path**

6.2.7.1 *Version 1.0*

Length unit	Bit	Byte (octet)	CAN MTU 8	CAN MTU 64
Message length	[8, 904]	[1, 113]	[1, 17]	[1, 2]

```
#
# 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.
#

uint8 SEPARATOR = '/'
uint8 MAX_LENGTH = 112

void1
uint8[<=MAX_LENGTH] path

@assert _offset_ % 8 == {0}
```

6.3 uavcan.internet.udp

6.3.1 HandleIncomingPacket

Full service type name: **`uavcan.internet.udp.HandleIncomingPacket`**

6.3.1.1 *Version 0.1, fixed service ID 500*

Length unit	Bit	Byte (octet)	CAN MTU 8	CAN MTU 64
Request length	[32, 2504]	[4, 313]	[1, 45]	[1, 5]
Response length	56	7	1	1

```
#
# 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.
#
# 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.
#


# This field must 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.
uint16 session_id

# 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.
void7
uint8[<=309] payload

@assert _offset_ % 8 == {0}
@assert _offset_.max == (313 * 8)      # At most five CAN FD frames

---

# 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.

void56      # Reserved for future use.
```

6.3.2 OutgoingPacket

Full message type name: **uavcan.internet.udp.OutgoingPacket**

6.3.2.1 Version 0.1, fixed subject ID 32750

Length unit	Bit	Byte (octet)	CAN MTU 8	CAN MTU 64
Message length	[56, 2456]	[7, 307]	[1, 45]	[1, 5]

```
#
# 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 must 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.
#

# 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.
uint32 NAT_ENTRY_MIN_TTL = 86400        # [second]

# 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 session_id

# UDP destination port number.
uint16 destination_port

# Option flags.
bool use_masquerading   # Expect data back (i.e. instruct the modem to use the NAT table)
void7

# 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)
void2
uint8[<=45] destination_address

# 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.
uint8[<256] payload

@assert _offset_ % 8 == {0}
```

6.4 uavcan.node

6.4.1 ExecuteCommand

Full service type name: **uavcan.node.ExecuteCommand**

6.4.1.1 Version 0.1, fixed service ID 435

Length unit	Bit	Byte (octet)	CAN MTU 8	CAN MTU 64
Request length	[24, 920]	[3, 115]	[1, 17]	[1, 2]
Response length	56	7	1	1

```
#
# 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.
#

# 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 must not use identifiers above 32767.
uint16 command

# 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_RESTART = 65535

# Shut down the node; further access will not be possible until the power is turned back on.
uint16 COMMAND_POWER_OFF = 65534

# 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_BEGIN_SOFTWARE_UPDATE = 65533

# 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_FACTORY_RESET = 65532

# Cease activities immediately, enter a safe state until restarted.
# Further operation may no longer be possible until a restart command is executed.
uint16 COMMAND_EMERGENCY_STOP = 65531

# 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.
uint16 COMMAND_STORE_PERSISTENT_STATES = 65530

# 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
void1
uint8[<=uavcan.file.Path.1.0.MAX_LENGTH] parameter

@assert _offset_ % 8 == {0}
@assert _offset_.max <= (124 * 8)     # Two CAN FD frames max

---

# The result of the request.
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

# Reserved for future use
void48
```

6.4.2 GetInfo

Full service type name: **`uavcan.node.GetInfo`**

6.4.2.1 Version 0.1, fixed service ID 430

Length unit	Bit	Byte (octet)	CAN MTU 8	CAN MTU 64
Request length	0	0	1	1
Response length	[264, 2504]	[33, 313]	[5, 45]	[1, 5]

```
#
# Full node info request.
# All of the returned information must be static (unchanged) while the node is running.
# It is highly recommended to support this service on all nodes.
#

---

# The UAVCAN protocol version implemented on this node, both major and minor.
# Not to be changed while the node is running.
Version.1.0 protocol_version

# The version information must not be changed while the node is running.
# The correct hardware version must 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.
Version.1.0 hardware_version
Version.1.0 software_version

# 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.
uint64 software_vcs_revision_id

# The unique node ID is a 128-bit long sequence that is likely to be globally unique per node.
# The vendor must 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 must use the same UID.
uint8[16] unique_id

# Manual serialization note: only fixed-size fields up to this point. The following fields are dynamically sized.
@assert _offset_ == {30 * 8}

# Human-readable non-empty ASCII node name. An empty name is not permitted.
# The name must 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".
void2
uint8[<=50] name

# 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.
void7
uint64[<=1] software_image_crc

# 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.
uint8[<=222] certificate_of_authenticity

@assert _offset_ % 8 == {0}
@assert _offset_.max == (313 * 8)      # At most five CAN FD frames
```

6.4.3 GetTransportStatistics

Full service type name: **`uavcan.node.GetTransportStatistics`**

6.4.3.1 Version 0.1, fixed service ID 434

Length unit	Bit	Byte (octet)	CAN MTU 8	CAN MTU 64
Request length	0	0	1	1
Response length	[128, 488]	[16, 61]	[3, 9]	1

```
#
# Returns a set of general low-level transport statistical counters.
# Servers are not required to sample the data atomically.
#

---

# UAVCAN supports up to triply modular redundant interfaces.
uint8 MAX_PHYSICAL_INTERFACES = 3

# UAVCAN transfer performance statistics:
# the number of UAVCAN transfers sent, received, and failed.
IOStatistics.0.1 transfer_statistics

# Physical 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.
void6
IOStatistics.0.1[<=MAX_PHYSICAL_INTERFACES] physical_interface_statistics

@assert _offset_ % 8 == {0}
@assert _offset_.max <= (63 * 8)      # One CAN FD frame
```

6.4.4 Heartbeat

Full message type name: **`uavcan.node.Heartbeat`**

6.4.4.1 Version 1.0, fixed subject ID 32085

Length unit	Bit	Byte (octet)	CAN MTU 8	CAN MTU 64
Message length	56	7	1	1

```
#
# Abstract node status information.
#
# All UAVCAN nodes are required to publish this message periodically.
# This is the only high-level function that must be implemented by all nodes.
#
# The default subject ID 32085 is 111110101010101 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.
#

# The publication period must not exceed this limit.
# The period should not change while the node is running.
uint16 MAX_PUBLICATION_PERIOD = 1   # [second]

# If the last message from the node was received more than this amount of time ago, it should be considered offline.
uint16 OFFLINE_TIMEOUT = 3          # [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.
uint32 uptime                       # [second]

# Abstract node health information. See constants below.
# 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
truncated uint2 health

# The node is functioning properly (nominal).
uint2 HEALTH_NOMINAL   = 0

# A critical parameter went out of range or the node encountered a minor failure that does not prevent
# the subsystem from performing any of its real-time functions.
uint2 HEALTH_ADVISORY = 1

# The node encountered a major failure and is performing in a degraded mode or outside of its designed limitations.
uint2 HEALTH_CAUTION   = 2

# The node suffered a fatal malfunction and is unable to perform its intended function.
uint2 HEALTH_WARNING   = 3

# The current operating mode. See constants below.
#
# The mode OFFLINE can be used for informing other network participants that the sending node has ceased its
# activities or about to do so. In this case, other nodes will not have to wait for the OFFLINE_TIMEOUT to
# expire before detecting that the sending node is no longer available.
#
# Reserved values can be used in future revisions of the specification.
truncated uint3 mode

# Normal operating mode.
uint3 MODE_OPERATIONAL      = 0

# Initialization is in progress; this mode is entered immediately after startup.
uint3 MODE_INITIALIZATION   = 1

# E.g. calibration, self-test, etc.
uint3 MODE_MAINTENANCE      = 2

# New software/firmware is being loaded or the bootloader is running.
uint3 MODE_SOFTWARE_UPDATE  = 3

# The node is no longer available.
```

```
uint3 MODE_OFFLINE          = 7

# Optional, vendor-specific node status code, e.g. a fault code or a status bitmask.
truncated uint19 vendor_specific_status_code

@assert _offset_ % 8 == {0}
@assert _offset_.max <= 56    # Must fit into one CAN 2.0 frame (least capable transport, smallest MTU)
```

6.4.5 ID

Full message type name: **`uavcan.node.ID`**

6.4.5.1 Version 1.0

Length unit	Bit	Byte (octet)	CAN MTU 8	CAN MTU 64
Message length	16	2	1	1

```
#
# 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

@assert _offset_ == {16}
```

6.4.6 IOStatistics

Full message type name: **`uavcan.node.IOStatistics`**

6.4.6.1 Version 0.1

Length unit	Bit	Byte (octet)	CAN MTU 8	CAN MTU 64
Message length	120	15	3	1

```
#
# A standard set of generic input/output statistical counters that should never overflow (in a realistic scenario).
# If a 40-bit counter is incremented every millisecond, it will overflow in ~35 years.
#

truncated uint40 num_emitted
truncated uint40 num_received
truncated uint40 num_errored
```

6.4.7 Version

Full message type name: **`uavcan.node.Version`**

6.4.7.1 Version 1.0

Length unit	Bit	Byte (octet)	CAN MTU 8	CAN MTU 64
Message length	16	2	1	1

```
#
# A shortened semantic version representation: only major and minor.
# The protocol generally does not concern itself with the patch version.
#

uint8 major
uint8 minor
```

6.5 uavcan.node.port

6.5.1 GetInfo

Full service type name: **`uavcan.node.port.GetInfo`**

6.5.1.1 Version 0.1, fixed service ID 432

Length unit	Bit	Byte (octet)	CAN MTU 8	CAN MTU 64
Request length	24	3	1	1
Response length	[32, 432]	[4, 54]	[1, 8]	1

```
#
# This service is used to obtain information about a port (either message or service).
# It can be used for advanced network inspections, e.g. for building computational graph maps.
#

# The port of interest; can be either a service or a message.
void7
```

```
ID.1.0 port_id

@assert _offset_ == {24}

---

# Flags describing the current port.
# Note that a message port can be both subscriber and publisher at the same time.
bool is_subscriber      # Message input port
bool is_publisher       # Message output port
bool is_server          # Server port
void5

# If such port exists, this field must contain the full name of its data type.
# If the requested port does not exist, this field must be empty.
# If this field is empty, the caller must ignore all other fields.
void2
uint8[<=50] data_type_full_name

# The version numbers of the data type used at this port.
uavcan.node.Version.1.0 data_type_version

@assert _offset_ % 8 == {0}
```

6.5.2 GetStatistics

Full service type name: **`uavcan.node.port.GetStatistics`**

6.5.2.1 Version 0.1, fixed service ID 433

Length unit	Bit	Byte (octet)	CAN MTU 8	CAN MTU 64
Request length	24	3	1	1
Response length	120	15	3	1

```
#
# This service is used to obtain statistical metrics of a port (either message or service).
# It can be used for advanced network inspections, e.g. for building real-time traffic maps.
#

# The port of interest; can be either a service or a message.
void7
ID.1.0 port_id

@assert _offset_ == {24}

---

# For messages, "emitted" applies to publications, and "received" applies to subscribers.
#
# For services, "emitted" applies to responses, and "received" applies to requests. Normally, they
# should be equal; however, under certain circumstances the number of responses ("emitted") can be
# lower, e.g. if the server could not or chose not to send a response.
#
# The error counter represents how many transfers could not be successfully received or emitted due to
# an error of any kind anywhere in the processing pipeline, excluding application-level errors.
# For example, if the message could not be decoded because of a data type compatibility problem,
# such incident should be reported here via this field. On the other hand, if the message or service
# call was processed properly, but the application could not act upon that data, it should not be
# reported here because that problem should be attributed to a different level of abstraction.
uavcan.node.IOStatistics.0.1 statistics

@assert _offset_ == {15 * 8}      # 15 bytes max; this service is optimized for high-frequency polling
```

6.5.3 List

Full service type name: **`uavcan.node.port.List`**

6.5.3.1 Version 0.1, fixed service ID 431

Length unit	Bit	Byte (octet)	CAN MTU 8	CAN MTU 64
Request length	24	3	1	1
Response length	[8, 2056]	[1, 257]	[1, 37]	[1, 5]

```
#
# Returns a subset of all ports of the specified kind (service or message) on the remote node.
# As the number of ports may exceed the capacity of the output array, the server will return only those which are
# greater than the number specified in the request. The caller should sweep the lower boundary upwards until
# the size of the returned array is less than its capacity. For example:
# 1. port_id_lower_boundary =    0; port_ids = [42, 567, ... 920]    (=128 elements, continue)
# 2. port_id_lower_boundary =  920; port_ids = [1052,    ... 5049]   (=128 elements, continue)
# 3. port_id_lower_boundary = 5049; port_ids = [6973,    ..., 24593] (<128 elements, stop)
#

# The server will return as many IDs as possible which are numerically greater than this lower boundary,
# and which are of the same kind as specified in the request (i.e., if the lower boundary is a service ID, only
# service IDs will be returned).
#
# If the number of matching (i.e., greater than this limit) IDs exceeds the capacity of the output array,
# the caller will need to repeat the call with this boundary set to the maximum value found in the returned array.
# The calls should be repeated while advancing the boundary as described until the returned array is not full
# (e.g. the number of elements is less than its capacity, possibly zero).
#
```

```
# The server is NOT required to ensure any ordering in the output array, meaning that the caller will have to
# search through the array in order to find the greatest value in it.
#
# The server is REQUIRED to ensure that every element in the returned array is unique.
#
# The server is REQUIRED to ensure that by performing the above actions the client will end up with a full set of
# port IDs, assuming that the set is not modified between invocations. This implicitly requires that there must be
# no port ID that is lower than the greatest value in the returned array that is not listed in the array, because
# in that case the client will never be able to obtain that value.
void7
ID.1.0 port_id_lower_boundary

@assert _offset_ == {24}

---

# See above for the full description of the logic.
uint16[<=128] port_ids

@assert _offset_ % 8 == {0}
```

6.5.4 ID

Full message type name: **`uavcan.node.port.ID`**

6.5.4.1 Version 1.0

Length unit	Bit	Byte (octet)	CAN MTU 8	CAN MTU 64
Message length	17	3	1	1

```
#
# 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
ServiceID.1.0 service_id
SubjectID.1.0 subject_id
@assert _offset_ == {17}
```

6.5.5 ServiceID

Full message type name: **`uavcan.node.port.ServiceID`**

6.5.5.1 Version 1.0

Length unit	Bit	Byte (octet)	CAN MTU 8	CAN MTU 64
Message length	16	2	1	1

```
#
# Service ID. The ranges are defined by the specification.
#

uint9 MAX_UNREGULATED_ID = 127

void7
uint9 value

@assert _offset_ == {16}
```

6.5.6 SubjectID

Full message type name: **`uavcan.node.port.SubjectID`**

6.5.6.1 Version 1.0

Length unit	Bit	Byte (octet)	CAN MTU 8	CAN MTU 64
Message length	16	2	1	1

```
#
# Subject ID. The ranges are defined by the specification.
#

uint16 MAX_UNREGULATED_ID = 24575

void1
uint15 value

@assert _offset_ == {16}
```

6.6 uavcan.pnp

6.6.1 NodeIDAllocationData

Full message type name: `uavcan.pnp.NodeIDAllocationData`

6.6.1.1 Version 0.1, fixed subject ID 32741

Length unit	Bit	Byte (octet)	CAN MTU 8	CAN MTU 64
Message length	144	18	3	1

```
#
# In order to be able to operate in a UAVCAN network, a node must 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.
#
# An allocated node-ID should not be persistent. This means that if a node is configured to use plug-and-play node-ID
# allocation, it must perform a new allocation every time it is started or rebooted. The allocated node-ID value
# should not be stored on the node itself, because there exist edge cases that may lead to node-ID conflicts under
# certain circumstances (reviewed later).
#
# 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 must 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 must support the service uavcan.node.GetInfo, and they must 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
# must 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 must 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 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 CAN 2.0 there is a dedicated message definition that takes into
# account the limitations of that transport (namely, the unique-ID value is replaced with a short hash in order to
# fit the data into one low-MTU single-frame transfer).
#
# When a node needs to request a PnP node-ID, it will transmit an anonymous message transfer of this type. The pseudo
# node-ID value of the anonymous transfer should be populated with pseudorandom data as described in the UAVCAN
# transport layer specification. Since pseudo node-ID collisions are likely to happen, nodes that are requesting
# PnP allocations need to be able to handle them correctly. Hence, a CSMA/CD collision resolution protocol is defined,
# which utilizes a randomized request period value referred to as Trequest.
#
# 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 must 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 must 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 simple 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:
#        node_id   - the preferred node-ID, or 125 if the allocatee doesn't have any specific preference.
#        unique_id - the 128-bit unique-ID of the allocatee, same value that is reported via uavcan.node.GetInfo.
#
#   Rule C. On any allocation message, even if other rules also match:
#     1. Request Timer restarts with a random interval of Trequest (chosen anew) (this is related to CSMA/CD).
#
#   Rule D. 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.
#

# 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, which is 125,
# because 126 and 127 are reserved for network maintenance tools. Requesting 126 and 127 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. In pseudocode:
#
#   uint8_t findFreeNodeID(const uint8_t preferred)     // Returns >127 on failure (valid node-ID range from 0 to 127)
#   {
#       uint8_t candidate = preferred;
#       while (candidate <= 125)
#       {
#           if (!isOccupied(candidate)) { return candidate; }
#           candidate++;
#       }
#       candidate = preferred;
#       while (candidate > 0)
#       {
#           if (!isOccupied(candidate)) { return candidate; }
#           candidate--;
#       }
#       return isOccupied(0) ? 255 : 0;
#   }
uavcan.node.ID.1.0 node_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 must 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 (except for the CSMA/CD clause, see rule C). If the IDs
# match, then the field node_id contains the allocated node-ID value for this node.
uint8[16] unique_id

@assert _offset_ % 8 == {0}
@assert _offset_.max / 8 == 18
```

6.6.2 NodeIDAllocationDataMTU8

Full message type name: **`uavcan.pnp.NodeIDAllocationDataMTU8`**

6.6.2.1 Version 0.1, fixed subject ID 32742

Length unit	Bit	Byte (octet)	CAN MTU 8	CAN MTU 64
Message length	[56, 72]	[7, 9]	[1, 2]	1

```
#
# The suffix "MTU8" indicates that this message is made to be compatible with transports where the maximum
# transmission unit size is only 8 bytes, such as CAN 2.0. For other kinds of transports there is a more general,
# more capable definition NodeIDAllocationData; it can't be used with low-MTU transports, hence the need for this
# special case. The PnP protocol is described in the documentation for the general case data type definition. The
# documentation provided here builds upon the general case, so read that first please.
#
# The motivation for the difference between this small-MTU case and the general case is that the full 128-bit
# unique-ID can't be accommodated in a small-MTU single-frame anonymous message transfer. The solution is to replace
# the full 128-bit ID with a smaller 55-bit hash of it, and remove support for preferred node-ID value in the
# allocation requests in order to save space.
#
# The 55-bit hash is obtained by applying an atbitrary hash function to the unique-ID that outputs at least 55 bit of
# data. For example, it could be the standard CRC-64WE function where only the lowest 55 bit of the result are used.
# The hash function must always produce the same hash value for the same unique-ID value, and it must be likely to
# produce a different hash value for any other unique-ID value.
#
# Allocators that support both the general case and the small-MTU case should maintain the same allocation table
# for both. Requests received via the small-MTU message obviously do not contain the full unique-ID; the allocators
# are recommended to extend the small 55-bit hash with zeros upwards (where "upwards" means that the most significant
# bits of the unique-ID will contain zeros, and the least significant 55 bits will contain the 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 transport is used, but they are considered tolerable.
#
# 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 simple 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 55-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) (this is related to CSMA/CD).
#
#   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 55-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.
#

```

```
# An arbitrary 55-bit hash of the unique-ID of the local node.
truncated uint55 unique_id_hash

# Must be empty in request messages.
# Must be populated in response messages.
uavcan.node.ID.1.0[<=1] allocated_node_id

@assert _offset_.min == 56        # Plus the tail byte yields 8 bytes; this is for requests only
@assert _offset_.max == 72        # 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

Length unit	Bit	Byte (octet)	CAN MTU 8	CAN MTU 64
Request length	[88, 264]	[11, 33]	[2, 5]	1
Response length	33	5	1	1

```
#
# 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 must 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 must 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 must 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 must
# 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.
#

# 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.
uint8 DEFAULT_MIN_ELECTION_TIMEOUT = 2      # [second]
uint8 DEFAULT_MAX_ELECTION_TIMEOUT = 4      # [second]

```

```
# Refer to the Raft paper for explanation.
uint32 term
uint32 prev_log_term
uint8 prev_log_index
uint8 leader_commit

# 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.
void7
Entry.1.0[<=1] entries

@assert _offset_ % 8 == {0}

---

# Refer to the Raft paper for explanation.
uint32 term
bool success
```

6.7.2 Discovery

Full message type name: **uavcan.pnp.cluster.Discovery**

6.7.2.1 Version 1.0, fixed subject ID 32740

Length unit	Bit	Byte (octet)	CAN MTU 8	CAN MTU 64
Message length	[8, 88]	[1, 11]	[1, 2]	1

```
#
# 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 must publish a Discovery message once. This condition
# allows other allocators to quickly re-discover the cluster after a restart.
#

# This message should be broadcasted by the allocator at this interval until all other allocators are discovered.
uint8 BROADCASTING_PERIOD = 1      # [second]

# The redundant allocator cluster cannot contain more than 5 allocators.
uint3 MAX_CLUSTER_SIZE = 5

# The number of allocators in the cluster as configured on the sender.
# This value must be the same across all allocators.
uint3 configured_cluster_size

# Node-IDs of the allocators that are known to the publishing allocator, including the publishing allocator itself.
void2
uavcan.node.ID.1.0[<=5] known_nodes

@assert _offset_ % 8 == {0}
```

6.7.3 RequestVote

Full service type name: **uavcan.pnp.cluster.RequestVote**

6.7.3.1 Version 1.0, fixed service ID 391

Length unit	Bit	Byte (octet)	CAN MTU 8	CAN MTU 64
Request length	72	9	2	1
Response length	33	5	1	1

```
#
# This type is a part of the Raft consensus algorithm. Please refer to the type AppendEntries for details.
#

# Refer to the Raft paper for explanation.
uint32 term
uint32 last_log_term
uint8 last_log_index

---

# Refer to the Raft paper for explanation.
uint32 term
bool vote_granted
```

6.7.4 Entry

Full message type name: **uavcan.pnp.cluster.Entry**

6.7.4.1 Version 1.0

Length unit	Bit	Byte (octet)	CAN MTU 8	CAN MTU 64
Message length	176	22	4	1

```
#
# 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.

@assert _offset_ % 8 == {0}
```

6.8 uavcan.register

6.8.1 Access

Full service type name: **`uavcan.register.Access`**

6.8.1.1 Version 0.1, fixed service ID 384

Length unit	Bit	Byte (octet)	CAN MTU 8	CAN MTU 64
Request length	[16, 2472]	[2, 309]	[1, 45]	[1, 5]
Response length	[64, 2120]	[8, 265]	[2, 39]	[1, 5]

```
#
# This service is used to write and read a register. Write is optional, it is performed if the value provided in
# the request is not empty.
#
# 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 value to the proper 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 bus.
#
# If only read is desired, but not write, the caller must provide a value of type Empty. That will signal the server
# that the write operation must 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.
#
# Registers must 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 must 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.
#
# In order to discover the type of a register, the caller needs to invoke this service with the write request set
# to Empty. The response will contain the current value of the register with the type information (which doesn't
# change).
#
# Register name may contain:
#   - All ASCII alphanumeric characters (a-z, A-Z, 0-9)
#   - Dot (.)
#   - Underscore (_)
#   - Minus (-)
# All other printable non-whitespace ASCII characters are reserved for standard functions;
# they may appear in register names to support such standard functions defined by the register protocol,
# but they cannot be freely used by applications outside of such standard functions.
#
# 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 following table specifies the register name patterns that are reserved by the specification for
# common functions. Implementers should always follow these conventions whenever applicable.
# The table uses the following abbreviations for register flags:
#   - M - mutable, I - immutable
#   - P - persistent, V - volatile
#
#   Name pattern          | Flags | Type       | Purpose
# ------------------------+-------+------------+-----------------------------------------------------------------------
#   uavcan.node_id        | MP    | uint16     | 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. The factory-default value should be 65535.
# ------------------------+-------+------------+-----------------------------------------------------------------------
#
#

# The name of the accessed register. Must not be empty.
# Use the List service to obtain the list of registers on the node.
Name.0.1 name

@assert _offset_ % 8 == {0}

# Value to be written. Empty if no write is required.
void4
Value.0.1 value

```

```
@assert _offset_.min % 8 == 0
@assert _offset_.max % 8 == 0

---

# The moment of time when the register was read (not written).
# Zero if the server does not support timestamping.
uavcan.time.SynchronizedTimestamp.1.0 timestamp

# 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 mutable

# 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.
bool persistent

void2

# 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.
Value.0.1 value

@assert _offset_.min % 8 == 0
@assert _offset_.max % 8 == 0
```

6.8.2 List

Full service type name: **uavcan.register.List**

6.8.2.1 Version 0.1, fixed service ID 385

Length unit	Bit	Byte (octet)	CAN MTU 8	CAN MTU 64
Request length	16	2	1	1
Response length	[8, 408]	[1, 51]	[1, 8]	1

```
#
# 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.
#

uint16 index

---

# Empty name in response means that the index is out of bounds, i.e., discovery is finished.
Name.0.1 name

@assert _offset_ % 8 == {0}
```

6.8.3 Name

Full message type name: **uavcan.register.Name**

6.8.3.1 Version 0.1

Length unit	Bit	Byte (octet)	CAN MTU 8	CAN MTU 64
Message length	[8, 408]	[1, 51]	[1, 8]	1

```
#
# An ASCII register name.
#

void2
uint8[<=50] name

@assert _offset_ % 8 == {0}
```

6.8.4 Value

Full message type name: **uavcan.register.Value**

6.8.4.1 Version 0.1

Length unit	Bit	Byte (octet)	CAN MTU 8	CAN MTU 64
Message length	[4, 2060]	[1, 258]	[1, 38]	[1, 5]

```
#
# 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.
#
# In order to ensure a byte alignment of the nested arrays, the outer type must pad this union with 4 bits.
#

@union                                          # The tag is 4 bits wide.

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]

# Tag 15 is reserved

@assert _offset_.min == 4                 # Empty and the tag
@assert _offset_.max == 257 * 8 + 4       # 257 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

Length unit	Bit	Byte (octet)	CAN MTU 8	CAN MTU 64
Request length	0	0	1	1
Response length	56	7	1	1

```
#
# 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 must
# use the same time system always.
#

---

# 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.
float32 error_variance      # [second^2]

# Time system currently in use by the master.
# Cannot be changed while the network is operating.
TimeSystem.0.1 time_system

# The fixed difference, in seconds, between TAI and GPS time. Does not change ever.
uint12 TIME_DIFFERENCE_TAI_MINUS_GPS = 19    # [second]

# The current difference between TAI and UTC (a.k.a. leap seconds), if known.
# If unknown, set to zero. This value can change states between known and unknown while the master is running,
# depending on its ability to obtain robust values from external sources.
# This value can change twice a year, possibly while the system is running; https://en.wikipedia.org/wiki/Leap_second
# Since Earth is decelerating, this value can be only positive. Do not use outside Earth.
uint12 TIME_DIFFERENCE_TAI_MINUS_UTC_UNKNOWN = 0
uint12 time_difference_tai_minus_utc

# Reserved for future use
void8

@assert _offset_ % 8 == {0}
@assert _offset_.max <= 56   # It is nice to have the response fit into one transport frame, although not required.
```

6.9.2 Synchronization

Full message type name: **`uavcan.time.Synchronization`**

6.9.2.1 Version 1.0, fixed subject ID 31744

Length unit	Bit	Byte (octet)	CAN MTU 8	CAN MTU 64
Message length	56	7	1	1

```
#
# 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 must
# 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 must 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 must 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 must 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)
#

# Publication period limits.
# A master should not change its publication period while running.
uint8 MAX_PUBLICATION_PERIOD = 1                 # [second]

# 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.
uint8 PUBLISHER_TIMEOUT_PERIOD_MULTIPLIER = 3

# 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 must be zero.
truncated uint56 previous_transmission_timestamp_microsecond

@assert _offset_ % 8 == {0}
@assert _offset_.max <= 56    # Must fit into one CAN 2.0 frame (least capable transport, smallest MTU)
```

6.9.3 SynchronizedAmbiguousTimestamp

Full message type name: **`uavcan.time.SynchronizedAmbiguousTimestamp`**

6.9.3.1 Version 1.0

Length unit	Bit	Byte (octet)	CAN MTU 8	CAN MTU 64
Message length	24	3	1	1

```
#
# Nested data type used for representing a network-wide synchronized timestamp with 10 (ten) microsecond resolution.
# Unlike the wider alternative, this timestamp value expires in ~167 seconds due to integer overflow of the
# microsecond timestamp field: 2**24 / 1e5 = 167.77216 seconds.
# This data type should not be used unless network throughput and latency are of utmost importance,
# because it is inherently unsafe: a delay of more than the specified above limit will invalidate the timestamp
# and its users will not be able to detect that. Instead, the full non-overflowing non-ambiguous timestamp
# should be preferred.
#

# Zero means that the time is not known.
uint24 UNKNOWN = 0

# Expiration timeout.
uint28 OVERFLOW_PERIOD_us = 167772160   # [microsecond]

# Overflowing sub-timestamp, 10 microseconds per least significant bit.
# Overflows every ~167 seconds.
truncated uint24 decamicrosecond        # [second*10^-5]
```

6.9.4 SynchronizedTimestamp

Full message type name: **`uavcan.time.SynchronizedTimestamp`**

6.9.4.1 Version 1.0

Length unit	Bit	Byte (octet)	CAN MTU 8	CAN MTU 64
Message length	56	7	1	1

```
#
# 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.
#

# Zero means that the time is not known.
uint56 UNKNOWN = 0

# 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.
#
#   - CAN 2.0-based transports carry up to 7 bytes per frame. Time sync messages must use single-frame
#     transfers, which means that the value can't be wider than 56 bits.
truncated uint56 microsecond
```

6.9.5 TimeSystem

Full message type name: **`uavcan.time.TimeSystem`**

6.9.5.1 Version 0.1

Length unit	Bit	Byte (octet)	CAN MTU 8	CAN MTU 64
Message length	4	1	1	1

```
#
# Time system enumeration.
# The time system must be the same for all masters in the network.
# It cannot be changed while the network is running.
#

# Monotonic time since boot.
# Monotonic time is a time reference that doesn't change rate or make leaps.
uint4 MONOTONIC_SINCE_BOOT = 0

# International Atomic Time; https://en.wikipedia.org/wiki/International_Atomic_Time
# 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.
# UAVCAN does not support GPS time directly on purpose, for reasons of consistency.
uint4 TAI = 1

# Application-specific time system of unknown properties.
uint4 APPLICATION_SPECIFIC = 15

truncated uint4 value
```

6.10 uavcan.primitive

6.10.1 Empty

Full message type name: **`uavcan.primitive.Empty`**

6.10.1.1 Version 1.0

Length unit	Bit	Byte (octet)	CAN MTU 8	CAN MTU 64
Message length	0	0	1	1

```

```

6.10.2 String

Full message type name: **`uavcan.primitive.String`**

6.10.2.1 Version 1.0

Length unit	Bit	Byte (octet)	CAN MTU 8	CAN MTU 64
Message length	[8, 2048]	[1, 256]	[1, 37]	[1, 5]

```
#
# 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

@assert _offset_ % 8 == {0}
@assert _offset_.max / 8 == 256
```

6.10.3 Unstructured

Full message type name: **`uavcan.primitive.Unstructured`**

6.10.3.1 Version 1.0

Length unit	Bit	Byte (octet)	CAN MTU 8	CAN MTU 64
Message length	[8, 2048]	[1, 256]	[1, 37]	[1, 5]

```
#
# An unstructured collection of bytes, e.g., raw binary image.
#

uint8[<256] value

@assert _offset_ % 8 == {0}
@assert _offset_.max / 8 == 256
```

6.11 uavcan.primitive.array

6.11.1 Bit

Full message type name: **`uavcan.primitive.array.Bit`**

6.11.1.1 Version 1.0

Length unit	Bit	Byte (octet)	CAN MTU 8	CAN MTU 64
Message length	[16, 2048]	[2, 256]	[1, 37]	[1, 5]

```
void5
bool[<=2032] value
@assert _offset_.min == 16
@assert _offset_.max == 2048      # 2032 bits + 11 bit length + 5 bit padding = 256 bytes
```

6.11.2 Integer8

Full message type name: **`uavcan.primitive.array.Integer8`**

6.11.2.1 Version 1.0

Length unit	Bit	Byte (octet)	CAN MTU 8	CAN MTU 64
Message length	[8, 2048]	[1, 256]	[1, 37]	[1, 5]

```
int8[<256] value
@assert _offset_ % 8 == {0}
@assert _offset_.max / 8 == 256
```

6.11.3 Integer16

Full message type name: **`uavcan.primitive.array.Integer16`**

6.11.3.1 Version 1.0

Length unit	Bit	Byte (octet)	CAN MTU 8	CAN MTU 64
Message length	[8, 2056]	[1, 257]	[1, 37]	[1, 5]

```
int16[<=128] value
@assert _offset_ % 8 == {0}
@assert _offset_.max / 8 == 257
```

6.11.4 Integer32

Full message type name: **`uavcan.primitive.array.Integer32`**

6.11.4.1 Version 1.0

Length unit	Bit	Byte (octet)	CAN MTU 8	CAN MTU 64
Message length	[8, 2056]	[1, 257]	[1, 37]	[1, 5]

```
void1
int32[<=64] value
@assert _offset_ % 8 == {0}
@assert _offset_.max / 8 == 257
```

6.11.5 Integer64

Full message type name: **`uavcan.primitive.array.Integer64`**

6.11.5.1 Version 1.0

Length unit	Bit	Byte (octet)	CAN MTU 8	CAN MTU 64
Message length	[8, 2056]	[1, 257]	[1, 37]	[1, 5]

```
void2
int64[<=32] value
@assert _offset_ % 8 == {0}
@assert _offset_.max / 8 == 257
```

6.11.6 Natural8

Full message type name: **`uavcan.primitive.array.Natural8`**

6.11.6.1 Version 1.0

Length unit	Bit	Byte (octet)	CAN MTU 8	CAN MTU 64
Message length	[8, 2048]	[1, 256]	[1, 37]	[1, 5]

```
uint8[<256] value
@assert _offset_ % 8 == {0}
@assert _offset_.max / 8 == 256
```

6.11.7 Natural16

Full message type name: **`uavcan.primitive.array.Natural16`**

6.11.7.1 Version 1.0

Length unit	Bit	Byte (octet)	CAN MTU 8	CAN MTU 64
Message length	[8, 2056]	[1, 257]	[1, 37]	[1, 5]

```
uint16[<=128] value
@assert _offset_ % 8 == {0}
@assert _offset_.max / 8 == 257
```

6.11.8 Natural32

Full message type name: **`uavcan.primitive.array.Natural32`**

6.11.8.1 Version 1.0

Length unit	Bit	Byte (octet)	CAN MTU 8	CAN MTU 64
Message length	[8, 2056]	[1, 257]	[1, 37]	[1, 5]

```
void1
uint32[<=64] value
@assert _offset_ % 8 == {0}
@assert _offset_.max / 8 == 257
```

6.11.9 Natural64

Full message type name: **`uavcan.primitive.array.Natural64`**

6.11.9.1 Version 1.0

Length unit	Bit	Byte (octet)	CAN MTU 8	CAN MTU 64
Message length	[8, 2056]	[1, 257]	[1, 37]	[1, 5]

```
void2
uint64[<=32] value
@assert _offset_ % 8 == {0}
@assert _offset_.max / 8 == 257
```

6.11.10 Real16

Full message type name: **`uavcan.primitive.array.Real16`**

6.11.10.1 Version 1.0

Length unit	Bit	Byte (octet)	CAN MTU 8	CAN MTU 64
Message length	[8, 2056]	[1, 257]	[1, 37]	[1, 5]

```
float16[<=128] value                    # Exactly representable integers: [-2048, +2048]
@assert _offset_ % 8 == {0}
@assert _offset_.max / 8 == 257
```

6.11.11 Real32

Full message type name: **`uavcan.primitive.array.Real32`**

6.11.11.1 Version 1.0

Length unit	Bit	Byte (octet)	CAN MTU 8	CAN MTU 64
Message length	[8, 2056]	[1, 257]	[1, 37]	[1, 5]

```
void1
float32[<=64] value                     # Exactly representable integers: [-16777216, +16777216]
@assert _offset_ % 8 == {0}
@assert _offset_.max / 8 == 257
```

6.11.12 Real64

Full message type name: **`uavcan.primitive.array.Real64`**

6.11.12.1 Version 1.0

Length unit	Bit	Byte (octet)	CAN MTU 8	CAN MTU 64
Message length	[8, 2056]	[1, 257]	[1, 37]	[1, 5]

```
void2
float64[<=32] value                     # Exactly representable integers: [-2**53, +2**53]
@assert _offset_ % 8 == {0}
@assert _offset_.max / 8 == 257
```

6.12 uavcan.primitive.scalar

6.12.1 Bit

Full message type name: **`uavcan.primitive.scalar.Bit`**

6.12.1.1 Version 1.0

Length unit	Bit	Byte (octet)	CAN MTU 8	CAN MTU 64
Message length	8	1	1	1

```
void7
bool value
```

6.12.2 Integer8

Full message type name: **`uavcan.primitive.scalar.Integer8`**

6.12.2.1 Version 1.0

Length unit	Bit	Byte (octet)	CAN MTU 8	CAN MTU 64
Message length	8	1	1	1

```
int8 value
```

6.12.3 Integer16

Full message type name: **`uavcan.primitive.scalar.Integer16`**

6.12.3.1 Version 1.0

Length unit	Bit	Byte (octet)	CAN MTU 8	CAN MTU 64
Message length	16	2	1	1

```
int16 value
```

6.12.4 Integer32

Full message type name: **`uavcan.primitive.scalar.Integer32`**

6.12.4.1 Version 1.0

Length unit	Bit	Byte (octet)	CAN MTU 8	CAN MTU 64
Message length	32	4	1	1

```
int32 value
```

6.12.5 Integer64

Full message type name: **`uavcan.primitive.scalar.Integer64`**

6.12.5.1 Version 1.0

Length unit	Bit	Byte (octet)	CAN MTU 8	CAN MTU 64
Message length	64	8	2	1

```
int64 value
```

6.12.6 Natural8

Full message type name: **`uavcan.primitive.scalar.Natural8`**

6.12.6.1 Version 1.0

Length unit	Bit	Byte (octet)	CAN MTU 8	CAN MTU 64
Message length	8	1	1	1

```
uint8 value
```

6.12.7 Natural16

Full message type name: **`uavcan.primitive.scalar.Natural16`**

6.12.7.1 Version 1.0

Length unit	Bit	Byte (octet)	CAN MTU 8	CAN MTU 64
Message length	16	2	1	1

```
uint16 value
```

6.12.8 Natural32

Full message type name: **`uavcan.primitive.scalar.Natural32`**

6.12.8.1 Version 1.0

Length unit	Bit	Byte (octet)	CAN MTU 8	CAN MTU 64
Message length	32	4	1	1

```
uint32 value
```

6.12.9 Natural64

Full message type name: **`uavcan.primitive.scalar.Natural64`**

6.12.9.1 Version 1.0

Length unit	Bit	Byte (octet)	CAN MTU 8	CAN MTU 64
Message length	64	8	2	1

```
uint64 value
```

6.12.10 Real16

Full message type name: **`uavcan.primitive.scalar.Real16`**

6.12.10.1 Version 1.0

Length unit	Bit	Byte (octet)	CAN MTU 8	CAN MTU 64
Message length	16	2	1	1

```
float16 value       # Exactly representable integers: [-2048, +2048]
```

6.12.11 Real32

Full message type name: **`uavcan.primitive.scalar.Real32`**

6.12.11.1 Version 1.0

Length unit	Bit	Byte (octet)	CAN MTU 8	CAN MTU 64
Message length	32	4	1	1

```
float32 value       # Exactly representable integers: [-16777216, +16777216]
```

6.12.12 Real64

Full message type name: **`uavcan.primitive.scalar.Real64`**

6.12.12.1 Version 1.0

Length unit	Bit	Byte (octet)	CAN MTU 8	CAN MTU 64
Message length	64	8	2	1

```
float64 value       # Exactly representable integers: [-2**53, +2**53]
```

6.13 uavcan.si.acceleration

6.13.1 Scalar

Full message type name: **`uavcan.si.acceleration.Scalar`**

6.13.1.1 Version 1.0

Length unit	Bit	Byte (octet)	CAN MTU 8	CAN MTU 64
Message length	56	7	1	1

```
uavcan.time.SynchronizedAmbiguousTimestamp.1.0 timestamp
float32 meter_per_second_per_second
@assert _offset_ / 8 == {7}               # Single CAN 2.0 frame
```

6.13.2 Vector3

Full message type name: **`uavcan.si.acceleration.Vector3`**

6.13.2.1 Version 1.0

Length unit	Bit	Byte (octet)	CAN MTU 8	CAN MTU 64
Message length	120	15	3	1

```
uavcan.time.SynchronizedAmbiguousTimestamp.1.0 timestamp
float32[3] meter_per_second_per_second
@assert _offset_ / 8 == {15}
@assert _offset_ % 8 == {0}
```

6.14 uavcan.si.angle

6.14.1 Quaternion

Full message type name: **`uavcan.si.angle.Quaternion`**

6.14.1.1 Version 1.0

Length unit	Bit	Byte (octet)	CAN MTU 8	CAN MTU 64
Message length	152	19	3	1

```
uavcan.time.SynchronizedAmbiguousTimestamp.1.0 timestamp
float32[4] wxyz
@assert _offset_ / 8 == {19}              # Optimized for three CAN 2.0 frames
@assert _offset_ % 8 == {0}
```

6.14.2 Scalar

Full message type name: **`uavcan.si.angle.Scalar`**

6.14.2.1 Version 1.0

Length unit	Bit	Byte (octet)	CAN MTU 8	CAN MTU 64
Message length	56	7	1	1

```
uavcan.time.SynchronizedAmbiguousTimestamp.1.0 timestamp
float32 radian
@assert _offset_ / 8 == {7}               # Single CAN 2.0 frame
```

6.15 uavcan.si.angular_velocity

6.15.1 Scalar

Full message type name: **`uavcan.si.angular_velocity.Scalar`**

6.15.1.1 Version 1.0

Length unit	Bit	Byte (octet)	CAN MTU 8	CAN MTU 64
Message length	56	7	1	1

```
uavcan.time.SynchronizedAmbiguousTimestamp.1.0 timestamp
float32 radian_per_second
@assert _offset_ / 8 == {7}               # Single CAN 2.0 frame
```

6.15.2 Vector3

Full message type name: **`uavcan.si.angular_velocity.Vector3`**

6.15.2.1 Version 1.0

Length unit	Bit	Byte (octet)	CAN MTU 8	CAN MTU 64
Message length	120	15	3	1

```
uavcan.time.SynchronizedAmbiguousTimestamp.1.0 timestamp
float32[3] radian_per_second
@assert _offset_ / 8 == {15}
@assert _offset_ % 8 == {0}
```

6.16 uavcan.si.duration

6.16.1 Scalar

Full message type name: **`uavcan.si.duration.Scalar`**

6.16.1.1 Version 1.0

Length unit	Bit	Byte (octet)	CAN MTU 8	CAN MTU 64
Message length	56	7	1	1

```
uavcan.time.SynchronizedAmbiguousTimestamp.1.0 timestamp
float32 second
@assert _offset_ / 8 == {7}               # Single CAN 2.0 frame
```

6.16.2 WideScalar

Full message type name: **`uavcan.si.duration.WideScalar`**

6.16.2.1 Version 1.0

Length unit	Bit	Byte (octet)	CAN MTU 8	CAN MTU 64
Message length	88	11	2	1

```
uavcan.time.SynchronizedAmbiguousTimestamp.1.0 timestamp
float64 second
@assert _offset_ / 8 == {11}              # Two CAN 2.0 frames
```

6.17 uavcan.si.electric_charge

6.17.1 Scalar

Full message type name: **`uavcan.si.electric_charge.Scalar`**

6.17.1.1 *Version 1.0*

Length unit	Bit	Byte (octet)	CAN MTU 8	CAN MTU 64
Message length	56	7	1	1

```
uavcan.time.SynchronizedAmbiguousTimestamp.1.0 timestamp
float32 coulomb
@assert _offset_ / 8 == {7}               # Single CAN 2.0 frame
```

6.18 uavcan.si.electric_current

6.18.1 Scalar

Full message type name: **`uavcan.si.electric_current.Scalar`**

6.18.1.1 *Version 1.0*

Length unit	Bit	Byte (octet)	CAN MTU 8	CAN MTU 64
Message length	56	7	1	1

```
uavcan.time.SynchronizedAmbiguousTimestamp.1.0 timestamp
float32 ampere
@assert _offset_ / 8 == {7}               # Single CAN 2.0 frame
```

6.19 uavcan.si.energy

6.19.1 Scalar

Full message type name: **`uavcan.si.energy.Scalar`**

6.19.1.1 *Version 1.0*

Length unit	Bit	Byte (octet)	CAN MTU 8	CAN MTU 64
Message length	56	7	1	1

```
uavcan.time.SynchronizedAmbiguousTimestamp.1.0 timestamp
float32 joule
@assert _offset_ / 8 == {7}               # Single CAN 2.0 frame
```

6.20 uavcan.si.length

6.20.1 Scalar

Full message type name: **`uavcan.si.length.Scalar`**

6.20.1.1 *Version 1.0*

Length unit	Bit	Byte (octet)	CAN MTU 8	CAN MTU 64
Message length	56	7	1	1

```
uavcan.time.SynchronizedAmbiguousTimestamp.1.0 timestamp
float32 meter
@assert _offset_ / 8 == {7}               # Single CAN 2.0 frame
```

6.20.2 Vector3

Full message type name: **`uavcan.si.length.Vector3`**

6.20.2.1 *Version 1.0*

Length unit	Bit	Byte (octet)	CAN MTU 8	CAN MTU 64
Message length	120	15	3	1

```
uavcan.time.SynchronizedAmbiguousTimestamp.1.0 timestamp
float32[3] meter
@assert _offset_ / 8 == {15}
@assert _offset_ % 8 == {0}
```

6.20.3 WideVector3

Full message type name: **`uavcan.si.length.WideVector3`**

6.20.3.1 *Version 1.0*

Length unit	Bit	Byte (octet)	CAN MTU 8	CAN MTU 64
Message length	216	27	5	1

```
uavcan.time.SynchronizedAmbiguousTimestamp.1.0 timestamp
float64[3] meter
@assert _offset_ / 8 == {27}
@assert _offset_ % 8 == {0}
```

6.21 uavcan.si.magnetic_field_strength

6.21.1 Scalar

Full message type name: **`uavcan.si.magnetic_field_strength.Scalar`**

6.21.1.1 Version 1.0

Length unit	Bit	Byte (octet)	CAN MTU 8	CAN MTU 64
Message length	56	7	1	1

```
uavcan.time.SynchronizedAmbiguousTimestamp.1.0 timestamp
float32 tesla
@assert _offset_ / 8 == {7}               # Single CAN 2.0 frame
```

6.21.2 Vector3

Full message type name: **`uavcan.si.magnetic_field_strength.Vector3`**

6.21.2.1 Version 1.0

Length unit	Bit	Byte (octet)	CAN MTU 8	CAN MTU 64
Message length	120	15	3	1

```
uavcan.time.SynchronizedAmbiguousTimestamp.1.0 timestamp
float32[3] tesla
@assert _offset_ / 8 == {15}
@assert _offset_ % 8 == {0}
```

6.22 uavcan.si.mass

6.22.1 Scalar

Full message type name: **`uavcan.si.mass.Scalar`**

6.22.1.1 Version 1.0

Length unit	Bit	Byte (octet)	CAN MTU 8	CAN MTU 64
Message length	56	7	1	1

```
uavcan.time.SynchronizedAmbiguousTimestamp.1.0 timestamp
float32 kilogram
@assert _offset_ / 8 == {7}               # Single CAN 2.0 frame
```

6.23 uavcan.si.power

6.23.1 Scalar

Full message type name: **`uavcan.si.power.Scalar`**

6.23.1.1 Version 1.0

Length unit	Bit	Byte (octet)	CAN MTU 8	CAN MTU 64
Message length	56	7	1	1

```
uavcan.time.SynchronizedAmbiguousTimestamp.1.0 timestamp
float32 watt
@assert _offset_ / 8 == {7}               # Single CAN 2.0 frame
```

6.24 uavcan.si.pressure

6.24.1 Scalar

Full message type name: **`uavcan.si.pressure.Scalar`**

6.24.1.1 Version 1.0

Length unit	Bit	Byte (octet)	CAN MTU 8	CAN MTU 64
Message length	56	7	1	1

```
uavcan.time.SynchronizedAmbiguousTimestamp.1.0 timestamp
float32 pascal
@assert _offset_ / 8 == {7}               # Single CAN 2.0 frame
```

6.25 uavcan.si.temperature

6.25.1 Scalar

Full message type name: **`uavcan.si.temperature.Scalar`**

6.25.1.1 Version 1.0

Length unit	Bit	Byte (octet)	CAN MTU 8	CAN MTU 64
Message length	56	7	1	1

```
uavcan.time.SynchronizedAmbiguousTimestamp.1.0 timestamp
float32 kelvin
@assert _offset_ / 8 == {7}               # Single CAN 2.0 frame
```

6.26 uavcan.si.velocity

6.26.1 Scalar

Full message type name: **`uavcan.si.velocity.Scalar`**

6.26.1.1 Version 1.0

Length unit	Bit	Byte (octet)	CAN MTU 8	CAN MTU 64
Message length	56	7	1	1

```
uavcan.time.SynchronizedAmbiguousTimestamp.1.0 timestamp
float32 meter_per_second
@assert _offset_ / 8 == {7}               # Single CAN 2.0 frame
```

6.26.2 Vector3

Full message type name: **`uavcan.si.velocity.Vector3`**

6.26.2.1 Version 1.0

Length unit	Bit	Byte (octet)	CAN MTU 8	CAN MTU 64
Message length	120	15	3	1

```
uavcan.time.SynchronizedAmbiguousTimestamp.1.0 timestamp
float32[3] meter_per_second
@assert _offset_ / 8 == {15}
@assert _offset_ % 8 == {0}
```

6.27 uavcan.si.voltage

6.27.1 Scalar

Full message type name: **`uavcan.si.voltage.Scalar`**

6.27.1.1 Version 1.0

Length unit	Bit	Byte (octet)	CAN MTU 8	CAN MTU 64
Message length	56	7	1	1

```
uavcan.time.SynchronizedAmbiguousTimestamp.1.0 timestamp
float32 volt
@assert _offset_ / 8 == {7}               # Single CAN 2.0 frame
```

6.28 uavcan.si.volume

6.28.1 Scalar

Full message type name: **`uavcan.si.volume.Scalar`**

6.28.1.1 Version 1.0

Length unit	Bit	Byte (octet)	CAN MTU 8	CAN MTU 64
Message length	56	7	1	1

```
uavcan.time.SynchronizedAmbiguousTimestamp.1.0 timestamp
float32 cubic_meter
@assert _offset_ / 8 == {7}               # Single CAN 2.0 frame
```

6.29 uavcan.si.volumetric_flow_rate

6.29.1 Scalar

Full message type name: **uavcan.si.volumetric_flow_rate.Scalar**

6.29.1.1 Version 1.0

Length unit	Bit	Byte (octet)	CAN MTU 8	CAN MTU 64
Message length	56	7	1	1

```
uavcan.time.SynchronizedAmbiguousTimestamp.1.0 timestamp
float32 cubic_meter_per_second
@assert _offset_ / 8 == {7}              # Single CAN 2.0 frame
```

7 Physical layer

This chapter contains the specification of the supported physical layers of UAVCAN, as well as some related hardware design recommendations.

Following the requirements and recommendations of this chapter will ensure the highest level of inter-vendor compatibility and allow the developers to avoid many common design pitfalls.

The sections that provide transport-specific physical layer specification directly correspond to those defined in the chapter 4.

7.1 CAN bus physical layer specification

This section specifies the CAN-based physical layer of UAVCAN.

Here and in the following parts of this section, “CAN” implies both CAN 2.0 and CAN FD, unless specifically noted otherwise.

7.1.1 Physical connector specification

The UAVCAN standard defines several connector types, targeted towards different application domains: from highly compact systems to large deployments, from low-cost to safety-critical applications.

The table 7.1 provides an overview of the currently defined connector types for the CAN bus transport implementation. Other connector types may be added in future revisions of the specification.

It is highly recommended to provide two identical parallel connectors for each CAN interface per device, and not using T-connectors. T-connectors should be avoided because they add another point of failure, increase the stub length, weight, and often require more complex and expensive wiring harnesses.

Table 7.1: Standard CAN connector types

Connector name	Base connector type	Bus power	Known compatible standards
UAVCAN D-Sub	Generic D-Subminiature DE-9	24 V, 3 A	De-facto standard connector for CAN, supported by many current specifications.
UAVCAN M8	Generic M8 5-circuit B-coded	24 V, 3 A	CiA 103 (CANopen)
UAVCAN Micro	JST GH 4-circuit	5 V, 1 A	Dronecode Autopilot Connector Standard

7.1.1.1 *UAVCAN D-Sub connector*

The UAVCAN D-Sub connector type is based upon, and compatible with, the D-Subminiature DE-9 CAN connector (this is the most popular CAN connector type, in effect the de-facto industry standard). This connector is fully compatible with CANopen and many other current specifications. An example connector pair is pictured on the figures 7.1 and 7.2.

Advantages	Disadvantages
• Highest level of compatibility with the existing commercial off the shelf (COTS) hardware. Connectors, cables, termination plugs, and other components can be easily purchased from many different vendors. • High-reliability options are available from multiple vendors. • Low-cost options are available from multiple vendors. • Both PCB mounted and panel mounted types are available.	D-Subminiature connectors are the largest connector type defined by UAVCAN. Due to its significant size and weight, it may be unsuitable for many vehicular applications.

The UAVCAN D-Sub connector is based on the industry-standard **D-Sub DE-9** (9-circuit) connector type. Devices are equipped with the male plug connector type mounted on the panel or on the PCB, and the cables are equipped with the female socket connectors on both ends (see the figures 7.1 and 7.2).

If the device uses two parallel connectors per CAN bus interface (as recommended), then all of the lines of the paired connectors, including those that are not used by the current specification, must be interconnected one to one. This will ensure compatibility with future revisions of the specification that make use of currently unused circuits of the connector.

The CAN physical layer standard that can be used with this connector type is ISO 11898-2[79].

Devices that deliver power to the bus are required to provide 23.0–30.0 V on the bus power line, 24 V nominal. The maximum current draw is up to 3 A per connector.

Devices that are powered from the bus should expect 18.0–30.0 V on the bus power line. The maximum recommended current draw from the bus is 0.5 A per device.

The table 7.2 documents the pinout specification for the UAVCAN D-Sub connector type. The provided pinout, as has been indicated above, is the de-facto industry standard for the CAN bus. Note that the signals “CAN High” and “CAN Low” must belong to the same twisted pair. Usage of twisted or flat wires for all other signals remains at the discretion of the implementer.

Table 7.2: UAVCAN D-Sub connector pinout

#	Function	Note
1		
2	CAN low	Twisted with “CAN high” (pin 7).
3	CAN ground	Must be interconnected with “Ground” (pin 6) within the device.
4		
5	CAN shield	Optional.
6	Ground	Must be interconnected with “CAN ground” (pin 3) within the device.
7	CAN high	Twisted with “CAN low” (pin 2).
8		
9	Bus power supply	24 V nominal. See the power supply requirements.

[79] Also known as *high-speed CAN*.

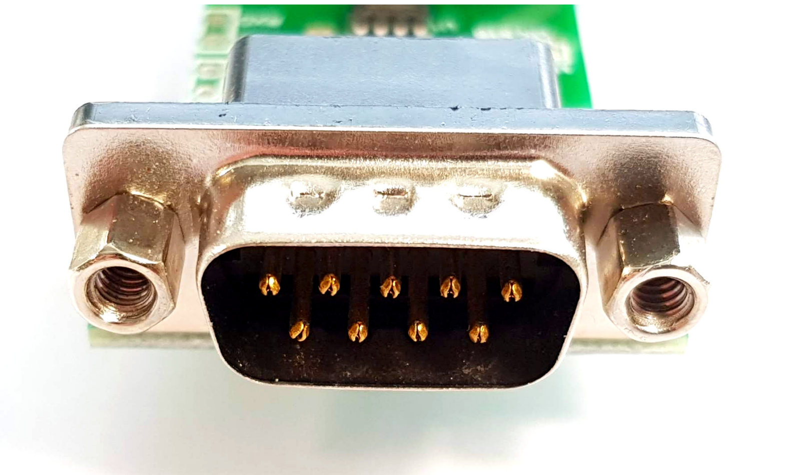


Figure 7.1: UAVCAN D-Sub device connector example.

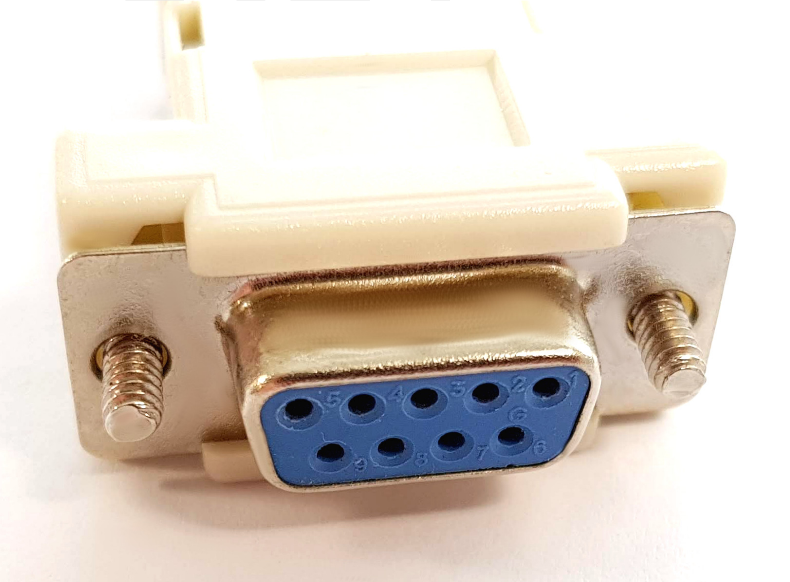


Figure 7.2: UAVCAN D-Sub cable connector example.

7.1.1.2 *UAVCAN M8 connector*

The UAVCAN M8 connector type is based on the generic circular M8 connector type, shown on the figure 7.3. This is a popular industry-standard connector, and there are many vendors that manufacture compatible components: connectors, cables, termination plugs, T-connectors, and so on. The pinning, physical layer, and supply voltages used in this connector type are compatible with CiA 103 (CANopen) and some other CAN bus standards.

The M8 connector is preferred for most UAVCAN applications (it should be the default choice, except when there are specific reasons to select another standard connector type).

Advantages	Disadvantages
• Compatibility with existing COTS hardware. Connectors, cables, termination plugs, and other components can be purchased from many different vendors. • High-reliability options are available from multiple vendors. • Low-cost options are available from multiple vendors. • Reasonably compact. M8 connectors are much smaller than D-Sub. • PCB mounted and panel mounted types are available.	• M8 connectors may be a poor fit for applications that have severe weight and space constraints. • The level of adoption in the industry is noticeably lower than that of the D-Sub connector type.

The UAVCAN M8 connector is based on the industry-standard **circular M8 B-coded 5-circuit** connector type. Devices are equipped with the male plug connector type mounted on the panel or on the PCB, and the cables are equipped with the female socket connectors on both ends (see the figure 7.3). *Do not confuse A-coded and B-coded M8 connectors – they are not mutually compatible.*

The CAN physical layer standard that can be used with this connector type is ISO 11898-2[80].

Devices that deliver power to the bus are required to provide 23.0–30.0 V on the bus power line, 24 V nominal. The maximum current draw is up to 3 A per connector.

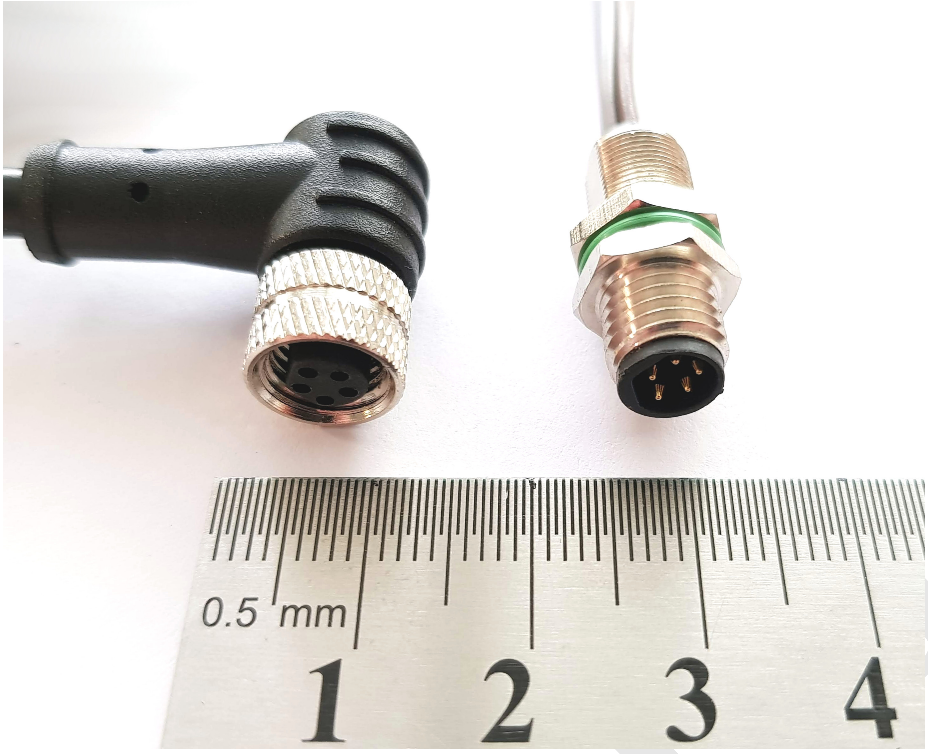
Devices that are powered from the bus should expect 18.0–30.0 V on the bus power line. The maximum recommended current draw from the bus is 0.5 A per device.

The table 7.3 documents the pinout specification for the UAVCAN M8 connector type. The provided pinout, as indicated above, is compatible with the CiA 103 specification (CANopen). Note that the wires "CAN high" and "CAN low" should be a twisted pair.

Table 7.3: UAVCAN M8 connector pinout

#	Function	Note
1	Bus power supply	24 V nominal. See the power supply requirements.
2	CAN shield	Optional.
3	CAN high	Twisted with "CAN low" (pin 4).
4	CAN low	Twisted with "CAN high" (pin 3).
5	Ground	

[80] Also known as *high-speed CAN*.



Example connectors: female socket cable (left) and male plug device connector (right). Different connector types are available from various vendors: PCB mounted, panel mounted; straight cables, angled cables, etc.

Figure 7.3: UAVCAN M8 connector pair example.

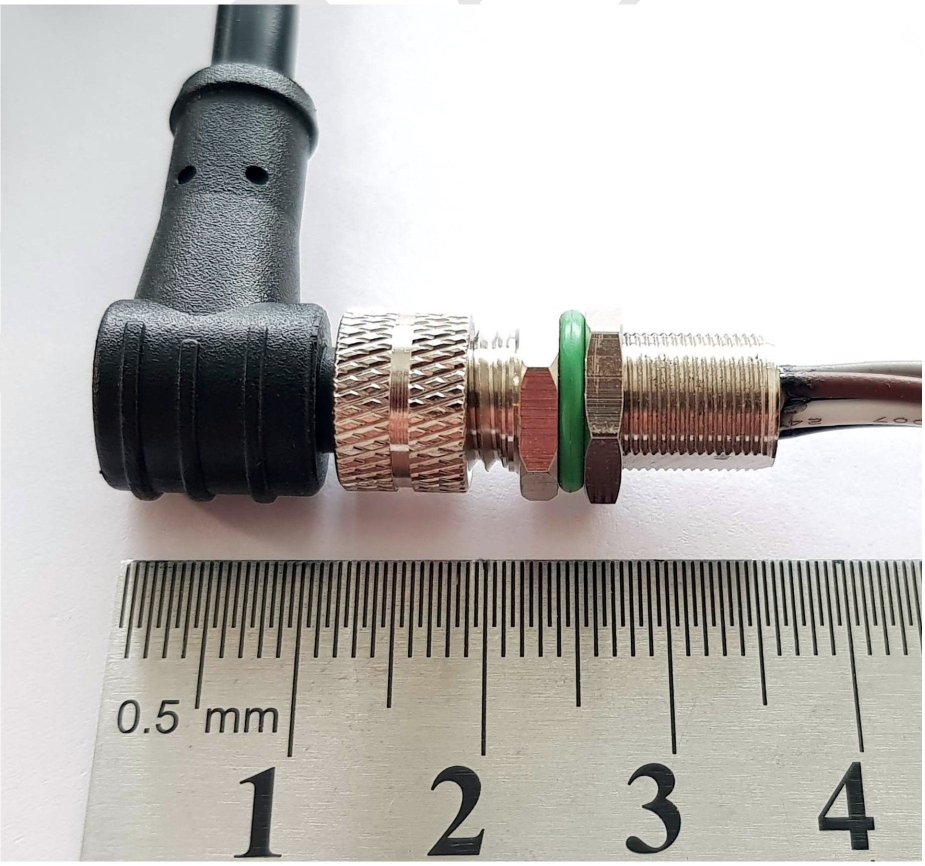


Figure 7.4: UAVCAN M8 assembled connector pair example.

7.1.1.3 *UAVCAN Micro connector*

The UAVCAN Micro connector is intended for weight- and space-sensitive applications. It is a board-level connector, meaning that it can be installed on the PCB rather than on the panel. An example is shown on the figure 7.5.

The Micro connector is compatible with the Dronecode Autopilot Connector Standard. This connector type is recommended for small UAV and nanosatellites. It is also the recommended connector for attaching external panel-mounted connectors (such as the M8 or D-Sub types) to the PCB inside the enclosure.

Advantages	**Disadvantages**
• Extremely compact, low-profile. The PCB footprint is under 9×5 millimeters. • Secure positive lock ensures that the connection will not self-disconnect when exposed to vibrations. • Low-cost, easy to stock.	• Board-level connections only. No panel-mounted options available. • No shielding available. • Not suitable for safety-critical hardware.

The UAVCAN Micro connector is based on the proprietary **JST GH 4-circuit** connector type. Beware that the top-entry type is not PCB footprint-compatible with the side-entry type – its pin ordering is reversed. The wire-side pinout, however, is compatible, so both types can be used interchangeably as long as their PCB footprints are correct.

The suitable cable types are flat or twisted pair #30 to #26 AWG, outer insulation diameter 0.8–1.0 mm, multistrand. Non-twisted (flat) cables can only be used in very small deployments free of significant EMI[81]; otherwise, reliable functioning of the bus cannot be guaranteed.

The CAN physical layer standard that can be used with this connector type is ISO 11898-2.

Devices that deliver power to the bus are required to provide 5.0–5.5 V on the bus power line. The anticipated current draw is up to 1 A per connector.

Devices that are powered from the bus should expect 4.0–5.5 V on the bus power line. The maximum recommended current draw from the bus is 0.5 A per device.

The table 7.4 documents the pinout specification for the UAVCAN M8 connector type. The provided pinout, as indicated above, is compatible with the Dronecode Autopilot Connector Standard. Note that the wires "CAN high" and "CAN low" should be a twisted pair.

Table 7.4: UAVCAN Micro connector pinout

#	Function	Note
1	Bus power supply	5 V nominal. See the power supply requirements.
2	CAN high	Should be twisted with "CAN low" (pin 3).
3	CAN low	Should be twisted with "CAN high" (pin 2).
4	Ground	

[81] Electromagnetic interference.

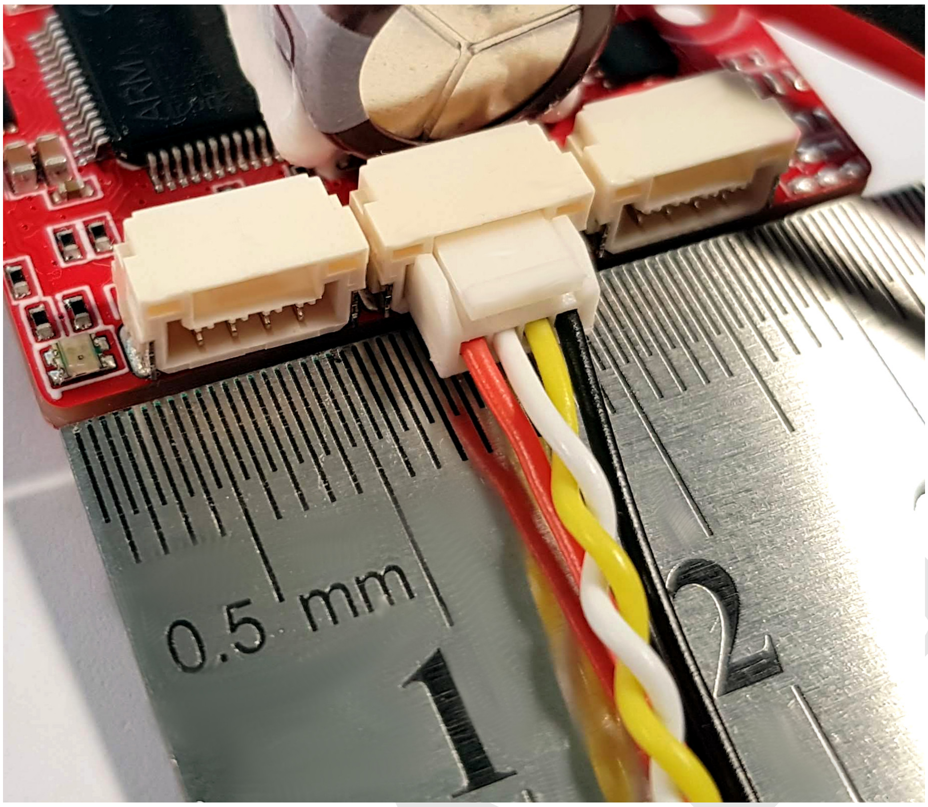


Figure 7.5: UAVCAN Micro right-angle connectors with a twisted pair patch cable connected.

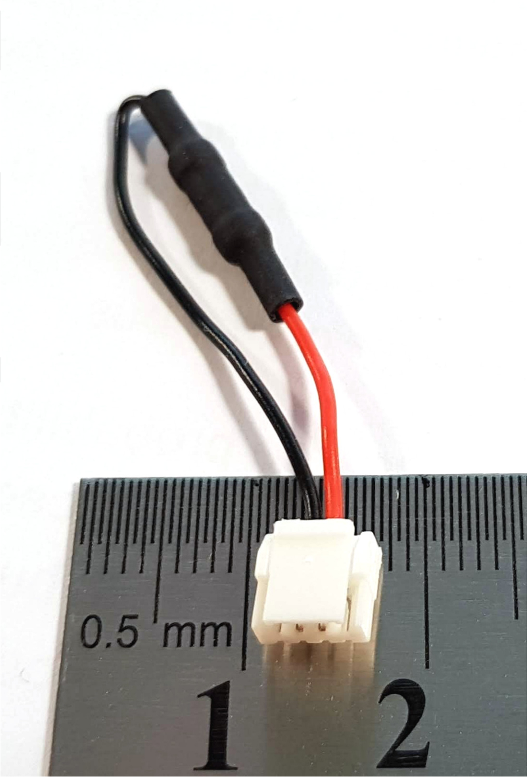


Figure 7.6: UAVCAN Micro CAN bus termination plug.

7.1.2 CAN bus physical layer parameters

As can be seen from the rest of the specification, UAVCAN is mostly agnostic of the parameters of the physical layer. However, vendors should follow the recommendations provided in this section to maximize the cross-vendor compatibility.

7.1.2.1 CAN 2.0

This section is dedicated to the legacy CAN 2.0 protocol.

The table 7.5 lists the standard parameters of the CAN PHY for ISO 11898-2. The estimated bus length limits are based on the assumption that the propagation delay does not exceed 5 ns/m, not including additional delay times of CAN transceivers and other components.

Table 7.5: Standard CAN 2.0 PHY parameters

Bit rate [kbit/s]	Valid range for location of sample point [%]	Recommended location of sample point [%]	Maximum bus length [m]	Maximum stub length [m]
1000	75 to 90	87.5	40	0.3
500	85 to 90	87.5	100	0.3
250	85 to 90	87.5	250	0.3
125	85 to 90	87.5	500	0.3

Designers are encouraged to implement CAN auto bit rate detection when applicable. Please refer to the CiA 801 application note for the recommended practices.

UAVCAN allows the use of a simple bit time measuring approach, as it is guaranteed that any functioning UAVCAN network will always exchange node status messages, which can be expected to be published at a rate no lower than 1 Hz, and that contain a suitable alternating bit pattern in the CAN ID field. Please refer to the chapter 5 for details.

7.1.2.2 CAN FD

This section will be populated in a later revision of the document.

7.2 Hardware design recommendations

This section contains certain generic hardware design recommendations that are agnostic of a particular physical layer implementation.

7.2.1 Non-uniform transport redundancy

Mission critical devices and non-mission critical devices often need to co-exist within the same UAVCAN network. Non-mission critical devices are likely to be equipped with a non-redundant transport interface, which can create the situation where multiple devices with different numbers of redundant interfaces need to be connected to the same network. In that case, the following rules should be followed:

- Each available bus is assigned a level of importance (primary, secondary, etc.).
- All nodes should be connected to the primary bus.
- Only nodes with redundant interfaces should be also connected to the non-primary bus/buses.

The figure 7.7 shows a doubly redundant bus transport as an example.

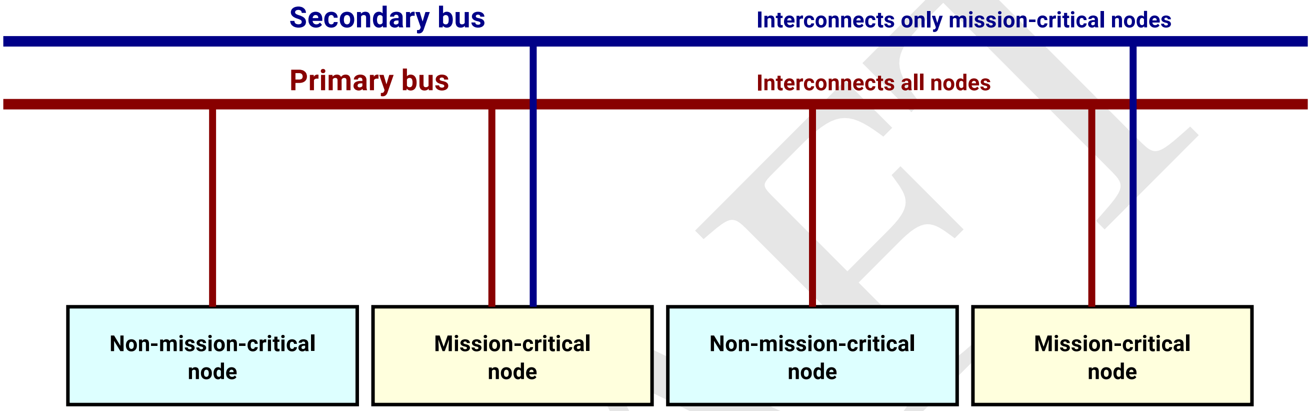

Secondary bus
Interconnects only mission-critical nodes
Primary bus
Interconnects all nodes
Non-mission-critical node
Mission-critical node
Non-mission-critical node
Mission-critical node

Figure 7.7: Non-uniform transport redundancy.

7.2.2 Bus power supply

The standard UAVCAN physical layers support power distribution between nodes. Integration of the power distribution functionality with the communication interface obviates the need for a dedicated power distribution network, which can greatly simplify the system design and reduce the complexity and weight of the wiring harnesses. Additionally, redundant power supply topologies can be easily implemented on top of redundant communication interfaces.

7.2.2.1 Power sinking nodes

This section applies to nodes that draw power from the network.

Each power input should be protected with an over-current protection circuit (for example, an electronic fuse), so that a short-circuit or a similar failure of the node does not propagate to the entire bus.

If the node incorporates redundant bus interfaces, it should prevent direct current flow between power inputs from different interface connectors, so that if one bus suffers a power failure (e.g. a short circuit) it is not propagated to the other buses.

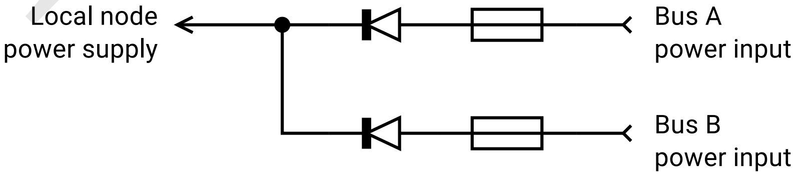


Figure 7.8: Simplified conceptual power sinking node design schematic.

7.2.2.2 Power sourcing nodes

This section applies to nodes that deliver power to the network.

Similar to the case of bus-powered nodes, UAVCAN power sources should take into account that one of the redundant interfaces may suffer a short-circuit or a failure of a similar mode. Should that happen, the power source should shut down the power supply of the failing bus and continue supplying the remaining bus interfaces.

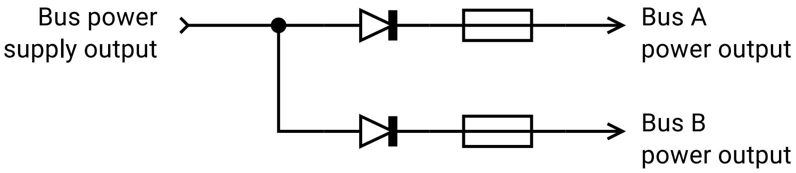


Figure 7.9: Simplified conceptual power sourcing node design schematic.