

UAVCAN

Specification v1.0

Revision 2019-03-23

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 (RPC¹) 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).

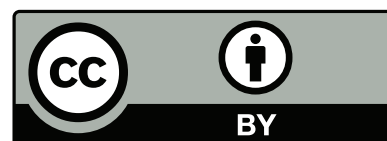
Support and feedback

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.

This work is licensed under the Creative Commons Attribution 4.0 International License. To view a copy of this license, visit <http://creativecommons.org/licenses/by/4.0/> or send a letter to Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.



¹Remote 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.

DRAFT

Table of contents

1	Introduction	1	6	List of standard data types	42
1.1	Document conventions	1	6.1	uavcan.diagnostic	45
1.2	Design principles	1	6.1.1	Record	45
1.3	Capabilities	2	6.1.2	Severity	45
1.4	Maintenance of the standard data type set	2	6.2	uavcan.file	46
1.5	Referenced sources	2	6.2.1	GetInfo	46
2	Basic concepts	4	6.2.2	List	46
2.1	Main principles.	4	6.2.3	Modify	47
2.1.1	Communication	4	6.2.4	Read	47
2.1.2	Data types	4	6.2.5	Write	48
2.1.3	High-level functions	6	6.2.6	Error	48
2.2	Message publication	6	6.2.7	Path	48
2.2.1	Anonymous message publication	6	6.3	uavcan.internet.udp	49
2.3	Service invocation.	6	6.3.1	HandleIncomingPacket	49
3	Data structure description language	8	6.3.2	OutgoingPacket	50
3.1	Architecture	8	6.4	uavcan.node.	52
3.1.1	General principles	8	6.4.1	ExecuteCommand	52
3.1.2	Data types and namespaces	8	6.4.2	GetInfo	53
3.1.3	File hierarchy	9	6.4.3	GetTransportStatistics	53
3.1.4	Elements of data type definition.	10	6.4.4	Heartbeat	54
4	Transport layer.	11	6.4.5	ID	55
4.1	Core concepts	11	6.4.6	IOStatistics	55
4.1.1	Transfer	11	6.4.7	Version	55
4.1.2	Message publication	12	6.5	uavcan.node.port	55
4.1.3	Service invocation	12	6.5.1	GetInfo	55
4.1.4	Transfer priority	13	6.5.2	GetStatistics	56
4.1.5	Transfer descriptor	14	6.5.3	List	56
4.2	Transfer emission	14	6.5.4	ID	57
4.2.1	Transfer ID computation.	14	6.5.5	ServiceID	57
4.2.2	Single frame transfers	15	6.5.6	SubjectID	57
4.2.3	Multi-frame transfers	15	6.6	uavcan.pnp	58
4.2.4	Redundant interface support	17	6.6.1	NodeIDAllocationData	58
4.3	Transfer reception	18	6.6.2	NodeIDAllocationDataMTU8	60
4.3.1	Transfer ID comparison	18	6.7	uavcan.pnp.cluster	61
4.3.2	State variables	18	6.7.1	AppendEntries	61
4.3.3	State update in a redundant interface configuration.	20	6.7.2	Discovery	62
4.3.4	State update in a non-redundant interface configuration	22	6.7.3	RequestVote	62
4.4	CAN bus transport layer specification	23	6.7.4	Entry	62
4.4.1	CAN ID structure	23	6.8	uavcan.register	64
4.4.2	CAN frame data	25	6.8.1	Access	64
4.4.3	Software design considerations	27	6.8.2	List	65
5	Application layer	30	6.8.3	Name	65
5.1	Application-level conventions.	31	6.8.4	Value	65
5.1.1	Identifier distribution	31	6.9	uavcan.time	67
5.1.2	Coordinate frames	31	6.9.1	GetSynchronizationMasterInfo	67
5.1.3	Rotation representation	32	6.9.2	Synchronization	67
5.1.4	Matrix representation	32	6.9.3	SynchronizedAmbiguousTimestamp	69
5.1.5	Physical quantity representation	33	6.9.4	SynchronizedTimestamp	69
5.2	Application-level functions.	34	6.9.5	TimeSystem	70
5.2.1	Node initialization	34	6.10	uavcan.primitive	71
5.2.2	Node heartbeat	34	6.10.1	Empty	71
5.2.3	Generic node information	35	6.10.2	String	71
5.2.4	Bus data flow monitoring	36	6.10.3	Unstructured	71
5.2.5	Network-wide time synchronization	36	6.11	uavcan.primitive.array	71
5.2.6	Primitive types and physical quantities	37	6.11.1	Bit	71
5.2.7	Remote file system interface.	39	6.11.2	Integer8	71
5.2.8	Generic node commands	39	6.11.3	Integer16	72
5.2.9	Node software update	39	6.11.4	Integer32	72
5.2.10	Register interface.	40	6.11.5	Integer64	72
5.2.11	Diagnostics and event logging	40	6.11.6	Natural8	72
5.2.12	Plug-and-play nodes	40	6.11.7	Natural16	72
5.2.13	Internet/LAN forwarding interface	41	6.11.8	Natural32	72
			6.11.9	Natural64	73
			6.11.10	Real16	73
			6.11.11	Real32	73
			6.11.12	Real64	73
			6.12	uavcan.primitive.scalar	73
			6.12.1	Bit	73
			6.12.2	Integer8	73
			6.12.3	Integer16	74
			6.12.4	Integer32	74

6.12.5	Integer64	74
6.12.6	Natural8	74
6.12.7	Natural16	74
6.12.8	Natural32	74
6.12.9	Natural64	74
6.12.10	Real16	75
6.12.11	Real32	75
6.12.12	Real64	75
6.13	uavcan.si.acceleration	75
6.13.1	Scalar	75
6.13.2	Vector3	75
6.14	uavcan.si.angle	75
6.14.1	Quaternion	75
6.14.2	Scalar	76
6.15	uavcan.si.angular_velocity	76
6.15.1	Scalar	76
6.15.2	Vector3	76
6.16	uavcan.si.duration	76
6.16.1	Scalar	76
6.16.2	WideScalar	76
6.17	uavcan.si.electric_charge	76
6.17.1	Scalar	76
6.18	uavcan.si.electric_current	77
6.18.1	Scalar	77
6.19	uavcan.si.energy	77
6.19.1	Scalar	77
6.20	uavcan.si.length	77
6.20.1	Scalar	77
6.20.2	Vector3	77
6.20.3	WideVector3	77
6.21	uavcan.si.magnetic_field_strength	78
6.21.1	Scalar	78
6.21.2	Vector3	78
6.22	uavcan.si.mass	78
6.22.1	Scalar	78
6.23	uavcan.si.power	78
6.23.1	Scalar	78
6.24	uavcan.si.pressure	78
6.24.1	Scalar	78
6.25	uavcan.si.temperature	79
6.25.1	Scalar	79
6.26	uavcan.si.velocity	79
6.26.1	Scalar	79
6.26.2	Vector3	79
6.27	uavcan.si.voltage	79
6.27.1	Scalar	79
6.28	uavcan.si.volume	79
6.28.1	Scalar	79
6.29	uavcan.si.volumetric_flow_rate	80
6.29.1	Scalar	80
7	Physical layer	81
7.1	CAN bus physical layer specification	82
7.1.1	Physical connector specification	82
7.1.2	CAN bus physical layer parameters	89
7.2	Hardware design recommendations	90
7.2.1	Non-uniform transport redundancy	90
7.2.2	Bus power supply	90

List of tables

2.1	Data type taxonomy	5
2.2	Published message properties	6
2.3	Service request/response properties	7
4.1	Common transfer properties	12
4.2	Service request transfer properties	13
4.3	Service response transfer properties	13
4.4	Transfer CRC algorithm parameters	16
4.5	Transfer reception state variables	19
4.6	CAN ID fields for message transfers	23
4.7	CAN ID fields for service transfers	23
4.8	CAN transfer priority level mapping	24
4.9	Tail byte structure	25
4.10	CAN frame data segments for single-frame transfers	26
4.11	CAN frame data segments for multi-frame transfers (except the last CAN frame of the transfer)	26
4.12	CAN frame data segments for multi-frame transfers (the last CAN frame of the transfer)	26
5.1	Port identifier distribution	31
5.2	Index of the nested namespace “uavcan.node.port”	36
5.3	Index of the nested namespace “uavcan.time”	37
5.4	Index of the nested namespace “uavcan.si”	38
5.5	Index of the nested namespace “uavcan.primitive”	39
5.6	Index of the nested namespace “uavcan.file”	39
5.7	Index of the nested namespace “uavcan.register”	40
5.8	Index of the nested namespace “uavcan.pnp”	41
5.9	Index of the nested namespace “uavcan.internet”	41
6.1	Index of the root namespace “uavcan”	43
7.1	Standard CAN connector types	82
7.2	UAVCAN D-Sub connector pinout	83
7.3	UAVCAN M8 connector pinout	85
7.4	UAVCAN Micro connector pinout	87
7.5	Standard CAN 2.0 PHY parameters	89

List of figures

2.1	UAVCAN architectural diagram.	6
3.1	Data type name structure.	9
3.2	Data type definition file name structure.	9
3.3	DSDL directory structure example.	10
4.1	CAN ID structure	23
5.1	Coordinate frame conventions.	31
7.1	UAVCAN D-Sub device connector example.	84
7.2	UAVCAN D-Sub cable connector example.	84
7.3	UAVCAN M8 connector pair example.	86
7.4	UAVCAN M8 assembled connector pair example.	86
7.5	UAVCAN Micro right-angle connectors with a twisted pair patch cable connected.	88
7.6	UAVCAN Micro CAN bus termination plug.	88
7.7	Non-uniform transport redundancy.	90
7.8	Simplified conceptual power sinking node design schematic.	90
7.9	Simplified conceptual power sourcing node design schematic.	91

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 footnotes² 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.

```
1 | # This is a source code listing.  
2 | print('Hello world!')
```

It is assumed that a byte always contains eight bits.

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 monitoring³, 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 frame⁴. UAVCAN should perform automatic data decomposition and reassembly at the protocol level, hiding the related complexity from the application.

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

Support for redundant interfaces and redundant nodes — UAVCAN 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.

²This is a footnote.

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

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

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

UAVCAN-based networks can accommodate up to 128 nodes on the same logical bus.

UAVCAN supports an unlimited number of 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”). 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 65536 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 least⁵ 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 Maintenance of the standard data type set

The UAVCAN maintainers are charged with advancing the standard data type set based on input from adopters. This feedback is gathered via the official discussion forum⁶ which is open to the general public.

The set of standard data type definitions⁷ is an integral part of the specification; however, there is only a small set of required 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 COTS⁸ equipment manufactured by different vendors. See chapter 5 for more information.

Within the same major version of the specification, the set of 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 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.
 - 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 website⁹.

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.

⁵Depending on the transport protocol.

⁶Please refer to uavcan.org.

⁷The *data structure description language* (DSDL) and related concepts are described in chapter 3.

⁸Commercial off-the-shelf equipment.

⁹uavcan.org

- 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.

DRAFT

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 identifier¹⁰ — *node ID* — ranging from 0 to 127, inclusively. 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 interactions¹¹.

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).

2.1.2 Data types

2.1.2.1 Data type definitions

Message and service data types¹² are defined using the *data structure description language* (DSDL) (chapter 3). A DSDL definition specifies the name, major version, minor version, the data structure, and an optional predefined port ID of the data type. Message data types always contain exactly one data structure, whereas service data types contain two data structures: one for request, and the other for response.

If a port identifier is specified in a data type definition, such identifier is said to be a *fixed port identifier* (by inheritance, the same naming principle applies to *fixed subject ID* and *fixed service ID*). A data type can be used with an arbitrary number of different port identifiers assigned on a per-application basis, but not more than one fixed port identifier.

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

¹⁰Here and elsewhere in this specification, *ID* and *identifier* are used interchangeably unless specifically indicated otherwise.

¹¹Like remote procedure call (RPC).

¹²Data types are the core concept of UAVCAN.

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*¹³.

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 collisions¹⁴. This restriction shall be followed at all times by all compliant implementations and systems¹⁵.

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 <i>“regulated port ID”</i> .	Definitions distributed separately from the UAVCAN specification. Fixed port identifiers are <i>not allowed</i> .
Private	Nonexistent category.	Definitions that are not available to anyone except their authors. Fixed port identifiers are permitted (although not recommended) as long as the definition is only used by its authors; they are called <i>“unregulated fixed port ID”</i> .

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 port identifiers) are segregated into three categories: application-specific (i.e., non-fixed or unregulated) identifiers, regulated non-standard port identifiers, and standard port identifiers. Fixed unregulated port identifiers can be allocated in the same range where application-specific identifiers are. The ranges are documented in chapter 5.

Data type authors that want to release regulated data type definitions or contribute to the standard data type set should contact the UAVCAN maintainers for coordination. The maintainers will choose unoccupied fixed port identifiers for use with the new definitions, if necessary. Since the set of regulated definitions is maintained in a highly centralized manner, it can be statically ensured that no identifier collisions will take place within it; also, since the identifier ranges used with regulated definitions are segregated, regulated port-IDs will not conflict with any other compliant UAVCAN node or system¹⁶.

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 data structures are exchanged by means of the transport layer (chapter 4), which implements automatic decomposition of long transfers into several transport frames¹⁷ and re-

¹³The 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.

¹⁴ 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.

¹⁵ 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.

¹⁶The 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.

¹⁷Here 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.

assembly from these transport frames back into a single atomic data block, allowing nodes to exchange data structures of arbitrary size.

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.

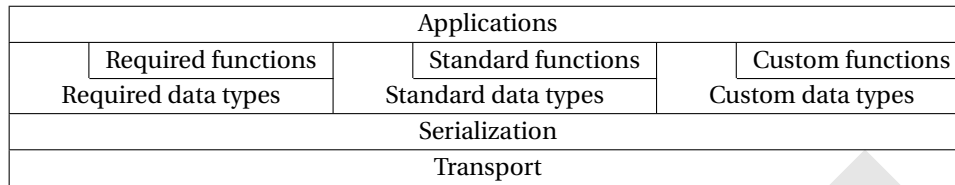


Figure 2.1: UAVCAN architectural diagram.

2.2 Message publication

Message publication refers to the transmission of a serialized data structure 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 data structure.
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. The properties of this field 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.

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 are¹⁸:

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.

¹⁸The 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.

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 data structure.
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. The properties of this field 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.

3 Data structure description language

The data structure description language, or *DSDL*, is a simple domain-specific language designed for defining compound data types. The defined data types are used for exchanging data between UAVCAN nodes via one of the standard UAVCAN transport protocols¹⁹.

This chapter of the specification is incomplete. Track the progress and submit proposals on the forum at <https://forum.uavcan.org>.

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 includes one data structure which forms the message body, and a service data type includes two data structures: one of them is used for service request transfers (from client to server), and the other is used for response transfers (from the server back to the client).

Following the deterministic nature of UAVCAN, the size of all data structures is bounded and statically known. Variable-size entities always have a fixed size limit defined by the data type designer.

DSDL definitions are statically typed.

DSDL provides well-defined means of data type versioning, which enable 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 structure 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 processor*.

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.

¹⁹The 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.

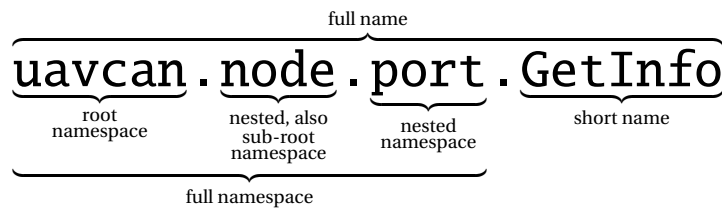


Figure 3.1: Data type name structure.

Data type names are case-sensitive. However, data type names that differ only in letter case are not permitted²⁰.

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 ??.

The length of a full data type name must not exceed 50 characters²¹.

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 ID²² value specified.

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

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

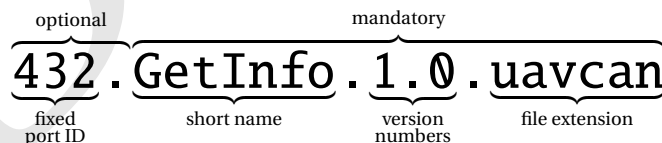


Figure 3.2: Data type definition file name structure.

DSDL namespaces are represented as directories, where one directory defines exactly one namespace, possibly nested. The name of the directory defines the name of its data type name component. A directory defining a namespace will always define said namespace in its entirety, meaning that the contents of a namespace cannot be spread across different directories sharing the same name. One directory cannot define more than one level of nesting²³.

An example DSDL directory structure is shown on the figure 3.2.

²⁰Because that may cause problems with case-insensitive file systems.

²¹This includes the name component separators.

²²Chapter 2.

²³For example, “foo.bar” is not a valid directory name. The valid representation would be “bar” nested in “foo”.

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 definition contains either one or two data structures, for message types and service types, respectively.

A data structure is an ordered, possibly empty collection of *fields* and/or unordered, possibly empty collection of *constants*. The value of a regular data structure is a function of the values of all of its fields. There is one special case of *tagged unions* — a tagged union is a special data structure for which at least two fields are defined, but it is capable of holding exactly one field at any given time. The value of a tagged union is a function of which field it is holding at the moment and the value of said field.

A field represents a named dynamically assigned value of a statically defined type that can be exchanged over the network as a member of its containing data structure. A field can be either of a *primitive type* (i.e., a type which is defined in this DSDL specification, known to DSDL processing tools a priori, which need not be defined by the user; for example, an integer) or a *compound type* (i.e., a type which is defined in DSDL notation; for example, a standard message type `uavcan.node.Heartbeat`). Fixed-size and variable-size array types are also available.

A padding field is a special kind of field which is used for data alignment purposes. Padding fields are not named.

A constant represents a named statically assigned value of a statically defined type that is provided directly in the DSDL source text by the data type designer. Constants are never exchanged over the network, since they are assumed to be known to all involved nodes by virtue of them sharing the same static definition of the data type.

There is a generic term — *attribute* — which is used to refer to either a field (including padding fields) or a constant.

Data type definitions can also contain auxiliary elements, 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).

This chapter of the specification is incomplete. Track the progress and submit proposals on the forum at <https://forum.uavcan.org>.

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 it²⁴. 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 metadata²⁵.

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.

²⁴The 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.

²⁵Such as the subject ID and the source node ID.

Table 4.1: Common transfer properties

Property	Description
Payload	The serialized data structure.
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 data structure. 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.

The set of message subject ID values that can be used with anonymous messages may be limited depending on the transport protocol in use²⁶. It is guaranteed, however, that the subject ID values in the range from 65528 (FFF8₁₆) to 65535 (FFFF₁₆), inclusively, are always available for use with anonymous messages regardless of the transport in use.

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.

²⁶This is considered to be an acceptable limitation because anonymous transfers are intended for an extremely limited set of use cases.

Table 4.2: Service request transfer properties

Property	Description
Payload	The serialized service request data structure.
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: <ul style="list-style-type: none"> 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 data structure.
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 one²⁷. 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.

²⁷The 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 communication²⁷; specifically:

²⁷One 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

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 problem²⁸ (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 structure²⁹) 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 CRC³⁰.
- 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

because it is 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.

²⁸This 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.

²⁹For example, simple static variables.

³⁰CRC stands for “cyclic redundancy check”, an error-detecting code added to data transmissions to reduce the likelihood of undetected data corruption.

byte order, and then emits the resulting byte set in chunks as an ordered sequence of transport frames, where the 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 transfers³¹. 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 basis³². 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 implementations³³ 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,...,56,57) → 29B1 ₁₆

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

³¹For 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.

³²Bit-level errors at the transport frame level may compromise the error-detecting properties of the transfer CRC.

³³Such as CAN FD.


```

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

3  #include <iostream>
4  #include <stdint>
5  #include <stddef>

6  class TransferCRC
7  {
8      std::uint16_t value_ = 0xFFFFU;

9  public:
10     void add(std::uint8_t byte)
11     {
12         value_ ^= static_cast<std::uint16_t>(byte) << 8U;
13         for (std::uint8_t bit = 8; bit > 0; --bit)
14         {
15             if ((value_ & 0x8000U) != 0)
16             {
17                 value_ = (value_ << 1U) ^ 0x1021U;
18             }
19             else
20             {
21                 value_ = value_ << 1U;
22             }
23         }
24     }

25     void add(const std::uint8_t* bytes, std::size_t length)
26     {
27         while (length-- > 0)
28         {
29             add(*bytes++);
30         }
31     }

32     [[nodiscard]] std::uint16_t get() const { return value_; }
33 };

34 int main()
35 {
36     TransferCRC crc;
37     crc.add(reinterpret_cast<const std::uint8_t*>("123456789"), 9);
38     std::cout << std::hex << "0x" << crc.get() << std::endl; // Outputs 0x29B1
39     return 0;
40 }

```

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 transfers³⁴.

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 synchro-

³⁴In single-frame transfers, transport frame deduplication is based on the transfer ID counter.

nization algorithm leveraged by UAVCAN (documented in chapter 5 of the specification).

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 desired³⁵. 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 arithmetic³⁶:

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

The *half range* of transfer ID is 16.

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

```

1 // UAVCAN transfer ID forward distance computation algorithm implemented in C++.
2 // License: CC0, no copyright reserved.
3
4 #include <stdint>
5 #include <iostream>
6 #include <cassert>
7
8 constexpr std::uint8_t TransferIDBitLength = 5; // Defined by the specification
9
10 [[nodiscard]]
11 constexpr std::uint8_t computeForwardDistance(std::uint8_t a, std::uint8_t b)
12 {
13     constexpr std::uint8_t MaxValue = (1U << TransferIDBitLength) - 1U;
14     assert((a <= MaxValue) && (b <= MaxValue));
15
16     std::int16_t d = static_cast<std::int16_t>(b) - static_cast<std::int16_t>(a);
17     if (d < 0)
18     {
19         d += 1U << TransferIDBitLength;
20     }
21
22     assert(d >= 0);
23     assert(d <= MaxValue);
24     assert(((a + d) & MaxValue) == b);
25     return static_cast<std::uint8_t>(d);
26 }
27
28 int main()
29 {
30     assert(0 == computeForwardDistance(0, 0));
31     assert(1 == computeForwardDistance(0, 1));
32     assert(7 == computeForwardDistance(0, 7));
33     assert(0 == computeForwardDistance(7, 7));
34     assert(31 == computeForwardDistance(31, 30)); // overflow
35     assert(1 == computeForwardDistance(31, 0)); // overflow
36     return 0;
37 }
```

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

³⁵Heterogeneous 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.

³⁶For example: $A = 0, B = 0, F \rightarrow 0$; $A = 0, B = 5, F \rightarrow 5$; $A = 5, B = 0, F \rightarrow 27$; $A = 31, B = 30, F \rightarrow 31$; $A = 31, B = 0, F \rightarrow 1$.

at the protocol level is functionally equivalent.

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*³⁷, 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.

³⁷Such behavior is not recommended for hard real-time applications, where deterministic static look-up tables should be preferred instead.

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

```

1  // Constants:
2  tid_timeout := 2 seconds;
3  tid_half_range := 16;
4  iface_switch_delay := UserDefinedConstant; // Or autodetect

5  // State variables:
6  initialized := 0;
7  payload;
8  this_transfer_timestamp;
9  current_transfer_id;
10 iface_index;
11 toggle;

12 function receiveFrame(frame)
13 {
14     // Resolving the state flags:
15     tid_timed_out := (frame.timestamp - this_transfer_timestamp) > tid_timeout;
16     same_iface := frame.iface_index == iface_index;
17     first_frame := frame.start_of_transfer;
18     non_wrapped_tid := computeForwardDistance(current_transfer_id, frame.transfer_id) < tid_half_range;
19     not_previous_tid := computeForwardDistance(frame.transfer_id, current_transfer_id) > 1;
20     iface_switch_allowed := (frame.timestamp - this_transfer_timestamp) > iface_switch_delay;
21     // Using the state flags from above, deciding whether we need to reset:
22     need_restart :=
23         (!initialized) or
24         (tid_timed_out) or
25         (same_iface and first_frame and not_previous_tid) or
26         (iface_switch_allowed and first_frame and non_wrapped_tid);

27     if (need_restart)
28     {
29         initialized := 1;
30         iface_index := frame.iface_index;
31         current_transfer_id := frame.transfer_id;
32         payload.clear();
33         toggle := 0;
34         if (!first_frame)
35         {
36             current_transfer_id.increment();
37             return; // Ignore this frame, since the start of the transfer has already been missed
38         }
39     }

40     if (frame.iface_index != iface_index)
41     {
42         return; // Wrong interface, ignore
43     }

44     if (frame.toggle != toggle)
45     {
46         return; // Unexpected toggle bit, ignore
47     }

48     if (frame.transfer_id != current_transfer_id)
49     {
50         return; // Unexpected transfer ID, ignore
51     }

52     if (first_frame)
53     {
54         this_transfer_timestamp := frame.timestamp;
55     }

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

58     if (frame.last_frame)
59     {
60         // CRC validation for multi-frame transfers is intentionally omitted for brevity
61         processTransfer(payload, ...);
62         current_transfer_id.increment();
63         toggle := 0;
64         payload.clear();
65     }
66 }

```

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.

```

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

3  // State variables:
4  initialized := 0;
5  payload;
6  this_transfer_timestamp;
7  current_transfer_id;
8  toggle;

9  function receiveFrame(frame)
10 {
11     // Resolving the state flags:
12     tid_timed_out := (frame.timestamp - this_transfer_timestamp) > tid_timeout;
13     first_frame := frame.start_of_transfer;
14     not_previous_tid := computeForwardDistance(frame.transfer_id, current_transfer_id) > 1;
15     // Using the state flags from above, deciding whether we need to reset:
16     need_restart :=
17         (!initialized) or
18         (tid_timed_out) or
19         (first_frame and not_previous_tid);

20     if (need_restart)
21     {
22         initialized := 1;
23         current_transfer_id := frame.transfer_id;
24         payload.clear();
25         toggle := 0;
26         if (!first_frame)
27         {
28             current_transfer_id.increment();
29             return; // Ignore this frame, since the start of the transfer has already been missed
30         }
31     }

32     if (frame.toggle != toggle)
33     {
34         return; // Unexpected toggle bit, ignore
35     }

36     if (frame.transfer_id != current_transfer_id)
37     {
38         return; // Unexpected transfer ID, ignore
39     }

40     if (first_frame)
41     {
42         this_transfer_timestamp := frame.timestamp;
43     }

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

46     if (frame.last_frame)
47     {
48         // CRC validation for multi-frame transfers is intentionally omitted for brevity
49         processTransfer(payload, ...);
50         current_transfer_id.increment();
51         toggle := 0;
52         payload.clear();
53     }
54 }
```

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.

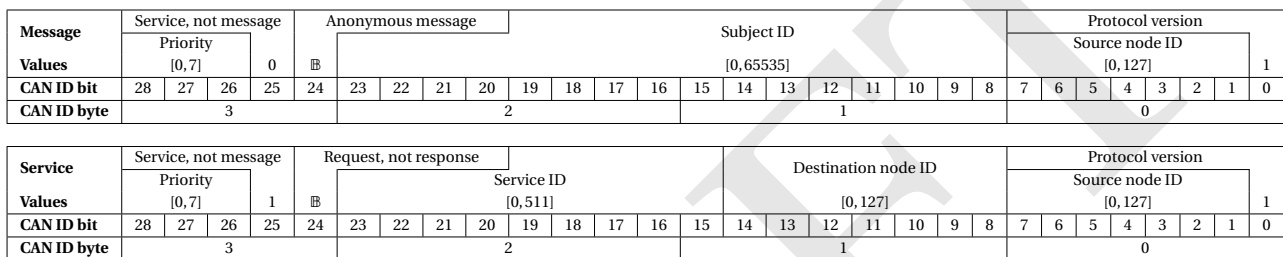


Figure 4.1: CAN ID structure

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, 65535] (any)	Subject identifier of the current message.
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 data structure (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 ID³⁸: 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 messages³⁹ 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 payloads⁴⁰. 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 detected⁴¹. 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 control⁴² should be used at the application level for further conflict resolution⁴³.

³⁸Subject ID or service ID.

³⁹Source node identifier is not defined for anonymous message transfers; see table 4.1.

⁴⁰As explained in section 4.1.2.1, anonymous message transfers are always single-frame transfers.

⁴¹I.e., if the CAN controller uses automatic retransmission, it should be disabled for anonymous frames.

⁴²E.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.

⁴³The 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 5.2. Deterministic applications are advised to avoid reliance on anonymous mes-

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

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 data structure). 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 3 2 1 0	Transfer ID	Modulo 32 (range [0, 31]) section 4.2.1 (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.

sages completely.

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 data structure, 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 data structure).
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 data structure). 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 order⁴⁴ 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 data structure).
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.

⁴⁴Most 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 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.

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 re-transmitted, the remote nodes will see it duplicated. Additionally, this approach increases the complexity of the driver and can possibly affect its throughput and latency.

Most CAN controllers offer a 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:

```

1 // Returns the index of the TX mailbox that can be used for the transmission of the newFrame
2 // If none are available, returns -1.
3 getFreeMailboxIndex(newFrame)
4 {
5     chosen_mailbox = -1 // By default, assume that no mailboxes are available
6
7     for i = 0...NumberOfTxMailboxes
8     {
9         if isTxMailboxFree(i)
10        {
11            chosen_mailbox = i
12            // Note: cannot break here, must check all other mailboxes as well.
13        }
14        else
15        {
16            if not isFramePriorityHigher(newFrame, getFrameFromTxMailbox(i))
17            {
18                chosen_mailbox = -1
19                break // Denied - must wait until this mailbox has finished transmitting
20            }
21        }
22    }
23
24    return chosen_mailbox
25 }
```

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:

$$\begin{aligned} m_M(R_A, R_B, M_A, M_B) &= M_A \wedge M_B \wedge \neg(R_A \oplus R_B) \\ m_R(R_A, R_B, M_A, M_B) &= R_A \wedge m_M(R_A, R_B, M_A, M_B) \end{aligned}$$

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:

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

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.

^aNotation: \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 structures and exchange them 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.

Unless stated otherwise, all of the high-level functions described in this chapter are optional, meaning that they do not necessarily have to be supported or implemented for an application or system to be UAVCAN-compliant. Excepting a very few mandatory functions and rules, users of UAVCAN are allowed to pick only those that are needed and suitable for their applications.

If a designer chooses to deviate from the rules and conventions presented in this chapter, every such deviation should be explicitly documented.

5.1 Application-level conventions

5.1.1 Identifier distribution

An overview of related concepts is provided in chapter 2.

5.1.1.1 Node ID

Valid values of node ID range from 0 to 127, inclusive.

The node ID values 126 and 127 are reserved for diagnostic and debugging tools; these node ID values should not be used in fielded systems.

5.1.1.2 Port ID

This rule is mandatory for all applications.

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.

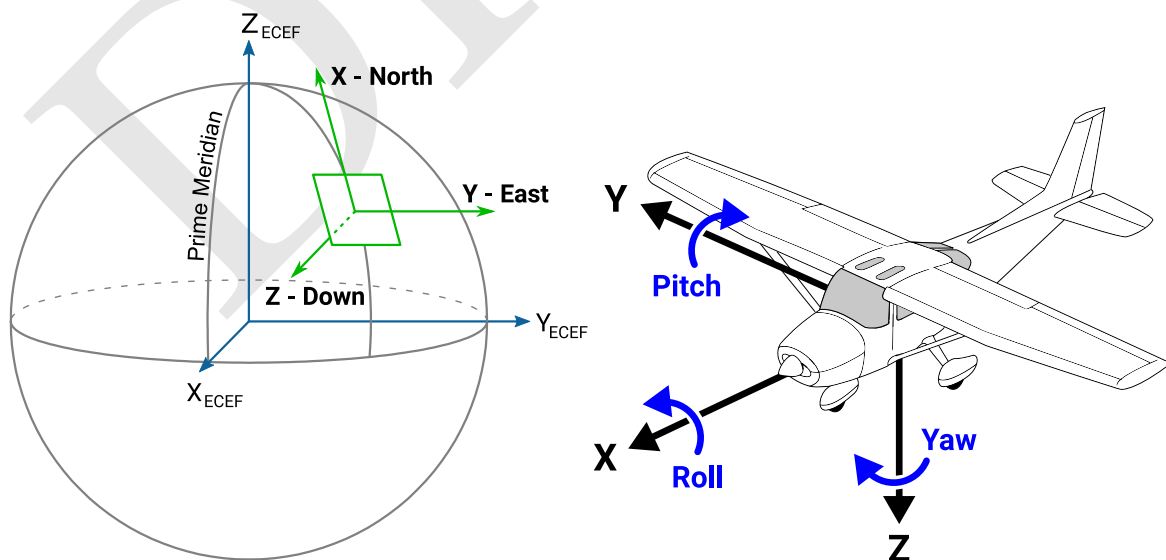
Table 5.1: Port identifier distribution

Subject ID	Service ID	Purpose
[0, 32767]	[0, 127]	Unregulated identifiers (both fixed and non-fixed).
[57344, 59391]	[256, 319]	Non-standard regulated identifiers (i.e., vendor-specific).
[62804, 65535]	[384, 511]	Standard regulated identifiers.

5.1.2 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.

5.1.2.1 *World frame*

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

X — northward;

Y — eastward;

Z — down.

5.1.2.2 *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.

5.1.2.3 *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.

5.1.3 **Rotation representation**

All applications should represent rotations using quaternions with the elements ordered as follows⁴⁵: 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.

5.1.4 **Matrix representation**

5.1.4.1 *General*

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

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

5.1.4.2 *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.

⁴⁵Assuming $w + xi + yj + zk$.

An array of $\frac{(1+n)n}{2}$ elements represents a symmetric matrix, where array members represent the elements of the upper-right triangle arranged in the row-major order.

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

This form is well-suited for covariance matrix representation.

An array of n elements represents a diagonal matrix, where an array member at position i (where $i = 1$ for the first element) represents the matrix element $x_{i,i}$ (where $x_{1,1}$ is the upper-left element).

$$\begin{bmatrix} a & 0 & 0 \\ 0 & b & 0 \\ 0 & 0 & c \end{bmatrix} \rightarrow (a, b, c)$$

An array of one element represents a scalar matrix.

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

An empty array represents a zero matrix.

5.1.4.3 Covariance matrices

A zero covariance matrix represents an unknown covariance⁴⁶.

Infinite error variance means that the associated value is undefined.

5.1.5 Physical quantity representation

All units should be SI⁴⁷ 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 units⁴⁸ or scaled SI units⁴⁹ should be suffixed with the standard abbreviation of the unit⁵⁰ and its metric prefix⁵¹ (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.

```

1 float16 temperature           # [kelvin] Suffix not needed because an unscaled SI unit is used here.
2 uint24 delay_us              # [microsecond] Scaled SI unit, suffix required. Mu replaced with "u".
3 uint24 MAX_DELAY_us = 600000 # [microsecond] Notice the letter case.
4 float32 kinetic_energy_GJ    # [gigajoule] Notice the letter case.
5 float16 estimated_charge_mAh # [milliampere hour] Scaled non-SI unit. Discouraged, use coulomb.
6 float16 MAX_CHARGE_mAh = 1e4 # [milliampere hour] Notice the letter case.
```

⁴⁶As described above, an empty array represents a zero matrix, from which follows that an empty array represents unknown covariance.

⁴⁷International System of Units.

⁴⁸E.g., degree Celsius instead of kelvin.

⁴⁹E.g., microsecond instead of second.

⁵⁰E.g., kg for kilogram, J for joule.

⁵¹E.g., M for mega, n for nano.

5.2 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, except for the node heartbeat feature (section 5.2.2), 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.

5.2.1 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 5.2.2.

5.2.2 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 54) 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 54.

```

1  #
2  # Abstract node status information.
3  #
4  # All UAVCAN nodes are required to publish this message periodically.
5  # This is the only high-level function that must be implemented by all nodes.
6  #
7  # The default subject ID 32085 is 1111101010101 in binary. The alternating bit pattern at the end
8  # helps transceiver synchronization (e.g., on CAN-based networks) and on some transports permits
9  # automatic bit rate detection.
10 #
11
12 # The publication period must not exceed this limit.
13 # The period should not change while the node is running.
14 uint16 MAX_PUBLICATION_PERIOD = 1 # [second]
15
16 # If the last message from the node was received more than this amount of time ago, it should be considered offline.
17 uint16 OFFLINE_TIMEOUT = 3 # [second]
18
19 # The uptime seconds counter should never overflow. The counter will reach the upper limit in ~136 years,
20 # upon which time it should stay at 0xFFFFFFFF until the node is restarted.
21 #
22 # Other nodes may detect that a remote node has restarted when this value leaps backwards.
23 uint32 uptime # [second]
24
25 # Abstract node health information. See constants below.
26 # Follows:
27 # https://www.law.cornell.edu/cfr/text/14/23.1322
28 # https://www.faa.gov/documentLibrary/media/Advisory_Circular/AC_25.1322-1.pdf section 6
29 truncated uint2 health
30
31 # The node is functioning properly (nominal).
32 uint2 HEALTH_NOMINAL = 0
33
34 # A critical parameter went out of range or the node encountered a minor failure that does not prevent
35 # the subsystem from performing any of its real-time functions.
36 uint2 HEALTH_ADVISORY = 1
37
38 # The node encountered a major failure and is performing in a degraded mode or outside of its designed limitations.
39 uint2 HEALTH_CAUTION = 2
40
41 # The node suffered a fatal malfunction and is unable to perform its intended function.
42 uint2 HEALTH_WARNING = 3
43
44 # The current operating mode. See constants below.
45 #
46 # The mode OFFLINE can be used for informing other network participants that the sending node has ceased its
47 # activities or about to do so. In this case, other nodes will not have to wait for the OFFLINE_TIMEOUT to
48 # expire before detecting that the sending node is no longer available.
49 #
50 # Reserved values can be used in future revisions of the specification.
51 truncated uint3 mode
52
53 # Normal operating mode.
54 uint3 MODE_OPERATIONAL = 0
55
56 # Initialization is in progress; this mode is entered immediately after startup.
57 uint3 MODE_INITIALIZATION = 1
58
59 # E.g. calibration, self-test, etc.
60 uint3 MODE_MAINTENANCE = 2

```



```

61
62 # New software/firmware is being loaded or the bootloader is running.
63 uint3 MODE_SOFTWARE_UPDATE = 3
64
65 # The node is no longer available.
66 uint3 MODE_OFFLINE = 7
67
68 # Optional, vendor-specific node status code, e.g. a fault code or a status bitmask.
69 truncated uint19 vendor_specific_status_code
70
71 @assert _offset_ % 8 == {0}
72 @assert _offset_.max <= 56 # Must fit into one CAN 2.0 frame (least capable transport, smallest MTU)

```

5.2.3 Generic node information

The service `uavcan.node.GetInfo` (section 6.4.2 on page 53) 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 53.

```

1 #
2 # Full node info request.
3 # All of the returned information must be static (unchanged) while the node is running.
4 # It is highly recommended to support this service on all nodes.
5 #
6
7 ---
8
9 # The UAVCAN protocol version implemented on this node, both major and minor.
10 # Not to be changed while the node is running.
11 Version.1.0 protocol_version
12
13 # The version information must not be changed while the node is running.
14 # The correct hardware version must be reported at all times, excepting software-only nodes, in which
15 # case it should be set to zeros.
16 # If the node is equipped with a UAVCAN-capable bootloader, the bootloader should report the software
17 # version of the installed application, if there is any; if no application is found, zeros should be reported.
18 Version.1.0 hardware_version
19 Version.1.0 software_version
20
21 # A version control system (VCS) revision number or hash. Not to be changed while the node is running.
22 # For example, this field can be used for reporting the short git commit hash of the current
23 # software revision.
24 # Set to zero if not used.
25 uint64 software_vcs_revision_id
26
27 # The unique node ID is a 128-bit long sequence that is likely to be globally unique per node.
28 # The vendor must ensure that the probability of a collision with any other node UID globally is negligibly low.
29 # UID is defined once per hardware unit and should never be changed.
30 # All zeros is not a valid UID.
31 # If the node is equipped with a UAVCAN-capable bootloader, the bootloader must use the same UID.
32 uint8[16] unique_id
33
34 # Manual serialization note: only fixed-size fields up to this point. The following fields are dynamically sized.
35 @assert _offset_ == {30 * 8}
36
37 # Human-readable non-empty ASCII node name. An empty name is not permitted.
38 # The name must not be changed while the node is running.
39 # Allowed characters are: a-z (lowercase ASCII letters) 0-9 (decimal digits) . (dot) - (dash) _ (underscore).
40 # Node name is a reversed Internet domain name (like Java packages), e.g. "com.manufacturer.project.product".
41 void2
42 uint8[<=50] name
43
44 # The value of an arbitrary hash function applied to the software image. Not to be changed while the node is running.
45 # This field can be used to detect whether the software or firmware running on the node is an exact
46 # same version as a certain specific revision. This field provides a very strong identity guarantee,
47 # unlike the version fields above, which can be the same for different builds of the software.
48 # As can be seen from its definition, this field is optional.
49 #
50 # The exact hash function and the methods of its application are implementation-defined.
51 # However, implementations are recommended to adhere to the following guidelines, fully or partially:
52 # - The hash function should be CRC-64-WE.
53 # - The hash function should be applied to the entire application image padded to 8 bytes.
54 # - If the computed image CRC is stored within the software image itself, the value of
55 #   the hash function becomes ill-defined, because it becomes recursively dependent on itself.
56 #   In order to circumvent this issue, while computing or checking the CRC, its value stored
57 #   within the image should be zeroed out.
58 void7
59 uint64[<=1] software_image_crc
60
61 # The certificate of authenticity (COA) of the node, 222 bytes max, optional. This field can be used for
62 # reporting digital signatures (e.g., RSA-1776, or ECDSA if a higher degree of cryptographic strength is desired).
63 # Leave empty if not used. Not to be changed while the node is running.
64 uint8[<=222] certificate_of_authenticity
65
66 @assert _offset_ % 8 == {0}
67 @assert _offset_.max == (313 * 8) # At most five CAN FD frames

```

5.2.4 Bus data flow monitoring

The combination of the following three services defined in the namespace `uavcan.node.port` (section 6.5 on page 55) (see table 5.2) enables a highly capable tool of network inspection and monitoring:

- `uavcan.node.port.List` (section 6.5.3 on page 56) — 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 55) — returns the static (unchanging or infrequently changing) information about the selected subject or service.
- `uavcan.node.port.GetStatistics` (section 6.5.2 on page 56) — 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	55 6.5.1	⇌ uavcan.node.port.GetInfo
GetStatistics	0.1	433	3	15	56 6.5.2	⇌ uavcan.node.port.GetStatistics
List	0.1	431	3	257	56 6.5.3	⇌ uavcan.node.port.List
ID	1.0		3		57 6.5.4	☑ uavcan.node.port.ID
ServiceID	1.0		2		57 6.5.5	☑ uavcan.node.port.ServiceID
SubjectID	1.0		2		57 6.5.6	☑ uavcan.node.port.SubjectID

5.2.5 Network-wide time synchronization

UAVCAN provides a simple and robust method of time synchronization⁵² that is built upon the work “Implementing a Distributed High-Resolution Real-Time Clock using the CAN-Bus” published by M. Gergeleit and H. Streich⁵³. 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 67).

An important service type `uavcan.time.GetSynchronizationMasterInfo` (section 6.9.1 on page 67) 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.

⁵²The 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.

⁵³Proceedings of the 1st international CAN-Conference 94, Mainz, 13.-14. Sep. 1994, CAN in Automation e.V., Erlangen.

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	67 6.9.1 ⇌ uavcan.time.GetSynchronizationMasterInfo
Synchronization	1.0	31744	7		67 6.9.2 ☑ uavcan.time.Synchronization
SynchronizedAmbiguousTimestamp	1.0		3		69 6.9.3 ☑ uavcan.time.SynchronizedAmbiguousTimestamp
SynchronizedTimestamp	1.0		7		69 6.9.4 ☑ uavcan.time.SynchronizedTimestamp
TimeSystem	0.1		1		70 6.9.5 ☑ uavcan.time.TimeSystem

5.2.6 Primitive types and physical quantities

The namespaces `uavcan.si` (section 6.13 on page 75) and `uavcan.primitive` (section 6.10 on page 71) 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 parties⁵⁴, 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.

5.2.6.1 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 69). 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.

⁵⁴As 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	75 6.13.1	☑ uavcan.si.acceleration.Scalar
Vector3	1.0		15	75 6.13.2	☑ uavcan.si.acceleration.Vector3
angle					
Quaternion	1.0		19	75 6.14.1	☑ uavcan.si.angle.Quaternion
Scalar	1.0		7	76 6.14.2	☑ uavcan.si.angle.Scalar
angular_velocity					
Scalar	1.0		7	76 6.15.1	☑ uavcan.si.angular_velocity.Scalar
Vector3	1.0		15	76 6.15.2	☑ uavcan.si.angular_velocity.Vector3
duration					
Scalar	1.0		7	76 6.16.1	☑ uavcan.si.duration.Scalar
WideScalar	1.0		11	76 6.16.2	☑ uavcan.si.duration.WideScalar
electric_charge					
Scalar	1.0		7	76 6.17.1	☑ uavcan.si.electric_charge.Scalar
electric_current					
Scalar	1.0		7	77 6.18.1	☑ uavcan.si.electric_current.Scalar
energy					
Scalar	1.0		7	77 6.19.1	☑ uavcan.si.energy.Scalar
length					
Scalar	1.0		7	77 6.20.1	☑ uavcan.si.length.Scalar
Vector3	1.0		15	77 6.20.2	☑ uavcan.si.length.Vector3
WideVector3	1.0		27	77 6.20.3	☑ uavcan.si.length.WideVector3
magnetic_field_strength					
Scalar	1.0		7	78 6.21.1	☑ uavcan.si.magnetic_field_strength.Scalar
Vector3	1.0		15	78 6.21.2	☑ uavcan.si.magnetic_field_strength.Vector3
mass					
Scalar	1.0		7	78 6.22.1	☑ uavcan.si.mass.Scalar
power					
Scalar	1.0		7	78 6.23.1	☑ uavcan.si.power.Scalar
pressure					
Scalar	1.0		7	78 6.24.1	☑ uavcan.si.pressure.Scalar
temperature					
Scalar	1.0		7	79 6.25.1	☑ uavcan.si.temperature.Scalar
velocity					
Scalar	1.0		7	79 6.26.1	☑ uavcan.si.velocity.Scalar
Vector3	1.0		15	79 6.26.2	☑ uavcan.si.velocity.Vector3
voltage					
Scalar	1.0		7	79 6.27.1	☑ uavcan.si.voltage.Scalar
volume					
Scalar	1.0		7	79 6.28.1	☑ uavcan.si.volume.Scalar
volumetric_flow_rate					
Scalar	1.0		7	80 6.29.1	☑ uavcan.si.volumetric_flow_rate.Scalar

5.2.6.2 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 5.2.10).

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	71 6.10.1	<input checked="" type="checkbox"/> uavcan.primitive.Empty
String	1.0		256	71 6.10.2	<input checked="" type="checkbox"/> uavcan.primitive.String
Unstructured	1.0		256	71 6.10.3	<input checked="" type="checkbox"/> uavcan.primitive.Unstructured
array					
Bit	1.0		256	71 6.11.1	<input checked="" type="checkbox"/> uavcan.primitive.array.Bit
Integer8	1.0		256	71 6.11.2	<input checked="" type="checkbox"/> uavcan.primitive.array.Integer8
Integer16	1.0		257	72 6.11.3	<input checked="" type="checkbox"/> uavcan.primitive.array.Integer16
Integer32	1.0		257	72 6.11.4	<input checked="" type="checkbox"/> uavcan.primitive.array.Integer32
Integer64	1.0		257	72 6.11.5	<input checked="" type="checkbox"/> uavcan.primitive.array.Integer64
Natural8	1.0		256	72 6.11.6	<input checked="" type="checkbox"/> uavcan.primitive.array.Natural8
Natural16	1.0		257	72 6.11.7	<input checked="" type="checkbox"/> uavcan.primitive.array.Natural16
Natural32	1.0		257	72 6.11.8	<input checked="" type="checkbox"/> uavcan.primitive.array.Natural32
Natural64	1.0		257	73 6.11.9	<input checked="" type="checkbox"/> uavcan.primitive.array.Natural64
Real16	1.0		257	73 6.11.10	<input checked="" type="checkbox"/> uavcan.primitive.array.Real16
Real32	1.0		257	73 6.11.11	<input checked="" type="checkbox"/> uavcan.primitive.array.Real32
Real64	1.0		257	73 6.11.12	<input checked="" type="checkbox"/> uavcan.primitive.array.Real64
scalar					
Bit	1.0		1	73 6.12.1	<input checked="" type="checkbox"/> uavcan.primitive.scalar.Bit
Integer8	1.0		1	73 6.12.2	<input checked="" type="checkbox"/> uavcan.primitive.scalar.Integer8
Integer16	1.0		2	74 6.12.3	<input checked="" type="checkbox"/> uavcan.primitive.scalar.Integer16
Integer32	1.0		4	74 6.12.4	<input checked="" type="checkbox"/> uavcan.primitive.scalar.Integer32
Integer64	1.0		8	74 6.12.5	<input checked="" type="checkbox"/> uavcan.primitive.scalar.Integer64
Natural8	1.0		1	74 6.12.6	<input checked="" type="checkbox"/> uavcan.primitive.scalar.Natural8
Natural16	1.0		2	74 6.12.7	<input checked="" type="checkbox"/> uavcan.primitive.scalar.Natural16
Natural32	1.0		4	74 6.12.8	<input checked="" type="checkbox"/> uavcan.primitive.scalar.Natural32
Natural64	1.0		8	74 6.12.9	<input checked="" type="checkbox"/> uavcan.primitive.scalar.Natural64
Real16	1.0		2	75 6.12.10	<input checked="" type="checkbox"/> uavcan.primitive.scalar.Real16
Real32	1.0		4	75 6.12.11	<input checked="" type="checkbox"/> uavcan.primitive.scalar.Real32
Real64	1.0		8	75 6.12.12	<input checked="" type="checkbox"/> uavcan.primitive.scalar.Real64

5.2.7 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 46), 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 46 6.2.1	<input checked="" type="checkbox"/> uavcan.file.GetInfo
List	0.1	406	121	117 46 6.2.2	<input checked="" type="checkbox"/> uavcan.file.List
Modify	1.0	407	230	2 47 6.2.3	<input checked="" type="checkbox"/> uavcan.file.Modify
Read	1.0	408	118	260 47 6.2.4	<input checked="" type="checkbox"/> uavcan.file.Read
Write	1.0	409	247	2 48 6.2.5	<input checked="" type="checkbox"/> uavcan.file.Write
Error	1.0		2	48 6.2.6	<input checked="" type="checkbox"/> uavcan.file.Error
Path	1.0		113	48 6.2.7	<input checked="" type="checkbox"/> uavcan.file.Path

5.2.8 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 52). The service also allows vendors to define vendor-specific commands alongside the standard ones.

It is recommended to support this service in all nodes.

5.2.9 Node software update

A simple software⁵⁵ update protocol is defined on top of the remote file system interface (section 5.2.7) and the generic node commands (section 5.2.8).

The process of software update involves the following data types:

- `uavcan.node.ExecuteCommand` (section 6.4.1 on page 52) – used to initiate the software update pro-

⁵⁵Or firmware – UAVCAN does not distinguish between the two.

cess.

- `uavcan.file.Read` (section 6.2.4 on page 47) – 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 52). The protocol is considered simple enough to be usable in embedded bootloaders with small memory-constrained microcontrollers.

5.2.10 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 UAVCAN⁵⁶ 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 64); 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 65).

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	64 6.8.1	⇕ uavcan.register.Access
List	0.1	385	2 51	65 6.8.2	⇕ uavcan.register.List
Name	0.1		51	65 6.8.3	☑ uavcan.register.Name
Value	0.1		258	65 6.8.4	☑ uavcan.register.Value

5.2.11 Diagnostics and event logging

The message type `uavcan.diagnostic.Record` (section 6.1.1 on page 45) is designed to facilitate emission of human-readable diagnostic messages and event logging, both for the needs of real-time display⁵⁷ and for long-term storage⁵⁸.

5.2.12 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.

⁵⁶And, possibly, other interfaces.

⁵⁷E.g., messages displayed to a human operator in real time.

⁵⁸E.g., flight data recording.

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 58) (where *pnp* stands for “plug-and-play”) (see table 5.8). The protocol is described in the documentation for the data type `uavcan.pnp.NodeIDAllocationData` (section 6.6.1 on page 58).

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”⁵⁹ 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	58 6.6.1	☑ uavcan.pnp.NodeIDAllocationData
NodeIDAllocationDataMTU8	0.1	32742	9	60 6.6.2	☑ uavcan.pnp.NodeIDAllocationDataMTU8
cluster					
AppendEntries	1.0	390	33	5 61 6.7.1	⇄ uavcan.pnp.cluster.AppendEntries
Discovery	1.0	32740	11	62 6.7.2	☑ uavcan.pnp.cluster.Discovery
RequestVote	1.0	391	9	5 62 6.7.3	⇄ uavcan.pnp.cluster.RequestVote
Entry	1.0		22	62 6.7.4	☑ uavcan.pnp.cluster.Entry

5.2.13 Internet/LAN forwarding interface

Data types defined in the namespace `uavcan.internet` (section 6.3 on page 49) (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*⁶⁰ (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 communication⁶¹.

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 49 6.3.1	⇄ uavcan.internet.udp.HandleIncomingPacket
OutgoingPacket	0.1	32750	307	50 6.3.2	☑ uavcan.internet.udp.OutgoingPacket

⁵⁹Proceedings of USENIX Annual Technical Conference, p. 305-320, 2014.

⁶⁰Normally, a modem node would be implemented using on-board cellular, radio frequency, or satellite communication hardware.

⁶¹Information 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 definitions⁶² 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.

⁶²I.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	45 6.1.1	☑ uavcan.diagnostic.Record
Severity	1.0		1	45 6.1.2	☑ uavcan.diagnostic.Severity
file					
GetInfo	0.1	405	113 21	46 6.2.1	⇕ uavcan.file.GetInfo
List	0.1	406	121 117	46 6.2.2	⇕ uavcan.file.List
Modify	1.0	407	230 2	47 6.2.3	⇕ uavcan.file.Modify
Read	1.0	408	118 260	47 6.2.4	⇕ uavcan.file.Read
Write	1.0	409	247 2	48 6.2.5	⇕ uavcan.file.Write
Error	1.0		2	48 6.2.6	☑ uavcan.file.Error
Path	1.0		113	48 6.2.7	☑ uavcan.file.Path
internet					
udp					
HandleIncomingPacket	0.1	500	313 7	49 6.3.1	⇕ uavcan.internet.udp.HandleIncomingPacket
OutgoingPacket	0.1	32750	307	50 6.3.2	☑ uavcan.internet.udp.OutgoingPacket
node					
ExecuteCommand	0.1	435	115 7	52 6.4.1	⇕ uavcan.node.ExecuteCommand
GetInfo	0.1	430	0 313	53 6.4.2	⇕ uavcan.node.GetInfo
GetTransportStatistics	0.1	434	0 61	53 6.4.3	⇕ uavcan.node.GetTransportStatistics
Heartbeat	1.0	32085	7	54 6.4.4	☑ uavcan.node.Heartbeat
ID	1.0		2	55 6.4.5	☑ uavcan.node.ID
IOStatistics	0.1		15	55 6.4.6	☑ uavcan.node.IOStatistics
Version	1.0		2	55 6.4.7	☑ uavcan.node.Version
port					
GetInfo	0.1	432	3 54	55 6.5.1	⇕ uavcan.node.port.GetInfo
GetStatistics	0.1	433	3 15	56 6.5.2	⇕ uavcan.node.port.GetStatistics
List	0.1	431	3 257	56 6.5.3	⇕ uavcan.node.port.List
ID	1.0		3	57 6.5.4	☑ uavcan.node.port.ID
ServiceID	1.0		2	57 6.5.5	☑ uavcan.node.port.ServiceID
SubjectID	1.0		2	57 6.5.6	☑ uavcan.node.port.SubjectID
pnp					
NodeIDAllocationData	0.1	32741	18	58 6.6.1	☑ uavcan.pnp.NodeIDAllocationData
NodeIDAllocationDataMTU8	0.1	32742	9	60 6.6.2	☑ uavcan.pnp.NodeIDAllocationDataMTU8
cluster					
AppendEntries	1.0	390	33 5	61 6.7.1	⇕ uavcan.pnp.cluster.AppendEntries
Discovery	1.0	32740	11	62 6.7.2	☑ uavcan.pnp.cluster.Discovery
RequestVote	1.0	391	9 5	62 6.7.3	⇕ uavcan.pnp.cluster.RequestVote
Entry	1.0		22	62 6.7.4	☑ uavcan.pnp.cluster.Entry
register					
Access	0.1	384	309 265	64 6.8.1	⇕ uavcan.register.Access
List	0.1	385	2 51	65 6.8.2	⇕ uavcan.register.List
Name	0.1		51	65 6.8.3	☑ uavcan.register.Name
Value	0.1		258	65 6.8.4	☑ uavcan.register.Value
time					
GetSynchronizationMasterInfo	0.1	510	0 7	67 6.9.1	⇕ uavcan.time.GetSynchronizationMasterInfo
Synchronization	1.0	31744	7	67 6.9.2	☑ uavcan.time.Synchronization
SynchronizedAmbiguousTimestamp	1.0		3	69 6.9.3	☑ uavcan.time.SynchronizedAmbiguousTimestamp
SynchronizedTimestamp	1.0		7	69 6.9.4	☑ uavcan.time.SynchronizedTimestamp
TimeSystem	0.1		1	70 6.9.5	☑ uavcan.time.TimeSystem
primitive					
Empty	1.0		0	71 6.10.1	☑ uavcan.primitive.Empty
String	1.0		256	71 6.10.2	☑ uavcan.primitive.String
Unstructured	1.0		256	71 6.10.3	☑ uavcan.primitive.Unstructured
array					
Bit	1.0		256	71 6.11.1	☑ uavcan.primitive.array.Bit
Integer8	1.0		256	71 6.11.2	☑ uavcan.primitive.array.Integer8
Integer16	1.0		257	72 6.11.3	☑ uavcan.primitive.array.Integer16
Integer32	1.0		257	72 6.11.4	☑ uavcan.primitive.array.Integer32
Integer64	1.0		257	72 6.11.5	☑ uavcan.primitive.array.Integer64
Natural8	1.0		256	72 6.11.6	☑ uavcan.primitive.array.Natural8
Natural16	1.0		257	72 6.11.7	☑ uavcan.primitive.array.Natural16
Natural32	1.0		257	72 6.11.8	☑ uavcan.primitive.array.Natural32
Natural64	1.0		257	73 6.11.9	☑ uavcan.primitive.array.Natural64
Real16	1.0		257	73 6.11.10	☑ uavcan.primitive.array.Real16
Real32	1.0		257	73 6.11.11	☑ uavcan.primitive.array.Real32
Real64	1.0		257	73 6.11.12	☑ uavcan.primitive.array.Real64

scalar					
Bit	1.0	1	73 6.12.1	<input checked="" type="checkbox"/>	uavcan.primitive.scalar.Bit
Integer8	1.0	1	73 6.12.2	<input checked="" type="checkbox"/>	uavcan.primitive.scalar.Integer8
Integer16	1.0	2	74 6.12.3	<input checked="" type="checkbox"/>	uavcan.primitive.scalar.Integer16
Integer32	1.0	4	74 6.12.4	<input checked="" type="checkbox"/>	uavcan.primitive.scalar.Integer32
Integer64	1.0	8	74 6.12.5	<input checked="" type="checkbox"/>	uavcan.primitive.scalar.Integer64
Natural8	1.0	1	74 6.12.6	<input checked="" type="checkbox"/>	uavcan.primitive.scalar.Natural8
Natural16	1.0	2	74 6.12.7	<input checked="" type="checkbox"/>	uavcan.primitive.scalar.Natural16
Natural32	1.0	4	74 6.12.8	<input checked="" type="checkbox"/>	uavcan.primitive.scalar.Natural32
Natural64	1.0	8	74 6.12.9	<input checked="" type="checkbox"/>	uavcan.primitive.scalar.Natural64
Real16	1.0	2	75 6.12.10	<input checked="" type="checkbox"/>	uavcan.primitive.scalar.Real16
Real32	1.0	4	75 6.12.11	<input checked="" type="checkbox"/>	uavcan.primitive.scalar.Real32
Real64	1.0	8	75 6.12.12	<input checked="" type="checkbox"/>	uavcan.primitive.scalar.Real64
si					
acceleration					
Scalar	1.0	7	75 6.13.1	<input checked="" type="checkbox"/>	uavcan.si.acceleration.Scalar
Vector3	1.0	15	75 6.13.2	<input checked="" type="checkbox"/>	uavcan.si.acceleration.Vector3
angle					
Quaternion	1.0	19	75 6.14.1	<input checked="" type="checkbox"/>	uavcan.si.angle.Quaternion
Scalar	1.0	7	76 6.14.2	<input checked="" type="checkbox"/>	uavcan.si.angle.Scalar
angular_velocity					
Scalar	1.0	7	76 6.15.1	<input checked="" type="checkbox"/>	uavcan.si.angular_velocity.Scalar
Vector3	1.0	15	76 6.15.2	<input checked="" type="checkbox"/>	uavcan.si.angular_velocity.Vector3
duration					
Scalar	1.0	7	76 6.16.1	<input checked="" type="checkbox"/>	uavcan.si.duration.Scalar
WideScalar	1.0	11	76 6.16.2	<input checked="" type="checkbox"/>	uavcan.si.duration.WideScalar
electric_charge					
Scalar	1.0	7	76 6.17.1	<input checked="" type="checkbox"/>	uavcan.si.electric_charge.Scalar
electric_current					
Scalar	1.0	7	77 6.18.1	<input checked="" type="checkbox"/>	uavcan.si.electric_current.Scalar
energy					
Scalar	1.0	7	77 6.19.1	<input checked="" type="checkbox"/>	uavcan.si.energy.Scalar
length					
Scalar	1.0	7	77 6.20.1	<input checked="" type="checkbox"/>	uavcan.si.length.Scalar
Vector3	1.0	15	77 6.20.2	<input checked="" type="checkbox"/>	uavcan.si.length.Vector3
WideVector3	1.0	27	77 6.20.3	<input checked="" type="checkbox"/>	uavcan.si.length.WideVector3
magnetic_field_strength					
Scalar	1.0	7	78 6.21.1	<input checked="" type="checkbox"/>	uavcan.si.magnetic_field_strength.Scalar
Vector3	1.0	15	78 6.21.2	<input checked="" type="checkbox"/>	uavcan.si.magnetic_field_strength.Vector3
mass					
Scalar	1.0	7	78 6.22.1	<input checked="" type="checkbox"/>	uavcan.si.mass.Scalar
power					
Scalar	1.0	7	78 6.23.1	<input checked="" type="checkbox"/>	uavcan.si.power.Scalar
pressure					
Scalar	1.0	7	78 6.24.1	<input checked="" type="checkbox"/>	uavcan.si.pressure.Scalar
temperature					
Scalar	1.0	7	79 6.25.1	<input checked="" type="checkbox"/>	uavcan.si.temperature.Scalar
velocity					
Scalar	1.0	7	79 6.26.1	<input checked="" type="checkbox"/>	uavcan.si.velocity.Scalar
Vector3	1.0	15	79 6.26.2	<input checked="" type="checkbox"/>	uavcan.si.velocity.Vector3
voltage					
Scalar	1.0	7	79 6.27.1	<input checked="" type="checkbox"/>	uavcan.si.voltage.Scalar
volume					
Scalar	1.0	7	79 6.28.1	<input checked="" type="checkbox"/>	uavcan.si.volume.Scalar
volumetric_flow_rate					
Scalar	1.0	7	80 6.29.1	<input checked="" type="checkbox"/>	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]

```

1  #
2  # Generic human-readable text message for logging and displaying purposes.
3  # Generally it should be published at the lowest priority level.
4  #
5
6  # Optional timestamp in the network-synchronized time system; zero if undefined.
7  # The timestamp value conveys the exact moment when the reported event took place.
8  uavcan.time.SynchronizedTimestamp.1.0 timestamp
9
10 # Standard severity, 3 bit wide.
11 Severity.1.0 severity
12
13 # Message text.
14 # Normally, messages should be kept as short as possible, especially those of high severity.
15 void6
16 uint8[<=112] text
17
18 @assert _offset_ % 8 == {0}
19 @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

```

1  #
2  # Generic message severity representation.
3  #
4
5  # The severity level ranging from 0 to 7, where low values represent low-severity (unimportant) messages, and
6  # high values represent high-severity (important) messages. Several mnemonics for the severity levels are
7  # defined below. Nodes are advised to implement output filtering mechanisms, allowing users to select
8  # the minimal severity for emitted messages; messages of the selected and higher severity levels will
9  # be published, and messages of lower severity will be suppressed (discarded).
10 uint3 value
11
12 # Messages of this severity can be used only during development.
13 # They must not be used in a fielded operational system.
14 uint3 TRACE = 0
15
16 # Messages that can aid in troubleshooting.
17 # Messages of this severity and lower should be disabled by default.
18 uint3 DEBUG = 1
19
20 # General informational messages of low importance.
21 # Messages of this severity and lower should be disabled by default.
22 uint3 INFO = 2
23
24 # General informational messages of high importance.
25 # Messages of this severity and lower should be disabled by default.
26 uint3 NOTICE = 3
27
28 # Messages reporting abnormalities and warning conditions.
29 # Messages of this severity and higher should be enabled by default.
30 uint3 WARNING = 4
31
32 # Messages reporting problems and error conditions.
33 # Messages of this severity and higher should be enabled by default.
34 uint3 ERROR = 5
35
36 # Messages reporting serious problems and critical conditions.
37 # Messages of this severity and higher should be always enabled.
38 uint3 CRITICAL = 6
39
40 # Notifications of dangerous circumstances that demand immediate attention.
41 # Messages of this severity should be always enabled.
42 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

```

1  #
2  # Information about a remote file system entry (file, directory, etc).
3  #
4
5  Path.1.0 path
6
7  ---
8
9  # Result of the operation
10 Error.1.0 error
11
12 # File size in bytes. Should be set to zero for directories.
13 truncated uint40 size
14
15 # The UNIX Epoch time when the entry was last modified. Zero if unknown.
16 truncated uint40 unix_timestamp_of_last_modification
17
18 # If such entry does not exist, all flags should be cleared/ignored.
19 bool is_file_not_directory # True if file, false if directory
20 bool is_link # This is a link to another entry; the above flag indicates the type of the target
21 bool is_readable # The item can be read by the caller (applies to files and directories)
22 bool is_writable # The item can be written by the caller (applies to files and directories)
23 void4
24
25 # Reserved for future use
26 void64
27
28 @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]

```

1  #
2  # This service can be used to list a remote directory, one entry per request.
3  #
4  # The client should query each entry independently, iterating 'entry_index' from 0 until the last entry.
5  # When the index reaches the number of elements in the directory, the server will report that there is
6  # no such entry by returning an empty name.
7  #
8  # The field entry_index must be applied to an ordered list of directory entries (e.g. alphabetically ordered).
9  # The exact sorting criteria does not matter as long as it provides the same ordering for subsequent service calls.
10 #
11 # Observe that this listing operation is fundamentally non-atomic. The caller shall beware of possible race conditions
12 # and is responsible for handling them properly. Particularly, consider what happens if a new item is inserted into
13 # the directory between two subsequent calls: if the item happened to be inserted at the index that is lower than the
14 # index of the next request, the next returned item (or several, if more items were inserted) will repeat the ones
15 # that were listed earlier. The caller should handle that properly, either by ignoring the repeated items or by
16 # restarting the listing operation from the beginning (index 0).
17 #
18
19 uint32 entry_index
20
21 void32 # Reserved for future use
22
23 Path.1.0 directory_path
24
25 @assert _offset_ % 8 == {0}
26
27 ---
28
29 void32 # Reserved for future use
30
31 # The base name of the referenced entry, i.e., relative to the outer directory.
32 # The outer directory path is not included to conserve bandwidth.
33 # Empty if such entry does not exist.
34 #
35 # For example, suppose there is a file "/foo/bar/baz.bin". Listing the directory with the path "/foo/bar/" (the slash
36 # at the end is optional) at the index 0 will return "baz.bin". Listing the same directory at the index 1 (or any
37 # higher) will return an empty name "", indicating that the caller has reached the end of the list.
38 Path.1.0 entry_base_name
39
40 @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

```

1  #
2  # Manipulate a remote file system entry. Applies to files, directories, and links alike.
3  # If the remote entry is a directory, all nested entries will be affected, too.
4  #
5  # The server should perform all operations atomically, unless atomicity is not supported by
6  # the underlying file system.
7  #
8  # Atomic copying can be effectively employed by remote nodes before reading or after writing
9  # the file to minimize the possibility of race conditions.
10 # For example, before reading a large file from the server, the client might opt to create
11 # a temporary copy of it first, then read the copy, and delete it upon completion. Likewise,
12 # a similar strategy can be employed for writing, where the file is first written at a
13 # temporary location, and then moved to its final destination. These approaches, however,
14 # may lead to creation of dangling temporary files if the client failed to dispose of them
15 # properly, so that risk should be taken into account.
16 #
17 # Move/Copy
18 #   Specify the source path and the destination path.
19 #   If the source does not exist, the operation will fail.
20 #   Set the preserve_source flag to copy rather than move.
21 #   If the destination exists and overwrite_destination is not set, the operation will fail.
22 #   If the target path includes non-existent directories, they will be created (like "mkdir -p").
23 #
24 # Touch
25 #   Specify the destination path and make the source path empty.
26 #   If the path exists (file/directory/link), its modification time will be updated.
27 #   If the path does not exist, an empty file will be created.
28 #   If the target path includes non-existent directories, they will be created (like "mkdir -p").
29 #   Flags are ignored.
30 #
31 # Remove
32 #   Specify the source path (file/directory/link) and make the destination path empty.
33 #   Fails if the path does not exist.
34 #   Flags are ignored.
35 #
36
37 bool preserve_source      # Do not remove the source. Used to copy instead of moving.
38 bool overwrite_destination # If the destination exists, remove it beforehand.
39 void30
40
41 Path.1.0 source
42 Path.1.0 destination
43
44 @assert _offset_ % 8 == {0}
45
46 ---
47
48 Error.1.0 error
49
50 @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]

```

1  #
2  # Read file from a remote node.
3  #
4  # There are two possible outcomes of a successful call:
5  # 1. Data array size equals its capacity. This means that the end of the file is not reached yet.
6  # 2. Data array size is less than its capacity, possibly zero. This means that the end of the file is reached.
7  #
8  # Thus, if the client needs to fetch the entire file, it should repeatedly call this service while increasing the
9  # offset, until a non-full data array is returned.
10 #
11 # If the object pointed by 'path' cannot be read (e.g. it is a directory or it does not exist), an appropriate error
12 # code will be returned, and the data array will be empty.
13 #
14 # It is easy to see that this protocol is prone to race conditions because the remote file can be modified
15 # between read operations which might result in the client obtaining a damaged file. To combat this,
16 # application designers are recommended to adhere to the following convention. Let every file whose integrity
17 # is of interest have a hash or a digital signature, which is stored in an adjacent file under the same name
18 # suffixed with the appropriate extension according to the type of hash or digital signature used.
19 # For example, let there be file "image.bin", integrity of which must be ensured by the client upon downloading.
20 # Suppose that the file is hashed using SHA-256, so the appropriate file extension for the hash would be
21 # ".sha256". Following this convention, the hash of "image.bin" would be stored in "image.bin.sha256".
22 # After downloading the file, the client would read the hash (being small, the hash can be read in a single

```

```

23 # request) and check it against a locally computed value. Some servers may opt to generate such hash files
24 # automatically as necessary; for example, if such file is requested but it does not exist, the server would
25 # compute the necessary signature or hash (the type of hash/signature can be deduced from the requested file
26 # extension) and return it as if the file existed. Obviously, this would be impractical for very large files;
27 # in that case, hash/signature should be pre-computed and stored in a real file. If this approach is followed,
28 # implementers are advised to use only SHA-256 for hashing, in order to reduce the number of fielded
29 # incompatible implementations.
30 #
31
32 truncated uint40 offset
33
34 Path.1.0 path
35
36 @assert _offset_ % 8 == {0}
37
38 ---
39
40 Error.1.0 error
41
42 void7
43 uint8[<=256] data
44
45 @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

```

1 #
2 # Write into a remote file.
3 # The server shall place the contents of the field 'data' into the file pointed by 'path' at the offset specified by
4 # the field 'offset'.
5 #
6 # When writing a file, the client should repeatedly call this service with data while advancing the offset until the
7 # file is written completely. When the write sequence is completed, the client shall call the service one last time,
8 # with the offset set to the size of the file and with the data field empty, which will signal the server that the
9 # transfer is finished.
10 #
11 # When the write operation is complete, the server shall truncate the resulting file past the specified offset.
12 #
13
14 truncated uint40 offset
15
16 Path.1.0 path
17
18 uint8[<=128] data
19
20 @assert _offset_ % 8 == {0}
21 @assert _offset_.max / 8 <= 300
22
23 ---
24
25 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

```

1 #
2 # Nested type.
3 # Result of a file system operation.
4 #
5
6 uint16 OK = 0
7 uint16 UNKNOWN_ERROR = 65535
8
9 uint16 NOT_FOUND = 2
10 uint16 IO_ERROR = 5
11 uint16 ACCESS_DENIED = 13
12 uint16 IS_DIRECTORY = 21 # I.e. attempted read/write on a path that points to a directory
13 uint16 INVALID_VALUE = 22 # E.g. file name is not valid for the target file system
14 uint16 FILE_TOO_LARGE = 27
15 uint16 OUT_OF_SPACE = 28
16 uint16 NOT_SUPPORTED = 38
17
18 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]

```

1  #
2  # Nested type.
3  # A file system path encoded in UTF8. The only valid separator is the forward slash "/".
4  # A single slash ("/") refers to the root directory (the location of which is defined by the server).
5  # Relative references (e.g. "..") are not defined and not permitted (although this may change in the future).
6  # Conventions (not enforced):
7  #   - A path pointing to a file or a link to file should not end with a separator.
8  #   - A path pointing to a directory or to a link to directory should end with a separator.
9  #
10 # The maximum path length limit is chosen as a trade-off between compatibility with deep directory structures and
11 # the worst-case transfer length. The limit is 112 bytes, which allows all transfers containing a single instance
12 # of path and no other large data chunks to fit into two CAN FD frames.
13 #
14
15 uint8 SEPARATOR = '/'
16
17 void1
18 uint8[<=112] path
19
20 @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

```

1  #
2  # This message carries UDP packets sent from a remote host on the Internet or a LAN to a node on the local UAVCAN bus.
3  # Please refer to the definition of the message type OutgoingPacket for a general overview of the packet forwarding
4  # logic.
5  #
6  # This data type has been made a service type rather than a message type in order to make its transfers addressable,
7  # allowing nodes to employ hardware acceptance filters for filtering out forwarded datagrams that are not addressed
8  # to them. Additionally, requiring the destination nodes to always respond upon reception of the forwarded datagram
9  # opens interesting opportunities for future extensions of the forwarding protocol.
10 #
11 # It should be noted that this data type definition intentionally leaves out the source address. This is done in
12 # order to simplify the implementation, reduce the bus traffic overhead, and because the nature of the
13 # communication patterns proposed by this set of messages does not provide a valid way to implement server hosts
14 # on the local UAVCAN bus. It is assumed that local nodes can be only clients, and therefore, they will be able to
15 # determine the address of the sender simply by mapping the field session_id to their internally maintained states.
16 # Furthermore, it is uncertain what is the optimal way of representing the source address for
17 # client nodes: it is assumed that the local nodes will mostly use DNS names rather than IP addresses, so if there
18 # was a source address field, modem nodes would have to perform reverse mapping from the IP address they received
19 # the datagram from to the corresponding DNS name that was used by the local node with the outgoing message. This
20 # approach creates a number of troubling corner cases and adds a fair amount of hidden complexities to the
21 # implementation of modem nodes.
22 #
23 # It is recommended to perform service invocations at the same transfer priority level as was used for broadcasting
24 # the latest matching message of type OutgoingPacket. However, meeting this recommendation would require the modem
25 # node to implement additional logic, which may be undesirable. Therefore, implementers are free to deviate from
26 # this recommendation and resort to a fixed priority level instead. In the case of a fixed priority level, it is
27 # advised to use the lowest transfer priority level.
28 #
29
30 # This field must contain the same value that was used by the local node when sending the corresponding outgoing
31 # packet using the message type OutgoingPacket. This value will be used by the local node to match the response
32 # with its local context.
33 uint16 session_id
34
35 # Effective payload. This data will be forwarded from the remote host verbatim.
36 # UDP packets that contain more than 508 bytes of payload may be dropped by some types of
37 # communication equipment. Refer to RFC 791 and 2460 for an in-depth review.
38 # UAVCAN further limits the maximum packet size to reduce the memory and traffic burden on the nodes.
39 # Datagrams that exceed the capacity of this field should be discarded by the modem node.
40 void7
41 uint8[<=309] payload
42
43 @assert _offset_ % 8 == {0}
44 @assert _offset_.max == (313 * 8) # At most five CAN FD frames
45
46 ---
47
48 # If the service invocation times out, the modem node is permitted to remove the corresponding entry from
49 # the NAT table immediately, not waiting for its TTL to expire.
50
51 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]

```

1  #
2  # This message carries UDP packets from a node on the local bus to a remote host on the Internet or a LAN.
3  #
4  # Any node can broadcast a message of this type.
5  #
6  # All nodes that are capable of communication with the Internet or a LAN should subscribe to messages
7  # of this type and forward the payload to the indicated host and port using exactly one UDP datagram
8  # per message (i.e. additional fragmentation is to be avoided). Such nodes will be referred to as
9  # "modem nodes".
10 #
11 # It is expected that some systems will have more than one modem node available.
12 # Each modem node is supposed to forward every message it sees, which will naturally create
13 # some degree of modular redundancy and fault tolerance. The remote host should therefore be able to
14 # properly handle possibly duplicated messages from different source addresses, in addition to
15 # possible duplications introduced by the UDP/IP protocol itself. There are at least two obvious
16 # strategies that can be employed by the remote host:
17 #
18 # - Accept only the first message, ignore duplicates. This approach requires that the UDP stream
19 #   should contain some metadata necessary for the remote host to determine the source and ordering
20 #   of each received datum. This approach works best for periodic data, such as telemetry, where
21 #   the sender does not expect any responses.
22 #
23 # - Process all messages, including duplicates. This approach assumes that the remote host acts
24 #   as a server, processing all received requests and providing responses to each. This arrangement
25 #   implies that the client may receive duplicated responses. It is therefore the client's
26 #   responsibility to resolve the possible ambiguity. An obvious solution is to accept the first
27 #   arrived response and ignore the later ones.
28 #
29 # Applications are free to choose whatever redundancy management strategy works best for them.
30 #
31 # If the source node expects that the remote host will send some data back, it must explicitly notify
32 # the modem nodes about this, so that they could prepare to perform reverse forwarding when the
33 # expected data arrives from the remote host. The technique of reverse forwarding is known in
34 # networking as IP Masquerading, or (in general) Network Address Translation (NAT). The notification
35 # is performed by means of setting one of the corresponding flags defined below.
36 #
37 # In order to be able to match datagrams received from remote hosts and the local nodes they should
38 # be forwarded to, modem nodes are required to keep certain metadata about outgoing datagrams. Such
39 # metadata is stored in a data structure referred to as "NAT table", where every entry would normally
40 # contain at least the following fields:
41 # - The local UDP port number that was used to send the outgoing datagram from.
42 # - Per RFC 4787, the port number is chosen by the modem node automatically.
43 # - The node ID of the local node that has sent the outgoing datagram.
44 # - Value of the field session_id defined below.
45 # - Possibly some other data, depending on the implementation.
46 #
47 # The modem nodes are required to keep each NAT table entry for at least NAT_ENTRY_MIN_TTL seconds
48 # since the last reverse forwarding action was performed. Should the memory resources of the modem node
49 # be exhausted, it is allowed to remove old NAT entries earlier, following the policy of least recent use.
50 #
51 # Having received a UDP packet from a remote host, the modem node would check the NAT table in order
52 # to determine where on the UAVCAN bus the received data should be forwarded to. If the NAT table
53 # contains no matches, the received data should be silently dropped. If a match is found, the
54 # modem node will forward the data to the recipient node using the service HandleIncomingPacket.
55 # If the service invocation times out, the modem node is permitted to remove the corresponding entry from
56 # the NAT table immediately (but it is not required). This will ensure that the modem nodes will not be
57 # tasked with translations for client nodes that are no longer online or are unreachable.
58 # Additionally, client nodes will be able to hint the modem nodes to remove translation entries they no
59 # longer need by simply refusing to respond to the corresponding service invocation. Please refer to
60 # the definition of that service data type for a more in-depth review of the reverse forwarding process.
61 #
62 # Modem nodes can also perform traffic shaping, if needed, by means of delaying or dropping UDP
63 # datagrams that exceed the quota.
64 #
65 # To summarize, a typical data exchange occurrence should amount to the following actions:
66 #
67 # - A local UAVCAN node broadcasts a message of type OutgoingPacket with the payload it needs
68 #   to forward. If the node expects the remote host to send any data back, it sets the masquerading flag.
69 #
70 # - Every modem node on the bus receives the message and performs the following actions:
71 #
72 #   - The domain name is resolved, unless the destination address provided in the message
73 #     is already an IP address, in which case this step should be skipped.
74 #
75 #   - The domain name to IP address mapping is added to the local DNS cache, although this
76 #     part is entirely implementation defined and is not required.
77 #
78 #   - The masquerading flag is checked. If it is set, a new entry is added to the NAT table.
79 #     If such entry already existed, its expiration timeout is reset. If no such entry existed
80 #     and a new one cannot be added because of memory limitations, the least recently used
81 #     (i.e. oldest) entry of the NAT table is replaced with the new one.
82 #
83 #   - The payload is forwarded to the determined IP address.
84 #
85 # - At this point, direct forwarding is complete. Should any of the modem nodes receive an incoming
86 #   packet, they would attempt to perform a reverse forwarding according to the above provided algorithm.
87 #
88 # It is recommended to use the lowest transport priority level when broadcasting messages of this type,
89 # in order to avoid interference with a real-time traffic on the bus. Usage of higher priority levels is

```

```

90  # unlikely to be practical because the latency and throughput limitations introduced by the on-board radio
91  # communication equipment are likely to vastly exceed those of the local CAN bus.
92  #
93
94  # Modem nodes are required to keep the NAT table entries alive for at least this amount of time, unless the
95  # table is overflowed, in which case they are allowed to remove least recently used entries in favor of
96  # newer ones. Modem nodes are required to be able to accommodate at least 100 entries in the NAT table.
97  uint32 NAT_ENTRY_MIN_TTL = 86400          # [second]
98
99  # This field is set to an arbitrary value by the transmitting node in order to be able to match the response
100 # with the locally kept context. The function of this field is virtually identical to that of UDP/IP port
101 # numbers. This value can be set to zero safely if the sending node does not have multiple contexts to
102 # distinguish between.
103 uint16 session_id
104
105 # UDP destination port number.
106 uint16 destination_port
107
108 # Option flags.
109 bool use_masquerading    # Expect data back (i.e. instruct the modem to use the NAT table)
110 void7
111
112 # Domain name or IP address where the payload should be forwarded to.
113 # Note that broadcast addresses are allowed here, for example, 255.255.255.255.
114 # Broadcasting with masquerading enabled works the same way as unicasting with masquerading enabled: the modem
115 # node should take care to channel all traffic arriving at the opened port from any source to the node that
116 # requested masquerading.
117 # The full domain name length may not exceed 253 octets, according to the DNS specification.
118 # UAVCAN imposes a stricter length limit in order to reduce the memory and traffic burden on the bus: 45 characters.
119 # 45 characters is the amount of space that is required to represent the longest possible form of an IPv6 address
120 # (an IPv4-mapped IPv6 address). Examples:
121 # "forum.uavcan.org"           - domain name
122 # "192.168.1.1"               - IPv4 address
123 # "2001:0db8:85a3:0000:0000:8a2e:0370:7334" - IPv6 address, full form
124 # "2001:db8:85a3::8a2e:370:7334" - IPv6 address, same as above, short form (preferred)
125 # "ABCD:ABCD:ABCD:ABCD:ABCD:ABCD:192.168.158.190" - IPv4-mapped IPv6, full form (length limit, 45 characters)
126 void2
127 uint8[<=45] destination_address
128
129 # Effective payload. This data will be forwarded to the remote host verbatim.
130 # UDP packets that contain more than 508 bytes of payload may be dropped by some types of
131 # communication equipment. Refer to RFC 791 and 2460 for an in-depth review.
132 # UAVCAN further limits the maximum packet size to reduce the memory and traffic burden on the nodes.
133 uint8[<256] payload
134
135 @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

```

1  #
2  # Instructs the server node to execute or commence execution of a simple predefined command.
3  # All standard commands are optional; i.e., not guaranteed to be supported by all nodes.
4  #
5
6  # Standard pre-defined commands are at the top of the range (defined below).
7  # Vendors can define arbitrary, vendor-specific commands in the bottom part of the range (starting from zero).
8  # Vendor-specific commands must not use identifiers above 32767.
9  uint16 command
10
11 # Reboot the node.
12 # Note that some standard commands may or may not require a restart in order to take effect; e.g., factory reset.
13 uint16 COMMAND_RESTART = 65535
14
15 # Shut down the node; further access will not be possible until the power is turned back on.
16 uint16 COMMAND_POWER_OFF = 65534
17
18 # Begin the software update process using uavcan.file.Read. This command makes use of the "parameter" field below.
19 # The parameter contains the path to the new software image file to be downloaded by the server from the client
20 # using the standard service uavcan.file.Read. Observe that this operation swaps the roles of the client and
21 # the server.
22 #
23 # Upon reception of this command, the server (updatee) will evaluate whether it is possible to begin the
24 # software update process. If that is deemed impossible, the command will be rejected with one of the
25 # error codes defined in the response section of this definition (e.g., BAD_STATE if the node is currently
26 # on-duty and a sudden interruption of its activities is considered unsafe, and so on).
27 # If an update process is already underway, the updatee should abort the process and restart with the new file,
28 # unless the updatee can determine that the specified file is the same file that is already being downloaded,
29 # in which case it is allowed to respond SUCCESS and continue the old update process.
30 # If there are no other conditions precluding the requested update, the updatee will return a SUCCESS and
31 # initiate the file transfer process by invoking the standard service uavcan.file.Read repeatedly until the file
32 # is transferred fully (please refer to the documentation for that data type for more information about its usage).
33 #
34 # While the software is being updated, the updatee should set its mode (the field "mode" in uavcan.node.Heartbeat)
35 # to MODE_SOFTWARE_UPDATE. Please refer to the documentation for uavcan.node.Heartbeat for more information.
36 #
37 # It is recognized that most systems will have to interrupt their normal services to perform the software update
38 # (unless some form of software hot swapping is implemented, as is the case in some high-availability systems).
39 #
40 # Microcontrollers that are requested to update their firmware may need to stop execution of their current firmware
41 # and start the embedded bootloader (although other approaches are possible as well). In that case,
42 # while the embedded bootloader is running, the mode reported via the message uavcan.node.Heartbeat should be
43 # MODE_SOFTWARE_UPDATE as long as the bootloader is running, even if no update-related activities
44 # are currently underway. For example, if the update process failed and the bootloader cannot load the software,
45 # the same mode MODE_SOFTWARE_UPDATE will be reported.
46 # It is also recognized that in a microcontroller setting, the application that served the update request will have
47 # to pass the update-related metadata (such as the node ID of the server and the firmware image file path) to
48 # the embedded bootloader. The tactics of that transaction lie outside of the scope of this specification.
49 uint16 COMMAND_BEGIN_SOFTWARE_UPDATE = 65533
50
51 # Return the node's configuration back to the factory default settings (may require restart).
52 # Due to the uncertainty whether a restart is required, generic interfaces should always force a restart.
53 uint16 COMMAND_FACTORY_RESET = 65532
54
55 # Cease activities immediately, enter a safe state until restarted.
56 # Further operation may no longer be possible until a restart command is executed.
57 uint16 COMMAND_EMERGENCY_STOP = 65531
58
59 # This command instructs the node to store the current configuration parameter values and other persistent states
60 # to the non-volatile storage. Nodes are allowed to manage persistent states automatically, obviating the need for
61 # this command by committing all such data to the non-volatile memory automatically as necessary. However, some
62 # nodes may lack this functionality, in which case this parameter should be used. Generic interfaces should always
63 # invoke this command in order to ensure that the data is stored even if the node doesn't implement automatic
64 # persistence management.
65 uint16 COMMAND_STORE_PERSISTENT_STATES = 65530
66
67 # A string parameter supplied to the command. The format and interpretation is command-specific.
68 # The standard commands do not use this field (ignore it), excepting the following:
69 # - COMMAND_BEGIN_SOFTWARE_UPDATE
70 void1
71 uint8[<=112] parameter
72
73 @assert _offset_ % 8 == {0}
74 @assert _offset_.max <= (124 * 8) # Two CAN FD frames max
75
76 ---
77
78 # The result of the request.
79 uint8 STATUS_SUCCESS = 0 # Started or executed successfully
80 uint8 STATUS_FAILURE = 1 # Could not start or the desired outcome could not be reached
81 uint8 STATUS_NOT_AUTHORIZED = 2 # Denied due to lack of authorization
82 uint8 STATUS_BAD_COMMAND = 3 # The requested command is not known or not supported
83 uint8 STATUS_BAD_PARAMETER = 4 # The supplied parameter cannot be used with the selected command
84 uint8 STATUS_BAD_STATE = 5 # The current state of the node does not permit execution of this command
85 uint8 STATUS_INTERNAL_ERROR = 6 # The operation should have succeeded but an unexpected failure occurred

```

```

86 | uint8 status
87 |
88 | # Reserved for future use
89 | 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]

```

1 | #
2 | # Full node info request.
3 | # All of the returned information must be static (unchanged) while the node is running.
4 | # It is highly recommended to support this service on all nodes.
5 | #
6 |
7 | ---
8 |
9 | # The UAVCAN protocol version implemented on this node, both major and minor.
10 | # Not to be changed while the node is running.
11 | Version.1.0 protocol_version
12 |
13 | # The version information must not be changed while the node is running.
14 | # The correct hardware version must be reported at all times, excepting software-only nodes, in which
15 | # case it should be set to zeros.
16 | # If the node is equipped with a UAVCAN-capable bootloader, the bootloader should report the software
17 | # version of the installed application, if there is any; if no application is found, zeros should be reported.
18 | Version.1.0 hardware_version
19 | Version.1.0 software_version
20 |
21 | # A version control system (VCS) revision number or hash. Not to be changed while the node is running.
22 | # For example, this field can be used for reporting the short git commit hash of the current
23 | # software revision.
24 | # Set to zero if not used.
25 | uint64 software_vcs_revision_id
26 |
27 | # The unique node ID is a 128-bit long sequence that is likely to be globally unique per node.
28 | # The vendor must ensure that the probability of a collision with any other node UID globally is negligibly low.
29 | # UID is defined once per hardware unit and should never be changed.
30 | # All zeros is not a valid UID.
31 | # If the node is equipped with a UAVCAN-capable bootloader, the bootloader must use the same UID.
32 | uint8[16] unique_id
33 |
34 | # Manual serialization note: only fixed-size fields up to this point. The following fields are dynamically sized.
35 | @assert _offset_ == {30 * 8}
36 |
37 | # Human-readable non-empty ASCII node name. An empty name is not permitted.
38 | # The name must not be changed while the node is running.
39 | # Allowed characters are: a-z (lowercase ASCII letters) 0-9 (decimal digits) . (dot) - (dash) _ (underscore).
40 | # Node name is a reversed Internet domain name (like Java packages), e.g. "com.manufacturer.project.product".
41 | void2
42 | uint8[<=50] name
43 |
44 | # The value of an arbitrary hash function applied to the software image. Not to be changed while the node is running.
45 | # This field can be used to detect whether the software or firmware running on the node is an exact
46 | # same version as a certain specific revision. This field provides a very strong identity guarantee,
47 | # unlike the version fields above, which can be the same for different builds of the software.
48 | # As can be seen from its definition, this field is optional.
49 | #
50 | # The exact hash function and the methods of its application are implementation-defined.
51 | # However, implementations are recommended to adhere to the following guidelines, fully or partially:
52 | # - The hash function should be CRC-64-WE.
53 | # - The hash function should be applied to the entire application image padded to 8 bytes.
54 | # - If the computed image CRC is stored within the software image itself, the value of
55 | #   the hash function becomes ill-defined, because it becomes recursively dependent on itself.
56 | #   In order to circumvent this issue, while computing or checking the CRC, its value stored
57 | #   within the image should be zeroed out.
58 | void7
59 | uint64[<=1] software_image_crc
60 |
61 | # The certificate of authenticity (COA) of the node, 222 bytes max, optional. This field can be used for
62 | # reporting digital signatures (e.g., RSA-1776, or ECDSA if a higher degree of cryptographic strength is desired).
63 | # Leave empty if not used. Not to be changed while the node is running.
64 | uint8[<=222] certificate_of_authenticity
65 |
66 | @assert _offset_ % 8 == {0}
67 | @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

```

1  #
2  # Returns a set of general low-level transport statistical counters.
3  # Servers are not required to sample the data atomically.
4  #
5
6  ---
7
8  # UAVCAN transfer performance statistics:
9  # the number of UAVCAN transfers sent, received, and failed.
10 IOStatistics.0.1 transfer_statistics
11
12 # Physical interface statistics, separate per interface.
13 # E.g., for a doubly redundant transport, this array would contain two elements,
14 # the one at the index zero would apply to the first interface, the other to the second interface.
15 void6
16 IOStatistics.0.1[<=3] physical_interface_statistics
17
18 @assert _offset_ % 8 == {0}
19 @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

```

1  #
2  # Abstract node status information.
3  #
4  # All UAVCAN nodes are required to publish this message periodically.
5  # This is the only high-level function that must be implemented by all nodes.
6  #
7  # The default subject ID 32085 is 111110101010101 in binary. The alternating bit pattern at the end
8  # helps transceiver synchronization (e.g., on CAN-based networks) and on some transports permits
9  # automatic bit rate detection.
10 #
11
12 # The publication period must not exceed this limit.
13 # The period should not change while the node is running.
14 uint16 MAX_PUBLICATION_PERIOD = 1 # [second]
15
16 # If the last message from the node was received more than this amount of time ago, it should be considered offline.
17 uint16 OFFLINE_TIMEOUT = 3 # [second]
18
19 # The uptime seconds counter should never overflow. The counter will reach the upper limit in ~136 years,
20 # upon which time it should stay at 0xFFFFFFFF until the node is restarted.
21 #
22 # Other nodes may detect that a remote node has restarted when this value leaps backwards.
23 uint32 uptime # [second]
24
25 # Abstract node health information. See constants below.
26 # Follows:
27 # https://www.law.cornell.edu/cfr/text/14/23.1322
28 # https://www.faa.gov/documentLibrary/media/Advisory_Circular/AC_25.1322-1.pdf section 6
29 truncated uint2 health
30
31 # The node is functioning properly (nominal).
32 uint2 HEALTH_NOMINAL = 0
33
34 # A critical parameter went out of range or the node encountered a minor failure that does not prevent
35 # the subsystem from performing any of its real-time functions.
36 uint2 HEALTH_ADVISORY = 1
37
38 # The node encountered a major failure and is performing in a degraded mode or outside of its designed limitations.
39 uint2 HEALTH_CAUTION = 2
40
41 # The node suffered a fatal malfunction and is unable to perform its intended function.
42 uint2 HEALTH_WARNING = 3
43
44 # The current operating mode. See constants below.
45 #
46 # The mode OFFLINE can be used for informing other network participants that the sending node has ceased its
47 # activities or about to do so. In this case, other nodes will not have to wait for the OFFLINE_TIMEOUT to
48 # expire before detecting that the sending node is no longer available.
49 #
50 # Reserved values can be used in future revisions of the specification.
51 truncated uint3 mode
52
53 # Normal operating mode.
54 uint3 MODE_OPERATIONAL = 0
55
56 # Initialization is in progress; this mode is entered immediately after startup.
57 uint3 MODE_INITIALIZATION = 1
58
59 # E.g. calibration, self-test, etc.
60 uint3 MODE_MAINTENANCE = 2
61
62 # New software/firmware is being loaded or the bootloader is running.
63 uint3 MODE_SOFTWARE_UPDATE = 3
64
65 # The node is no longer available.
66 uint3 MODE_OFFLINE = 7
67
68 # Optional, vendor-specific node status code, e.g. a fault code or a status bitmask.

```



```

69 | truncated uint19 vendor_specific_status_code
70 |
71 | @assert _offset_ % 8 == {0}
72 | @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

```

1 | #
2 | # Defines a node ID.
3 | # The maximum valid value is dependent on the underlying transport layer.
4 | # Values lower than 128 are always valid for all transports.
5 | # Refer to the specification for more info.
6 | #
7 |
8 | uint16 value
9 |
10 | @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

```

1 | #
2 | # A standard set of generic input/output statistical counters that should never overflow (in a realistic scenario).
3 | # If a 40-bit counter is incremented every millisecond, it will overflow in ~35 years.
4 | #
5 |
6 | truncated uint40 num_emitted
7 | truncated uint40 num_received
8 | 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

```

1 | #
2 | # A shortened semantic version representation: only major and minor.
3 | # The protocol generally does not concern itself with the patch version.
4 | #
5 |
6 | uint8 major
7 | 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

```

1 | #
2 | # This service is used to obtain information about a port (either message or service).
3 | # It can be used for advanced network inspections, e.g. for building computational graph maps.
4 | #
5 |
6 | # The port of interest; can be either a service or a message.
7 | void7
8 | ID.1.0 port_id
9 |
10 | @assert _offset_ == {24}

```



```

11 | ---
12 |
13 |
14 | # Flags describing the current port.
15 | # Note that a message port can be both subscriber and publisher at the same time.
16 | bool is_subscriber # Message input port
17 | bool is_publisher  # Message output port
18 | bool is_server     # Server port
19 | void5
20 |
21 | # If such port exists, this field must contain the full name of its data type.
22 | # If the requested port does not exist, this field must be empty.
23 | # If this field is empty, the caller must ignore all other fields.
24 | void2
25 | uint8[<=50] data_type_full_name
26 |
27 | # The version numbers of the data type used at this port.
28 | uavcan.node.Version.1.0 data_type_version
29 |
30 | @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

```

1 | #
2 | # This service is used to obtain statistical metrics of a port (either message or service).
3 | # It can be used for advanced network inspections, e.g. for building real-time traffic maps.
4 | #
5 |
6 | # The port of interest; can be either a service or a message.
7 | void7
8 | ID.1.0 port_id
9 |
10 | @assert _offset_ == {24}
11 |
12 | ---
13 |
14 | # For messages, "emitted" applies to publications, and "received" applies to subscribers.
15 | #
16 | # For services, "emitted" applies to responses, and "received" applies to requests. Normally, they
17 | # should be equal; however, under certain circumstances the number of responses ("emitted") can be
18 | # lower, e.g. if the server could not or chose not to send a response.
19 | #
20 | # The error counter represents how many transfers could not be successfully received or emitted due to
21 | # an error of any kind anywhere in the processing pipeline, excluding application-level errors.
22 | # For example, if the message could not be decoded because of a data type compatibility problem,
23 | # such incident should be reported here via this field. On the other hand, if the message or service
24 | # call was processed properly, but the application could not act upon that data, it should not be
25 | # reported here because that problem should be attributed to a different level of abstraction.
26 | uavcan.node.IOStatistics.0.1 statistics
27 |
28 | @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]

```

1 | #
2 | # Returns a subset of all ports of the specified kind (service or message) on the remote node.
3 | # As the number of ports may exceed the capacity of the output array, the server will return only those which are
4 | # greater than the number specified in the request. The caller should sweep the lower boundary upwards until
5 | # the size of the returned array is less than its capacity. For example:
6 | # 1. port_id_lower_boundary = 0; port_ids = [42, 567, ... 920] (=128 elements, continue)
7 | # 2. port_id_lower_boundary = 920; port_ids = [1052, ... 5049] (=128 elements, continue)
8 | # 3. port_id_lower_boundary = 5049; port_ids = [6973, ..., 24593] (<128 elements, stop)
9 | #
10 |
11 | # The server will return as many IDs as possible which are numerically greater than this lower boundary,
12 | # and which are of the same kind as specified in the request (i.e., if the lower boundary is a service ID, only
13 | # service IDs will be returned).
14 | #
15 | # If the number of matching (i.e., greater than this limit) IDs exceeds the capacity of the output array,
16 | # the caller will need to repeat the call with this boundary set to the maximum value found in the returned array.
17 | # The calls should be repeated while advancing the boundary as described until the returned array is not full
18 | # (e.g. the number of elements is less than its capacity, possibly zero).
19 | #
20 | # The server is NOT required to ensure any ordering in the output array, meaning that the caller will have to
21 | # search through the array in order to find the greatest value in it.
22 | #

```

```

23 | # The server is REQUIRED to ensure that every element in the returned array is unique.
24 | #
25 | # The server is REQUIRED to ensure that by performing the above actions the client will end up with a full set of
26 | # port IDs, assuming that the set is not modified between invocations. This implicitly requires that there must be
27 | # no port ID that is lower than the greatest value in the returned array that is not listed in the array, because
28 | # in that case the client will never be able to obtain that value.
29 | void7
30 | ID.1.0 port_id_lower_boundary
31 |
32 | @assert _offset_ == {24}
33 |
34 | ---
35 |
36 | # See above for the full description of the logic.
37 | uint16[<=128] port_ids
38 |
39 | @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

```

1 | #
2 | # Used to refer either to a Service or to a Subject.
3 | # The chosen tag identifies the kind of the port, then the numerical ID identifies the port within the kind.
4 | #
5 |
6 | @union
7 | ServiceID.1.0 service_id
8 | SubjectID.1.0 subject_id
9 | @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

```

1 | #
2 | # Service ID. The ranges are defined by the specification.
3 | #
4 |
5 | uint9 MAX_UNREGULATED_ID = 127
6 |
7 | void7
8 | uint9 value
9 |
10 | @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

```

1 | #
2 | # Subject ID. The ranges are defined by the specification.
3 | #
4 |
5 | uint16 MAX_UNREGULATED_ID = 24575
6 |
7 | void1
8 | uint15 value
9 |
10 | @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

```

1  #
2  # In order to be able to operate in a UAVCAN network, a node must have a node-ID that is unique within the network.
3  # Typically, a valid node-ID can be configured manually for each node; however, in certain use cases the manual
4  # approach is either undesirable or impossible, therefore UAVCAN defines the high-level feature of plug-and-play
5  # nodes that allows nodes to obtain a node-ID value automatically upon connection to the network. When combined
6  # with automatic physical layer configuration (such as auto bit rate detection), this feature allows one to implement
7  # nodes that can join a UAVCAN network without any prior manual configuration whatsoever. Such nodes are referred to
8  # as "plug-and-play nodes" (or "PnP nodes" for brevity).
9  #
10 # The feature is fundamentally non-deterministic and is likely to be unfit for some high-reliability systems;
11 # the designers need to carefully consider the trade-offs involved before deciding to rely on this feature.
12 # Normally, static node-ID settings should be preferred.
13 #
14 # This feature relies on the concept of "anonymous message transfers", please consult with the UAVCAN transport
15 # layer specification for details.
16 #
17 # An allocated node-ID should not be persistent. This means that if a node is configured to use plug-and-play node-ID
18 # allocation, it must perform a new allocation every time it is started or rebooted. The allocated node-ID value
19 # should not be stored on the node itself, because there exist edge cases that may lead to node-ID conflicts under
20 # certain circumstances (reviewed later).
21 #
22 # The process of plug-and-play node-ID allocation always involves two types of nodes: "allocators", which serve
23 # allocation requests; and "allocates", which request PnP node-ID from allocators. A UAVCAN network may implement
24 # the following configurations of allocators:
25 #
26 # - Zero allocators, in which case plug-and-play node-ID allocation cannot be used, only nodes with statically
27 #   configured node-ID can communicate.
28 #
29 # - One allocator, in which case the feature of plug-and-play node-ID allocation will become unavailable
30 #   if the allocator fails. In this configuration, the role of the allocator can be performed even by a very
31 #   resource-constrained system, e.g. a low-end microcontroller.
32 #
33 # - Three allocators, in which case the allocators will be using a replicated allocation table via a
34 #   distributed consensus algorithm. In this configuration, the network can tolerate the loss of one
35 #   allocator and continue to serve allocation requests. This configuration requires the allocators to
36 #   maintain large data structures for the needs of the distributed consensus algorithm, and may therefore
37 #   require a slightly more sophisticated computational platform, e.g., a high-end microcontroller.
38 #
39 # - Five allocators, it is the same as the three allocator configuration reviewed above except that the network
40 #   can tolerate the loss of two allocators and still continue to serve allocation requests.
41 #
42 # In order to get a PnP node-ID, an allocatee must have a globally unique 128-bit integer identifier, known as
43 # unique-ID (where "globally unique" means that the probability of having two nodes anywhere in the world that share
44 # the same unique-ID is negligibly low). This is the same value that is used in the field unique_id of the data type
45 # uavcan.node.GetInfo. All PnP nodes must support the service uavcan.node.GetInfo, and they must use the same
46 # unique-ID value when requesting node-ID allocation and when responding to the GetInfo requests (there may exist
47 # other usages of the unique-ID value, but they lie outside of the scope of the PnP protocol).
48 #
49 # During allocation, the allocatee communicates its unique-ID to the allocator (or allocators in the case of a
50 # redundant allocator configuration), which then use it to produce an appropriate allocation response. Unique-ID
51 # values are kept by allocators in the "allocation table" - a data structure that contains the mapping between
52 # unique-ID and the corresponding node-ID values. The allocation table is a write-only data structure that can
53 # only expand. When a new allocatee requests a PnP node-ID, its unique-ID is recorded in the allocation table,
54 # and all subsequent allocation requests from the same allocatee will be served with the same node-ID value.
55 #
56 # In configurations with redundant allocators, every allocator maintains a replica of the same allocation table
57 # (a UAVCAN network cannot contain more than one allocation table, regardless of the number of allocators employed).
58 # While the allocation table is a write-only data structure that can only grow, it is still possible to wipe the
59 # table completely (as long as it is removed from all redundant allocators on the network simultaneously),
60 # forcing the allocators to forget known nodes and perform all future allocations anew.
61 #
62 # In the context of the following description, nodes that use a manually-configured node-ID will be referred to as
63 # "static nodes". It is assumed that allocators are always static nodes themselves since there is no other authority
64 # on the network that can grant a PnP node-ID, so allocators are unable to request a PnP node-ID for themselves.
65 # Excepting allocators, it is not recommended to mix PnP and static nodes on the same network; i.e., normally,
66 # a UAVCAN network should contain either all static nodes, or all PnP nodes (excepting allocators). If this
67 # recommendation cannot be followed, the following rules of safe co-existence of PnP nodes with static nodes should
68 # be adopted:
69 # - It is safe to connect PnP nodes to the bus at any time.
70 # - A static node can be connected to the bus if the allocator (allocators) is (are) already aware of it.
71 #   I.e., the static node is already listed in the allocation table.
72 # - A new static node (i.e., a node that does not meet the above criterion) can be connected to the bus only if
73 #   no PnP allocation requests are happening at the moment.
74 #
75 # Due to the possibility of coexistence of static nodes with PnP nodes, allocators are tasked with monitoring
76 # the nodes present in the network. If the allocator detects an online node in the network the node-ID of which is
77 # not found in the allocation table (or the local copy thereof in the case of redundant allocators), the allocator
78 # must create a new mock entry where the node-ID matches that of the newly detected node and the unique-ID is set to
79 # zero (i.e., a 128-bit long sequence of zero bits). This behavior ensures that PnP nodes will never be granted
80 # node-ID values that are already taken by static nodes. Allocators are allowed to request the true unique-ID of the
81 # newly detected nodes by issuing requests uavcan.node.GetInfo instead of using mock zero unique-IDs, but this is not
82 # required for the sake of simplicity and determinism (some nodes may fail to respond to the GetInfo request, e.g.,
83 # if this service is not supported). Note that in the case of redundant allocators, some of them may be relieved of
84 # this task due to the intrinsic properties of the distributed consensus algorithm; please refer to the documentation
85 # for the data type uavcan.pnp.cluster.AppendEntries for more information.
86 #

```

```

87 # The unique-ID & node-ID pair of each allocator must be kept in the allocation table as well. It is allowed to replace
88 # the unique-ID values of allocators with zeros at the discretion of the implementer.
89 #
90 # As can be inferred from the above, the process of PnP node-ID allocation involves up to two types of communications:
91 #
92 # - "Allocatee-allocator exchange" - this communication is used when an allocatee requests a PnP node-ID from the
93 #   allocator (or redundant allocators), and also when the allocator transmits a response back to the allocatee.
94 #   This communication is invariant to the allocator configuration used, i.e., the allocatees are not aware of
95 #   how many allocators are available on the network and how they are configured. In configurations with
96 #   non-redundant (i.e., single) allocator, this is the only type of PnP allocation exchanges.
97 #
98 # - "Allocator-allocator exchange" - this communication is used by redundant allocators for the maintenance of
99 #   the replicated allocation table and for other needs of the distributed consensus algorithm. Allocatees are
100 #   completely isolated and are unaware of these exchanges. This communication is not used with the single-allocator
101 #   configuration, since there is only one server and the allocation table is not distributed. The data types
102 #   used for the allocator-allocator exchanges are defined in the namespace uavcan.pnp.cluster.
103 #
104 # As has been said earlier, the logic used for communication between allocators (for the needs of the maintenance of
105 # the distributed allocation table) is completely unrelated to the allocatees. The allocatees are unaware of these
106 # exchanges, and they are also unaware of the allocator configuration used on the network: single or redundant.
107 # As such, the documentation you're currently reading does not describe the logic and requirements of the
108 # allocator-allocator exchanges for redundant configurations; for that, please refer to the documentation for the
109 # data type uavcan.pnp.cluster.AppendEntries.
110 #
111 # Allocatee-allocator exchanges are performed using only this message type uavcan.pnp.NodeIDAllocationData. Allocators
112 # use it with regular message transfers; allocatees use it with anonymous message transfers. The specification and
113 # usage info for this data type is provided below.
114 #
115 # The general idea of the allocatee-allocator exchanges is that the allocatee communicates to the allocator its
116 # unique-ID and, if applicable, the preferred node-ID value that it would like to have. The allocatee uses
117 # anonymous message transfers of this type. The allocator performs the allocation and sends a response using
118 # the same message type, where the field for unique-ID is populated with the unique-ID of the requesting node
119 # and the field for node-ID is populated with the allocated node-ID. All exchanges from allocatee to allocator use
120 # single-frame transfers only (see the specification for more information on the limitations of anonymous messages).
121 #
122 # The allocatee-allocator exchange logic differs between allocators and allocatees. For allocators, the logic is
123 # trivial: upon reception of a request, the allocator performs an allocation and sends a response back. If the
124 # allocation could not be performed for any reason (e.g., the allocation table is full, or there was a failure),
125 # no response is sent back (i.e., the request is simply ignored). The allocator that could not complete an allocation
126 # for any reason is recommended to emit a diagnostic message with a human-readable description of the problem.
127 # For allocatees, the logic is described below.
128 #
129 # This message is used for PnP node-ID allocation on all transports where the maximum transmission unit size is
130 # sufficiently large. For low-MTU transports such as CAN 2.0 there is a dedicated message definition that takes into
131 # account the limitations of that transport (namely, the unique-ID value is replaced with a short hash in order to
132 # fit the data into one low-MTU single-frame transfer).
133 #
134 # When a node needs to request a PnP node-ID, it will transmit an anonymous message transfer of this type. The pseudo
135 # node-ID value of the anonymous transfer should be populated with pseudorandom data as described in the UAVCAN
136 # transport layer specification. Since pseudo node-ID collisions are likely to happen, nodes that are requesting
137 # PnP allocations need to be able to handle them correctly. Hence, a CSMA/CD collision resolution protocol is defined,
138 # which utilizes a randomized request period value referred to as Trequest.
139 #
140 # Generally, the randomly chosen values of the request period (Trequest) should be in the range from 0 to 1 seconds.
141 # Applications that are not concerned about the allocation time are recommended to pick higher values, as it will
142 # reduce interference with other nodes where faster allocations may be desirable. The random interval must be chosen
143 # anew per transmission, whereas the pseudo node-ID value is allowed to stay constant per node.
144 #
145 # The source of random data for Trequest must be likely to yield different values for participating nodes, avoiding
146 # common sequences. This implies that the time since boot alone is not a sufficiently robust source of randomness,
147 # as that would be probable to cause nodes powered up at the same time to emit colliding messages repeatedly.
148 #
149 # The response timeout is not explicitly defined for this protocol, as the allocatee will request a new allocation
150 # Trequest units of time later again, unless an allocation has been granted. Since the request and response messages
151 # are fully idempotent, accidentally repeated messages (e.g., due to benign race conditions that are inherent to this
152 # protocol) are harmless.
153 #
154 # On the allocatee's side the protocol is defined through the following set of simple rules:
155 #
156 # Rule A. On initialization:
157 #   1. The allocatee subscribes to this message.
158 #   2. The allocatee starts the Request Timer with a random interval of Trequest.
159 #
160 # Rule B. On expiration of the Request Timer (started as per rules A, B, or C):
161 #   1. Request Timer restarts with a random interval of Trequest (chosen anew).
162 #   2. The allocatee broadcasts an allocation request message, where the fields are populated as follows:
163 #       node_id - the preferred node-ID, or 125 if the allocatee doesn't have any specific preference.
164 #       unique_id - the 128-bit unique-ID of the allocatee, same value that is reported via uavcan.node.GetInfo.
165 #
166 # Rule C. On any allocation message, even if other rules also match:
167 #   1. Request Timer restarts with a random interval of Trequest (chosen anew) (this is related to CSMA/CD).
168 #
169 # Rule D. On an allocation message WHERE (source node-ID is non-anonymous, i.e., regular allocation response)
170 #         AND (the field unique_id matches the allocatee's unique-ID):
171 #   1. Request Timer stops.
172 #   2. The allocatee initializes its node-ID with the received value.
173 #   3. The allocatee terminates its subscription to allocation messages.
174 #   4. Exit.
175 #
176 # As can be seen, the algorithm assumes that the allocatee will continue to emit requests at random intervals
177 # until an allocation is granted or the allocatee is disconnected.
178 #
179 #
180 # If the message transfer is anonymous (i.e., allocation request), this is the preferred ID.
181 # If the message transfer is non-anonymous (i.e., allocation response), this is the allocated ID.
182 #
183 # If the allocatee does not have any preference, it should request the highest possible node-ID, which is 125,
184 # because 126 and 127 are reserved for network maintenance tools. Requesting 126 and 127 is not prohibited,
185 # but the allocator is recommended to avoid granting these node-ID, using nearest available lower value instead.
186 # The allocator will traverse the allocation table starting from the preferred node-ID upward,
187 # until a free node-ID is found (or the first ID reserved for network maintenance tools is reached).

```

```

188 # If a free node-ID could not be found, the allocator will restart the search from the preferred node-ID
189 # downward, until a free node-ID is found. In pseudocode:
190 #
191 #   uint8_t findFreeNodeID(const uint8_t preferred)    // Returns >127 on failure (valid node-ID range from 0 to 127)
192 #   {
193 #       uint8_t candidate = preferred;
194 #       while (candidate <= 125)
195 #       {
196 #           if (!isOccupied(candidate)) { return candidate; }
197 #           candidate++;
198 #       }
199 #       candidate = preferred;
200 #       while (candidate > 0)
201 #       {
202 #           if (!isOccupied(candidate)) { return candidate; }
203 #           candidate--;
204 #       }
205 #       return isOccupied(0) ? 255 : 0;
206 #   }
207 uavcan.node.ID.1.0 node_id
208
209 # The unique-ID of the allocatee. This is the SAME value that is reported via uavcan.node.GetInfo.
210 # The value is subjected to the same set of constraints; e.g., it can't be changed while the node is running,
211 # and the same value must be unlikely to be used by any two different nodes anywhere in the world.
212 #
213 # If this is a non-anonymous transfer (i.e., allocation response), allocatees will match this value against their
214 # own unique-ID, and ignore the message if there is no match (except for the CSMA/CD clause, see rule C). If the IDs
215 # match, then the field node_id contains the allocated node-ID value for this node.
216 uint8[16] unique_id
217
218 @assert _offset_ % 8 == {0}
219 @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

```

1 #
2 # The suffix "MTU8" indicates that this message is made to be compatible with transports where the maximum
3 # transmission unit size is only 8 bytes, such as CAN 2.0. For other kinds of transports there is a more general,
4 # more capable definition NodeIDAllocationData; it can't be used with low-MTU transports, hence the need for this
5 # special case. The PnP protocol is described in the documentation for the general case data type definition. The
6 # documentation provided here builds upon the general case, so read that first please.
7 #
8 # The motivation for the difference between this small-MTU case and the general case is that the full 128-bit
9 # unique-ID can't be accommodated in a small-MTU single-frame anonymous message transfer. The solution is to replace
10 # the full 128-bit ID with a smaller 55-bit hash of it, and remove support for preferred node-ID value in the
11 # allocation requests in order to save space.
12 #
13 # The 55-bit hash is obtained by applying an arbitrary hash function to the unique-ID that outputs at least 55 bit of
14 # data. For example, it could be the standard CRC-64WE function where only the lowest 55 bit of the result are used.
15 # The hash function must always produce the same hash value for the same unique-ID value, and it must be likely to
16 # produce a different hash value for any other unique-ID value.
17 #
18 # Allocators that support both the general case and the small-MTU case should maintain the same allocation table
19 # for both. Requests received via the small-MTU message obviously do not contain the full unique-ID; the allocators
20 # are recommended to extend the small 55-bit hash with zeros upwards (where "upwards" means that the most significant
21 # bits of the unique-ID will contain zeros, and the least significant 55 bits will contain the hash) in order to
22 # obtain a "pseudo unique-ID", and use this value in the allocation table as a substitute for the real unique-ID.
23 # It is recognized that this behavior will have certain side effects, such as the same allocatee obtaining different
24 # allocated node-ID values depending on which transport is used, but they are considered tolerable.
25 #
26 # In order to save space for the hash, the preferred node-ID is removed from the request. The allocated node-ID
27 # is provided in the response, however; this is achieved by means of an optional field that is not populated in
28 # the request but is populated in the response. This implies that the response may be a multi-frame transfer,
29 # which is acceptable since responses are sent by allocators, which are regular nodes, and therefore they are
30 # allowed to use regular message transfers rather than being limited to anonymous message transfers as allocatees are.
31 #
32 # On the allocatee's side the protocol is defined through the following set of simple rules:
33 #
34 # Rule A. On initialization:
35 #   1. The allocatee subscribes to this message.
36 #   2. The allocatee starts the Request Timer with a random interval of Trequest.
37 #
38 # Rule B. On expiration of the Request Timer (started as per rules A, B, or C):
39 #   1. Request Timer restarts with a random interval of Trequest (chosen anew).
40 #   2. The allocatee broadcasts an allocation request message, where the fields are populated as follows:
41 #       unique_id_hash - a 55-bit hash of the unique-ID of the allocatee.
42 #       allocated_node_id - empty (not populated).
43 #
44 # Rule C. On any allocation message, even if other rules also match:
45 #   1. Request Timer restarts with a random interval of Trequest (chosen anew) (this is related to CSMA/CD).
46 #
47 # Rule D. On an allocation message WHERE (source node-ID is non-anonymous, i.e., regular allocation response)
48 #       AND (the field unique_id_hash matches the allocatee's 55-bit unique-ID hash)
49 #       AND (the field allocated_node_id is populated):
50 #   1. Request Timer stops.
51 #   2. The allocatee initializes its node-ID with the received value.
52 #   3. The allocatee terminates its subscription to allocation messages.
53 #   4. Exit.
54 #
55 #

```



```

56 # An arbitrary 55-bit hash of the unique-ID of the local node.
57 truncated uint55 unique_id_hash
58
59 # Must be empty in request messages.
60 # Must be populated in response messages.
61 uavcan.node.ID.1.0[<=1] allocated_node_id
62
63 @assert _offset_.min == 56 # Plus the tail byte yields 8 bytes; this is for requests only
64 @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

```

1 #
2 # This type is a part of the Raft consensus algorithm. The Raft consensus is used for the maintenance of the
3 # distributed allocation table between redundant allocators. The following description is focused on the exchanges
4 # between redundant PnP node-ID allocators. It does not apply to the case of non-redundant allocators, because
5 # in that case the allocation table is stored locally and the process of node-ID allocation is trivial and fully local.
6 # Exchanges between allocatees and allocators are documented in the appropriate message type definition.
7 #
8 # The algorithm used for replication of the allocation table across redundant allocators is a fairly direct
9 # implementation of the Raft consensus algorithm, as published in the paper
10 # "In Search of an Understandable Consensus Algorithm (Extended Version)" by Diego Ongaro and John Ousterhout.
11 # The following text assumes that the reader is familiar with the paper.
12 #
13 # The Raft log contains entries of type Entry (in the same namespace), where every entry contains the Raft term
14 # number, the unique-ID, and the corresponding node-ID value (or zeros if it could not be requested from a static
15 # node). Therefore, the Raft log is the allocation table itself.
16 #
17 # Since the maximum number of entries in the allocation table is limited by the range of node-ID values, the log
18 # capacity is bounded. Therefore, the snapshot transfer and log compaction functions are not required,
19 # so they are not used in this implementation of the Raft algorithm.
20 #
21 # When an allocator becomes the leader of the Raft cluster, it checks if the Raft log contains an entry for its own
22 # node-ID, and if it doesn't, the leader adds its own allocation entry to the log (the unique-ID can be replaced with
23 # zeros at the discretion of the implementer). This behavior guarantees that the Raft log always contains at least
24 # one entry, therefore it is not necessary to support negative log indices, as proposed by the Raft paper.
25 #
26 # Since the log is write-only and limited in growth, all allocations are permanent. This restriction is acceptable,
27 # since UAVCAN is a vehicle bus, and configuration of vehicle's components is not expected to change frequently.
28 # Old allocations can be removed in order to free node-IDs for new allocations by clearing the Raft log on all
29 # allocators; such clearing must be performed simultaneously while the network is down, otherwise the Raft cluster
30 # will automatically attempt to restore the lost state on the allocators where the table was cleared.
31 #
32 # 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
33 # node-ID values, the allocators broadcast messages of type Discovery (in the same namespace) until the cluster is
34 # fully discovered and all allocators know of each other's node-ID. This extension to the Raft algorithm makes the
35 # cluster almost configuration-free - the only parameter that must be configured on all allocators of the cluster
36 # is the number of nodes in the cluster (everything else will be auto-detected).
37 #
38 # Runtime cluster membership changes are not supported, since they are not needed for a vehicle bus.
39 #
40 # As has been explained in the general description of the PnP node-ID allocation feature, allocators must watch for
41 # unknown static nodes appearing on the bus. In the case of a non-redundant allocator, the task is trivial, since the
42 # allocation table can be updated locally. In the case of a Raft cluster, however, the network monitoring task must
43 # be performed by the leader only, since other cluster members cannot commit to the shared allocation table (i.e.,
44 # the Raft log) anyway. Redundant allocators should not attempt to obtain the true unique-ID of the newly detected
45 # static nodes (use zeros instead), because the allocation table is write-only: if the unique-ID of a static node
46 # ever changes (e.g., a replacement unit is installed, or network configuration is changed manually), the change
47 # will be impossible to reflect in the allocation table.
48 #
49 # Only the current Raft leader can process allocation requests and engage in communication with allocatees.
50 # An allocator is allowed to send allocation responses only if both conditions are met:
51 #
52 # - The allocator is currently the Raft leader.
53 # - Its replica of the Raft log does not contain uncommitted entries (i.e. the last allocation request has been
54 #   completed successfully).
55 #
56 # All cluster maintenance traffic should normally use either the lowest or the next-to-lowest transfer priority level.
57 #
58 #
59 # Given the minimum election timeout and the cluster size,
60 # the maximum recommended request interval can be derived as follows:
61 #
62 # max recommended request interval = (min election timeout) / 2 requests / (cluster size - 1)
63 #
64 # The equation assumes that the Leader requests one Follower at a time, so that there's at most one pending call
65 # at any moment. Such behavior is optimal as it creates a uniform bus load, although it is implementation-specific.
66 # Obviously, the request interval can be lower than that if needed, but higher values are not recommended as they may
67 # cause Followers to initiate premature elections in case of frame losses or delays.
68 #
69 # The timeout value is randomized in the range [MIN, MAX], according to the Raft paper. The randomization granularity
70 # should be at least one millisecond or higher.
71 uint8 DEFAULT_MIN_ELECTION_TIMEOUT = 2 # [second]
72 uint8 DEFAULT_MAX_ELECTION_TIMEOUT = 4 # [second]
73

```

```

74 | # Refer to the Raft paper for explanation.
75 | uint32 term
76 | uint32 prev_log_term
77 | uint8 prev_log_index
78 | uint8 leader_commit
79 |
80 | # Worst case replication time per Follower can be computed as:
81 | #
82 | #   worst replication time = (node-ID capacity) * (2 trips of next_index) * (request interval per Follower)
83 | #
84 | # E.g., given the request interval of 0.5 seconds, the worst case replication time for CAN bus is:
85 | #
86 | #   128 nodes * 2 trips * 0.5 seconds = 128 seconds.
87 | #
88 | # This is the amount of time it will take for a new Follower to reconstruct a full replica of the distributed log.
89 | void7
90 | Entry.1.0[<=1] entries
91 |
92 | @assert _offset_ % 8 == {0}
93 |
94 | ---
95 |
96 | # Refer to the Raft paper for explanation.
97 | uint32 term
98 | 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

```

1 | #
2 | # This message is used by redundant allocators to find each other's node-ID.
3 | # Please refer to the type AppendEntries for details.
4 | #
5 | # An allocator should stop publishing this message as soon as it has discovered all other allocators in the cluster.
6 | #
7 | # An exception applies: when an allocator receives a Discovery message where the list of known nodes is incomplete
8 | # (i.e. len(known_nodes) < configured_cluster_size), it must publish a Discovery message once. This condition
9 | # allows other allocators to quickly re-discover the cluster after a restart.
10 | #
11 |
12 | # This message should be broadcasted by the allocator at this interval until all other allocators are discovered.
13 | uint8 BROADCASTING_PERIOD = 1 # [second]
14 |
15 | # The redundant allocator cluster cannot contain more than 5 allocators.
16 | uint3 MAX_CLUSTER_SIZE = 5
17 |
18 | # The number of allocators in the cluster as configured on the sender.
19 | # This value must be the same across all allocators.
20 | uint3 configured_cluster_size
21 |
22 | # Node-IDs of the allocators that are known to the publishing allocator, including the publishing allocator itself.
23 | void2
24 | uavcan.node.ID.1.0[<=5] known_nodes
25 |
26 | @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

```

1 | #
2 | # This type is a part of the Raft consensus algorithm. Please refer to the type AppendEntries for details.
3 | #
4 | #
5 | # Refer to the Raft paper for explanation.
6 | uint32 term
7 | uint32 last_log_term
8 | uint8 last_log_index
9 |
10 | ---
11 |
12 | # Refer to the Raft paper for explanation.
13 | uint32 term
14 | 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

```
1  #
2  # One PnP node-ID allocation entry.
3  # This type is a part of the Raft consensus algorithm. Please refer to the type AppendEntries for details.
4  #
5
6  uint32 term           # Refer to the Raft paper for explanation.
7
8  uint8[16] unique_id   # Unique-ID of this allocation; zero if unknown.
9
10 uavcan.node.ID.1.0 node_id # Node-ID of this allocation.
11
12 @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]

```

1  #
2  # This service is used to write and read a register. Write is optional, it is performed if the value provided in
3  # the request is not empty.
4  #
5  # The write operation is performed first, unless skipped by sending an empty value in the request.
6  # The server may attempt to convert the type of the value to the proper type if there is a type mismatch
7  # (e.g. uint8 may be converted to uint16); however, servers are not required to perform implicit type conversion,
8  # and the rules of such conversion are not explicitly specified, so this behavior should not be relied upon.
9  #
10 # On the next step the register will be read regardless of the outcome of the write operation. As such, if the write
11 # operation could not be performed (e.g. due to a type mismatch or any other issue), the register will retain its old
12 # value. By evaluating the response the caller can determine whether the register was written successfully.
13 #
14 # The write-read sequence is not guaranteed to be atomic, meaning that external influences may cause the register to
15 # change its value between the write and the subsequent read operation. The caller is responsible for handling that
16 # case properly.
17 #
18 # The timestamp provided in the response corresponds to the time when the register was read. The timestamp may
19 # be empty if the server does not support timestamping or its clock is not yet synchronized with the bus.
20 #
21 # If only read is desired, but not write, the caller must provide a value of type Empty. That will signal the server
22 # that the write operation must be skipped, and it will proceed to read the register immediately.
23 #
24 # If the requested register does not exist, the write operation will have no effect and the returned value will be
25 # empty. Existing registers should not return Empty when read since that would make them indistinguishable from
26 # nonexistent registers.
27 #
28 # Registers must never change their type or flags as long as the server is running. Meaning that:
29 # - Mutability and persistence flags cannot change their states.
30 # - Read operations must always return values of the same type and same dimensionality.
31 # - The dimensionality requirement does not apply to inherently variable-length values such as strings and
32 #   unstructured chunks.
33 #
34 # In order to discover the type of a register, the caller needs to invoke this service with the write request set
35 # to Empty. The response will contain the current value of the register with the type information (which doesn't
36 # change).
37 #
38 # Register name may contain:
39 # - All ASCII alphanumeric characters (a-z, A-Z, 0-9)
40 # - Dot (.)
41 # - Underscore (_)
42 # - Minus (-)
43 # All other printable non-whitespace ASCII characters are reserved for standard functions;
44 # they may appear in register names to support such standard functions defined by the register protocol,
45 # but they cannot be freely used by applications outside of such standard functions.
46 #
47 # The following optional special function register names are defined:
48 # - suffix '<' is used to define an immutable persistent value that contains the maximum value
49 #   of the respective register.
50 # - suffix '>' is like above, used to define the minimum value of the respective register.
51 # - suffix '=' is like above, used to define the default value of the respective register.
52 # - prefix '*' is reserved for raw memory access (to be defined later).
53 # Examples:
54 # - register name "system.parameter"
55 # - maximum value is contained in the register named "system.parameter<" (optional)
56 # - minimum value is contained in the register named "system.parameter>" (optional)
57 # - default value is contained in the register named "system.parameter=" (optional)
58 #
59 # The following table specifies the register name patterns that are reserved by the specification for
60 # common functions. Implementers should always follow these conventions whenever applicable.
61 # The table uses the following abbreviations for register flags:
62 # - M - mutable, I - immutable
63 # - P - persistent, V - volatile
64 #
65 # Name pattern | Flags | Type | Purpose
66 #-----|-----|-----|-----
67 # uavcan.node_id | MP | uint8 | Contains the node ID of the local node. Values above 127 indicate that
68 # | | | | the node ID is not set; if plug-and-play is supported, it will be
69 # | | | | used by the node to obtain an automatic node ID. Values above 128 should
70 # | | | | be avoided for consistency. The factory-default value should be 128.
71 #-----|-----|-----|-----
72 #
73 #
74 #
75 # The name of the accessed register. Must not be empty.
76 # Use the List service to obtain the list of registers on the node.
77 Name.0.1 name
78
79 @assert_offset % 8 == {0}
80
81 # Value to be written. Empty if no write is required.
82 void4
83 Value.0.1 value
84
85 @assert_offset_min % 8 == 0

```

```

86 | @assert _offset_.max % 8 == 0
87 |
88 | ---
89 |
90 | # The moment of time when the register was read (not written).
91 | # Zero if the server does not support timestamping.
92 | uavcan.time.SynchronizedTimestamp.1.0 timestamp
93 |
94 | # Mutable means that the register can be written using this service.
95 | # Immutable registers cannot be written, but that doesn't imply that their values are constant (unchanging).
96 | bool mutable
97 |
98 | # Persistence means that the register retains its value permanently across power cycles or any other changes
99 | # in the state of the server, until it is explicitly overwritten (either via UAVCAN, any other interface,
100 | # or by the device itself).
101 | #
102 | # The server is recommended to manage persistence automatically by committing changed register values to a
103 | # non-volatile storage automatically as necessary. If automatic persistence management is not implemented, it
104 | # can be controlled manually via the standard service uavcan.node.ExecuteCommand. The same service can be used
105 | # to return the configuration to a factory-default state. Please refer to its definition for more information.
106 | #
107 | # Consider the following examples:
108 | # - Configuration parameters are usually both mutable and persistent.
109 | # - Diagnostic values are usually immutable and non-persistent.
110 | # - Registers that trigger an activity when written are typically mutable but non-persistent.
111 | # - Registers that contain factory-programmed values such as calibration coefficients that can't
112 | #   be changed are typically immutable but persistent.
113 | bool persistent
114 |
115 | void2
116 |
117 | # The value of the register when it was read (beware of race conditions).
118 | # Registers never change their type and dimensionality while the node is running.
119 | # Empty value means that the register does not exist (in this case the flags should be cleared/ignored).
120 | # By comparing the returned value against the write request the caller can determine whether the register
121 | # was written successfully, unless write was not requested.
122 | Value.0.1 value
123 |
124 | @assert _offset_.min % 8 == 0
125 | @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

```

1 | #
2 | # This service allows the caller to discover the names of all registers available on the server
3 | # by iterating the index field from zero until an empty name is returned.
4 | #
5 |
6 | uint16 index
7 |
8 | ---
9 |
10 | # Empty name in response means that the index is out of bounds, i.e., discovery is finished.
11 | Name.0.1 name
12 |
13 | @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

```

1 | #
2 | # An ASCII register name.
3 | #
4 |
5 | void2
6 | uint8[<=50] name
7 |
8 | @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]

```

1  #
2  # This union contains all possible value types supported by the register protocol.
3  # Numeric types can be either scalars or arrays; the former is a special case of the latter.
4  #
5  # In order to ensure a byte alignment of the nested arrays, the outer type must pad this union with 4 bits.
6  #
7
8  @union                                # The tag is 4 bits wide.
9
10 uavcan.primitive.Empty.1.0          empty      # Tag 0      Used to represent an undefined value
11 uavcan.primitive.String.1.0         string     # Tag 1      UTF-8 encoded text
12 uavcan.primitive.Unstructured.1.0   unstructured # Tag 2      Raw unstructured binary image
13 uavcan.primitive.array.Bit.1.0      bit        # Tag 3      Bit array
14
15 uavcan.primitive.array.Integer64.1.0 integer64 # Tag 4
16 uavcan.primitive.array.Integer32.1.0 integer32 # Tag 5
17 uavcan.primitive.array.Integer16.1.0 integer16 # Tag 6
18 uavcan.primitive.array.Integer8.1.0  integer8  # Tag 7
19
20 uavcan.primitive.array.Natural64.1.0 natural64 # Tag 8
21 uavcan.primitive.array.Natural32.1.0 natural32 # Tag 9
22 uavcan.primitive.array.Natural16.1.0 natural16 # Tag 10
23 uavcan.primitive.array.Natural8.1.0  natural8  # Tag 11
24
25 uavcan.primitive.array.Real64.1.0    real64    # Tag 12      Exactly representable integers: [-2**53, +2**53]
26 uavcan.primitive.array.Real32.1.0    real32    # Tag 13      Exactly representable integers: [-16777216, +16777216]
27 uavcan.primitive.array.Real16.1.0    real16    # Tag 14      Exactly representable integers: [-2048, +2048]
28
29 # Tag 15 is reserved
30
31 @assert _offset_.min == 4            # Empty and the tag
32 @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

```

1  #
2  # Every node that acts as a time synchronization master, or is capable of acting as such,
3  # should support this service.
4  # Its objective is to provide information about which time system is currently used in the network.
5  #
6  # Once a time system is chosen, it cannot be changed as long as at least one node on the network is running.
7  # In other words, the time system cannot be changed while the network is operating.
8  # An implication of this is that if there are redundant time synchronization masters, they all must
9  # use the same time system always.
10 #
11 ---
12
13 # Error variance, in second^2, of the time value reported by this master.
14 # This value is allowed to change freely while the master is running.
15 # For example, if the master's own clock is synchronized with a GNSS, the error variance is expected to increase
16 # as signal reception deteriorates. If the signal is lost, this value is expected to grow steadily, the rate of
17 # growth would be dependent on the quality of the time keeping hardware available locally (bad hardware yields
18 # faster growth). Once the signal is regained, this value would drop back to nominal.
19 float32 error_variance # [second^2]
20
21 # Time system currently in use by the master.
22 # Cannot be changed while the network is operating.
23 TimeSystem.0.1 time_system
24
25 # The fixed difference, in seconds, between TAI and GPS time. Does not change ever.
26 uint12 TIME_DIFFERENCE_TAI_MINUS_GPS = 19 # [second]
27
28 # The current difference between TAI and UTC (a.k.a. leap seconds), if known.
29 # If unknown, set to zero. This value can change states between known and unknown while the master is running,
30 # depending on its ability to obtain robust values from external sources.
31 # This value can change twice a year, possibly while the system is running; https://en.wikipedia.org/wiki/Leap_second
32 # Since Earth is decelerating, this value can be only positive. Do not use outside Earth.
33 uint12 TIME_DIFFERENCE_TAI_MINUS_UTC_UNKNOWN = 0
34 uint12 time_difference_tai_minus_utc
35
36 # Reserved for future use
37 void8
38
39 @assert _offset_ % 8 == {0}
40 @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

```

1  #
2  # Network-wide time synchronization message.
3  # Any node that publishes timestamped data should use this time reference.
4  #
5  # The time synchronization algorithm is based on the work
6  # "Implementing a Distributed High-Resolution Real-Time Clock using the CAN-Bus" by M. Gergeleit and H. Streich.
7  # The general idea of the algorithm is to have one or more nodes that periodically publish a message of this type
8  # containing the exact timestamp of the PREVIOUS transmission of this message.
9  # A node that publishes this message periodically is referred to as a "time synchronization master",
10 # whereas nodes that synchronize their clocks with the master are referred to as "time synchronization slaves".
11 #
12 # Once a time base is chosen, it cannot be changed as long as at least one node on the network is running.
13 # In other words, the time base cannot be changed while the network is operating.
14 # An implication of this is that if there are redundant time synchronization masters, they all must
15 # use the same time base.
16 #
17 # The resolution is dependent on the transport and its physical layer, but generally it can be assumed
18 # to be close to one bit time but not better than one microsecond (e.g., for a 500 kbps CAN bus,
19 # the resolution is two microseconds). The maximum accuracy is achievable only if the transport layer
20 # supports precise timestamping in hardware; otherwise, the accuracy may be degraded.
21 #
22 # This algorithm allows the slaves to precisely estimate the difference (i.e., phase error) between their
23 # local time and the master clock they are synchronized with. The algorithm for clock rate adjustment
24 # is entirely implementation-defined (for example, a simple phase-locked loop or a PID rate controller can be used).
25 #
26 # The network can accommodate more than one time synchronization master for purposes of increased reliability:
27 # if one master fails, the others will continue to provide the network with accurate and consistent time information.
28 # The risk of undesirable transients while the masters are swapped is mitigated by the requirement that all masters
29 # use the same time base at all times, as described above.

```

```

30 #
31 # The master with the lowest node ID is called the "dominant master". The current dominant master ceases to be one
32 # if its last synchronization message was published more than 3X seconds ago, where X is the time interval
33 # between the last and the previous messages published by it. In this case, the master with the next-higher node ID
34 # will take over as the new dominant master. The current dominant master will be displaced immediately as soon as
35 # the first message from a new master with a lower node ID is seen on the bus.
36 #
37 # In the presence of multiple masters, they all publish their time synchronization messages concurrently at all times.
38 # The slaves must listen to the master with the lowest node ID and ignore the messages published by masters with
39 # higher node ID values.
40 #
41 # Currently, there is a work underway to develop and validate a highly robust fault-operational time synchronization
42 # algorithm where the slaves select the median time base among all available masters rather than using only the
43 # one with the lowest node ID value. Follow the work at https://forum.uavcan.org. When complete, this algorithm
44 # will be added in a backward-compatible way as an option for high-reliability systems.
45 #
46 # For networks with redundant transports, the timestamp value published on different interfaces is likely to be
47 # different, since different transports are generally not expected to be synchronized. Synchronization slaves
48 # are allowed to use any of the available redundant interfaces for synchronization at their discretion.
49 #
50 # The following pseudocode shows the logic of a time synchronization master. This example assumes that the master
51 # does not need to synchronize its own clock with other masters on the bus, which is the case if the current master
52 # is the only master, or if all masters synchronize their clocks with a robust external source, e.g., a GNSS system.
53 # If several masters need to synchronize their clock through the bus, their logic will be extended with the
54 # slave-side behavior explained later.
55 #
56 # // State variables
57 # transfer_id := 0;
58 # previous_tx_timestamp_per_iface[NUM_IFACES] := {0};
59 #
60 # // This function publishes a message with a specified Transfer ID using only one transport interface.
61 # function publishMessage(transfer_id, iface_index, msg);
62 #
63 # // This callback is invoked when the transport layer completes the transmission of a time sync message.
64 # // Observe that the time sync message is always a single-frame message by virtue of its small size.
65 # // The tx_timestamp argument contains the exact timestamp when the transport frame was delivered to the bus.
66 # function messageTxTimestampCallback(iface_index, tx_timestamp)
67 # {
68 #     previous_tx_timestamp_per_iface[iface_index] := tx_timestamp;
69 # }
70 #
71 # // Publishes messages of type uavcan.time.Synchronization to each available transport interface.
72 # // It is assumed that this function is invoked with a fixed frequency not lower than 1 hertz.
73 # function publishTimeSync()
74 # {
75 #     for (i := 0; i < NUM_IFACES; i++)
76 #     {
77 #         message := uavcan.time.Synchronization();
78 #         message.previous_transmission_timestamp_usec := previous_tx_timestamp_per_iface[i];
79 #         previous_tx_timestamp_per_iface[i] := 0;
80 #         publishMessage(transfer_id, i, message);
81 #     }
82 #     transfer_id++; // Overflow must be handled correctly
83 # }
84 #
85 # (end of the master-side logic pseudocode)
86 # The following pseudocode describes the logic of a time synchronization slave.
87 #
88 # // State variables:
89 # previous_rx_real_timestamp := 0; // This clock is being synchronized
90 # previous_rx_monotonic_timestamp := 0; // Monotonic time -- doesn't leap or change rate
91 # previous_transfer_id := 0;
92 # state := STATE_UPDATE; // Variants: STATE_UPDATE, STATE_ADJUST
93 # master_node_id := -1; // Invalid value
94 # iface_index := -1; // Invalid value
95 #
96 # // This function adjusts the local clock by the specified amount
97 # function adjustLocalTime(phase_error);
98 #
99 # function adjust(message)
100 # {
101 #     // Clock adjustment will be performed every second message
102 #     local_time_phase_error := previous_rx_real_timestamp - msg.previous_transmission_timestamp_microsecond;
103 #     adjustLocalTime(local_time_phase_error);
104 #     state := STATE_UPDATE;
105 # }
106 #
107 # function update(message)
108 # {
109 #     // A message is assumed to have two timestamps:
110 #     // Real - sampled from the clock that is being synchronized
111 #     // Monotonic - clock that never leaps and never changes rate
112 #     previous_rx_real_timestamp := message.rx_real_timestamp;
113 #     previous_rx_monotonic_timestamp := message.rx_monotonic_timestamp;
114 #     master_node_id := message.source_node_id;
115 #     iface_index := message.iface_index;
116 #     previous_transfer_id := message.transfer_id;
117 #     state := STATE_ADJUST;
118 # }
119 #
120 # // Accepts the message of type uavcan.time.Synchronization
121 # function handleReceivedTimeSyncMessage(message)
122 # {
123 #     time_since_previous_msg := message.monotonic_timestamp - previous_rx_monotonic_timestamp;
124 #
125 #     needs_init := (master_node_id < 0) or (iface_index < 0);
126 #     switch_master := message.source_node_id < master_node_id;
127 #
128 #     // The value publisher_timeout is computed as described in the specification (3x interval)
129 #     publisher_timed_out := time_since_previous_msg > publisher_timeout;
130 # }

```

```

131 #         if (needs_init or switch_master or publisher_timed_out)
132 #         {
133 #             update(message);
134 #         }
135 #     else if ((message.iface_index == iface_index) and (message.source_node_id == master_node_id))
136 #     {
137 #         // Revert the state to STATE_UPDATE if needed
138 #         if (state == STATE_ADJUST)
139 #         {
140 #             msg_invalid := message.previous_transmission_timestamp_microsecond == 0;
141 #             // Overflow must be handled correctly
142 #             wrong_tid := message.transfer_id != (previous_transfer_id + 1);
143 #             wrong_timing := time_since_previous_msg > MAX_PUBLICATION_PERIOD;
144 #             if (msg_invalid or wrong_tid or wrong_timing)
145 #             {
146 #                 state := STATE_UPDATE;
147 #             }
148 #         }
149 #         // Handle the current state
150 #         if (state == STATE_ADJUST)
151 #         {
152 #             adjust(message);
153 #         }
154 #         else
155 #         {
156 #             update(message);
157 #         }
158 #     } // else ignore
159 # }
160 #
161 # (end of the slave-side logic pseudocode)
162 #
163 #
164 # Publication period limits.
165 # A master should not change its publication period while running.
166 uint8 MAX_PUBLICATION_PERIOD = 1 # [second]
167
168 # Synchronization slaves should normally switch to a new master if the current master was silent
169 # for thrice the interval between the reception of the last two messages published by it.
170 # For example, imagine that the last message was received at the time X, and the previous message
171 # was received at the time (X - 0.5 seconds); the period is 0.5 seconds, and therefore the
172 # publisher timeout is (0.5 seconds * 3) = 1.5 seconds. If there was no message from the current
173 # master in this amount of time, all slaves will synchronize with another master with the next-higher
174 # node ID.
175 uint8 PUBLISHER_TIMEOUT_PERIOD_MULTIPLIER = 3
176
177 # The time when the PREVIOUS message was transmitted from the current publisher, in microseconds.
178 # If this message is published for the first time, or if the previous transmission was more than
179 # one second ago, this field must be zero.
180 truncated uint56 previous_transmission_timestamp_microsecond
181
182 @assert _offset_ % 8 == {0}
183 @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

```

1 #
2 # Nested data type used for representing a network-wide synchronized timestamp with 10 (ten) microsecond resolution.
3 # Unlike the wider alternative, this timestamp value expires in ~167 seconds due to integer overflow of the
4 # microsecond timestamp field: 2**24 / 1e5 = 167.77216 seconds.
5 # This data type should not be used unless network throughput and latency are of utmost importance,
6 # because it is inherently unsafe: a delay of more than the specified above limit will invalidate the timestamp
7 # and its users will not be able to detect that. Instead, the full non-overflowing non-ambiguous timestamp
8 # should be preferred.
9 #
10 #
11 # Zero means that the time is not known.
12 uint24 UNKNOWN = 0
13
14 # Expiration timeout.
15 uint28 OVERFLOW_PERIOD_us = 167772160 # [microsecond]
16
17 # Overflowing sub-timestamp, 10 microseconds per least significant bit.
18 # Overflows every ~167 seconds.
19 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

```

1 #
2 # Nested data type used for representing a network-wide synchronized timestamp with microsecond resolution.

```



```

3 | # This data type is highly recommended for use both in standard and vendor-specific messages alike.
4 | #
5 |
6 | # Zero means that the time is not known.
7 | uint56 UNKNOWN = 0
8 |
9 | # The number of microseconds that have passed since some arbitrary moment in the past.
10 | # The moment of origin (i.e., the time base) is defined per-application. The current time base in use
11 | # can be requested from the time synchronization master, see the corresponding service definition.
12 | #
13 | # This value is to never overflow. The value is 56-bit wide because:
14 | #
15 | # - 2^56 microseconds is about 2285 years, which is plenty. A 64-bit microsecond counter would be
16 | #   unnecessarily wide and its overflow interval of 585 thousand years induces a mild existential crisis.
17 | #
18 | # - CAN 2.0-based transports carry up to 7 bytes per frame. Time sync messages must use single-frame
19 | #   transfers, which means that the value can't be wider than 56 bits.
20 | 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

```

1 | #
2 | # Time system enumeration.
3 | # The time system must be the same for all masters in the network.
4 | # It cannot be changed while the network is running.
5 | #
6 |
7 | # Monotonic time since boot.
8 | # Monotonic time is a time reference that doesn't change rate or make leaps.
9 | uint4 MONOTONIC_SINCE_BOOT = 0
10 |
11 | # International Atomic Time; https://en.wikipedia.org/wiki/International_Atomic_Time
12 | # TAI is always a fixed integer number of seconds ahead of GPS time.
13 | # Systems that use GPS time as a reference should convert that to TAI by adding the fixed difference.
14 | # UAVCAN does not support GPS time directly on purpose, for reasons of consistency.
15 | uint4 TAI = 1
16 |
17 | # Application-specific time system of unknown properties.
18 | uint4 APPLICATION_SPECIFIC = 15
19 |
20 | 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

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]

```

1 | #
2 | # A UTF8-encoded string of text.
3 | # Since the string is represented as a dynamic array of bytes, it is not null-terminated. Like Pascal string.
4 | #
5 |
6 | uint8[<256] value
7 |
8 | @assert _offset_ % 8 == {0}
9 | @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]

```

1 | #
2 | # An unstructured collection of bytes, e.g., raw binary image.
3 | #
4 |
5 | uint8[<256] value
6 |
7 | @assert _offset_ % 8 == {0}
8 | @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]

```

1 | void5
2 | bool[<=2032] value
3 | @assert _offset_.min == 16
4 | @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]

```

1 | int8[<256] value
2 | @assert _offset_ % 8 == {0}
3 | @assert _offset_.max / 8 == 256

```

6.11.3 Integer16Full 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]

```

1 | int16[<=128] value
2 | @assert _offset_ % 8 == {0}
3 | @assert _offset_.max / 8 == 257

```

6.11.4 Integer32Full 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]

```

1 | void1
2 | int32[<=64] value
3 | @assert _offset_ % 8 == {0}
4 | @assert _offset_.max / 8 == 257

```

6.11.5 Integer64Full 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]

```

1 | void2
2 | int64[<=32] value
3 | @assert _offset_ % 8 == {0}
4 | @assert _offset_.max / 8 == 257

```

6.11.6 Natural8Full 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]

```

1 | uint8[<256] value
2 | @assert _offset_ % 8 == {0}
3 | @assert _offset_.max / 8 == 256

```

6.11.7 Natural16Full 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]

```

1 | uint16[<=128] value
2 | @assert _offset_ % 8 == {0}
3 | @assert _offset_.max / 8 == 257

```

6.11.8 Natural32Full 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]

```

1 | void1
2 | uint32[<=64] value
3 | @assert _offset_ % 8 == {0}
4 | @assert _offset_.max / 8 == 257

```

6.11.9 Natural64Full 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]

```

1 | void2
2 | uint64[<=32] value
3 | @assert _offset_ % 8 == {0}
4 | @assert _offset_.max / 8 == 257

```

6.11.10 Real16Full 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]

```

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

```

6.11.11 Real32Full 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]

```

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

```

6.11.12 Real64Full 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]

```

1 | void2
2 | float64[<=32] value # Exactly representable integers: [-2**53, +2**53]
3 | @assert _offset_ % 8 == {0}
4 | @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

```

1 | void7
2 | bool value

```

6.12.2 Integer8Full 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

```

1 | int8 value

```

6.12.3 Integer16Full 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

1 | int16 value

6.12.4 Integer32Full 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

1 | int32 value

6.12.5 Integer64Full 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

1 | int64 value

6.12.6 Natural8Full 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

1 | uint8 value

6.12.7 Natural16Full 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

1 | uint16 value

6.12.8 Natural32Full 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

1 | uint32 value

6.12.9 Natural64Full 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

1 | uint64 value

6.12.10 Real16Full 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

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

6.12.11 Real32Full 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

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

6.12.12 Real64Full 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

```
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

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

6.13.2 Vector3Full 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

```
1 | uavcan.time.SynchronizedAmbiguousTimestamp.1.0 timestamp
2 | float32[3] meter_per_second_per_second
3 | @assert _offset_ / 8 == {15}
4 | @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

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

6.14.2 ScalarFull 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

```

1 | uavcan.time.SynchronizedAmbiguousTimestamp.1.0 timestamp
2 | float32 radian
3 | @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

```

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

```

6.15.2 Vector3Full 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

```

1 | uavcan.time.SynchronizedAmbiguousTimestamp.1.0 timestamp
2 | float32[3] radian_per_second
3 | @assert _offset_ / 8 == {15}
4 | @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

```

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

```

6.16.2 WideScalarFull 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

```

1 | uavcan.time.SynchronizedAmbiguousTimestamp.1.0 timestamp
2 | float64 second
3 | @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

```

1 | uavcan.time.SynchronizedAmbiguousTimestamp.1.0 timestamp
2 | float32 coulomb
3 | @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

```

1 | uavcan.time.SynchronizedAmbiguousTimestamp.1.0 timestamp
2 | float32 ampere
3 | @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

```

1 | uavcan.time.SynchronizedAmbiguousTimestamp.1.0 timestamp
2 | float32 joule
3 | @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

```

1 | uavcan.time.SynchronizedAmbiguousTimestamp.1.0 timestamp
2 | float32 meter
3 | @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

```

1 | uavcan.time.SynchronizedAmbiguousTimestamp.1.0 timestamp
2 | float32[3] meter
3 | @assert _offset_ / 8 == {15}
4 | @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

```

1 | uavcan.time.SynchronizedAmbiguousTimestamp.1.0 timestamp
2 | float64[3] meter
3 | @assert _offset_ / 8 == {27}
4 | @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

```

1 | uavcan.time.SynchronizedAmbiguousTimestamp.1.0 timestamp
2 | float32 tesla
3 | @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

```

1 | uavcan.time.SynchronizedAmbiguousTimestamp.1.0 timestamp
2 | float32[3] tesla
3 | @assert _offset_ / 8 == {15}
4 | @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

```

1 | uavcan.time.SynchronizedAmbiguousTimestamp.1.0 timestamp
2 | float32 kilogram
3 | @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

```

1 | uavcan.time.SynchronizedAmbiguousTimestamp.1.0 timestamp
2 | float32 watt
3 | @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

```

1 | uavcan.time.SynchronizedAmbiguousTimestamp.1.0 timestamp
2 | float32 pascal
3 | @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

```

1 | uavcan.time.SynchronizedAmbiguousTimestamp.1.0 timestamp
2 | float32 kelvin
3 | @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

```

1 | uavcan.time.SynchronizedAmbiguousTimestamp.1.0 timestamp
2 | float32 meter_per_second
3 | @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

```

1 | uavcan.time.SynchronizedAmbiguousTimestamp.1.0 timestamp
2 | float32[3] meter_per_second
3 | @assert _offset_ / 8 == {15}
4 | @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

```

1 | uavcan.time.SynchronizedAmbiguousTimestamp.1.0 timestamp
2 | float32 volt
3 | @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

```

1 | uavcan.time.SynchronizedAmbiguousTimestamp.1.0 timestamp
2 | float32 cubic_meter
3 | @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

```
1 | uavcan.time.SynchronizedAmbiguousTimestamp.1.0 timestamp
2 | float32 cubic_meter_per_second
3 | @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](#).

DRAFT

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
<ul style="list-style-type: none"> • 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. 	<p>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.</p>

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 11898-2⁶³.

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.

⁶³Also known as *high-speed CAN*.

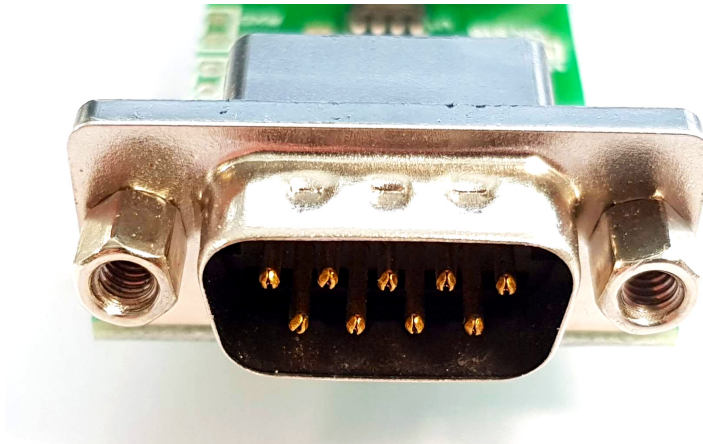


Figure 7.1: UAVCAN D-Sub device connector example.

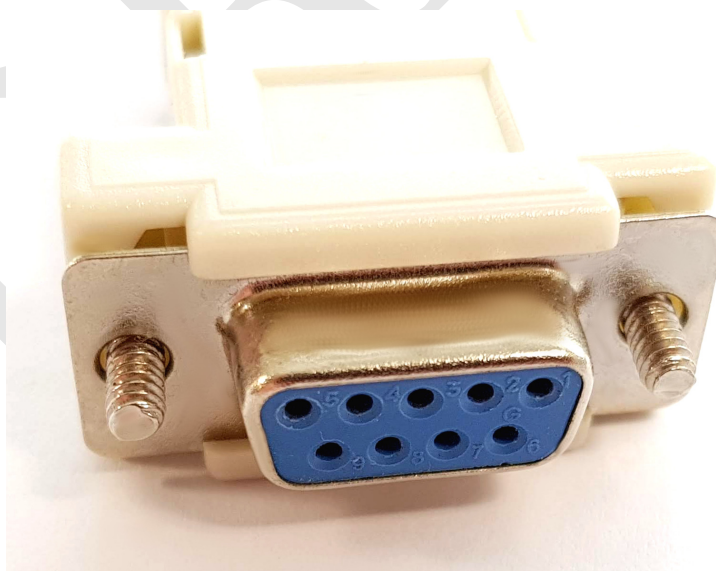


Figure 7.2: UAVCAN D-Sub cable connector example.

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
<ul style="list-style-type: none"> • 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. 	<ul style="list-style-type: none"> • 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 11898-2⁶⁴.

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.

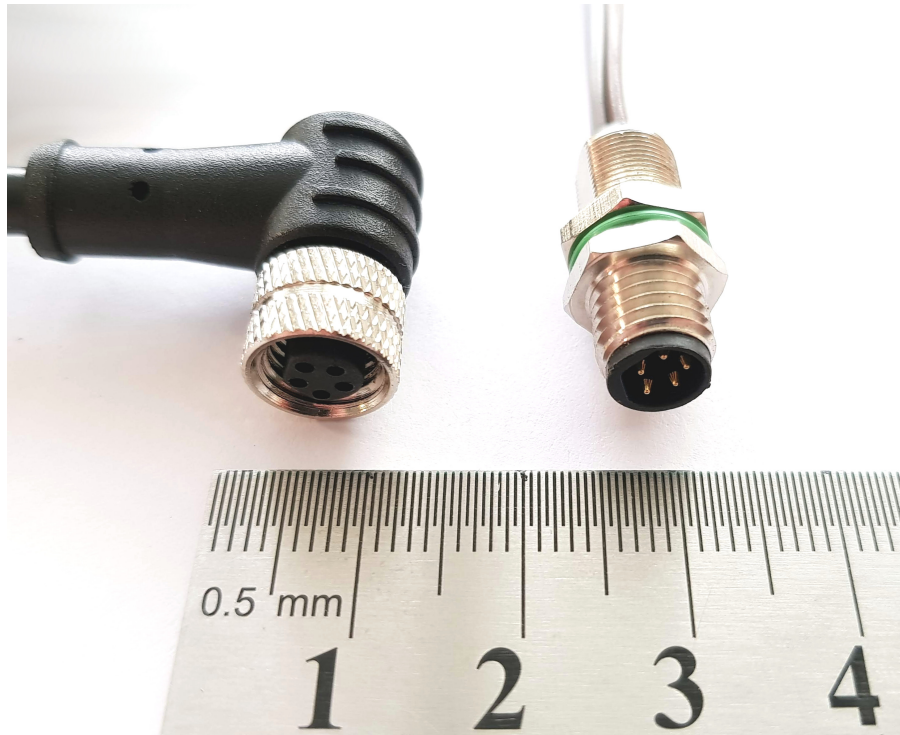
Devices that are powered from the bus should expect 18.0–30.0 V on the bus power line. The maximum recommended current draw from the bus is 0.5 A per device.

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	

⁶⁴Also 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.

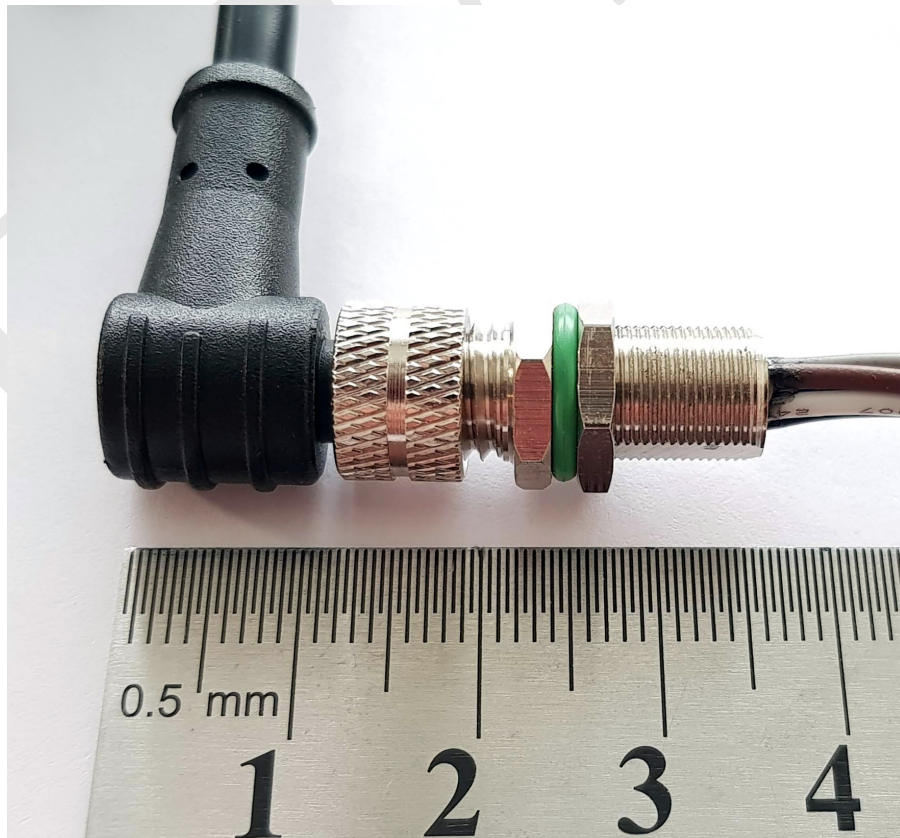


Figure 7.4: UAVCAN M8 assembled connector pair example.

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
<ul style="list-style-type: none"> 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. 	<ul style="list-style-type: none"> 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, multi-strand. Non-twisted (flat) cables can only be used in very small deployments free of significant EMI⁶⁵; 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	

⁶⁵Electromagnetic interference.

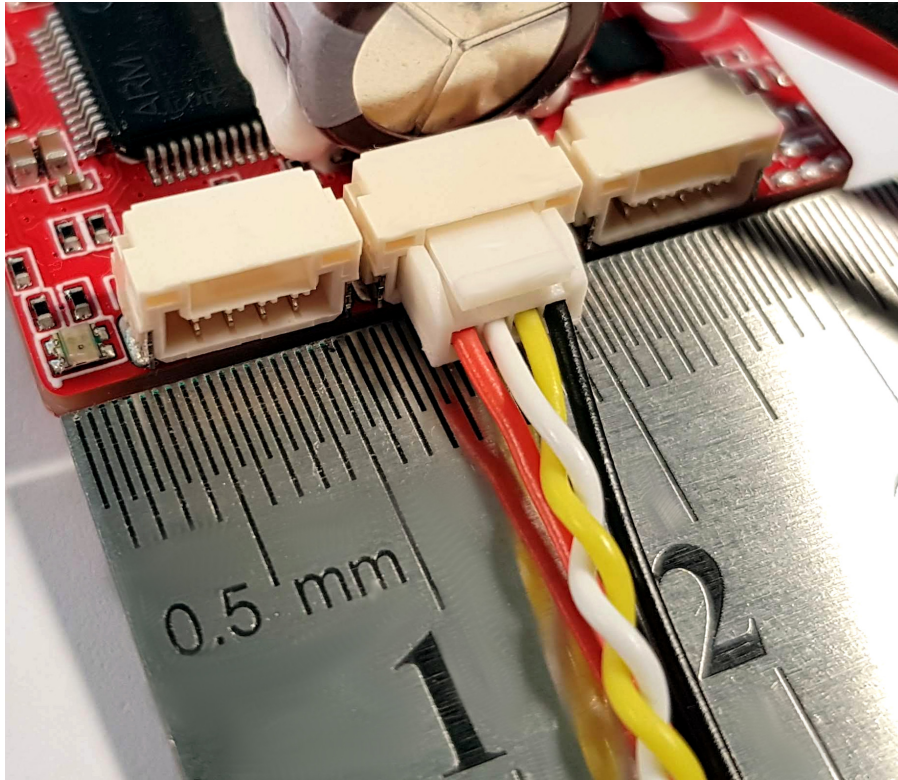


Figure 7.5: UAVCAN Micro right-angle connectors with a twisted pair patch cable connected.

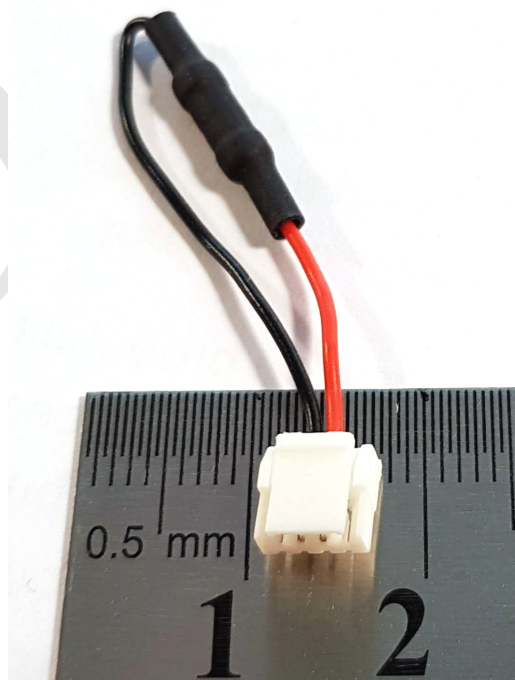


Figure 7.6: UAVCAN Micro CAN bus termination plug.

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.

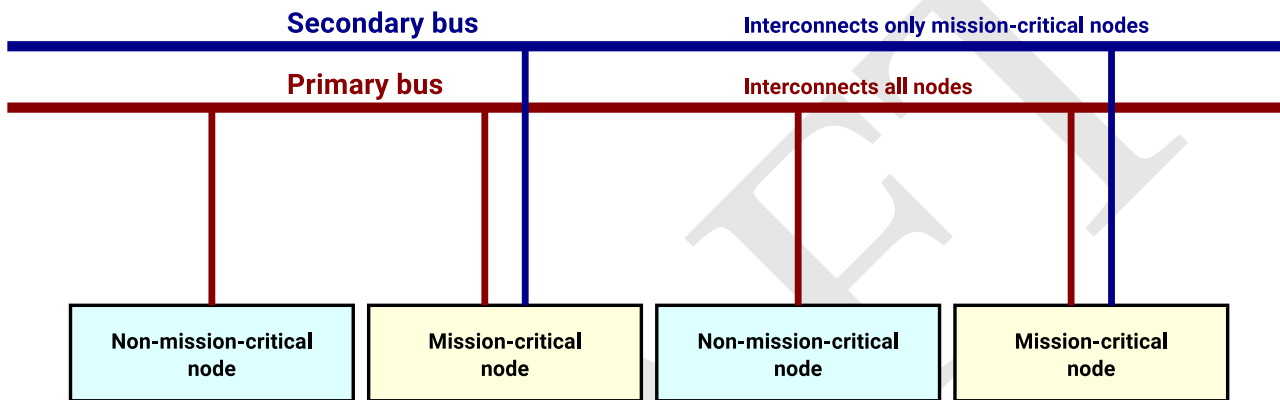


Figure 7.7: Non-uniform transport redundancy.

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.

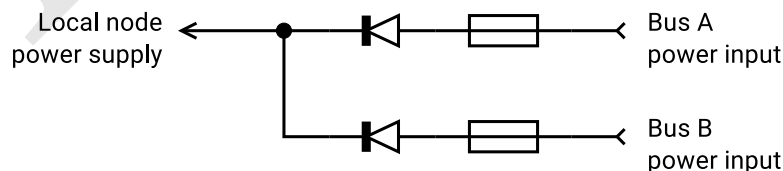


Figure 7.8: Simplified conceptual power sinking node design schematic.

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.

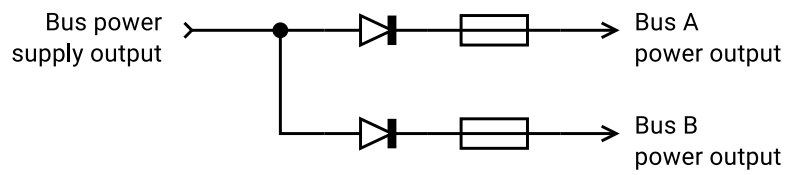


Figure 7.9: Simplified conceptual power sourcing node design schematic.